HAL
open science

Vérification des systèmes multi-agents comportementalistes par le moyen des simulations formellement guidées

Paulo Salem Da Salem da Silva Silva

To cite this version:

Paulo Salem Da Salem da Silva Silva. Vérification des systèmes multi-agents comportementalistes par le moyen des simulations formellement guidées. Other [cs.OH]. Université Paris Sud - Paris XI; Universidade de São Paulo (Brésil), 2011. English. NNT : 2011PA112267 . tel-00656809

HAL Id: tel-00656809

https://theses.hal.science/tel-00656809

Submitted on 5 Jan 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITÉ PARIS-SUD 11

ÉCOLE DOCTORALE: Informatique

Laboratoire de Recherche en Informatique

DISCIPLINE *Informatique*

THÈSE DE DOCTORAT

en co-tutelle avec l'Universidade de São Paulo

soutenu le 28/11/2011

par

Paulo SALEM DA SILVA

Verification of behaviourist multi-agent systems by means of formally guided simulations

Co-directrice de thèse:	Marie-Claude GAUDEL	Professeur émérite (Université Paris-Sud 11)
Co-directrice de thèse:	Ana Cristina VIEIRA DE MELO	Professeur (Universidade de São Paulo)

Composition du jury:

Président du jury:	Flavio SOARES CORREA DA SILVA	Professeur (Universidade de São Paulo)
Rapporteurs:	Augusto SAMPAIO	Professeur (Universidade Federal de Pernambuco)
	Jean-Pierre BRIOT	Directeur de recherche (CNRS)
Examinateurs:	Marcelo FINGER	Professeur (Universidade de São Paulo)

Science in the course of the few centuries of its history has undergone an internal development which appears to be not yet completed. One may sum up this development as the passage from contemplation to manipulation.

– Bertrand Russell, *The Scientific Outlook*

Acknowledgements

My gratitude goes to the people who helped me either in the technical and intellectual developments themselves or in the surprisingly hard work involved in getting these developments through the academic bureaucracy. As it is usual, I fear I may forget important names and contributions, but an attempt must be made to give credit to whom credit is due.

This thesis is the result of a cooperation between two universities, with an advisor from each one. Both were essential during the whole process, without them this work would not exist. Thanks to my Brazilian advisor, Ana Cristina, I got to know many of the central technical notions used in this thesis. Moreover, her many advices and comments during the whole time enriched my work considerably. I am also particularly grateful for her incentive for me to pursue my interests and to work on my own ideas, with all the advantages and disadvantages involved. Ana was also very helpful in every bureaucratic challenge, which were numerous. My French advisor, Marie-Claude, provided crucial technical feedback and suggestions, which greatly improved my work. Her attention to detail and high standards were fundamental in bringing the thesis to its current form. I am also thankful for her logistic and bureaucratic support in France in every respect, often going well beyond her obligations, without which I doubt my stay there – previously only an abstract concept to me – could have worked as well as it did.

During the doctorate I also had very fruitful interactions with several friends, colleagues and professors, many of whom I found along the way. Through technical remarks, enlightening conversation and friendly support, they helped in creating a sound and stimulating environment. I am specially thankful for my time with Agnes Helena Chiuratto, Alvaro Heiji Miyazawa, Marcelo Hashimoto, Matthias P. Krieger, Markus (Makarius) Wenzel, Zahia Guessoum, Leliane Nunes de Barros, Renata Wassermann, Domingos Soares Neto, Carlos Cardonha, Mounir Lallali, Abderrahmane (Abdu) Feliachi, Márcio Moretto Ribeiro, Ricardo Herrmann, Johan Oudinet, Lina Bentakouk, François-Élie Calvier, Domingos Dellamonica Jr, Juliana Barby Simão, Ellen Hidemi Fukuda, Thiago Palmieri, Gordana Manic, Ricardo Andrade and Wendel Scardua. My parents, who put great value on a solid education, were also very

supportive of this enterprise.

This is an unusually long thesis, and I am grateful that the jury members actually took the time to read it. They provided detailed and useful feedback, which not only improved the text itself, but also gave me an opportunity to reflect in new ways about my work.

Finally, I would like to acknowledge the effective and professional secretarial support I got from Pinho at USP and Stéphanie Druetta at Paris-Sud.

Abstract

Multi-agent systems (MASs) can be used to model phenomena that can be decomposed into several interacting agents which exist within an environment. In particular, they can be used to model human and animal societies, for the purpose of analysing their properties by computational means. This thesis is concerned with the automated analysis of a particular kind of such social models, namely, those based on behaviourist principles, which contrasts with the more dominant cognitive approaches found in the MAS literature. The hallmark of behaviourist theories is the emphasis on the definition of behaviour in terms of the interaction between agents and their environment. In this manner, not merely reflexive actions, but also learning, drives, and emotions can be defined. More specifically, in this thesis we introduce a formal agent architecture (specified with the Z Notation) based on the Behaviour Analysis theory of B. F. Skinner, and provide a suitable formal notion of environment (based on the π-calculus process algebra) to bring such agents together as an MAS.

Simulation is often used to analyse MASs. The techniques involved typically consist in implementing and then simulating a MAS several times to either collect statistics or see what happens through animation. However, simulations can be used in a more verification-oriented manner if one considers that they are actually explorations of large state-spaces. In this thesis we propose a novel verification technique based on this insight, which consists in simulating a MAS in a guided way in order to check whether some hypothesis about it holds or not. To this end, we leverage the prominent position that environments have in the MASs of this thesis: the formal specification of the environment of a MAS serves to compute the possible evolutions of the MAS as a transition system, thereby establishing the state-space to be investigated. In this computation, agents are taken into account by being simulated in order to determine, at each environmental state, what their actions are. Each simulation execution is a sequence of states in this state-space, which is computed on-the-fly, as the simulation progresses.

The hypothesis to be investigated, in turn, is given as another transition system, called a simulation purpose, which defines the desirable and undesirable

simulations (e.g., "every time the agent does X, it will do Y later"). It is then possible to check whether the MAS satisfies the simulation purpose according to a number of precisely defined notions of satisfiability. Algorithmically, this corresponds to building a synchronous product of these two transitions systems (i.e., the MAS's and the simulation purpose) on-the-fly and using it to operate a simulator. That is to say, the simulation purpose is used to guide the simulator, so that only the relevant states are actually simulated. By the end of such an algorithm, it delivers either a conclusive or an inconclusive verdict. If conclusive, it becomes known whether the MAS satisfies the simulation purpose with respect to the observations made during simulations. If inconclusive, it is possible to perform some adjustments and try again.

In summary, then, in this thesis we provide four novel elements: (i) an agent architecture; (ii) a formal specification of the environment of these agents, so that they can be composed into an MAS; (iii) a structure to describe the property of interest, which we named simulation purpose; and (iv) a technique to formally analyse the resulting MAS with respect to a simulation purpose. These elements are implemented in a tool, called Formally Guided Simulator (FGS). Case studies executable in FGS are provided to illustrate the approach.

Keywords: multi-agent systems, behaviourism, environments, formal methods, formal verification, simulation, model-based testing.

An extended version of this abstract is given in Section F.1 of Appendix F.

Resumo

Sistemas multi-agentes (SMAs) podem ser usados para modelar fenômenos que podem ser decompostos em diversos agentes que interagem entre si dentro de um ambiente. Em particular, eles podem ser usados para modelar sociedades humanas e animais, com a finalidade de se analisar as suas propriedades computacionalmente. Esta tese trata da análise automatizada de um tipo particular de tais modelos sociais, a saber, aqueles baseados em princípios behavioristas, o que contrasta com as abordagens cognitivas mais dominante na literatura de SMAs. A principal característica das teorias behaviorista é a ênfase na definição do comportamento em termos da interação entre agentes e seu ambiente. Desta forma, não apenas ações reflexivas, mas também de aprendizado, motivações, e as emoções podem ser definidas. Mais especificamente, nesta tese apresentamos uma arquitetura de agentes formal (especificada através da Notação Z) baseada na teoria da Análise do Comportamento de B. F. Skinner, e fornecemos uma noção adequada e formal de ambiente (com base na álgebra de processos π-calculus) para colocar tais agentes juntos em um SMA.

Simulações são freqüentemente utilizadas para se analisar SMAs. As técnicas envolvidas tipicamente consistem em simular um SMA diversas vezes, seja para coletar estatísticas, seja para observar o que acontece através da animações. Contudo, simulações podem ser usadas de forma a pertmitir a realização de verificações automatizadas do SMA caso sejam entendidas como explorações de grandes espaços-de-estados. Nesta tese propomos uma técnica de verificação baseada nessa observação, que consiste em simular um SMA de uma forma guiada, a fim de se determinar se uma dada hipótese sobre ele é verdadeira ou não. Para tal fim, tiramos proveito da importância que os ambientes têm nesta tese: a especificação formal do ambiente de um SMA serve para calcular as evoluções possíveis do SMA como um sistema de transição, estabelecendo assim o espaço-de-estados a ser investigado. Neste cálculo, os agentes são levados em conta simulando-os, a fim de determinar, em cada estado do ambiente, quais são suas ações. Cada execução da simulação é uma seqüência de estados nesse espaço-de-estados, que é calculado em tempo de execução, conforme a simulação progride.

A hipótese a ser investigada, por sua vez, é dada como um outro sistema de transição, chamado propósito de simulação, o qual define as simulações desejáveis e indesejáveis (e.g., "sempre que o agente fizer X, ele fará Y depois"). Em seguida, é possível verificar se o SMA satisfaz o propósito de simulação de acordo com uma série de relações de satisfatibilidade precisamente definidas. Algoritmicamente, isso corresponde a construir um produto síncrono desses dois sistemas de transições (i.e., o do SMA e o do propósito de simulação) em tempo de execução e usá-lo para operar um simulador. Ou seja, o propósito de simulação é usado para guiar o simulador, de modo que somente os estados relevantes sejam efetivamente simulados. Ao terminar, um tal algoritmo pode fornecer um veredito conclusivo ou inconclusivo. Se conclusivo, descobre-se se o SMA satisfaz ou não o propósito de simulação com relação às observações feitas durante as simulações. Se inconclusivo, é possível realizar alguns ajustes e tentar novamente.

Em resumo, portanto, nesta tese propomos quatro novos elementos: (i) uma arquitetura de agente, (ii) uma especificação formal do ambiente desses agentes, de modo que possam ser compostos em um SMA, (iii) uma estrutura para descrever a propriedade de interesse, a qual chamamos de propósito de simulação, e (iv) uma técnica para se analisar formalmente o SMA resultante com relação a um propósito de simulação. Esses elementos estão implementados em uma ferramenta, denominada Simulador Formalmente Guiado (FGS, do inglês *Formally Guided Simulator*). Estudos de caso executáveis no FGS são fornecidos para ilustrar a abordagem.

Palavras-chave: sistemas multi-agentes, comportamentalismo, ambientes, métodos formais, verificação formal, simulação, teste baseado em modelos.

Uma versão estendida deste resumo é dada na Seção F.2 do Apêndice F.

Résumé

Les systèmes multi-agents (SMA) peuvent être utilisé pour modéliser les phénomènes qui peuvent être décomposés en plusieurs agents qui interagissent et qui existent au sein d'un environnement. En particulier, ils peuvent être utilisés pour modéliser les sociétés humaines et animales, aux fins de l'analyse de leurs propriétés par des moyens de calcul. Cette thèse est consacrée à l'analyse automatisée d'un type particulier de ces modèles sociaux, à savoir, celles qui sont fondées sur les principes comportementalistes, qui contrastent avec les approches cognitives plus dominante dans la littérature des SMAs. La caractéristique des théories comportementalistes est l'accent mis sur la définition des comportements basée sur l'interaction entre les agents et leur environnement. De cette manière, non seulement des actions réflexives, mais aussi d'apprentissage, les motivations, et les émotions peuvent être définies. Plus précisément, dans cette thèse, nous introduisons une architecture formelle d'agent (spécifiée avec la Notation Z) basée sur la théorie d'analyse comportementale de B. F. Skinner, ainsi que une notion appropriée et formelle de l'environnement (basée sur l'algèbre de processus π-calculus) pour mettre ces agents ensemble dans un SMA.

La simulation est souvent utilisée pour analyser les SMAs. Les techniques consistent généralement à simuler le SMA plusieurs fois, soit pour recueillir des statistiques, soit pour voir ce qui se passe à travers l'animation. Toutefois, les simulations peuvent être utilisés d'une manière plus orientée vers la vérification si on considère qu'elles sont en réalité des explorations de grandes espaces d'états. Dans cette thèse nous proposons une technique de vérification nouvelle basé sur cette idée, qui consiste à simuler un SMA de manière guidée, afin de vérifier si quelques hypothèses sur lui sont confirmées ou non. À cette fin, nous tirons profit de la position privilégiée que les environnements sont dans les SMAs de cette thèse: la spécification formelle de l'environnement d'un SMA sert à calculer les évolutions possibles du SMA comme un système de transition, établissant ainsi l'espace d'états à vérifier. Dans ce calcul, les agents sont pris en compte en les simulant afin de déterminer, à chaque état de l'environnement, quelles sont leurs actions. Chaque exécution de la simulation est une séquence d'états dans cet espace d'états, qui est calculée à la volée, au

fur et à mesure que la simulation progresse.

L'hypothèse à étudier, à son tour, est donnée comme un autre système de transition, appelé objectif de simulation, qui définit les simulations désirables et indésirables (e.g., "chaque fois que l'agent fait X, il fera Y plus tard"). Il est alors possible de vérifier si le SMA est conforme à l'objectif de simulation selon un certain nombre de notions de satisfiabilité très précises. Algorithmiquement, cela correspond à la construction d'un produit synchrone de ces deux systèmes de transitions (i.e., celui du SMA et l'objectif de simulation) à la volée et à l'utiliser pour faire fonctionner un simulateur. C'est-à-dire, l'objectif de simulation est utilisé pour guider le simulateur, de sorte que seuls les états concernés sont en réalité simulés. À la fin d'un tel algorithme, il délivre un verdict concluant ou non concluant. Si c'est concluant, il est connu que le SMA est conforme à l'objectif de simulation par rapport aux observations qui ont été faites lors des simulations. Si c'est non-concluant, il est possible d'effectuer quelques ajustements et essayer à nouveau.

En résumé, donc, dans cette thèse nous fournissons quatre nouveaux éléments: (i) une architecture d'agent; (ii) une spécification formelle de l'environnement de ces agents, afin qu'ils puissent être composés comme un SMA; (iii) une structure pour décrire les propriétés d'intérêt, que nous avons nommée objectif de simulation, et (iv) une technique pour l'analyse formelle du SMA résultant par rapport à un objectif de simulation. Ces éléments sont mis en œuvre dans un outil, appelé Simulateur Formellement Guidé (FGS, de l'Anglais *Formally Guided Simulator*). Des études de cas exécutables dans FGS sont fournies pour illustrer l'approche.

Mots-clés: systèmes multi-agents, comportementalisme, environements, méthodes formelles, vérification formelle, simulation, test basé sur des modèles.

Une version étendu de ce résumé est donné dans la Section F.3 de l'Annexe F.

Contents

I Overview **1**

1 Introduction **3**
1.1 Automation of Experiments 9
1.2 Notation and other Conventions 11
1.3 Thesis Organization 11

2 Related Work **13**
2.1 Autonomous Agents and Multi-Agent Systems 13
2.1.1 Agent Models and Architectures 14
2.1.2 Environments in Multi-Agent Systems 20
2.1.3 Multi-Agent Based Simulation 22
2.2 Formal Verification 23
2.2.1 Model Checking 23
2.2.2 Model-Based Testing 24
2.2.3 Runtime Verification 26
2.2.4 Process Algebras 27
2.2.5 Formal Development 28
2.3 Other Influences 28
2.3.1 Non-Agent Based Simulation Methods 28
2.3.2 Software Components 29
2.4 Formal Approaches to Multi-Agent Systems 30
2.4.1 Formal Specification of Agent Architectures 30
2.4.2 Formal Specification of Environments 30
2.4.3 Formal Verification and MAS Simulation 31
2.4.4 Model Checking of MAS 31

3 Contribution of this Thesis **33**
3.1 Agent Architecture 33
3.1.1 Comparison with Other Approaches 36
3.2 Environment Model 41
3.2.1 Comparison with Other Approaches 43
3.3 Transition Systems and Semantics 45
3.3.1 Comparison with Other Approaches 45

3.4 Verification Technique 46
3.4.1 Comparison with Other Approaches 49
3.5 Tool Implementation 51
3.5.1 Comparison with Other Approaches 52
3.6 Conclusion 55

II Multi-Agent Systems 57

4 Behaviourist Agent Architecture 59
4.1 Adaptation and Learning 60
4.2 Formal Specification 62
4.2.1 Preliminary Definitions 63
4.2.2 Stimulation 65
4.2.3 General Responding 73
4.2.4 Operant Behaviour 83
4.2.5 Respondent Behaviour 92
4.2.6 Drives 95
4.2.7 Emotions 98
4.2.8 Subsystems Integration 102
4.3 Conclusion 104

5 EMMAS 105
5.1 The Role of Environments 107
5.2 Environment Model 108
5.2.1 Underlying Elementary π-Calculus Events 109
5.2.2 Environment Operations 110
5.2.3 Environment Structures 111
5.3 Convenience Elements and Operations 115
5.3.1 Composition Operators 115
5.3.2 Core Operations 117
5.3.3 Sets 119
5.3.4 Predicates and Logical Formulas 120
5.3.5 Quantification 120
5.3.6 Complex Operations 121
5.4 Conclusion 122

III Formal Analysis and Verification 125

6 Transition Systems and Semantics 127
6.1 Annotated Transition Systems 127
6.2 EMMAS Semantics 130
6.2.1 Preliminary Definitions 130

6.2.2 Building the Transition System 132
6.3 Conclusion 137

7 Verification Technique 139
7.1 Making the Env. ATS Suitable for Sim. and Verif. 140
7.1.1 Step 1 143
7.1.2 Step 2 145
7.2 Simulation Purposes 147
7.3 Synchronous Product of an ATS and a SP 148
7.4 Satisfiability Relations 151
7.5 Verification Algorithms 154
7.5.1 Simulator Interface 154
7.5.2 Feasibility Verification 155
7.5.3 Refutability Verification 162
7.5.4 Certainty Verification 162
7.5.5 Impossibility Verification 162
7.6 Analysis of the Algorithms 162
7.6.1 Justification of Completeness 166
7.6.2 Justification of Soundness 167
7.6.3 Justification of Termination 176
7.6.4 Justification of Correctness 179
7.6.5 Justification of Worst-Case Complexities 179
7.7 Conclusion 183

IV Implementation 185

8 Simulator Implementation 187
8.1 Architecture of FGS 188
8.1.1 Components 189
8.1.2 Scenarios 190
8.1.3 Experiments 190
8.2 Simulation Execution and Analysis 190
8.3 Behaviourist Agent Architecture Component 192
8.4 π-Calculus Simulation Library 192
8.4.1 Optimizations 193
8.5 Conclusion 197

9 Case Studies 199
9.1 Single Agent Examples 200
9.1.1 Pavlovian Dog: Classical Conditioning 200
9.1.2 Worker: Operant Chaining 206
9.2 Multi-Agent Examples 210
9.2.1 Violent Child: Behaviour Elimination 210

9.2.2 Factory: Rearranging a Social Network 216
9.2.3 School Children: From Chaos to Order 224
9.2.4 Online Social Networks: Spreading a Message 232
9.3 Conclusion 237

V Conclusion 241

10 Conclusion 243

A Full Agent Specification 247
A.1 Formal Specification of Agent Behaviour 247
A.1.1 Preliminary Definitions 247
A.1.2 Stimulation 251
A.1.3 General Responding 262
A.1.4 Operant Behaviour 275
A.1.5 Respondent Behaviour 284
A.1.6 Drives 287
A.1.7 Emotions 289
A.1.8 Subsystems Integration 295

B Input Files and Tool Output for the Case Studies 297
B.1 Pavlovian Dog 297
B.1.1 Agent 297
B.1.2 Scenario 300
B.1.3 Experiment 301
B.1.4 Result 304
B.2 Worker 306
B.2.1 Agent 306
B.2.2 Scenario 308
B.2.3 Experiment 309
B.2.4 Result 311
B.3 Violent Child 312
B.3.1 Agents 312
B.3.2 Scenario 317
B.3.3 Experiment 319
B.3.4 Result 324
B.4 Factory 327
B.4.1 Agents 327
B.4.2 Scenario 338
B.4.3 Experiment 345
B.4.4 Result 349
B.5 School Children 351
B.5.1 Agents 351

B.5.2 Scenario 358
B.5.3 Experiment 362
B.5.4 Result 363
B.6 Online Social Network 364
B.6.1 Agents 364
B.6.2 Scenario 376
B.6.3 Experiment 379
B.6.4 Result 381

C Simulator Input Format and Parameters 385
C.1 Input Format 385
C.1.1 Behaviourist Agent Architecture 386
C.1.2 Scenarios 391
C.1.3 Experiments 395
C.2 Parameters 398

D Z Notation Overview 399
D.1 Types, Functions and Predicates 399
D.2 Stateless Definitions 399
D.3 State Schemas 401
D.4 Operation Schemas 402
D.5 Schema Calculus 403
D.6 Refinement 404

E π-calculus Overview 407

F Extended Abstracts 411
F.1 Extended Abstract 411
F.2 Resumo Estendido 422
F.3 Résumé Étendu 433

Glossary 445

Acronyms 449

Bibliography 449

Index 467

Part I

Overview

CHAPTER 1

Introduction

Is there a way to model and analyse human as well as animal societies? This rather general question, either implicitly or explicitly, has been asked since at least classical antiquity[1]. Whole scientific fields, such as economics, psychology and sociology have been developed to address specific versions of it. In all of them, though, the difficulties involved are great: not only is it necessary to somehow make sense of individual organisms, but it is also imperative to understand their mutual relations; theories must be simple to be manageable, but not so simple as to be meaningless; variety between individuals must be taken into account, and yet cannot be considered beyond a point in which there is nothing but confusion; in the face of complexity, analysis methods must at the same time be effective and efficient.

Modern computing has given scientists a new set of tools to deal with these matters. In particular, as long as a theory can be put in the terms of a computer program, it can also be subject to systematic and automated scrutiny that would otherwise be too tiresome for human beings to pursue. Simplifications can be made less simple and more accurate because part of the job can be transferred to a machine. This possibility raises the similar – but fundamentally new – question: is there a way to model and analyse human as well as animal societies *through computing*? Such is the motivation, in its most general aspect, that guides us in this thesis. The problems we address here, then, are steps in this direction.

It is not difficult to see why this new *computational* problem is distinct from the old one. Imagine, if you will, that a careful economist surveys his town and describes, through some complicated set of rules, the idiosyncratic personalities of each townsman, as well as their relations to each other. How

[1] For instance, in Plato's *Republic* and Aristotle's *Politics*.

is he to go about analysing this descriptive model? Perhaps he can address certain questions, such as determining the most influential person by looking at the individual that has more friends. Nonetheless, questions concerning how his model will evolve over time quickly become difficult, for the possibilities offered by a sufficiently rich model are staggering. Will there be an economic depression or not? The answer depends partly on the behaviour of each individual. If there are many different and complicated individuals that relate in non-homogeneous ways, it is clear that the possible evolutions of the model are many, each possibly leading to different conclusions. This requires explicit – and boring – calculations of how the model will actually evolve. Without computing machinery, therefore, it is not a practical method. The traditional, non-computational, solution to such problems is to simplify the model so that solutions can be calculated more easily. But with computing one may actually pursue the evolution of complicated models in many different circumstances. This brings two new issues: how to describe models so that their evolutions can be computed, and how to make sense of the wealth of computed evolutions.

In computer science, questions such as these have been considered in the field of *Multi-Agent Systems*. Many situations can be described in terms of independent *agents* which interact within some *environment* in order to achieve their aims. For example, not only phenomena related to human societies, but also those concerning neural tissue and computer networks, different as they may be, all share this characteristic. A system that can be seen in this way is called a *multi-agent system (MAS)*(Weiss, 1999; Wooldridge, 2009).

As the above examples suggest, agents can be either artificial entities (e.g., computers, software) or natural ones (e.g., humans, animals). Roughly, in the former case one is mostly worried about how to *implement* an agent so that it is capable of performing certain tasks, whereas in the latter case the focus is on *modelling* the behaviour found in nature so that it can be investigated by computational means. It is this latter possibility that concerns us in this thesis.

To describe an MAS, one needs specific notions of agents and environments. With respect to agents, much work has been done in trying to understand and model so-called *intelligent* and *cognitive* agents. These approaches focus largely on what constitute rational decisions, specially in the case of agents with limited computing capabilities (e.g., all of us). The *Beliefs-Desires-Intentions* (BDI) architecture (Bratman, 1987; Cohen and Levesque, 1990; Rao and Georgeff, 1995) is a well-known example.

Behaviour of organisms, however, is sometimes better described in different terms. A dog does not reason that it will die if it does not eat[2]; rather, it

[2]Assuming, of course, that dogs cannot foresee their own deaths in the same way that

has a drive to seek food when hungry. If it has learned that whenever his master whistles he provides food, the dog will salivate at the sound of the whistle – without thinking. These observations suggest that a different focus in agent modelling is possible. This thesis provides such a model, based on the psychology theory known as Behaviour Analysis (Skinner, 1953), a particular branch of the behaviourist school of thought. In this theory, the actions of agents are seen as the result of past stimulation and certain innate parameters according to behavioural laws. One is not interested in mental qualities such as the nature of reason, but merely in the prediction and control of behaviour by means of environmental stimulation. This point of view, though classical within psychology, is scarce in the MAS literature. As a contribution in this sense, this thesis introduces the **Behaviourist Agent Architecture**.

In relation to agents, environments of MASs have received comparatively very little attention, as the survey of Weyns *et al.* (2005) points out. The environment model of Ferber and Müller (1996) is one exception. In this thesis we propose the **Environment Model for Multi-Agent Systems (EMMAS)**, which is designed to work with our agent architecture. Since the psychology theory from which we draw from puts great emphasis in the relation between agents and their environment, it is clear that this is an important aspect of our MASs. Furthermore, we shall see that our environments have certain particular mathematical features that help in their analysis.

The purpose of an MAS model is to be studied so that its properties can be understood. There are, of course, a number of ways in which this can be accomplished, ranging from traditional mathematical approaches (e.g., by using equations and calculating their properties) to fully automated and exhaustive formal verification (e.g., by means of Model Checking). The technique to be employed, however, is not arbitrary, for it both imposes restrictions on how the MAS model can be specified and defines what kinds of properties can be investigated. In general, the more details are allowed in a model, the harder it is to determine its properties. Consider again a model of a society composed of many different agents, each containing its own independent set of possible behaviours. We have seen that unless simplifications can be found, a manual mathematical analysis would be too tedious and error prone to be carried out. On the other hand, automated and exhaustive analyses would also face severe challenges, for the state-spaces involved can easily become too large. This is known as the *state explosion problem*, and is usually caused by the combinatorial nature of the possible communications between agents.

Nevertheless, it is usually possible to simulate complicated MASs. That is to say, given an MAS, one may calculate several sequences of states in order to explore some of its possible behaviours, and thus gain at least some partial

we humans can.

knowledge about its properties.[3] A technique often employed to this end involves programming the agents and their environment using some general purpose programming language (e.g., Java) and then running the resulting program several times and under different circumstances (Gilbert and Bankers, 2002). Since such programming languages allow any computation to be specified, it follows that very detailed models can be built in this way. In such works, the analysis method of choice is usually the collection or optimization of statistics over several simulation runs (e.g., the mean value of a numeric variable over time). Examples of this approach include platforms such as Swarm (Minar *et al.*, 1996), MASON (Luke *et al.*, 2004) and Repast (North *et al.*, 2006).

A simulation performed in this manner is typically constrained only by its initial state. This is quite reasonable if the objective is merely to "see what happens" to the system under different circumstances of interest. However, as soon as the objective includes a more sophisticated assessment, this lack of constraints may become a hindrance. Consider, for example, a situation in which the objective is to study what happens to an agent when it is, say, hungry. The logical strategy in this case would be to simulate only the situations in which the agent may indeed become hungry. But since only the initial state is constrained, the simulation could possibly go constantly through states in which this is not the case (e.g., because food is often available).

Though they are not usually found on MAS simulations, such constraints over relevant states are common place in formal verification. They are usually specified in terms of temporal logic formulas or automata which define the property of interest and, consequently, the relevant portion of the state-space. Unfortunately, as we pointed out above, these techniques suffer from efficiency problems owing to the large size of state-spaces.

To address these problems, in this thesis we propose a way to combine the strengths of simulation with those of formal verification, thus creating a new technique to model and verify MASs. Our method consists in systematically guiding the simulation runs so that only the states relevant for the property of interest are actually simulated. We call the property being investigated a **simulation purpose**, because it defines the purpose of the simulation. During simulations, for reasons we explain in Section 1.1 below, agents and environment are used in different ways. The former is implemented (in accordance with the **Behaviourist Agent Architecture**), executed and examined as a black-box with interfaces, whereas the latter – together with a **simulation**

[3]Notice that by "simulation" we do not mean the formal relation among two transition systems, such as what Milner (1999) employs. As we explain, in this thesis a "simulation" refers – broadly – to an abstract reproduction of some target system by means of a detailed and executable model, in the same sense that, for instance, Ferber (1999) uses to describe multiagent simulations.

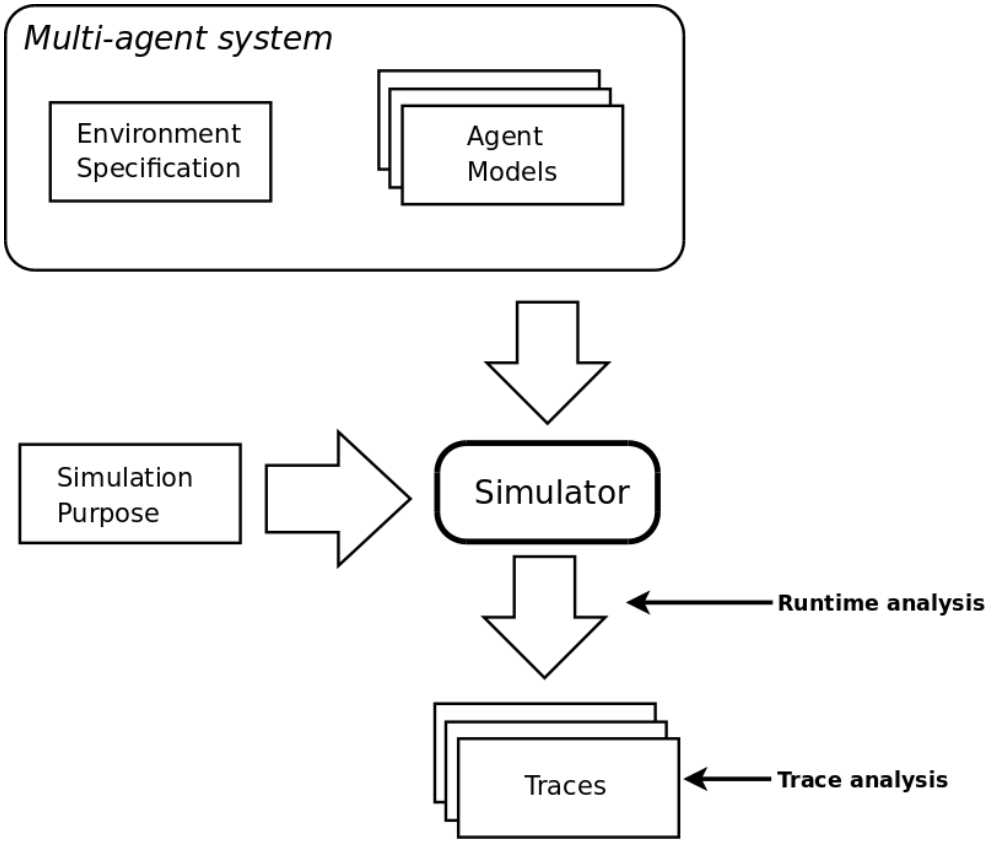


Figure 1.1: General architecture of FGS, our proposed tool. The simulator takes two inputs: (i) an MAS, composed by agent models and an environment specification; (ii) a *simulation purpose* to be tested. The simulation purpose specifies the property to be analysed and is used to guide the simulations (i.e., defines their purpose). The simulator then produces traces as outputs. In principle, verification can be done at the simulation runtime level, as well as at the trace level. However, the verification technique developed in this thesis concerns only runtime analysis, because this allows the simulations to be controlled in such a way that only the ones relevant to the specified simulation purpose are actually performed. It is worth to note, though, that trace level analysis can also be profitably employed for verification of MASs, and some examples found in the literature are given in Section 2.4.3 of Chapter 2.

purpose – provides the formal elements that are manipulated in a fine grained manner by the verification algorithms. This method is inspired by the use of formal *test purposes* in TGV (Jard and Jéron, 2005), a model-based software testing approach.

In short, then, in this thesis we provide four novel elements: (i) an agent architecture; (ii) a formal specification of the environment of these agents, so that they can be composed into an MAS; (iii) a structure to describe the property

of interest, which we named **simulation purpose**; and (iv) a technique to formally analyse the resulting MAS with respect to a **simulation purpose**. These elements are combined in a tool, called Formally Guided Simulator (FGS), as shown in Figure 1.1. In Chapter 3 we shall provide a technical summary of these artefacts and explain in more detail their contribution with respect to the current state of the art.

The analysis technique we propose is rather general, and as a consequence can be applied to many kinds of MASs. Nevertheless, we have opted to develop them in the context of a particularly suitable class of agents and environments (i.e., *behaviourist* MASs). As a consequence, the novel contribution of this thesis is twofold:

- New ways to model both agents and environments based on behaviourist principles;
- A technique to perform partial, but automated, formal verification in the MAS thus described.

These are closely related. The technique depends on the possibility of systematically exploring an environment, which thus has to be explicitly separated from agents. But this only makes sense if the phenomena concerning the agents themselves can be expressed in terms of environmental conditions, which is a strong point of the behaviourist perspective we adopt to design our agents.

It is important to note that despite the fact that the MASs defined and simulated in this thesis are based on an underlying psychology theory, we have not attempted to forecast the results of actual empirical experiments (e.g., using real animals) with them. Such an endeavour would require the consideration of many other complex issues, such as how to establish an exact match between real and simulated organisms, and it would also involve validation work concerning the psychology theory itself, which would be out of our scope. The approach developed in this thesis aims only at providing an approximation of actual animal behaviour, so that it can be represented in a computational form, though in a qualitative and limited manner. As we shall see throughout the text, our approach allows the investigation of many fundamental issues, such as the role of environments and agents, the importance of observable events, and the kinds of questions that can be formulated. Hence, this thesis is a step towards a more complete understanding of the computational modelling and analysis of such behavioural phenomena, and it provides an important basis for further progress. Nonetheless, as a more immediate practical application, the developments presented here can also be used in circumstances where only imitation of real behaviour is relevant, such as in games and other forms of interactive fiction.

1.1 Automation of Experiments

Programs are usually designed in order to accomplish something. That is to say, they are supposed to obey a specification[4]. If they indeed do so, they are deemed correct. Otherwise, they are considered incorrect. Verification, and formal verification in particular (where one has *formal* specifications of the expected behaviour), is thus concerned with determining whether or not programs satisfy specifications – from which one may infer the correctness of the program.

However, one may have a slightly different point of view on the matter. In our case, we use programs to model MASs. From our perspective of modellers, the MAS is not necessarily supposed to accomplish something, for it is merely a representation of a certain state of affairs, which may be outside of our control. In investigating it, we are thus not necessarily concerned with whether it is doing its job correctly. Indeed, we may very well ignore why the MAS was designed in the way it was. We just want to discover *what* it can or cannot do. To this end, we may also employ a specification. But it is the specification of a *hypothesis* to be investigated, and which can be either true or false. Notice the crucial difference: when verifying a program, the fact that the specification was violated indicates a problem in the program, and thus it is always undesirable; however, in our case, the fact that the hypothesis is violated is not, in principle, an indication of a problem either in the MAS or in the hypothesis itself. The judgement to be made depends of our objectives in each particular circumstance. Are we trying to discover some law about the MAS? In this case, if a hypothesis that represents this law turns out to be false, it is the hypothesis that is incorrect, not the MAS. Are we trying to engineer an MAS that obey some law? In this case we have the opposite, a falsified hypothesis indicates a problem in the MAS. This view is akin to that found in empirical sciences, in which scientists investigate hypotheses and make judgements in a similar manner.[5] In this respect, the main difference is that the empirical scientist studies the natural world *directly*, while we are concerned with *models* of nature in the form of MASs.

In an MAS, the description of environments is often much simpler than that of the agents. When this is the case, we can give a formal model for the environment and treat the agents therein as black-boxes (which obey certain interface requirements). As an example, let us consider a model of an online social network, where several persons exist and can interact with each other through the features of a website.[6] Clearly, the behaviour of each individual

[4]Even if the specification exists only in the mind of the programmer.

[5]In particular, the scientific implications of the falsifiability of hypotheses have been deeply investigated by Popper (1959), to whom we own part of the philosophical position outlined here.

[6]Actual examples of such networks currently include popular websites such

person is likely to be very complex, and if a model is given to them, it is possible that it will not be a simple one. But the environment, on the other hand, can be described by some formalism that merely define relations among agents (e.g., a process algebra such as the π-calculus of Milner, 1999), providing a much more tractable model. The purely formal manipulations, then, can be restricted to the environment model.

Notice that this is analogous to an experimental scientist working in his laboratory. The scientist is usually interested in discovering the properties of some agents, such as animals, chemicals, or elementary particles. He has no control over the internal mechanism of these agents – that is why experiments are needed. But he can control everything around them, so that they can be subject to conditions suitable for their study. These scientific experiments have some important characteristics:

- *Inputs* should be given to agents under experimentation;
- *Outputs* should be collected from these agents;
- Sometimes it is not possible to make some important measurement, and therefore experiments often yield *incomplete knowledge*;
- The experiment is designed to either confirm some expectation (a *success*) or refute it (a *failure*);
- The experiment should last a *finite amount of time*, since the life of the scientist is also finite;
- The experiment should be as *systematic* as possible, though exhaustiveness is not required. The important thing is to try as many *relevant situations* as possible. In particular, the scientist may control how to continue the experiment depending on how the agents react;
- The experiment should define a clear course of action from the start;
- The experiment can be a way to find out how to achieve a certain end, after trying many things. Therefore, it must be performed in a *constructive* manner, and not merely by deriving a contradiction;
- Absence of a success does not necessarily mean that there is no way to achieve a desired effect. Hence, it is convenient to know when something clearly indicates a failure.

as `www.facebook.com`, `www.twitter.com`, `plus.google.com`, `www.orkut.com` and `www.myspace.com`.

A simulation purpose is like such a scientist: it controls the direction of the simulation and determines whether something constitutes a success or a failure by following similar principles. An environment model, in turn, is similar to the experimental setup, with its several instruments and agents. The simulation purpose interacts with the environment model in order to achieve its aims. In this sense, hence, our approach can be seen as the automation of experiments to investigate the properties of an MAS.

1.2 Notation and other Conventions

Notation and other conventions particular to the subject of a chapter are introduced in the beginning of the relevant chapter itself. Technical terms are introduced by using ***this special font***, and referred to elsewhere by using **this other special font**.

THIS SPECIAL FONT

Our general mathematical notation is mostly standard and therefore does not require presentation. However, it is worth to define the meaning of a few symbols which do not enjoy such a universal acceptance:

- $\neg\, P$ denotes the set $\{\neg\, p \mid p \in P\}$, where P is a set of propositions.
- $\mathbb{P}(S)$ denotes the power set of a set S (i.e., the set of all subsets of S).

1.3 Thesis Organization

This thesis can be read serially, since chapters are given mostly in the order in which their content is needed. Exceptions to this order are clearly marked in the text with the appropriate references.

Very briefly, the content of the chapters are as follows:

Chapter 2 Describes the relevant related work, thus providing an account of the current state of the art.

Chapter 3 Provides a technical overview of the main elements introduced by this thesis and compare them to the existing state of the art.

Chapter 4 Contains an in-depth account of the **Behaviourist Agent Architecture**. This architecture is given using the Z Notation, of which a summary is provided in Appendix D.

Chapter 5 Defines the environment model, **EMMAS**. The theory developed here has a close relation to the agents described in Chapter 4:

those agents expect a certain kind of interaction with their environment, whereas the environment expects a certain kind of agent. Methodologically, this means that particular kinds of questions (i.e., of behaviourist nature) can be addressed because both agents and environments subscribe to common principles. Technically, this relation implies in certain formal requirements on both sides.

Chapter 6 Introduces the **annotated transition systems (ATSs)** which are to be subject to verification. After introducing such structures in their general form, the chapter employs them to provide the semantics of **EMMAS**. This allows the reduction of the MASs developed in Chapters 4 and 5 into a structure that can be subject to formal analyses. This semantics is quite general and is not tied to any particular application (e.g., simulation).

Chapter 7 Provides a more concrete version of the semantics of **EMMAS** (to allow its simulation) and presents the verification technique. Introduces **simulation purposes**, associated satisfiability relations, mathematical concepts required to perform verification and, finally, the verification algorithms themselves.

Chapter 8 Presents FGS, our tool, whose several parts implement our agents, environments and verification algorithms. This chapter is concerned with design principles and architectural choices, not with details on how to run the system, which is covered in Appendix C.

Chapter 9 Provides case studies to illustrate the use of the **Behaviourist Agent Architecture** and **EMMAS**. These are all executed using the FGS tool. The actual input files to FGS (and the corresponding outputs), however, are given only in Appendix B.

Chapter 10 Concludes the thesis by summarizing what has been achieved and pointing out to further developments that can be made upon what we have proposed.

A number of appendixes are also provided in the end of the thesis, which are meant to be used as references whenever further details are needed. Appropriate pointers to these appendixes are given whenever relevant.

CHAPTER 2

Related Work

Our work draws inspiration and techniques from a number of disciplines. In this chapter we present these several influences, including the current state of the art. However, their relation with our own work is not treated here – we postpone this to Chapter 3, in which a technical summary of our approach shall make it easier to establish this relation, as it depends on a number of technicalities.

The relevant works can be broadly collected in four large groups, which correspond to the organization of the present chapter. In Section 2.1 we explore the area of Autonomous Agents and Multi-Agent Systems. In Section 2.2 we present the pertinent works in the area of Formal Verification. These are the main influences upon this thesis. Whatever cannot be classified in one of these areas is dealt with in Section 2.3. Finally, in Section 2.4 we present works that combine ideas from these different domains.

2.1 Autonomous Agents and Multi-Agent Systems

The area of Autonomous Agents and Multi-Agent Systems provided the main motivation for the problems we pose in this thesis. These *multi-agent systems* (Weiss, 1999; Ferber, 1999; Wooldridge, 2009), the objects of our study, are systems composed by agents that exist and interact within an environment. This of course only makes sense in the light of appropriate accounts of both agents and environments. Thus, in what follows we examine the most pertinent notions for our purposes. Section 2.1.1 presents agents and surveys related models. Section 2.1.2, in turn, explore environments. Finally, since this thesis concerns the simulation of MASs, Section 2.1.3 addresses this topic.

2.1.1 Agent Models and Architectures

There is no precise and universally accepted definition of what an *agent* is. However, the undisputed characteristics of this concept can be summarized as follows: an agent is as an entity that exists in an environment and that interacts with it and other agents in an autonomous way.

This notion, in spite of its intuitive appeal, is all too abstract and informal, and thus cannot by itself provide the basis of a useful theory. For this reason, researchers have proposed a number of more precise models in order to define agents of particular classes. Usually, the objective of such models is to be general enough so that many agents of interest can be defined with them. Such general models are often called *agent architectures*. Wooldridge (2009) defines an agent architecture as:

> A software architecture for autonomous decision making: specifies the data structures, control flow, and, usually, the methodology to be used for constructing an agent for a particular task in a particular environment.

Let us rephrase this and call attention to other relevant points so that we may have a definition to use in this thesis. In this thesis, an agent architecture is an abstract, structured and integrated description of a class of agents. It provides the necessary elements to build particular agents of such a class.

In this section we present several such models and architectures, grouped according to the main idea or underlying theory that they employ.

2.1.1.1 Rational Agents

In Artificial Intelligence the notion of *intelligent* or *rational* agents plays a large role (Russell and Norvig, 2002). It was McCarthy (1958) who first proposed that programs could be endowed with common sense, establishing the basis for much of the future research on the topic. To McCarthy, logic could be used to describe the knowledge of an agent, and then it would be just a matter of automatically performing deduction in order to produce such intelligent actions. There are, however, two problems with this approach. The first is that theorem proving turns out to be very expensive computationally. The second problem is that not all agents one might be interested in follow such deduction principles to guide their actions. Each of these problems, in turn, motivated further research.

2.1.1.2 Practical Reasoning

While, ideally, an intelligent agent should perform perfect logical deductions, in reality no such perfect being exists. Men and animals alike have limited resources and must make decisions which are often not optimal. This insight motivated the work on *bounded rationality*, which aims at determining how to compute the best answers given a limited amount of resources. Russell and Subramanian (1995), for instance, show how optimality can be treated under such limits. Another well-known model built along these lines can be found in the *Beliefs-Desires-Intentions* (BDI) approach, originated by Bratman (1987). In this work, Bratman introduces the notion of *practical rationality* and proposes a theoretical explanation to the decision process employed by limited rational beings such as humans. In particular, he goes on to show how one can use planning in order to take decisions, and that intentions play an important role in this process. Further theoretical and computational development of this approach can be found on Cohen and Levesque (1990) and Rao and Georgeff (1995). There are a number of implementations of the BDI model, among which we can cite the PRS (Ingrand *et al.*, 1992) and dMARS (d'Inverno *et al.*, 1997). Kakas *et al.* (2008) show a model inspired by BDI, aiming mainly at solving certain difficulties involved in the implementation and formal verification of BDI theories.

Agent-oriented programming is an area closely related to these architectures. It was first proposed by Shoham (1993) as a means of describing general programs in terms of mentalistic notions. Indeed, the BDI approach offers a suitable set of such mentalistic definitions, and as a consequence the languages in this area largely adopted the BDI approach as their underlying agency model. AgentSpeak(L) (Rao, 1996) is one such language, and Jason (Bordini *et al.*, 2007) is one of its modern implementations.

2.1.1.3 Cognitive Psychology

The concept of practical reasoning does not solve the problem that agents need more than such general logical constraints in order to produce behaviour. In fact, much of what many real agents do depend on mechanisms much more *ad hoc* to their nature. For example, human memory is not just a mathematical set in which one can put knowledge. Rather, it has a detailed structure, which makes knowledge retention and retrieval a complex task. The impact of such idiosyncrasies can be seen by considering an agent that believes in some proposition, but fails to use it in its reasoning because of problems in the retrieval of the relevant memory. These discoveries, which came mainly from Cognitive Psychology (Neisser, 1967), led researchers to try to endow their agents with the same properties. This *cognitive* approach produced a number

of results. Simon (1996) is one of the pioneers on this area and provided many insights on the computational properties of the human mind. More recent work includes the SOAR architecture (Laird *et al.*, 1987), which provides a platform for the development of artificial cognitive agents, and ACT-R (Anderson *et al.*, 2004), which, in particular, employs functional Magnetic Resonance Imaging (fMRI) to validate its proposed models.

2.1.1.4 Behaviourism

The approaches we have reviewed so far have in common the fact that they place great importance on internal, *mental*, states of agents. Behaviourism provides a contrasting point of view, by focusing on their external, observable, behaviour.

To understand what is special about a behaviourist point of view, we must first examine the history of psychology. By the end of the XIXth century, psychology was still a new discipline. Many of its first proponents saw introspection and other forms of inner knowledge to be paramount to the understanding of the human mind. However, such an inner knowledge is often unreliable, for it lacks the objectivity of precise measurement. And it was because of this fundamental limitation that some psychologists started to go against these initial ideas in the search of an objective science. Thus, these *behaviourists*, as they became known, maintained that psychology should derive its theories only from the observable and measurable behaviour. A celebrated defence of these fundamental principles was given by Watson (1913).

The behaviourist tradition produced several important thinkers, from which Burrhus Frederic Skinner was, perhaps, the most notorious one. Between the decades of 1930 and 1950 he developed his own kind of behaviourism, called Behaviour Analysis. The classical exposition of this theory was given by Skinner (1953), whereas a more modern reference to the area can be found in Catania (1998). In Behaviour Analysis, an *organism* is an entity which receives *stimuli* from its environment, and produces *behavioural responses* that affect this same environment. It is assumed that these behavioural responses are a function of the stimulation history of the organism, governed by certain innate mechanisms. Therefore, the central aim of the theory is to establish how such relations work. That is to say, to discover the laws which allow one to either control behaviour by means of stimulation or predict behaviour by considering the organism's stimulation history.

Organisms are assumed to be constantly seeking pleasure and avoiding pain. That is to say, their fundamental purpose is the maximization of pleasure and the minimization of pain during their existence. This search is the basis for most of the organism's behavioural responses. And while at first it might

seem a rather simple motivation, it turns out that it can be used in order to describe a number of interesting phenomena.

A distinctive feature of the behaviourist tradition is its insistence on the irrelevancy of how organisms are really implemented. It maintains that it does not matter how the mind, the brain or any other organ works, as long as one can provide abstract laws with predictive power. In this respect, then, it differs from other psychology schools, which often describe behaviour in terms of the internal components of organisms. In behaviourism, any such reference to internal structures must be seen merely as a technical device, which could be completely substituted if an alternative offering superior predictive capabilities could be found. This does not imply that behaviourism denies the existence of internal structures responsible for behaviour (e.g., in the brain). Rather, it merely takes the point of view of an external observer to the limit by elaborating abstract concepts and laws that relate stimulation to observed behaviour.

Behaviour Analysis, in particular, offers a rich set of such abstract concepts, relations and laws. We shall examine them in detail as we formalize them in Chapter 4, but for the moment we may provide the following summary:

Stimulus (or classical) conditioning. Organisms may learn that a stimulus is followed by another. For example, a dog may be taught that a whistle is always followed by the provision of food. Hence, the dog may react to the whistle as if it was the food itself. By such associations, an organism can build a useful model of its environment.

Classes of behaviour. Behavioural responses are produced according to laws. Such laws, in turn, can be grouped in different classes of behaviour. Behaviour Analysis defines the classes of *respondent behaviour* and *operant behaviour*:

Respondent behaviour Also known as *reflexive* behaviour, this class accounts for reflexes, which are innate automatic responses to stimuli. Reflexes, then, are integral parts of an organism, and cannot be neither learned nor unlearned.

Operant behaviour Operant behaviour, on othe other hand, allows an organism to learn what actions are appropriate to achieve certain ends. An operant is a learning structure that records which action may lead to a stimulus, and how this takes place. By *reinforcing* (i.e., rewarding) or *punishing* an organism's actions, it may be taught new operants. With such operants, the organism may then choose the action that best suits its interests.

Drives. These are innate needs for certain stimuli. The organism may then be either *satiated* or *deprived* with respect to its drives. For example,

thirst is a drive which is satiated by the provision of water. If no water is provided, the organism becomes increasingly interested in water.

Emotions. These account for other temporary changes in the organism's behaviour. Each emotion has its own effect, but all of them are fully characterized by behavioural changes. This is a particularly interesting feature of Behaviour Analysis, for it contrasts with accounts of emotion which depend on internal factors (e.g., the reduction of some neurotransmitter in the brain). *Depression*, for instance, can be characterized by a generalized reduction of behavioural responses with respect to some normal level of response.

These elements interact in several ways in order to generate behaviour. For example, when choosing an appropriate action, the organism will not only use the laws of some behavioural class, but also the model that he has built of the environment using stimulus conditioning.

Notice that agents thus defined are different from what is usually called reflex agents (e.g., by Russell and Norvig, 2002) or reactive agents (e.g., by Wooldridge, 2009), whose actions are elicited by stimulation according to very direct relationships. As we have just seen, Behaviour Analysis does define reflexes as a behavioural class, but it goes far beyond them, and its value lies precisely on the richness that is achieved by the several behavioural structures that it establishes.

Despite the importance of behaviourism within psychology, computational models of agency based on Behaviour Analysis are scarce in the literature. To the best of our knowledge, the approach of Touretzky and Saksida (1997) is the most pertinent one. They propose agents called *skinnerbots*, which are endowed with learning capabilities based on classical and operant conditioning. Their model is particularly interesting because the learning that results is capable of synthesizing more complex behavioural phenomena, notably operant chaining (i.e., a sequence of learned actions in which the execution of an action sets the appropriate conditions for the execution of the next one), and has also been implemented in robots. However, the proposed model is more like a particular algorithm for calculating some aspects of classical and operant conditioning than a general framework for a behavioural agent. The main formal structure defined is a kind of rule (i.e., $A \leftarrow B\,[p]$), in which the first term (i.e., A, a stimulus consequence) is contingent upon the second one (i.e., B, a conjunction of stimuli and actions) with a probability p. The approach reduces to developing ways to learn these rules and chaining them. Despite its qualities, then, this approach is limited to a very particular aspect of agent behaviour, and it is also unclear how it could be extended or changed, since no provision is explicit made for this.

Such an extensibility is specially important because, as McDowell (2004) remarks, there is no universally accepted mathematical model that predicts quantitatively the exact way in which animals compute behavioural responses, despite the fact that some relations between the overall rate of reinforcement and the corresponding behaviour are known. McDowell (2004) then provides its own computational model for this problem and argues that it generates empirically plausible results. The main characteristic of the method proposed is that it makes no reference to an utility function being maximized by the agent. Rather, a genetic algorithm is employed to generate possible operants *a priori*, which are then emitted. It is the environment then that selects correct responses by reinforcing them, and this is used to generate other similar operants. This work is further extended by McDowell *et al.* (2006), where it is shown how this local training can be used to compose operant chains.

There exists a program called *Sniffy, the virtual rat* which aims at providing an interactive simulation of a rat for the purpose of teaching classical and operant conditioning (Alloway, 2005). However, neither the underlying computational model nor the actual source code are provided, so one cannot understand precisely how the simulation works. It seems, though, that much of it is hard-coded for very specific tasks, since, for example, possible actions and stimuli are all fixed, as are also the experiments that can be conducted. Therefore, despite being a program, *Sniffy* does not provide an actual computational account of behavioural phenomena, but merely a tool for teaching known concepts in an interactive manner (Jakubow, 2007).

Gaudiano and Phone (1997) propose a particular version of operant conditioning using neural networks intended specifically to allow robots to avoid obstacles. Naturally, though, this is too specific to constitute a general model for operant conditioning. Hutchison (2010), on the other hand, claims to have used neural networks to create a general adaptive autonomous agent that follows principles of Behaviour Analysis and is capable, in particular, of verbal behaviour. However, it is not clear exactly what has been accomplished in this work, since neither technical details nor concrete examples are provided.

Though the Behaviour Analysis perspective to agent modelling is uncommon, some specific ideas concerning learning by reinforcement, originated on this behaviourist literature, have been widely employed in Artificial Intelligence (Russell and Norvig, 2002). In particular, Q-learning theory (Watkins, 1989) seeks to abstract the notion that an action's value may change over time according to experience, similarly to the operants of Behaviour Analysis. However, Q-learning formulation assumes a particular calculation strategy when seeking the optimal action, which is not necessarily employed by agents (e.g., for efficiency reasons, or other idiosyncrasies, agents might not perform the kind of optimization postulated by Q-learning). Furthermore, it is not directed towards obtaining some particular stimulus (i.e., utility is calculated

over states, not over stimuli).

2.1.1.5 Behaviour-Based Robotics

As some of the previous examples suggested, robotics often employs learning by reinforcement techniques, and indeed one of its branches is called *behaviour-based robotics* (Matarić, 1998). This name is misleading in the context of our work. Behaviour-based robotics' emphasis is not on behaviourist psychological approaches, but on a parallel and decentralized architecture. In such an architecture, independent and parallel "behaviours" account for particular goals and tasks, with the objective of providing real-time decision making necessary for robots. It originated specially on the *subsumption architecture* of Brooks (1986, 1991). Some elements familiar to behaviourists (e.g., reflexes) can be found in such an approach, though they do not constitute its essence.

Behaviour-based robotics is a biologically inspired approach, but very general, and therefore bears no direct relationship with the particular behaviourist theories found on psychology. To make matters worse, agents of this kind are sometimes referred to as *behavioural agents* (e.g., Wooldridge, 2009). The reader should thus be careful to distinguish what concerns this method from what pertains to behaviourist psychology properly, where the adjective *behavioural* is also widely employed. In this thesis, unless noted otherwise, our use shall be of the latter kind.

2.1.2 Environments in Multi-Agent Systems

The term "environment" is not used consistently in the MAS literature (Weyns *et al.*, 2005). Sometimes, it is used to mean the conceptual entity in which the agents and other objects exist and that allows them to interact; sometimes, it is used to mean the computational infrastructure that supports the MAS (e.g., a simulator). We use the term in the former sense in this thesis. In Chapter 5 we shall give a precise formal notion of our particular kind of environment, but for the moment this intuitive notion suffices.

In this sense, then, environments are conceptually as important as the agents themselves. Despite this crucial role, the survey of Weyns *et al.* (2005) also points out that not much attention has been given to environments, which often do not receive detailed technical treatment. For instance, although Russell and Norvig (2002) present the notion of environments explicitly and analyse some of their possible properties, they do not develop any sophisticated environment model in the same depth and detail that agent models are developed. Nonetheless, there exist works that take environments and related notions as first-class entities and which are particularly relevant to this thesis.

Ferber and Müller (1996) presents a synchronous model for environment construction. In it, the environment acts as a coordinator which receives *influences* from the agents and that generates *reactions* towards them. In this manner, first agents act, and then their actions are taken into account by the environment, thus allowing simultaneous actions to be specified.

Okuyama *et al.* (2005) defines the Environment Description Language for Multi-Agent Simulation (ELMS). It allows the specification, in XML, of agent's potential actions and perceptions, as well as other resources present in the environment. Such specifications may carry certain logical preconditions which must be satisfied, thus constraining their execution. The language also supports the definition of the structure of the environment as a grid, which can be used in calculating preconditions or assigning effects (e.g., an action's effect might be to change the agent's position from one grid cell to another). The simulation itself is performed by combining an environment specification with agent implementations.

Part of the purpose of an environment is to allow agents to interact. A way to deal with such interaction is through protocols, which define how messages must be exchanged between agents. A number of initiatives exists in this sense. The COOrdination Language (COOL) (Barbuceanu and Fox, 1995) is an early example of such an approach. In COOL, a protocol is a conversation represented by a finite state machine (FSM), in which transitions represent message exchanges based on speech act theory. Each agent must instantiate such an FSM, which regulates the agent's individual state in the conversation. To choose a transition, agents must comply with certain rules, which are part of the coordination protocol.

A more recent and well-known approach to describing agent interaction is the AUML sequence diagram (FIPA, 2003), an extension of UML sequence diagram for MASs. However, like UML itself, this is largely an abstract graphical notation, and lacks both a formal and programming model. The IOM/T language (Doi *et al.*, 2005) is a Java-like language designed to allow the actual programming of such AUML interactions. Quenum *et al.* (2006) proposes a similar framework, but designed to emphasize the separation between agents and their role in protocols, thereby making the protocols generic. This requires the addition of features, such as the differentiation between agent actions and messages, to those provided by AUML and variants. Moreover, contrary to imperative approaches such as IOM/T, Quenum *et al.* (2006) argue in favour of a declarative specification language.

The notion of an *organization* (Ferber, 1999) can be related with environments. An organization, in this sense, is divided in two parts: the abstract organizational structure and the concrete organizations. The organizational structure defines the roles that agents might occupy, independently of the

actual agents that will eventually fulfil the roles. Roles define powers and responsibilities, and are related to other roles too. A concrete organization, in turn, is an actual MAS that fulfils the constraints imposed by an organizational structure. The relation to environments can be established at this concrete level, since the MAS environment, if represented explicitly, may as well be subject to organizational constraints. If the environment is not represented explicitly, the mechanism that allow the agents to interact according to the organizational structure can be seen as a kind of environment, although possibly quite an abstract one. MOISE (Hannoun *et al.*, 2000) is an example of such an organization model.

Finally, it is worth to mention the work done in reasoning involving dependency networks (Sichman *et al.*, 1998). By modelling the capabilities of other agents and the dependencies among them, an agent can build a network of dependencies that comprises the whole MAS, thereby creating a model of its environment in so far as such dependencies are concerned. While this is not an environment in itself, this kind of social reasoning can be used to operate in an environment rationally. Social networks such as these in fact present a general way of modelling and reasoning about societies, which has also been studied in the sociology area of Social Network Analysis (Wasserman *et al.*, 1994).

2.1.3 Multi-Agent Based Simulation

In the context of a scientific inquiry, a *model* is an abstract representation of a *target system*, the entity one wishes to study (Frigg and Hartmann, 2009). Accordingly, in this thesis we are concerned with *simulation models*. A *simulation*, in turn, is the execution of such a model by a *simulator*, which produces a sequence of simulator states.

Multi-Agent Systems can be used to develop models of interesting situations in order to analyse their properties. Simulation is often used to perform such an analysis, and a number of tools are available to this end (Gilbert and Bankers, 2002). These tools usually provide both a programming framework in which to define agents and a simulation tool to actually perform simulations. This general architecture was introduced by the SWARM platform (Minar *et al.*, 1996). More recent examples can be found on the RePast (North *et al.*, 2006) and MASON (Luke *et al.*, 2004) platforms. Tobias and Hofmann (2004) survey a number of such platforms and compare them.

Simulation platforms often require the user to program. This, of course, prevents many potential users from employing them. To mitigate this problem, some tools, such as NetLogo (Wilensky, 1999) and SeSAm (Klugl and Puppe, 1998), have easy of use as an explicit goal

Multi-Agent models have been used, in particular, to simulate social phenomena. Notably, Epistein and Axtell (1996) explored phenomena such as trade, war and disease transmission using the Sugarscape platform they developed. Their objective was to show that these phenomena can be explained by finding the right simulation rules that generate them.

Many similar studies through simulation have been devised, addressing many different matters, such as: natural resources management (Briot *et al.*, 2010), epidemiology (Alam *et al.*, 2009; Bearman *et al.*, 2004; Eubank *et al.*, 2004; Mysore V. *et al.*, 2005), terrorism (Tsvetovat and Latek, 2009), climate (Downing *et al.*, 2001; Balbi *et al.*, 2010), crowd behaviour (Henein and White, 2005; Bansal *et al.*, 2008), opinion formation (Stocker *et al.*, 2001), archaeology (Doran and Palmer, 1995; Dean *et al.*, 2000), economics (McCarthy *et al.*, 2008), crime (Bosse and Gerritsen, 2008), stem cells (d'Inverno and Saunders, 2005) and computer networks (Bhargavan *et al.*, 2002).

2.2 Formal Verification

The idea of analysing the properties of MASs automatically came mostly from the broad field of Formal Verification, to which we now turn our attention. Section 2.2.1 concerns Model Checking, from which we took the idea of systematically exploring state-spaces. Section 2.2.2 addresses Model-Based Testing, an area that combines formal specifications with actual program execution. Section 2.2.3 presents Runtime Verification, an approach that verifies executions of programs, not their specifications. Finally, Section 2.2.4 presents process algebras, among which we found semantic models useful for verification.

2.2.1 Model Checking

In modal logics, a *model* $\mathscr{M}$ for a formula ϕ is a graph with labelled states that provides the semantics of ϕ. Model Checking is a verification method first proposed by Clarke and Emerson (1981) and Queille and Sifakis (1982) in which the properties of interest are evaluated directly on the model for a system, instead of the specification's syntax. That is to say, instead of trying to produce a proof, one merely scans a model searching for violations of the desired formula ϕ. We denote that some state s of the model $\mathscr{M}$ satisfies ϕ by writing:

$$\mathscr{M}, s \models \phi$$

In Model Checking, $\mathscr{M}$ typically represents some computational system of interest (e.g., a set of computers that communicate through some protocol), and

ϕ some property concerning such a system (e.g., that no deadlock occurs). $\mathscr{M}$ can be obtained automatically from higher-level descriptions that specify the behaviour of the system (i.e., from a program), and ϕ can be given explicitly in terms of temporal logics (a kind of modal logic) such as the Computation Tree Logic (CTL) or the Linear Temporal Logic (LTL). For instance, in LTL one can formalize the assertion

"It will always be the case that when process p requests resource r, it will eventually receive it."

by writing something like

$$G(p_requests_r \Rightarrow F(p_receives_r))$$

where G ("Globally") and F ("in the Future") are temporal modalities.

Model Checking has seen considerable progress since its inception. Symbolic Model Checking (Burch *et al.*, 1990) has made the treatment of large state-spaces possible by using special data structures (i.e., Binary Decision Diagrams) to encode them succinctly. More recently, Bounded Model Checking (Biere *et al.*, 1999, 2003; Clarke *et al.*, 2001) has profited from the developments in SAT solvers. This is done by limiting the length of the counterexamples one is searching for, which allows an efficient translation of the resulting problem to an instance of SAT.

Clarke *et al.* (1999) and Baier and Katoen (2008) provide long and self-contained texts covering much of the developments in Model Checking. Finally, Clarke (2008) gives a historical account of this development.

2.2.2 Model-Based Testing

Software Testing is a form of verification in which a *system under test* (SUT) is systematically executed according to *test cases* in order to identify *defects*. Model-Based Testing (MBT) (e.g., Gaudel, 1995; Brinksma and Tretmans, 2001), in turn, is a formal approach to testing which employs mathematical models of the SUT to generate test cases. This brings two main advantages:

- test cases can be generated automatically from the model;
- test cases can be chosen in such a way that some coverage guarantee can be given. For instance, if the system is modelled as a control-flow graph, one can aim at covering all possible execution paths of such a graph.

A well-known example of such an approach was proposed by Tretmans (2008). There, the SUT is specified as a labelled transition system, which can be used to generate test cases. By this method, it is possible to systematically test whether the SUT conforms to its specification with respect to the so-called *ioco* relation.

In order to produce test cases directed for some particular end, one can employ *test purposes*. Such test purposes can, in particular, be represented formally. This approach is used on the TGV tool (Jard and Jéron, 2005), where both test purposes and SUTs are modelled as input-output transition systems (IOTSs). By performing a special synchronized product between them, along with other transformations, one gets another smaller automaton from which test cases can be extracted in order to assess whether the SUT is *ioco*-conformant to the specification. By this method, only relevant tests are executed. TGV itself is based on a more general approach to on-the-fly verification, which can also be used to perform Model Checking and to determine bisimulations (Fernandez *et al.*, 1992).

Often, the state-space relevant for testing is very large. For this reason, techniques to partially explore the state-space have been devised. In particular, the use of statistical methods allows the performance of *random testing*. As the name implies, random testing is concerned with generating test cases in a random way. Randomness can be introduced in a variety of manners, such as providing random inputs, or performing a random walk on a control graph. The latter approach is of particular interest because it allows the exploration of the state-space according to desired statistical criteria. Moreover, formal approaches often assume the existence of a graph model of the SUT, which make such random walks a natural choice for their testing.

There are a number of ways in which one can perform a random walk. First, one can proceed using an uninformed random walk, which simply chooses randomly between the successors states of the current execution state. This method, however, produces biased coverages of the graph. To correct this, Denise *et al.* (2004) proposed a method in which each possible execution trace has the same probability of been chosen (i.e., a uniform distribution over traces). Later improvements of this method allow the uniform selection of traces in a concurrent system, in which each program has its own, smaller, control graph (Denise *et al.*, 2008). This allows the uniform analysis of much larger state spaces.

At last, we consider the area of *passive testing* (Lee *et al.*, 1997). In passive testing, checks are performed *a posteriori*. That is to say, the SUT is not exercised by test cases; rather, checks are performed on the execution traces of the SUT's normal behaviour, and thus have a passive role. Usually, an automaton represents the property to be tested. It is then just a matter of

performing language recognition on the traces. This approach, though from a different community, resembles Runtime Verification in that one has not much (or none at all) control over the concerned system, and checks are performed merely by observing its normal behaviour. Passive testing, however, make such observations after the system has been executed (by examining the logged traces), whereas Runtime Verification, which we shall now turn our attention to, focus on *runtime* observations.

2.2.3 Runtime Verification

Formal Verification techniques are designed to be applied to specifications of systems so that it can be guaranteed that it conforms to some property of interest. This, however, is often unfeasible, owing to the large models that need to be analysed. Runtime Verification (RV) is an alternative to such usual methods. In RV, instead of proving that the specification of a system conforms to a property, one merely checks whether the execution of the system is conformant. Thus, while it cannot guarantee that the system is conformant, it can at least provide a way to detect and respond to deviations from the desired behaviour.

Typically, this is achieved using a *monitor*, which is an extra component that is added to the system in order to perform the verification. The precise nature of such monitors vary according to the kind of property to be analysed. It turns out that linear-time properties are more suitable such an architecture, and thus most approaches employ some variation of a linear-time logics, such as Linear Temporal Logic (LTL).

However, the traditional semantics for LTL assumes an infinite execution trace. Thus, it is unable to cope with cases in which only finite traces are available. For example, a liveness property such as GFp (i.e., "in the future, p will always happen again") cannot be verified because one cannot know whether p will happen again after a trace terminates. To solve this problem, one may redefine the semantics of LTL to account for the case in which traces are finite entities. This approach is followed by Finkbeiner and Sipma (2004), where an upper-bound (i.e., the trace length) is introduced in the semantics and it is shown how to build monitors for it. A similar approach is taken by Geilen (2001), where the technique presented is capable of ensuring certain kinds of LTL properties.

Another way of solving the problem of finite traces is to modify LTL more extensively. This is achieved, for instance, by Bauer *et al.* (2007), where the Runtime Verification Linear-Temporal Logic (RV-LTL) – a four-valued version of LTL – is defined, alongside an appropriate monitor. This work is itself built upon two other modifications of LTL for finite traces, namely, FLTL

(Lichtenstein *et al.*, 1985) and LTL_3 (Bauer *et al.*, 2006).

An even more sophisticated approach can be found on Eagle (Barringer *et al.*, 2004a; Goldberg and Havelund, 2005), which is a general logic framework which can be specialized to a number of particular logics. It is designed to allow the creation of monitors and is implemented as a Java library. An LTL specialization for Eagle is given by Barringer *et al.* (2004b).

The RV community also emphasizes practical implementations, and therefore a number of software architectures have been designed in order to support the verification techniques. Monitoring-Oriented Programming (MOP) (Chen and Roşu, 2007), for instance, is an effort to build a whole paradigm using monitoring ideas. Using MOP, one can employ a number of different formalisms to express properties, and the generated monitors can be written in a number of different programming languages. Java-MOP (Chen and Roşu, 2005) is the version for the Java platform. The MaC architecture (Kim *et al.*, 2001) follows a similar platform-independence principle, but is tied to its own particular specification language. Java-MaC is the version for Java of this general architecture.

2.2.4 Process Algebras

Concurrent systems are notably difficult to design correctly. This has led to the development of formal approaches to their specification and verification, among which process algebras have been particularly fruitful. In such a formalism, *processes* are algebraic expressions that model, in an abstract manner, the communication capabilities of individual systems. By putting such processes in parallel, it is possible to assess their combined behaviour, which is the main source of complications arising in concurrent systems. Examples of process algebras include ACP (Bergstra and Klop, 1984), CSP (Hoare, 1985), CCS and π-calculus (Milner, 1999) (see Appendix E for an overview), and Ambient Calculus (Cardelli and Gordon, 1998).

Besides serving as formalisms for specifications, some of these process algebras have been actually implemented in some form, so that specifications can actually be executed as programs. Pict (Pierce and Turner, 1997) is an example of programming language based on the π-calculus. Peschanski and Hym (2006) develops the cube-calculus, based on the π-calculus, which is actually a language designed to run in an interpreter called the CubeVM. JCSP (Lea, 1999) provides a Java framework for the implementation of CSP, so that it is possible to create Java programs whose communication structures follow a CSP specification. Applications of JCSP include the works of Oliveira (2005) and Freitas and Cavalcanti (2006).

2.2.5 Formal Development

In a formal development approach, one starts with a formal specification of the system to be created and, through some technique that guarantees correctness, transforms this specification in either another specification or actual software. Typically, the formal specification defines the high-level requirements of the system, without considering implementation details, and is thus more focused on the abstract properties of the problem to be solved. Another advantage of proceeding in this manner is that one may, at each formal specification level, verify whether certain properties hold or not (e.g., by means of logical proofs or Model Checking), which may be more difficult or even impossible in an implementation.

The Z Notation (ISO/IEC, 2002; Woodcock and Davies, 1996; Jacky, 1996) is a well-known formalism for writing such specifications, based on set theory and first-order logic (see Appendix D for an overview of Z). Z is designed for the specification of systems composed of states and transitions among states, although stateless definitions are also possible. A calculus allows the composition of more complex specifications out of simpler ones. Circus (Woodcock and Cavalcanti, 2001) is a related method, which integrates Z with CSP in order to allow the communication aspects of the system to be more properly specified. Given such a specification, the question of how to transform it in correct software arises. A solution is to proceed with formal refinements, which are transformations that allow one to go from abstract specifications to less abstract ones, until an implementation is reached. Sampaio *et al.* (2002) provides an example of such a refinement technique for Circus. The B Method (Abrial, 1996) is another approach, based largely on Z notions, but with a focus on facilitating refinement to executable code.

2.3 Other Influences

2.3.1 Non-Agent Based Simulation Methods

Schruben (2010) points out the simulation modelling and analysis are often seen as two entirely different activities, and argues that it would be more productive to design models considering how they are supposed to be analysed.

The Discrete Event System Specification (DEVS) (Zeigler *et al.*, 2000) family of simulation formalisms provides conceptual frameworks to put simulation under rigorous definitions. In particular, DEVS defines the notion of *experimental frame* as an entity which provides inputs to a simulation model and judges its outputs. For the sake of uniformity, experimental frames can be expressed with the same formalism used to specify the simulation model itself.

Experiments run in this fashion, though, have no control over the simulation once it is started, and can only evaluate its final result. This is sufficient to devise certain optimization techniques, by which several input parameters are tested in order to find the ones that generate the best output according to some optimization criteria (Halim and Seck, 2011).

Even though a DEVS model is meant for simulation, it can sometimes be subject to formal verification through model checking, provided that the model can be reduced to a particular subset of DEVS such as FD-DEVS (Hwang, 2005). Model-based testing can also be applied (Li *et al.*, 2011).

2.3.2 Software Components

Software components are used in the implementation of our tool, so let us examine what they are. The fundamental ideas concerning components were given by McIlroy (1968), in which it was envisioned that software should be built using reusable parts, much like electronics are built using reusable integrated circuits. To this end, the task of developing software would have to be divided into two branches. One that would take care of building components useful in many different situations, and another that would develop the final software using these reusable components. This way, developers would save time by not having to rewrite software parts.

These ideas have developed through the years, and today we have a Component-Based Software Engineering field. Following the contemporary treatment of the subject found by Szyperski (1999), a software component is characterized as follows:

- *It is an independent unit of deployment.* That is, it can be packaged and transmitted independently of anything else;
- *It is a unity of third-party composition.* Components are designed to be reused in unknown applications, built by different people;
- *It has no externally observable state.* This is just a technical detail to make sure that the same components will always perform the same functions;
- *It has contractually specified interfaces and explicit context dependencies only.* In other words, one can know what the component requires from and provides to an application;
- *It targets a particular component platform.* Components frequently assume the existence of a platform that provides useful services.

To be used, software components must provide *component instances* (i.e., objects that do have an observable state[1] and are, thus, useful in particular applications) and such instances must be *composed*.

2.4 Formal Approaches to Multi-Agent Systems

Benerecetti *et al.* (1998) remarks that, in 1998, there were few approaches to the formal verification of MASs. Since then, however, there has been an increasing interest in applying formal methods to MASs, including formal verification techniques. In this section we present the approaches which are most significant for our own attempt in bringing these areas together.

2.4.1 Formal Specification of Agent Architectures

Some researchers are particularly interested in establishing precise basis in which to define agents. The SMART framework (d'Inverno and Luck, 2003), for instance, employs the Z Notation in order to formalize a general theory of agency. Its aim is to allow any other agency theory to be specified in its terms, provided that a few minimal obligations are met. One such extension can be found in da Silva (2005), where a theory of business management is formalized as a multi-agent system.

Another example is the dMARS system we saw in Section 2.1.1.2, which was formalized by d'Inverno *et al.* (1997).

2.4.2 Formal Specification of Environments

Since process algebras (see Section 2.2.4) are designed to model and verify communications in concurrent systems, it would be natural to employ them in the specification of MAS environments. Yet, this is seldom done in the context of MAS simulation. One exception is the work of Wang and Wysk (2008), which uses a modified π-calculus to express a certain class of agents and their environments. Another example is the IOM/T language (Doi *et al.*, 2005), which is used to specify interaction protocols, and whose semantics can be given using the π-calculus. IOM/T is actually designed to be a textual representation of AUML sequence diagrams (FIPA, 2003), and the π-calculus semantics is used to formally demonstrate their equivalence.

As we saw in Section 2.1.2 above, Ferber and Müller (1996) develop a model to the specification of environments of multi-agent systems. This model can

[1] Even if, owing to information hiding, only partly or indirectly observable.

be formalized to some extent by the Block-like Representation of Interactive Components (BRIC) developed by Ferber (1999). A BRIC specification is defined by *blocks* which possess their own behaviour (specified as Petri nets) and can be connected to each other. In this way, agents and their environment, as well as the mechanisms of synchronization and message passing that relate them, can be specified as individual but interconnected blocks.

Although not actually part of the computer science community, it is worth to note that in the sociology area of Social Network Analysis (Wasserman *et al.*, 1994), social networks are precisely defined (as graphs), along with properties of interest (such as the *centrality* of an individual in a network). All this is provided using formal definitions, and indeed software that, given a social network (e.g., as an adjacency matrix), can calculate these properties. Examples of such software include Pajek (Batagelj and Mrvar, 1998) and UCINet (Borgatti *et al.*, 2002).

2.4.3 Formal Verification and MAS Simulation

Most approaches to MAS simulation do not employ any form of automated formal analysis. There are, however, a few notable exceptions. Let us review them.

Bosse *et al.* (2009) presents the Temporal Trace Language (TTL), which has an associated tool, designed to define simulation models (in a sublanguage called LEADSTO), as well as linear-time properties about such models. The approach is to execute the simulation model and check whether the resulting traces obey the specified linear-time properties. An example of this method is given by Bosse and Gerritsen (2008), where criminal behaviour is modelled, simulated and analysed. A similar method was given by Mysore V. *et al.* (2005), who developed a multi-agent model of food poisoning using the RePast (North *et al.*, 2006) simulation platform and analysed it by checking the resulting simulation traces with respect to LTL formulas.

Despite the clear possibility, Runtime Verification (see Section 2.2.3) is not usually applied to simulations. An exception is the network simulator Verisim (Bhargavan *et al.*, 2002). This tool runs the simulation normally, but checks linear-time properties as it proceeds using runtime *monitors*. Indeed, the MaC architecture (Kim *et al.*, 2001) is employed to implement such monitors.

2.4.4 Model Checking of MAS

While MAS simulation is usually treated in an informal manner, there are a number of approaches to formally specify and verify MASs (not necessarily meant for simulation). These, in essence, are merely the application of usual

model checking techniques to particular kinds of formal specifications (i.e., specifications of MAS), as we shall see below. Furthermore,
van der Hoek and Wooldridge (2003) note that there is a difficulty in relating an agent's *program* to its formal *specification*. Indeed, though the pioneer work of Rao and Georgeff (1993) shows how to model check a BDI-based modal logic specification, the problem of how to implement such a specification remains. This is an important gap, since the ultimate objective is to understand the properties of an actual agent, which must exist as an implementation too.

Benerecetti *et al.* (1998) attempted to solve this problem by demanding that one codifies agents in an extension of the input language PROMELA of the SPIN model checker (Holzmann, 2003). In a more high-level manner, this issue has also been addressed by devising special purpose programming languages, which are then translated to the input of a model checker. For example, MABLE (Wooldridge *et al.*, 2006) is a programming language which, in addition to usual imperative constructs, adds the possibility of specifying *mental states* in accordance to the BDI theory we saw previously (e.g., by specifying an agent's *beliefs*). The verification of a MABLE program is achieved by translating it in PROMELA and using the SPIN model checker. Hence, the approach reduces to devising a translation scheme to the input accepted by a traditional model checker.

Similarly, Bordini *et al.* (2003) have shown that AgentSpeak(F), a (finite state) subset of the AgentSpeak(L) language to specify BDI agents, is reducible to PROMELA. Moreover, Bordini *et al.* (2004) have shown how to reduce these same agents to Java, in which case verification can be done using the JPF2 model checker (Visser *et al.*, 2003).

MCMAS (Lomuscio *et al.*, 2009) follows a similar approach, but instead of reducing an MAS program to another formalism, it provides a model checker that operates directly on the program provided. This is done by incrementing existing BDD-based algorithms with procedures for the new epistemic modalities introduced by the approach (e.g., *knowledge*). This approach has been particularly relevant to the analysis of communication protocols among agents, whose properties can often be expressed more elegantly using the provided epistemic modalities.

CHAPTER 3

Contribution of this Thesis

In previous chapters we have seen the motivation and general aspects of our approach, as well as the works related to it. Let us now turn to its technical characteristics and show how it compares with the state of the art. In what follows we present an overview of the main elements introduced by this thesis and explain how they relate to existing methods, thereby providing both a detailed account of our scientific contribution and a summary of its technical content.

This presentation is done by, in each section: (i) summarizing the contents of a particular chapter, which is specified in the beginning of the section; and then (ii) comparing our contribution with existing approaches. Chapters that are not mentioned are of course original as well, but they play a support role with respect to the ones which are dealt with here (e.g., Chapter 9, which provides examples of uses of our approach, and thus supports the theory). Notice that in the present chapter the technical terms are **emphasized** but used informally. Their precise definitions are left for their respective chapters. This is done to avoid introducing unnecessary complications and minutiae at this point. The interested reader may refer directly to the definition in the appropriate chapter.

3.1 Agent Architecture

The complete contribution is presented in Chapter 4.

It is clear that owing to its focus on the organism as a whole (i.e., not on isolated details of particular internal structures), as seen in Section 2.1.1.4, Behaviour Analysis incidentally provides a useful basis for a computational agent architecture. That is to say, a framework with which to define agents capable of receiving stimuli and performing actions in a rather general manner.

In this thesis we introduce a new agent architecture based on the core elements of Behaviour Analysis[1], which we call the **Behaviourist Agent Architecture**[2]. Besides the overall perspective of agent behaviour that it is capable of providing, this behaviourist theory is valuable to us also for the following reasons: (i) it places great importance on defining and analysing behaviour from an external point of view, which in the light of the methodology suggested in Section 1.1 is clearly important; (ii) it is based on an empirical science, and therefore is capable of modelling many realistic animal phenomena, such as learning; and (iii) the underlying psychological theory is sufficiently well defined in order to allow the possibility of a formalization, which is necessary for a computational implementation. Moreover, its practical usefulness can arise in the following situations:

- if the agents are to be studied and manipulated using similar techniques to those allowed by Behaviour Analysis;
- if the agents are actually models of real organisms. In such a case, the agents can be simulated in order to infer results about the organisms they model;
- if one believes that copying these natural mechanisms provides more efficient ways to solve problems.

The architecture gives a computational account of the Behaviour Analytic elements we presented in Section 2.1.1.4, namely: (i) stimulus conditioning; (ii) respondent behaviour (i.e., reflexes); (iii) operant behaviour; (iv) drives; and (v) emotions. Agent behaviour arises from the interaction of these several parts among themselves as well as with the surrounding environment.

While a direct implementation of such an architecture would in fact be a formalization (i.e., because the program's text is written in a formal language), we chose to write an *abstract* formalization first to serve as the specification of the implementation. In this way we were able to separate the theory itself from

[1] Part of which we published in (da Silva and de Melo, 2007).

[2] We realize that the word "behaviour" and its variants are quite broad and have many intuitive meanings. Nevertheless, we have chosen to keep them as technical terms here in order to remain faithful to the naming conventions usually employed in the behaviourist literature we draw from. Thus, all of our references to the "behaviour" of agents should be seen from this perspective, unless explicitly noted otherwise.

implementation details. This formal specification is written in the Z Notation (ISO/IEC, 2002; Woodcock and Davies, 1996; Jacky, 1996) and constitutes the subject of Chapter 4. Its implementation, which we use in the simulations, is done in Java in a rather straightforward manner. More details about the implementation are given in Chapter 8, and a reference of the required input format is provided in Appendix C.

The agent architecture defines an agent as an *organism* which is capable of receiving *stimuli* and providing *behavioural responses*. These responses can constitute instances either of *reflexive behaviour*, or *operant behaviour*. The former accounts for actions which are directly elicited by stimulation, whereas the latter accounts for actions that are emitted autonomously because of previous learning. Operant behaviour is formed and maintained by *reinforcing* actions through pleasant stimuli.

The purpose of an organism is to find pleasant stimuli and avoid unpleasant ones. This task is complicated by the fact that the *utility* of stimuli can change over time, as the organism may learn new relations among them. For example, a neutral stimulus may become pleasant if the organisms finds out that the first is always followed by the latter in its environment.

Drives and *emotions* can regulate behaviour by modifying either the behaviour emission itself or stimulus utility. Drives formalize the notion that some stimuli becomes increasingly important to organism while it is deprived of them (e.g., thirst is a drive in animals). Emotions specify other behavioural modifications suitable in particular circumstances (e.g., a *frustrated* organism becomes more likely to emit actions carelessly).

As an example of what this architecture can represent, let us consider a behaviour analytic description of a typical laboratory experiment that one could perform on, say, a pigeon. The pigeon is put on a cage, where both a button and a light bulb are present. Before giving food to the pigeon, and only then, the experimenter tuns the light on. After some time, the pigeon learns that light is followed by food. So every time the light is on, the pigeon acts as if the food has arrived. This is an example of *classical conditioning*. Moreover, the pigeon initially does only random actions, because it does not know how its environment works. But eventually it discovers that by pushing a button, the light is turned on. This is an example of *operant conditioning*. By combining these two conditionings, the pigeon then becomes likely to *emit* the behaviour of pushing the button when it wants to eat. Its hunger, in turn, is given by a *drive*, which changes the utility of stimuli according to how much the pigeon has already eaten. Finally, the experimenter might decide that no food shall be given in association with the light. In this case, the pigeon will be gradually unconditioned, and the behaviour of pushing the button will be *extinct*. This causes *frustration* on the pigeon.

The precise meaning of the emphasized terms will be given in Chapter 4, but for the moment it suffices to note that they provide a relevant vocabulary to describe the experiment. Remarkably, it is a vocabulary whose expressions are ultimately defined in terms of externally observable behavioural responses and stimulation. Take hunger, for instance. It is a drive, which means that it is defined by the existence of two operations, namely, *satiation* and *deprivation*, with respect to a particular kind of stimulus, in this case food. By depriving the organism of food, it becomes more likely to emit behaviours that lead to food. By satiating it, the contrary happens. This is the definition found on the psychology theory. To make it computational, we add the notion of *stimulus utility*, which provides a minimal explanation for the phenomenon: drives affect the utility of stimuli, which in turn affect behaviour emission.

Since behaviourist principles were largely developed using animals, examples like the one we gave above abound on the related literature.[3] Nevertheless, the underlying principles that arise from such experiments are applicable to humans as well. Consider, for example, any interactive website. In this case, one may well be interested in how often, and under which conditions, some users will perform some actions. For instance, how often they click on certain links, use certain features, or which kind of advertisement is more effective. Because of the abstract nature of such a model, it can be put in behaviourist terms much like the experiments with, say, pigeons. To the extent that the user is interacting in this well-defined and abstract space, he can be seen just as the pigeon in its experimental chamber. We shall see other examples involving people in Chapter 9.

3.1.1 Comparison with Other Approaches

We avoid introducing constructs which we do not find necessary for the computational formalization of the original definitions of Behaviour Analysis, thus upholding its values as much as possible. In particular, though agents thus defined have state, which is necessary for computation, we do not ascribe usual mental qualities to them, such as will, belief, intention, knowledge, memory, and reasoning. This is so because Behaviour Analysis rejects these usual explanations of behaviour, and puts in their place a different set of concepts, focused on the properties of externally observable events – that is to say, behavioural responses and stimulation.

In this way, we differentiate our architecture from a number of others. Many of the fundamental ideas of Artificial Intelligence are related to the view that human intellect can be understood as an information processing device, much like a computer (e.g., Simon, 1996). This view finds considerable support

[3] For a curious one, see Skinner (1948), where it is shown experimentally that some superstitions can be explained in behaviourist terms.

on approaches to psychology that seek to identify and analyse these internal information processing mechanisms. This excludes Behaviour Analysis, but includes both folk psychology (e.g., as pointed out by Bratman, 1987), as well as academic branches such as Cognitive Psychology (Neisser, 1967). It is only natural, then, that many well-known agent architectures, such as the examples we cited in Section 2.1.1, should be based on similar principles, that for convenience we shall call *mentalistic*. These are appropriate in many cases, but behaviour of humans and other animals alike is often determined by mechanisms which cannot be effectively presented using such mentalistic descriptions alone.

Though not incompatible, mentalistic and behaviourist theories are very different. In particular, cognitive approaches treat agents as information processing units endowed with certain mental mechanisms, such as memory and planning capabilities, and try to analyse the properties of such mechanisms experimentally (e.g., the capacity of working memory as shown by Miller, 1956). Behaviourism, in contrast, seeks mathematical relations between stimulation and behavioural responses without assuming any intermediary mental mechanism (e.g., the rate of behavioural responses in relation to the rate of reinforcement as studied by Herrnstein, 1970). Hence, the two perspectives can be seen as complementary, but with distinct focuses and theories.

To see this more clearly, we may return to the example of the pigeon in a cage given previously. How could one use mentalistic notions to describe the experiment with the pigeon? Let us examine some possibilities. If knowledge is represented explicitly and the agent relies only on reasoning, it would be necessary to add knowledge concerning the several phenomena involved, which would state that it is right to make such conditioning, and specify how they are to be defined, maintained and eventually dissolved. Knowledge for the regulation of hunger over time, and of the emotional effects of frustration, would also have to be provided. But then one would be using mental terms to describe not the knowledge of the agent, but unconscious mechanisms that regulate its behaviour. And in this case it would be better to describe such mechanisms directly, instead of relying on an agent's rational machinery to deduce their consequences. Conversely, if instead of explicit knowledge one employed intricate internal cognitive or neural mechanisms, it would possibly add more complexity than it is required for the description of the observed phenomenon. For instance, if one knows a mathematical formula capable of describing how classical conditioning takes place, there is no need to provide a detailed neural explanation in so far as the simulation of the behaviour is concerned. In summary, it seems that certain classes of behaviour are better understood without reference to mental entities, but merely environmental ones.

It is clear that the modelling of organisms can be done at many points in a

continuum of abstraction: from the use of disembodied reason (very abstract) down to the mimicking the physical properties of actual organisms and their brains (very concrete). Owing to its emphasis in the interface between organisms and their environments, a behaviour analytic approach provides an interesting alternative, a middle ground between these two extremes. It is not, of course, the only possible alternative, but it is one with an established, distinct and coherent psychological theory, which was created with certain particular problems in mind, and therefore merits attention. Nonetheless, it must be be emphasized that our objective is not to dismiss or substitute any of the existing agency models, mentalistic or otherwise. Rather, our approach complements existing ideas and allows the study of agent behaviour from a different point of view, suitable for different purposes.

Computational models for Behaviour Analysis are scarce in the literature, and none of the existing ones gives a unified account of its main elements, which we group as follows: (i) stimulus conditioning; (ii) respondent behaviour (i.e., reflexes); (iii) operant behaviour; (iv) drives; and (v) emotions. The approaches that do exist, such as the work of Touretzky and Saksida (1997), focus mostly on algorithmic aspects of operant conditioning, a form of learning by reinforcement. Important as this may be, it is not sufficient as an architectural basis, which requires a more extensive and structural specification of what constitutes an agent. It must be extensive because there is great dependency among the several behavioural phenomena, and to represent one it is often necessary to represent another. In particular, operant conditioning itself depends on other aspects of the agent, such as drives and emotions. It must also be structural because it serves as a fundamental basis for both implementation and further theoretical development. Therefore, its elements must be organized in such a way that they can be easily identified, analysed, related, changed and extended – that is to say, highly structured.

Concerning the work of Touretzky and Saksida (1997) specifically, despite its merits, it is limited in ways our approach is not. No account is given of drives and emotional behaviour, nor is stimulus utility represented explicitly. Reflexes are also not represented. Moreover, the treatment of operant behaviour itself, though interesting, is limited in some important ways. For instance, it assumes that reinforcement must be provided in the instant following the action to be reinforced, whereas it could be a longer delay. It is also unclear how the model could be extended or changed, since no other structures are presented. Our approach, in contrast, provides a general framework for agent modelling which defines several aspects of agent behaviour, shows how they relate to each other and can also be extended and changed modularly. This implies, in particular, that different calculations for operant conditioning can be specified within our architecture.

Extensibility is an important design goal for us. While we provide abstract

definitions and corresponding refinements, it should be possible to devise alternative refinements. This is possible by the use of the Z Notion, and can be useful in order to specialize or change the architecture in a systematic manner. For example, if one wishes to consider a different way of learning in operant behaviour, it is possible to define it and integrate in the architecture. This flexibility contrasts, for instance, with Q-learning theory (Watkins, 1989), in which a certain learning mechanism is defined and is not supposed to be changed. Our approach to operant modelling, on the other hand, aims at being flexible enough so that a wide range of utility calculation strategies can be defined, and it is designed to model the usual operant contingency as a triple of antecedent stimuli, action and consequent stimulus. Moreover, our approach is closely linked to other agent's aspects, such as its stimuli processing facilities and its emotional state.

It is also worth to note that McCarthy (2008) has argued that some innate notions and mechanisms can be useful in complementing an agent's learning process. Our approach can be seen under this light, since we provide a number of supporting mechanisms that the agent uses to guide and structure its learning. However, instead of relying on theorem proving, as McCarthy suggests, we define specialized structures for the several mechanisms. We thus avoid the efficiency complications that might arise from automatic theorem proving.

A number of other specific comparisons can be made. The following points are particularly relevant:

- The conditioning mechanisms we define, namely, stimulus and operant conditioning, provide a learning framework suitable for understanding and acting upon an environment. By structuring the relations between the observed environment elements, these mechanisms allow the agent to reuse these experiences in future situations in order to either manipulate or react properly to the environment. In contrast, rationality-inspired architectures, such as BDI (Bratman, 1987; Rao and Georgeff, 1995), do not define the learning mechanisms involved, but merely assume that learning is performed and its results transformed into beliefs. In other words, we assume *a priori* features of the agent's environment (e.g., events that happened together may happen together again in the future), and profit from them by defining associate learning mechanisms. But this also reduces the generality of our approach. In particular, it is not clear if we could develop a useful agent-oriented programming (Shoham, 1993) language based on our architecture, although this might be attempted. To this end, one would have to isolate tasks that a programmer may wish to accomplish and that find a good representation using behaviourist notions. Currently, though, our architecture seems more appropriate for modelling actual organisms, including whatever

idiosyncrasies they might have.

- Historically, Behaviour Analysis has developed a distinct perspective and theory on organism behaviour, different from approaches based on mental mechanisms such as Cognitive Psychology. The **Behaviourist Agent Architecture** helps in bringing these differences to computational agent architectures, in which cognitive approaches, such as SOAR (Laird *et al.*, 1987) and ACT-R (Anderson *et al.*, 2004), predominate.

- Because our agent architecture is geared towards representing actual animal behaviour, it is more suitable for modelling such behaviour than approaches that lack similar empirical basis. This feature might be useful, for instance, in modelling and forecasting social behaviour. This kind of study has already been done using much simpler models of agency. Epistein and Axtell (1996), for example, employ a kind of cellular automaton to this end. It would be interesting to undertake similar studies using behaviourist models instead.

- The structures used to represent conditioning, such as the stimulus graph, are themselves objects to be studied. Questions might be asked about how exactly organisms search over these structures, and therefore the architecture can be refined to reflect actual behaviour more closely. These structures can be seen as a type of semantic network (Sowa, 1987) specialized for behaviourist phenomena.

- The several parts of our architecture work with each other in order to provide the final organism behaviour. For instance, the operant utility is calculated considering the stimulus utility, which in turn might be changed by an emotion or a drive. Naturally, this deep relationships are missing in many approaches that seek to capture only specific parts of organisms, such as it is usually done in the reinforcement learning area (Russell and Norvig, 2002), and even in the treatment given to operant conditioning in the *skinnerbots* of Touretzky and Saksida (1997). While there are benefits in such an isolation, it is clear that there is also some loss if the objective is to model a whole organism, whose behaviour is a consequence of several interacting mechanisms.

- Moreover, learning by reinforcement techniques, though inspired by behaviourism, often restrict themselves to the notions of *reinforcement* and *punishment* without further analysis. Our formalization, however, provides finer structures to model learning. For instance, we have seen that both reinforcement and punishment can be subdivided into *positive* and *negative*. And each might have particular characteristics. Negative punishment, for example, triggers the emotion of depression. Furthermore, extensions of our architecture may profit from this finer structure in order to add other characteristics in an equally fine manner.

- We have shown that drives and emotions can be defined in behaviourist terms and are relevant to model actual organism behaviour. However, they are not usually considered by other agent architectures. Even the few approaches based on behaviourism that we are aware of ignore this.
- Our characteristic employment of an utility function differentiates our approach from that of McDowell *et al.* (2006), which explicitly avoids any similar scheme. Hence, it is not clear whether it is possible to express drives and emotions in their framework. In any case, they do not show how to do such a thing.
- Operant utility calculation can be seen as a form of automated planning (Nau *et al.*, 2004), because it involves the composition of a sequence of operants in order to achieve a goal (i.e., the best stimulus available). As such, our architecture could benefit from the research done in this area. This need for planning is also at the heart of BDI approaches and related agent-oriented programming languages, such as AgentSpeak(L) (Rao, 1996).
- Though d'Inverno and Luck (2003) provides an extensible agent theory based on the Z Notation, we define our own model from scratch. The reason is that we are interested in specifying very basic mechanisms, such as the notion of behaviour itself, and in this case the foundation provided by the SMART framework would not be suitable. Moreover, in SMART the bound between agents is the goals they share, whereas in our approach agents can only be related by mutual stimulation, not some abstract goal. Nevertheless, its use of Z clearly influenced our choice of a formal notation for agent modelling.

3.2 Environment Model

The complete contribution is presented in Chapter 5.

As we remarked in Section 1.1, the environment of an MAS can be used as a crucial element in its automated analysis. Because the greater complexity of interest is often within the agents, which are thus simulated as black-boxes, simpler and merely coordinating functions can be attributed to their environment. These functions, in turn, lend themselves to simple and explicit formal representations, which is attractive from a formal verification standpoint. It was with this in mind that we designed the **Environment Model for Multi-Agent Systems (EMMAS)**[4].

[4]Part of which we published in (da Silva and de Melo, 2011a).

EMMAS provides both a suitable coordination mechanism for our behaviourist agents and a way to specify several possible experimentation scenarios in a succinct manner. This latter point is particularly distinctive, as it provides the basis upon which we can perform formal verification.

To achieve this, we employ ideas from process algebra by realising that our agents can be seen as communicating processes. Indeed, **EMMAS** itself is defined on top of the π-calculus process algebra (Milner, 1999). Among other qualities, π-calculus has a simple operational semantics (i.e., the meaning of expressions is given by considering a transition system). This means that it is possible to transform an environment specification into a transition system, in which runs (i.e., sequences of states and events) denote the possible evolutions of the environment.

An **EMMAS** specification defines three aspects of an environment:

- Which agents are present. The agents are seen as black-boxes, and only a minimal interface is explicit in the specification (i.e., what stimuli each can receive, and actions each can emit).
- How these agents relate to each other. These relations are given by defining how the action of an agent is transformed into a stimulus for another. The environment structure, then, is a social network (in the sense of Wasserman *et al.*, 1994).
- A number of behaviours of the environment itself, which are specified as **operations**.

All of these elements are put in parallel composition in the specification so that they can interact.

It is mostly through environment behaviours that experimental situations can be specified in **EMMAS**. For example, suppose that one wishes to test two different ways to manipulate a group of agents. One can define an operation Op_1 to account for the first, and another operation Op_2 for the second. Then, to specify that the two are to be experimented with, one requests a non-deterministic choice in the form of the following composed operation:

$$Op_1 + Op_2$$

During simulation, this means that there are two distinct possible courses of actions – one employing Op_1 and another employing Op_2. During verification, both of these simulation paths might be examined to ensure that some property holds in both cases. The mechanism by which such choices are made can vary, but in this thesis they are defined by the verification algorithms we provide and which we present later.

Owing to its process algebraic foundation, there are a number of such composition operators, which can be used in a structurally recursive way to define intricate operations. Moreover, **EMMAS** also provides primitive operations which must be used when building more complex ones. These primitive operations include, for instance, the stimulation of agents and the creation of new relations between agents. As a further example, consider the following operation:

$$((Stimulate(s_1, ag) + Stimulate(s_2, ag));$$
$$(Stimulate(s_3, ag) + Stimulate(s_4, ag))) \parallel$$
$$Stimulate(s_5, ag)$$

It defines: (i) that an agent ag be stimulated first by either s_1 or s_2, and then by either s_3 or s_4; but also (ii) that at any moment ag might be stimulated by s_5. As in the previous example, though not as obviously, there exists a transition system that represents all of the possibilities contained in this operation.

3.2.1 Comparison with Other Approaches

The ELMS (Okuyama *et al.*, 2005) approach we saw in Section 2.1.2 also employs the idea of combining environment specifications with agent implementations to perform simulations. However, there are a number of important differences. Most crucially, ELMS does not seem designed with formal verification goals in mind. Thus, differently from our method, no underlying semantics amenable to formal analyses (e.g., transition systems) is provided in ELMS. In particular, it is not possible to specify multiple situations via non-deterministic operators, which hinders its applicability to the specification of experimental situations. Furthermore, ELMS's agents are assumed to be implemented using AgentSpeak(L)(Rao, 1996), whereas our approach assumes a different agent architecture, as explained previously.

Ferber and Müller (1996) develops an execution scheme similar to ours for defining environments, which is formalized by Ferber (1999). However, the formalism itself is entirely different, as it is based on Petri nets and not on process algebras. Other important differences are:

- Although Petri nets are used, the environment reactions to agent actions are actually not given in the Petri net formalism. Rather, any description language can be used to do so. This presents a difficulty to formally analyse these reactions. In contrast, our **EMMAS** is entirely formal, and therefore poses no such difficulty. This restricts what can be expressed (since it is not an arbitrary description language), but

allows entirely formal investigations with respect to the environment's structure and behaviour.

- The focus of Ferber's (1999) environments is on the synchronization of agent influences and environment reactions, as required by the theory developed by Ferber and Müller (1996). No provision is made for the concise definition of multiple experiments and situations, which is a main concern in **EMMAS**. This arises, in part, from the previous point, since if the behaviour of environments (beyond synchronization) is not given explicitly by Ferber's (1999) formalism, then it is not possible to explore them systematically.

- Ferber's (1999) agents are supposed to signal when they are done with their actions, so that one can determine which group of actions can be considered simultaneous. **EMMAS** requires no such thing from the agents. Rather, all possibilities of simultaneous actions are automatically considered at the semantic level. This arises from our focus on verification.

- Ferber's (1999) environment aims at being domain independent, while **EMMAS** has no such ambition, and addresses only a class of MASs.

It is also enlightening that Ferber (1999, p. 211) shares one of our main concerns:

> Unfortunately, very little work has been done on modelling environments, and details relating to environments are usually lost in explanations of systems which have implemented them, or indeed completely buried in the code for their implementation.

Both our contribution and Ferber's (1999) aim at this problem.

We employ a process algebra to provide the semantics of **EMMAS** and, as seen in the previous section, use the Z Notation to specify the agents. Thus, there are similarities with the Circus (Woodcock and Cavalcanti, 2001) method, in which the process algebra CSP and Z are combined in a uniform framework. However, here we use the π-calculus instead of CSP. Moreover, although the agents are specified in Z, this specification is used mostly as a guide for implementing them, so that during verification and simulation the algorithms do not manipulate its internal structures. That is to say, verification is achieved by manipulating the structures of the environment, and agents are considered mostly as black-boxes with interfaces.

3.3 Transition Systems and Semantics

The complete contribution is presented in Chapter 6.

We saw in the previous section that **EMMAS** environments can be put in terms of transition systems. In this thesis we develop a particular kind of such structures, which we call **annotated transition system (ATS)**. Thus, a crucial step to formally analyse an MAS in our approach is to translate an **EMMAS** specification into an **ATS**. We do this in two different ways.

The first is by giving a general semantics of **EMMAS** in terms of an **ATS**. This is achieved by considering the operational semantics of π-calculus, which provides a transition system, and then removing certain undesirable runs. The resulting **ATS** provides a rigorous semantics, but is does not include implementation details that are needed for particular applications – it is an abstract semantic model, and shows that **EMMAS** is not limited to simulation applications.

However, since the technique developed in this thesis is based on the possibility of simulating an **EMMAS** specification, it is necessary to provide a more concrete translation from such a specification to an **ATS**. This second way of performing a translation is tied to the simulator and the associated verification technique, and therefore is considered in Section 3.4.

3.3.1 Comparison with Other Approaches

Transition systems are classical mathematical entities to represent semantics of concurrent systems. For example, they are used by Milner (1999) to provide the semantics of both CCS and π-calculus process algebras. Model checking techniques (Baier and Katoen, 2008) employ transition systems as well to represent the systems to be analysed. Therefore, with respect to transition systems, our contribution, if any, is merely the definition of a particular kind of transition system (the **ATS**s) that groups a number of well-known features useful in our work, in particular: input and output events, internal events, labelled transitions and labelled states.

Our significant contribution, though, is the provision of a semantics for our **EMMAS** in terms of such transition systems. To our knowledge, it is the only MAS environment to have such a semantics. The practical importance of this is that, as we shall see in the next section, such a semantics is fundamental for the verification technique we develop.

3.4 Verification Technique

The complete contribution is presented in Chapter 7.

To apply the verification technique developed here, we first need a suitable **ATS** to represent the MAS, which can be derived from an **EMMAS** specification. As explained previously, the general semantics to be developed in Chapter 6 does not contemplate the particular needs of the simulator required for the application of the technique. The problem, essentially, is that there must be a way to explicitly request, during simulation, that agents change their current actions and acknowledge changes in environmental stimulation (i.e., the agents must be updated). This is an implementation issue, and therefore we have not included in the general **EMMAS** semantics. Nevertheless, a more concrete semantics is given by introducing a new event, called *commit*, which works as a signal to make such a request. This produces an **ATS** whose runs can be directly interpreted by the simulator. In particular, whenever the simulator finds a commit event in a run of the **ATS**, it enforces the update of the agents. Moreover, this **ATS** is mathematically well-defined in its entirety, but its actual construction is done during simulations, as each particular state becomes relevant - that is, on-the-fly.

One models an MAS in order to study its properties. In this thesis, we propose a way to do so by formulating hypotheses about the MAS and automatically checking whether they hold or not (e.g., "every time the agent does X, will it do Y later?").[5] If a hypothesis does not hold, it means that either the hypothesis is false or the MAS has not been correctly specified. The judgement to be made depends on our objectives in each particular circumstance. Are we trying to discover some law about the MAS? In this case, if a hypothesis that represents this law turns out to be false, it is the hypothesis that is incorrect, not the MAS. Are we trying to engineer an MAS that obey some law? In this case we have the opposite, a falsified hypothesis indicates a problem in the MAS. This view is akin to that found in empirical sciences, in which scientists investigate hypotheses and make judgements in a similar manner. In this respect, the main difference is that the empirical scientist studies the natural world *directly*, while we are concerned with *models* of nature in the form of MASs.

In this thesis, such a hypothesis is defined by specifying a **simulation purpose** and a satisfiability relation. If the MAS satisfies the specified **simulation purpose** with respect to the desired satisfiability relation, then the hypothesis is corroborated. Otherwise, it is falsified. Formally, a **simula-**

[5] We have published part of this technique in (da Silva and de Melo, 2011b).

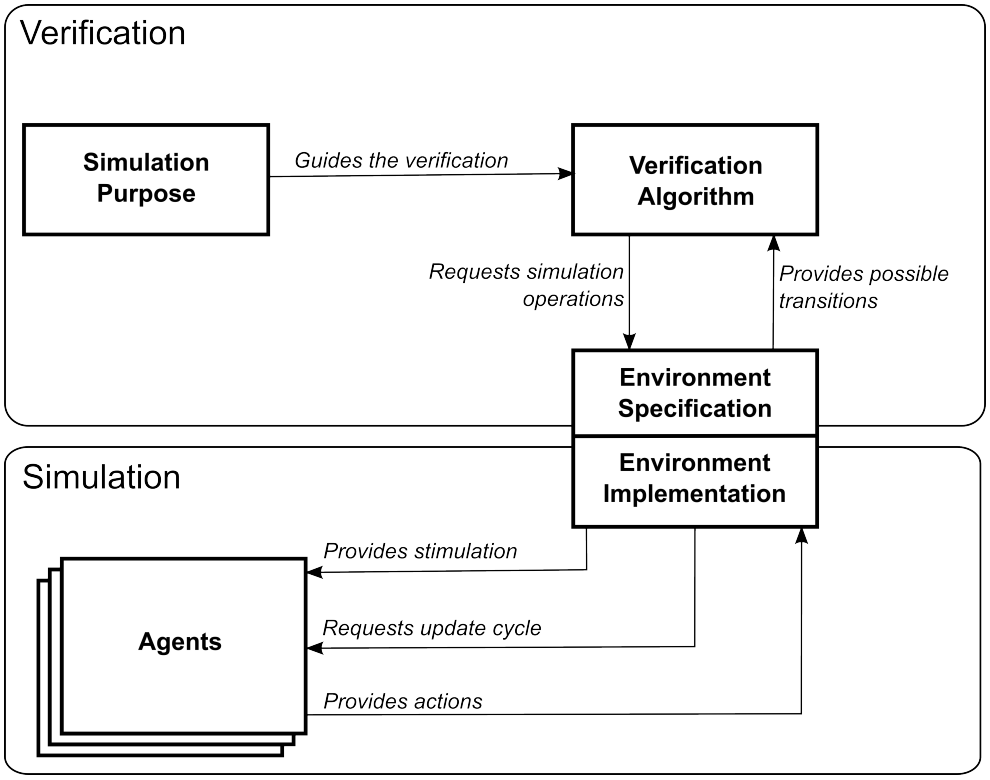


Figure 3.1: Verification and simulation elements interaction. Notice, in particular, the important role that the environment has in relating verification and simulation. It acts as a coordinator which, on the one hand, formally defines what can be done, while on the other hand requests actual simulator operations. The environment's implementation is provided mainly by a π-calculus simulation library, as explained in Section 3.5 below.

tion purpose is an **ATS** subject to further restrictions. In particular, it has finitely many states and defines two special states, *Success* and *Failure*. All **runs** that lead to *Success* denote desirable simulations, whereas all that lead to *Failure* denote undesirable ones. Moreover, differently from the **ATS** that is automatically and progressively derived from an **EMMAS** specification, these **simulation purposes** must be specified explicitly and *a priori*.

Simulation purposes not only give criteria for correctness but are also employed to guide the simulation, so that states irrelevant for the property are not explored. The verification is achieved by building – on-the-fly – a special kind of ***synchronous product*** (written $\mathcal{SP} \otimes \mathcal{M}$) between an **ATS** $\mathcal{M}$ representing the MAS of interest and a **simulation purpose** $\mathcal{SP}$ denoting the property of interest. This **synchronous product** is itself an **ATS**, in which states are of the form (q, s), where q is a state of $\mathcal{SP}$ and s is a state of $\mathcal{M}$.

SYNCHRONOUS PRODUCT

A **feasible run** in this product is a run whose last state (q, s) is such that $q = \mathit{Success}$.

This formal construction depends on and influences the behaviour of agents, which are simulated as black-boxes (but have a known interface). For example, a transition that states that a certain agent has performed a certain action can only take place if the agent in question really performed that action (i.e., if the simulator, after being queried, informs that the agent did so in the simulation). Conversely, if a transition that specifies that a certain agent receives a certain stimulus takes place, then it is necessary that the agent really receives the stimulus (i.e., by requesting the simulator to stimulate it). This interaction with the simulator is formalized by the provision of an abstract simulator interface, which can be incorporated in formal definitions and implemented by the actual simulator. Hence, there is an interplay between the formal analyses and the simulation. The formal structures provide order to the simulation in the form of an abstract representation, but they would be pointless without an actual simulation to put in order in the first place. This close relationship between these two aspects is a distinguishing feature of the work developed in this thesis (see Figure 3.1).

This **synchronous product** can be used in various ways to define whether $\mathcal{SP}$ satisfies $\mathcal{M}$. In this thesis we define the following such satisfiability relations:

FEASIBILITY

- ***Feasibility***: $\mathcal{SP}$ is **feasible** with respect to $\mathcal{M}$ if there is at least one **run** in $\mathcal{SP} \otimes \mathcal{M}$ which terminates in a state (q, s) such that $q = \mathit{Success}$. There are weak and strong variants of this.

REFUTABILITY

- ***Refutability***: $\mathcal{SP}$ is **refutable** with respect to $\mathcal{M}$ if there is at least one **run** in $\mathcal{SP} \otimes \mathcal{M}$ which terminates in a state (q, s) such that $q = \mathit{Failure}$. There are weak and strong variants of this.

CERTAINTY

- ***Certainty***: $\mathcal{SP}$ is **certain** with respect to $\mathcal{M}$ if all **runs** in $\mathcal{SP} \otimes \mathcal{M}$ terminate in a state (q, s) such that $q = \mathit{Success}$.

IMPOSSIBILITY

- ***Impossibility***: $\mathcal{SP}$ is **impossible** with respect to $\mathcal{M}$ if all **runs** in $\mathcal{SP} \otimes \mathcal{M}$ terminate in a state (q, s) such that $q = \mathit{Failure}$.

Each satisfiability relation is verified by a different, but similar, algorithm. They all share the following main characteristics:

- They perform a depth-first search on the **synchronous product** of $\mathcal{SP}$ and $\mathcal{M}$;
- The search has a maximum depth, depth_{max};

- $\mathcal{SP} \otimes \mathcal{M}$ is computed on-the-fly (i.e., it is not computed *a priori*; rather, at each state, the algorithm calculates the next states necessary to continue), because $\mathcal{M}$ itself is obtained on-the-fly from the π-calculus expressions present in each of its states. Visited states in $\mathcal{M}$ are not marked as such.
- A simulator interface is assumed to exist. This is used to control the simulation execution, including the possibility of storing simulator states and backtracking to them later.
- The algorithm is guaranteed to terminate, and the result is a conclusive or inconclusive verdict. If conclusive, it becomes known whether the MAS satisfies the **simulation purpose** with respect to the observations made during simulations. If inconclusive, it is possible to perform some adjustments and try again.

Regarding the complexities of the algorithms, this means that they must be given mainly in terms of $depth_{max}$ and the maximum branching factor (i.e., the maximum number of possible successors of any state), instead of the number of states and transitions in the complete transition system. Moreover, since states in $\mathcal{M}$ are actually computed from an **EMMAS environment** specification, the complexities of these computations must be taken into account as well. These characteristics lead to many parameters to be accounted for in the statement of the complexities. The complete development is given in Chapter 7. In a few words, the complexity in space is polynomial with respect to the size of the **environment** and other parameters, and the complexity in time is exponential with respect to $depth_{max}$.

What is important in this technique is that, once given a **simulation purpose**, it chooses which simulations to execute automatically and in a systematic manner, instead of depending on a user to guide and inspect the simulation manually, thereby exploring the possible simulations more efficiently, even if inconclusively. Moreover, the algorithms are carefully shown to be correct according to precise notions of soundness and completeness.

3.4.1 Comparison with Other Approaches

Our approach is largely inspired by TGV (Jard and Jéron, 2005), which we saw in Section 2.2.2. But we differentiate ourselves fundamentally because our objective is not the generation of test cases, and in particular we are not tied to the *ioco* conformance relation. Indeed, our **simulation purpose** is itself the structure that shall determine success or failure of a verification procedure (i.e., not some *a posteriori* test cases). As a consequence, different criteria of success or failure can be given, and then computed on-the-fly. As we saw in

Section 1.1, a number of particular methodological considerations are at the heart of these definitions. Moreover, there are also other technical differences, such as the fact that we use labelled states (and not only transitions), and that **simulation purposes** need not be input complete.

Although uncommon, there are works on the verification of simulation traces, as we have seen in Section 2.4.3, such as that of Bosse *et al.* (2009). Our method, however, distinguishes itself mainly by actually *guiding* the simulation, and not only checking properties over traces *a posteriori*.

Though quite different, our technique has nevertheless common characteristics with Model Checking, presented in Section 2.2.1. Most importantly, both assume the existence of an explicit set of states and transitions to be analysed. In Model Checking this set is examined exhaustively, so that a conclusive verdict can always be given, provided that there are enough computational resources. In our case, by contrast, only a small part of the state-space is explored (i.e., those that are reached by the simulations performed), and one can never be sure of having explored every possible state, since agents are given as black-boxes. Moreover, both methods allow the specification of a property of interest to be analysed with respect to a system, thus establishing a difference between the model and the properties of the model. As we have seen, in Model Checking such a property is typically given in terms of some temporal logics. In our approach we use **simulation purposes** instead. By similar reasons, it is also clear that our approach is distinct from the application of Model Checking to formal MASs specifications, as surveyed in Section 2.4.4.

The particular case of *Bounded* Model Checking is also worth commenting. In this approach, one limits the length of a counterexample run to some constant. In this manner, the problem can be translated to an instance of SAT and addressed using SAT solvers. In our algorithms, we also limit the runs we examine to some constant. However, this is done not to allow a translation to another format such as SAT, but simply because the search in the **synchronous product** of a **simulation purpose** and an **ATS** must have a maximum depth. Otherwise, the search could never end, since it is possible to have infinite branches in the **synchronous product**.

In our verification algorithms we shall need a preprocessing procedure to calculate shortest distances from a certain vertex in the graph induced by the specified **simulation purpose**. To this end, we could use Dijkstra's algorithm (Cormen *et al.*, 2001). However, the edges in our graph all have weight 1 (i.e., we count hops between a vertex and its successors), which permitted the development of a more specific algorithm. The reason is that, in this case, it is not necessary to keep a priority queue with unexplored vertices (ordered according to their current distances), which needs to be regularly re-ordered to account for updated entries. It suffices to explore the vertices in a depth-first

manner.

3.5 Tool Implementation

The complete contribution is presented in Chapter 8.

The theory presented in this thesis has been implemented as a software tool following the architecture outlined in Figure 1.1. In this respect, there are two main distinct artefacts:

- The implementation of the agent model;
- The simulation and verification tool itself, FGS.

The implementation of agent model is actually a Java library that can be used in many different manners, since it is merely a realization of an agency theory. It allows the creation of *Organism*s objects, which are initialized by the provision of an XML[6] configuration file. In this file one specifies all that makes the particular organism unique, such as the stimuli it is capable of receiving, the actions it can perform, the operants it has already learned, and so on.

In this thesis, we use these agents to specify MASs subject to simulation and verification. To this end, we provide the FGS tool, also written in Java, which takes the following as its main inputs:

- *A component repository.* Components provide the implementation of particular kinds of agents or properties. Their instantiations are the "black-boxes" that are simulated. Our agent model is provided as one such component. Propositions about these agents (e.g., "stimulus X is reinforcing to agent Y.") can also be provided as a special kind of component, called properties, whose values are calculated during simulation as well.
- *A scenario description.* A scenario is an XML file that specifies the MAS to be simulated. It defines the agents that are present, as well as the environment in which they exist. To define an agent, one specifies the component that implements it and the configuration file to initialize it. The environment, in turn, can be specified by using tags that map to **EMMAS** elements (i.e., the `<choice>` tag maps to the the + operator).

[6]Extensible Markup Language.

- *An experiment description.* An experiment is an XML file that specifies what kind of simulation and verification should be done with the given scenario. **Simulation purposes** can be defined by explicitly listing their states, events and transitions.

Since the semantics of **EMMAS** is given in terms of the π-calculus, we have implemented a π-calculus simulation library that is used by FGS to simulate as directly as possible **EMMAS** specifications. A main advantage of proceeding in this manner is that modifications and additions to **EMMAS** can be more easily implemented. On the other hand, the direct simulation of π-calculus brings some efficiency issues, which require the implementation of certain optimizations.

Finally, it is worth to emphasize that, besides its theoretical foundation, FGS is designed with some practical engineering concerns in mind. In particular, the fact that it is based on components allows the substitution of agents without affecting the simulation and verification infrastructure. So, for instance, if a different agent implementation is devised, it can be immediately employed with the existing tool using the existing scenarios and experiments.[7]

3.5.1 Comparison with Other Approaches

The main innovation of FGS, of course, is the implementation of the novel techniques and models introduced in this thesis, which we have already considered. Apart from that, two other characteristics are noteworthy: the π-calculus simulation library, and the component-based architecture of the system.

Existing implementations of the π-calculus, such as Pict (Pierce and Turner, 1997) and CubeVM (Peschanski and Hym, 2006), are geared towards using the π-calculus as a foundation for programming languages. This means that given a specification (indeed, a program), only one possible execution path is considered. Since for the purpose of verification it is necessary to be able to consider all possible executions (at least up to a certain length), these approaches are insufficient to address our concerns. The π-calculus simulation library of FGS is designed to fulfil this need. It provides an executable implementation of the π-calculus which can be used to systematically investigate all possible executions (up to a certain length). It does that by providing access to the current state of the underlying transition system of the π-calculus process, which can then be used to explicitly calculate the possible successors. A

[7] An earlier version of this simulation infrastructure was designed exactly to show the value of a component based approach to multi-agent simulation. We published this result in (da Silva and de Melo, 2008), but at that time the simulator had not incorporated yet the MAS models and verification techniques proposed in this thesis.

number of optimizations are required in order to make this calculation more efficient.

The architecture of FGS itself, in turn, has certain distinctive features worth commenting in what relates to its focus on the separation of concerns and reuse of artefacts (i.e., components, scenarios and experiments).

It is generally acknowledged that it is important to separate the simulator infrastructure from the models being simulated. Swarm (Minar *et al.*, 1996), MASON (Luke *et al.*, 2004) and Repast (North *et al.*, 2006), which are popular multi-agent simulation platforms, try to achieve this by providing both a framework to program models and a simulation engine to run them. This helps reuse simulation infrastructure, but does not simplify the reuse of parts of simulation models in different simulations built by different people.

A new version of Repast, called Repast S (North *et al.*, 2005), is being developed in order, partly, to address this issue. This new version is similar to our system in that external Java classes can be arranged together declaratively (i.e., without Java programming) to compose simulation models. However, we differ from them in a number of ways. First, the simulation models of Repast S are mostly restrictions on which components are in the model, while our models carry information regarding not only the components, but also their actual instantiation (i.e., we represent a complete state of affairs). Second, Repast S has a very inclusive definition of component, so that any Java class can be a component, while we enforce several requirements in order to attain more semantics. Third, Repast S aims at being a general platform, while we prefer to adopt a domain-specific approach, which we believe to lead to more elegant and manageable simulation models, albeit with more limited applications. Besides the points we shall discuss later, we think that the simplicity thus achieved is also important in order to make simulators more accessible to non-programmers, which is one of our objectives.

The idea of a component-based agent simulation environment is also used in the Quicksilver project (Burse, 2000). In that work, any compiled Java class can be treated as a component. Some predefined classes of agents are provided and a special tool allows the user to instantiate classes, connect instances and run the simulation thus assembled. In this way, agents can be reused in several simulations. This approach, however, suffers from some problems. Like Repast S, it relies on a very inclusive definition of component, which implies that such components do not bring any advantage over normal Java classes. The reuse technology is in the composition tool that allows arbitrary instances to be easily connected by the user, but this is not really specific to simulation, nor does it help in enforcing any special semantics to the underlying classes. Hence, such a reuse mechanism is mostly a general Java technology, which allows one to build programs in a different manner. This contrasts with our approach,

in which components are highly structured entities designed for simulation: they must implement predefined interfaces, be annotated in special ways and be deployed to a special location. Furthermore, Quicksilver assumes that the user has knowledge of Java programming. This makes it inaccessible to non-programmers, whereas our approach has the contrary goal of facilitating their access to simulation.

This goal is also shared by NetLogo (Wilensky, 1999), an environment designed to simplify the creation of simulation models. To this end, a graphical editor and a set of controls (e.g., buttons, sliders, plotters) are provided in order to build the simulation front-end, while a special procedural scripting language (assumed to be easier to learn than a general purpose language such as Java) can be employed to specify the actual simulation behaviour. The system, however, does not offer any reuse mechanism beyond copy-and-paste of scripts.

With similar purposes, but in a more sophisticated realization, SeSAm (Klugl and Puppe, 1998) provides a rich application in which users may create agents, setup simulations and run them. Agent creation relies on a base library of agent properties, which must be used in order to define new agents. Though simple agents can be built easily with point-and-click interaction, more advanced ones require the use of a custom scripting language. The created agents are then instantiated in order to build one or more simulation situations, all of which are stored in a model file. SeSAm, however, does not provide advanced facilities to reuse such agents across different simulation models. The only way to do so is through importing a model into another, which amounts to a copy-and-paste technique. On the other hand, programmers have the possibility of extending SeSAm through plugins written in Java. Such plugins allow the definition of new elements that the final user may employ when building his agents (e.g., new functions to be used when specifying the behavior of agents). Therefore, SeSAm does provide an interesting reuse technology, but whose purpose differs from ours, which aims at the easy reuse of whole agents and other simulation elements.

Finally, in (Okuyama *et al.*, 2005) we find the ELMS markup language, which bears some similarities to our scenario language in that both describe features of the simulation environment. However, the objective of ELMS is to restrict the kinds of entities that exist, while ours is to explicitly define and compose individual entities. Moreover, ELMS is geared towards agent-oriented programming, while our underlying programming paradigm is the more common object-oriented one.

3.6 Conclusion

In this chapter we have presented an overview of the main technical contributions of this thesis and related them to existing work. In order to develop our approach, we build upon a number of ideas from different areas, thereby integrating them in a coherent framework. To clarify this overall relationship, Table 3.1 selects some of the most relevant related work discussed here and summarizes the comparison with our method by describing crucial aspects of each. From this comparison, it can be seen that our work borrows, integrates, contrasts with and adds to ideas from disparate sources.

Work	**Specification Formalisms**	**Semantic Model**	**Property Checked**	**Automated Analysis**	**Agents**	**Environments**
This thesis	Z, π-calculus (base of **EM-MAS**)	**ATS**	**Simulation Purpose** satisfiability	Guided simulations; hypothesis testing	Behaviourist	Social networks; context for experiments; dynamic; non-deterministic
Skinnerbots (Touretzky and Saksida, 1997)	–	–	–	–	Behaviourist	–
SMART (d'Inverno and Luck, 2003)	Z	–	–	–	Any goal oriented	Goal dependencies
BDI architecture (Rao and Georgeff, 1995)	–	–	–	–	Practical reasoning	Dynamic; non-deterministic
Influences and reactions (Ferber and Müller, 1996; Ferber, 1999)	Petri nets	–	–	–	Any	Any synchronous environment
ELMS Okuyama *et al.* (2005)	–	–	–	Simulation; animation	BDI-based	Grids; reactive resources
TGV (Jard and Jéron, 2005)	–	IOLTS	*ioco* conformance	Test case generation	Reactive systems	–
TTL (Bosse *et al.*, 2009)	TTL	Based on predicate logic's	TTL property	Verification of traces	LEADSTO specification	LEADSTO specification
MAS model checking (e.g., Wooldridge *et al.*, 2006; Lomuscio *et al.*, 2009)	Various languages	Transition systems; interpreted systems	Temporal logic formulas	Exhaustive model checking	BDI-based; epistemic	–
MAS simulation such as SWARM (Minar *et al.*, 1996), RePast (North *et al.*, 2006) and MASON (Luke *et al.*, 2004)	–	–	–	Simulation; animation; statistics compilation; input optimization	Any	Grids; social networks; dynamic

Table 3.1: Comparison of this thesis with some important related works. Columns designate the aspect to be analysed, and lines show the related works. Only the most relevant references discussed in this chapter are given. A dash (–) indicates that the aspect is not significantly addressed in the work. From this table, it is clear that the approach proposed in this thesis relates to ideas from very different areas.

Part II

Multi-Agent Systems

CHAPTER 4

Behaviourist Agent Architecture

Behaviourist Agent Architecture

In this chapter we present the ***Behaviourist Agent Architecture***, the novel agent architecture introduced by this thesis. Its structure follows core principles of Behaviour Analysis, as presented in Section 2.1.1.4, which we organize in five classes: (i) stimulus conditioning; (ii) respondent behaviour (i.e., reflexes); (iii) operant behaviour; (iv) drives; and (v) emotions.

Despite its many details, one can abstract two themes common to all of them: adaptation and learning. These concern how environmental stimuli affect the actions of agents over time. In this way, phenomena pertaining to agents are closely related to the possibilities offered by an external environment. In this chapter, however, we focus on agents – the corresponding environments are addressed in Chapter 5.

We explain the main concepts of our work in an informal manner, but the architecture itself is given as a formal specification written in the Z Notation. This provision ensures that the architecture is defined in a precise and compositional form. The benefits of precision are evident. But compositionality should also be valued, for it allows each part of the specification to be examined and modified separately, and thus allows further progresses to be made upon it. Indeed, certain parts in our specification, which we call **extension points**, are designed to be changed[1] in order to allow the experimentation with variations of particularly important mechanisms (e.g., the computation of the utilities assigned to stimuli).

[1] However, since the schema calculus is not monotonic with respect to refinement, special care must be taken when refining the specification. See Section D.6 of Appendix D.

The implementation of the specification is given as a Java program. In a complete formal development approach, this Java program should be proved correct with respect to the Z specification (e.g., by means of formal refinements). This was not done in this thesis, where the main formal effort has been devoted to formally guided simulations and the related verification algorithms, which are proved to be correct. However, the Java implementation of the agent architecture follows very closely the structure of its Z specification, which helps in avoiding errors that could arise in more complex implementation strategies. In the present chapter only the formal specification is considered. More details about its implementation are given in Chapter 8.

This chapter is organized as follows. First, in Section 4.1, we comment on the overall themes of adaptation and learning that permeate the architecture. In Section 4.2, then, we present our **Behaviourist Agent Architecture** specification in great detail, and this is the main contribution of the chapter. Finally, in Section 4.3 we conclude with some observations. A summary of the Z Notation is provided in Appendix D.

4.1 Adaptation and Learning

Despite the many specialized parts of the architecture we introduce in this chapter, there are two main themes that tie them together, namely, that of adaptation and learning with respect to an environment. It is thus worth to examine the general features of these two activities before proceeding to the architecture specification itself.

The agents considered in this chapter:

- exist within an environment that provides them with stimulation and which receives their actions;
- prefer certain stimuli to others;
- assume that relations exist between their actions and the stimuli that they receive.

Given these characteristics, adaptation concerns any change in the agent that is caused by external stimuli. Moreover, since agents have preferences, such changes often imply in behavioural modifications that make it more likely to obtain the preferred stimuli. For example, a hungry agent will be more likely to engage in actions that lead to food. In general, reflexes, emotions and drives are adaptations.

Learning is a form of adaptation with certain particularities. First, whatever is learned, is not present in the agent *a priori*. Rather, it must be gained through experience. Second, whatever is learned can be forgotten. That is to say, something learned is not intrinsic to the agent's constitution. An agent cannot learn to be hungry, but it can both learn how to get food and forget this when it is no longer useful. For learning to work, however, it is necessary that the agent possesses the predisposition to observe, remember and eventually forget relations between stimuli and actions. Hence, this is *a priori* with respect to learning. In this chapter classical conditioning and operant behaviour are defined as the main learning mechanisms of organisms.

Adaptation and learning experiences influence each other. For instance, an agent may know how to get food (through learning), but because it has eaten too much already, it has no interest in doing so (an adaptation to having eaten). This brings unity to these experiences, as they affect the agent as a whole.

All this is associated with an environment with certain characteristics. Clearly, the environment must be ordered in a way that there is something useful to learn about it. But perhaps less clear is the fact that this adaptive relation with the environment can be used to study the agents themselves. If, from an environmental perspective, one assumes that agents adapt and learn in particular ways, one is then in a position to manipulate stimulation and observe the behaviour of agents in order to infer individual as well as collective agent properties. This close relation between agents and their environment is inherited from the behaviourist tradition we subscribe to. The reason is plain: in behaviourist approaches, by definition, organisms are studied solely by means of stimulation and observation of the resulting actions. That is to say, by producing an ordered environment that the agents can adapt to *and* that reveals the mechanisms used for such an adaptation.

In a psychological setting, as this discussion implies, organisms are black-boxes and the objective is to discover, through experimentation, the mechanisms "hidden" therein. In this thesis, however, organisms are simulated, and for this it is necessary to provide these mechanisms in an executable form. As explained previously, in this chapter we do this by formalizing central characteristics of the Behaviour Analysis theory as a computable agent architecture. The corresponding environment, in turn, is the subject of Chapter 5. There, we shall see how these agents can be manipulated in order to reveal more about their behaviour, by putting the basic behaviourist mechanisms described here in motion.

4.2 Formal Specification

In this section we present our **Behaviourist Agent Architecture**. It is formalized using the Z Notation, which employs logic as part of its specification language. However, we emphasize that this does not mean that implementations following a specification should perform automated deductions. Rather, the logical statements in a Z specification serve only as criteria of correctness. They can be implemented by whatever means are available, as long as the final product respects the constraints imposed by the specification. This means that specialized and efficient algorithms can often be provided in an implementation. Moreover, the specification can be refined by adding further constraints, which allows one to extend it.

Let us then proceed with a systematic description of an agent's structures as defined by our architecture. We divide an agent into several parts, and to each one we provide the following information:

- *The rationale for its existence:* it is important to understand why it is needed before defining it formally.

- *The main elements of its formal specification:* since the full specification is quite large, for the sake of readability we show only its main elements in this chapter. Nevertheless, this gives the reader a comprehensive understanding of its main features. Furthermore, the full specification is provided in Appendix A. We reference this appendix whenever we omit some part of the specification, so that the reader may pursue the details if he or she so wishes.

- *How it could be changed:* the specification we provide can be changed in a number of ways. In particular, certain parts, which we refer to as ***extension points***, are designed to be specialized (e.g., through refinements). They are important aspects of agent behaviour that one may wish to modify. These **extension points** are commented throughout the text.

EXTENSION POINTS

These subsystems come together to form an *organism*, which is the name we give to agents that follow our specifications. Figure 4.1 presents an overview diagram of these subsystems and their relations. Formally, an organism is given by the *Organism* schema.

Organism

StimulationSubsystem

RespondingSubsystem

DriveSubsystem

EmotionSubsystem

In the following sections we present each of its constituent parts. But we begin by presenting some preliminary definitions which will be used throughout this presentation. And in the last section we show how to group all subsystems' operations.

4.2.1 Preliminary Definitions

The Z Notation does not provide a built-in manner to treat rational numbers. Therefore, we have built our own definition of rational numbers on top of the definitions available for integers. Moreover, our use of rational numbers is always restricted to a certain interval. Thus, the arithmetic operators that we define ensure that the upper and lower bound of this interval are respected. We call *magnitudes* this bounded kind of rational number, but denote them by the usual Q symbol.

$$
\begin{aligned}
Q == \{ & q : \mathbb{Z} \times \mathbb{Z} \mid \mathbf{let}\ a == \mathit{first}\ q;\ b == \mathit{second}\ q \bullet \\
& b \neq 0 \land \\
& a \ \mathsf{div}\ b \leq 1 \land \\
& a \ \mathsf{div}\ b \geq -1 \}
\end{aligned}
$$

There are also *positive magnitudes*, whose minimum element is neutral.

$$
\begin{aligned}
\mathit{PositiveQ} == \{ & q : \mathbb{Z} \times \mathbb{Z} \mid \mathbf{let}\ a == \mathit{first}\ q;\ b == \mathit{second}\ q \bullet \\
& b \neq 0 \land \\
& a \ \mathsf{div}\ b \leq 1 \land \\
& a \ \mathsf{div}\ b \geq 0 \}
\end{aligned}
$$

See Appendix A for more details about these elements.

The operators and the relations over magnitudes are denoted by their usual symbol followed by a subscript. This subscript is merely a technical convention to differentiate these symbols from the ones concerning integers. Thus, we have symbols such as $+_1$, $\leq_1$, and $=_1$, whose intuitive meaning should be clear (for the formal definitions, see Appendix A).

As a convenience, we define special kinds of magnitudes according to their use in the specification. Hence, *Intensity*, *Correlation* and *Probability* are

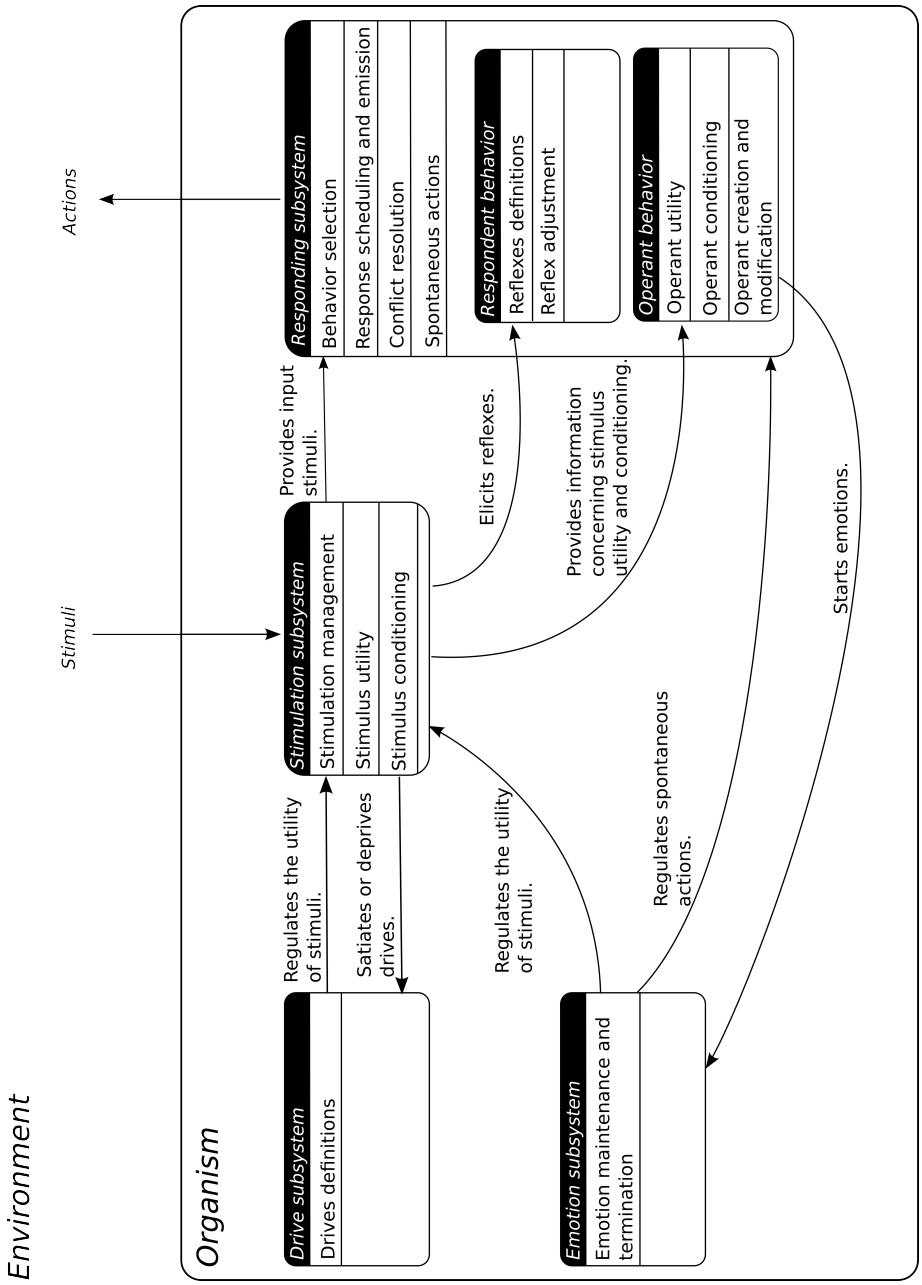


Figure 4.1: Overview of the main parts that form an organism. Each box denotes one such part and its main responsibilities. Arrows indicate important relations between these parts. An arrow from A to B means that A provides something that influences B.

all formally magnitudes (see Appendix A), but are used in different manners throughout the specification.

The passage of time is represented by natural numbers. But we also define new names for such integers according to their use, and thus we have the *Instant* and the *Duration* types (see Appendix A). Importantly, the time an organism has access to is based on counting its interactions with an external environment, and is therefore independent of any global, absolute, clock. The passage of time, thus, is a function of an organism's perception of an external environment, much like its behaviour (see Section 4.2.8 for more details).

Finally, we assume the existence of a *random* function which is capable of generating a random magnitude from any given instant (see Appendix A).

4.2.2 Stimulation

As we have seen, stimulation is one of the main concepts in behaviourist theories. It is only by means of stimulation that an organism can be influenced by its environment (which includes other agents in the environment). In this section we will see how stimuli are defined and perceived, how they relate among themselves, how the organism learns about such relations.

4.2.2.1 Basic Entities

First of all, there is a primitive set of stimuli.

$$[Stimulus]$$

Each particular organism will have its own, particular, set. But for the purpose of this specification, it suffices to have such an abstract set.

Recall that the organism is divided in a number of subsystems. The stimulation subsystem is one of them. It is defined by the *StimulationSubsystem* schema, and it holds the data structures relevant for stimulation.

StimulationSubsystem

$StimulationParameters$

$StimulusImplication$

$StimulusEquivalence$

$currentStimuli : \mathbb{P}\, Stimulus$

$pastStimuli : Instant \rightarrowtail \mathbb{P}\, Stimulus$

$stimulus_status : Stimulus \rightarrow StimulusStatus$

$stimulusBeginning : Stimulus \rightarrowtail Instant$

In particular, the subsystem is parametrized by the *StimulationParameters* schema, which isolates some important parameters that vary from organism to organism. An important parameter concerns the utility that the organism attaches to stimuli. Some stimuli are naturally pleasant or painful. These are called *primary stimuli*, for they have utilities *a priori*.

StimulationParameters

$StimulationHints$

$Conditioning_Ref1_Parameters$

$stimuli : \mathbb{P}\, Stimulus$

$primaryStimuli : \mathbb{P}\, Stimulus$

$primary_utility : Stimulus \rightarrowtail Utility$

$max_delay : Duration$

$\mathrm{dom}\, primary_utility = primaryStimuli$

Every stimulus has an associated status information, which records if the stimulation is beginning, ending, stable or absent.

$$StimulusStatus ::= Beginning \mid Ending \mid Stable \mid Absent$$

We call *hints* the stimuli that "give hints" about the state of the environment or another organism. The *StimulationHints* schema accounts for the hints available to the organism (see Appendix A). These hints are particularly useful when defining emotions (i.e., for instance, an angry organism will want to cause harm, and therefore there must be a way for him to detect harm).

Stimuli are delivered to organisms in the form of *Stimulation* schema, which carries information about their intensity and status. Moreover, the status is restricted to the two values that make sense in this context.

Stimulation

$stimulus : Stimulus$

$intensity : Intensity$

$status : StimulusStatus$

$status = Beginning \lor status = Ending$

4.2.2.2 Relations Among Stimuli

Organisms establish relations among stimuli. These relations help them in understanding their environment, and therefore in achieving their aims. For example, a dog may be taught that a whistle is always followed by the provision of food. Once the dog learns this relation, he will start salivating once he hears the whistle. The principle behind this phenomenon is known as *classical conditioning*.

Here we generalize this principle in the form of a relation we call *stimulus implication*. It accounts for the cases in which the organism believes that, given the presence of a certain stimulus, another stimulus will come. We use a reflexive and transitive order relation plus a *sCorrelation* function to model these beliefs. The *StimulusImplication* schema models this formally.

StimulusImplication

$sCauses : \mathbb{P}(Stimulus \times Stimulus)$

$sCorrelation : Stimulus \times Stimulus \nrightarrow Correlation$

$\forall\, s_1, s_2, s_3 : Stimulus \bullet$

$\quad (s_1 \ \underline{sCauses}\ s_1) \land$

$\quad (((s_1 \ \underline{sCauses}\ s_2) \land (s_2 \ \underline{sCauses}\ s_3)) \Rightarrow (s_1 \ \underline{sCauses}\ s_3))$

$\forall\, s_1, s_2 : Stimulus \mid s_1 \ \underline{sCauses}\ s_2 \bullet$

$\quad \exists\, c : Correlation \bullet ((s_1, s_2) \mapsto c) \in sCorrelation$

Notice that stimulus implication may be regarded as a directed graph (Figure 4.2), in which vertices represent stimuli and edges are the conditioning between stimuli. Furthermore, edges might have weight, if the correlation of the conditioning is to be taken into account.[2]

Stimulus equivalence captures the notion that, under some circumstances, a stimulus might be treated as if it is another. We define such a notion as

[2]Notice that this graph can also be seen as a semantic network (Sowa, 1987), but specialized for the representation of stimulation phenomena.

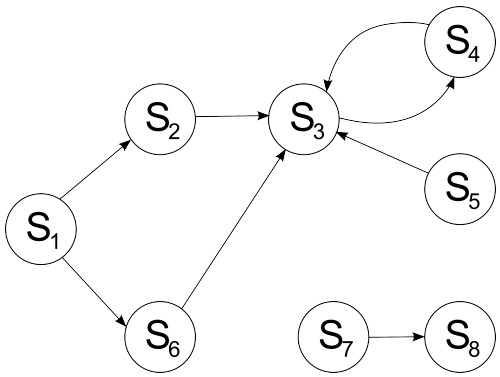


Figure 4.2: An example of stimulus implication represented as a directed graph.

a standard mathematical equivalence relation (i.e., reflexive, symmetric and transitive). The definition is given in schema *StimulusEquivalence* using the previous stimulus relation definition.

StimulusEquivalence

$$\begin{array}{l}
\mathit{StimulusImplication} \\
\mathit{equals} : \mathbb{P}(\mathit{Stimulus} \times \mathit{Stimulus}) \\
\hline
\forall\, s_1, s_2 : \mathit{Stimulus} \bullet \\
\qquad (s_1 \ \underline{\mathit{equals}}\ s_2) \Leftrightarrow (s_1 \ \underline{\mathit{sCauses}}\ s_2) \wedge (s_2 \ \underline{\mathit{sCauses}}\ s_1) \\
\forall\, s_1, s_2 : \mathit{Stimulus} \mid s_1 \ \underline{\mathit{equals}}\ s_2 \bullet \\
\qquad \mathit{sCorrelation}(s_1, s_2) = \mathit{sCorrelation}(s_2, s_1)
\end{array}$$

As in implication relations, stimulus equivalence may be represented by a graph (Figure 4.3). But the symmetry in equivalence relations requires the graph to be undirected.

4.2.2.3 Stimulus Utility

Recall from Section 2.1.1.4 that Behaviour Analysis assumes that the fundamental purpose of organisms is the maximization of pleasure and the minimization of pain throughout their lives. An organism, thus, can be thought of as an agent trying to maximize an *utility function*. Such a function, in turn, can be manipulated in a number of ways in order to modify the organism's behaviour.

The utility function associates an utility to each stimulus. The *Utility* type formalizes this quantity as a rational number between *min_utility* and *max_utility*,

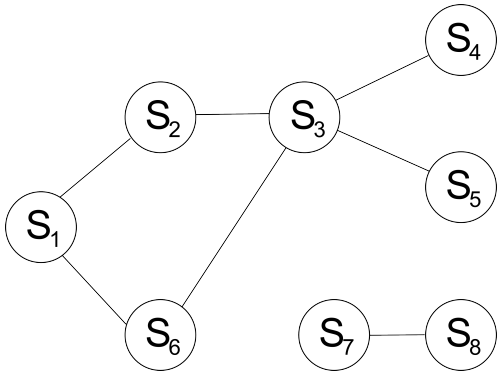


Figure 4.3: An example of stimulus equivalence represented as an undirected graph.

where the first indicates the greatest pain and the second stands for the greatest pleasure. There is also a *neutral_utility* element, which accounts for indifference (see Appendix A).

$$\mid \; Utility : \mathbb{P}\, Q$$

Having established the possible relations that hold among stimuli and the existence of some primary stimuli, it is then possible to define an utility for any given stimulus. Computationally, this requires the definition of a search that, for any stimulus s in a stimuli graph (e.g., the one shown in Figure 4.2), seeks the primary stimuli that s can reach and use their primary utility to assign an utility to s itself. There are, however, many possible and reasonable ways of giving such a definition. Thus, we first establish a very general schema, *StimulusUtility*.

StimulusUtility

$$StimulationSubsystem$$

$$EmotionSubsystem$$

$$DriveSubsystem$$

$$sUtility : Stimulus \rightarrow Utility$$

Refinements must then be provided. Our own refinement is given by the *StimulusUtility_Ref*1 schema, where the utility of a stimulus is defined as the utility of the best stimulus it can reach by the stimulus implication relation, but modified according to certain regulators (see Appendix A). These regulators account for the influence of drives and emotions, which we examine in Sections 4.2.6.2 and 4.2.7.2, respectively.

The computation of stimulus utility lends itself to different definitions. Different organisms may employ different mechanisms, and therefore each concrete case could be improved by an appropriate refinement of the stimulus utility we provide. For instance, for efficiency reasons, it might be the case that a particular search strategy is used when exploring the stimuli graph (e.g., depth-first, breadth-first, some kind of bounded search). This is an important **extension point** in the agent architecture, since it allows the extension of the architecture by merely providing a new refinement of *StimulusUtility*.

4.2.2.4 Stimulus Conditioning

Stimuli that are not primary gain their utility through association to primary stimuli. In general, this process is known as *stimulus conditioning* and the stimulus that has its utility modified is called a *conditioned stimulus*. Usually, stimulus conditioning resembles a causal law. That is, a stimulus is conditioned because it seems to cause another.

As a learning process, stimulus conditioning has two fundamental operations. The first is the conditioning itself, which strengthens the association between two stimuli. *ConditioningOp*_1 operation formalizes this. It states that if a stimulus s_1 is followed by a stimulus s_2 within a maximum delay, then the pair (s_1, s_2) must become part of the stimulus implication relation. If the maximum delay is not respected, than nothing changes, and this neutral behaviour is specified by the *ConditioningOp*_2 schema. (See Appendix A).

$$T_ConditioningOp \mathrel{\widehat{=}} ConditioningOp_1 \lor ConditioningOp_2$$

The second fundamental operation is the decay of the conditioning, which happens every time a stimulus is not followed by the expected consequence. That is to say, once the organism learns that a stimulus s_2 follows a stimulus s_1, it expects this to happen. If it does not happen, it loses confidence in this implication, and consequently the correlation between s_1 and s_2 is reduced. Moreover, if this correlation is below a certain minimum, the implication relation between the two stimuli is simply unlearned. This is specified by the *UnconditioningOp*_1 operation. If the conditions for such a decay are not met, nothing changes, and this is defined by *UnconditioningOp*_2. (See Appendix A).

$$T_UnconditioningOp \mathrel{\widehat{=}} UnconditioningOp_1 \lor UnconditioningOp_2$$

These definitions of conditioning and its decay are very general. For instance, they do not specify at what rate the conditioning should take place. To supply these details, one must refine these operations. We provide a simple linear policy refinement to conditioning. That is, a policy given by the following rules:

- the correlation grows with discrete increments, which are calculated every time the stimuli happen together;
- each increment is inversely proportional to the delay between the two stimuli. The proportion constant is c.

*Conditioning_Ref*1*_Parameters* schema factors out the definition of the conditioning parameters, such as the constant c. *ConditioningOp_Ref*1 operation, then, provides a refinement of *ConditioningOp*_1. Similarly, *UnconditioningOp_Ref*1 operation refines *UnconditioningOp*_1.[3] (See Appendix A).

$$T_UnconditioningOp_Ref1 \mathrel{\widehat{=}} UnconditioningOp_Ref1 \lor UnconditioningOp_2$$

Clearly, other refinements can be provided. This is another useful **extension point** of our agent model.

4.2.2.5 Stimulation

Stimuli may be delivered to or removed from the organism. Once delivered, the stimuli remain active until they are removed. That is, we assume that the environment does not signal the presence of stimuli, but only the change of such a presence. This convention will be useful later on, when modelling operant behaviour. Once a stimulus is delivered, the organism updates its properties until it is removed.

This delivery is controlled by three schemas, namely, *StimulationUpdateOp*_1, *StimulationUpdateOp*_2 and *StimulationUpdateOp*_3, each addressing a different step in the stimulation process (see Appendix A).

$$\begin{aligned} T_StimulationUpdateOp \mathrel{\widehat{=}} & \\ & StimulationUpdateOp_1 \lor \\ & StimulationUpdateOp_2 \lor \\ & StimulationUpdateOp_3 \end{aligned}$$

Once a stimulus is delivered, it becomes part of the set of *current stimuli* of schema *StimulationSubsystem*. The importance of this set of stimuli is in that it establishes a context for the organism's actions and observations, which is

[3] For the sake of illustration, the proof that *ConditioningOp* is refined by *ConditioningOp_Ref*1 is given in Appendix A right after the definition of *ConditioningOp_Ref*1. However, the refinements defined in this thesis are all rather simple, so we do not provide other similar proofs.

essential to learning processes such as the stimulus conditioning we have just seen and operant behaviour that is examined in Section 4.2.4.

Each stimulus has an associated *StimulusStatus*, which changes with time. For example, a stimulus that originally had the *Beginning* status must be changed to the *Stable* status. This change has also implications for the set of current stimuli, since an absent stimulus clearly should not be in this set. Since we have four possible such statuses, this update operation is divided in four schemas, namely, *CurrentStimuliUpdateOp*_1, *CurrentStimuliUpdateOp*_2, *CurrentStimuliUpdateOp*_3 and *CurrentStimuliUpdateOp*_4 (see Appendix A).

$$
\begin{aligned}
T_CurrentStimuliUpdateOp \mathrel{\widehat{=}} & \\
& CurrentStimuliUpdateOp_1 \lor \\
& CurrentStimuliUpdateOp_2 \lor \\
& CurrentStimuliUpdateOp_3 \lor \\
& CurrentStimuliUpdateOp_4
\end{aligned}
$$

Finally, it is necessary to record the stimuli present at the current instant. This will be important in order to assess the context that a past action has been performed, and hence determine the most favorable conditions for such an action. This is achieved in *T_PastStimuliUpdateOp* schema (see Appendix A).

4.2.2.6 Integration

At every instant, the organism may both receive new stimulation and process the current stimuli. Hence, concerning stimuli, its main tasks are as follows:

- Apply the *T_ConditioningOp* operation for each new stimulation;
- Apply the *T_UnconditioningOp* operation for each pair of stimuli in the stimulus implication relation;
- Deliver stimulation by means of the *T_StimulationUpdateOp* operation;
- Update the current stimuli using the *T_CurrentStimuliUpdateOp* operation;
- Record the current stimuli for later reference using the *T_PastStimuliUpdateOp* operation.

The schema *Organism_StimulusProcessing* groups all of these tasks together (see Appendix A).

4.2.3 General Responding

Behavioural responses constitute the means through which organisms alter their environments. As such, responses are also the only way we can gain knowledge about organisms. They are, thus, the counterpart of stimuli.

As we have seen, Behaviour Analysis defines two fundamental classes of behaviours, namely, operant behaviour and reflexive behaviour (also known as respondent behaviour). We can, however, abstract common properties of these two classes. For example, both classes require primitive actions to be performed and both require a scheduling mechanism.

It is worth to notice that one may imagine other classes of behaviour besides reflexes and operants. If such classes were defined, they could be integrated in the responding subsystem by providing structures similar to those for reflexes and operants. This constitutes a possible way to improve the architecture, although it would require changes to many schemas (probably refinements would not suffice), and therefore it would not be a straightforward task.

In this section we present these common features, while in the next two we explore each behavioural class in its specificities. Here we see what primitive actions are, what property they have, how behaviours are scheduled, how conflicts among potential behaviours are solved, and finally how behavioural responses are managed.

4.2.3.1 Basic Entities

The Responding Subsystem aggregates all definitions of behavioural classes and also hold the particular behaviour available to the organism. While we specify the details of operant and respondent behaviour in the next two sections, the present definitions use them.

The *RespondingSubsystem* schema imports a number of other schemas, which we will examine shortly. It also defines the sets *operants* and *reflexes*, which contain the behaviours available to the organism.

RespondingSubsystem

$CurrentBehaviors$

$CurrentResponses$

$Actions$

$ActionHistory$

$ActionConflict$

$ActionBaselevel$

$operants : \mathbb{P}\ Operant$

$reflexes : \mathbb{P}\ Reflex$

At every instant, the organism may or may not wish to employ a behaviour. The behaviours which are planned to be performed are defined in the *CurrentBehaviors* schema. Notice that this schema contains, in particular, the *spontaneous* actions set. This accounts for actions that are to be performed independently of any reflex or operant. The introduction of such a set, however, is a mere technicality, for ultimately these spontaneous actions are expected to lead to the formation of operants according to the consequences that they bring. But before any such consequence can be perceived by the organism, there must be a way to specify that certain actions can happen without elicitation (i.e., do not arise from reflexes) even though they are not associated with any consequence (i.e., are not operants). Hence, with respect to the present specification, operants should not be confused with spontaneous actions, although informally one may say that operants are spontaneous (i.e., because they are emitted and not elicited) much like Skinner (1953) himself does.

CurrentBehaviors

$elicited : \mathbb{P}\ Reflex$

$emitted : \mathbb{P}\ Operant$

$spontaneous : \mathbb{P}\ Action$

When a behaviour is actually performed, it generates a *behavioural response*. These are kept by the *CurrentResponses* schema, which defines the current responses and map the pertinent behaviours to them.

$$
\begin{array}{l}
\textit{CurrentResponses} \\
\hline
\textit{responses} : \mathbb{P}\, \textit{Response} \\
\textit{activeResponses} : \mathbb{P}\, \textit{Response} \\
\textit{inactiveResponses} : \mathbb{P}\, \textit{Response} \\
\textit{reflexResponse} : \textit{Reflex} \rightarrowtail \textit{Response} \\
\textit{operantResponse} : \textit{Operant} \rightarrowtail \textit{Response} \\
\textit{spontaneousResponse} : \textit{Action} \rightarrowtail \textit{Response} \\
\textit{reflexElicitationTime} : \textit{Reflex} \rightarrowtail \textit{Instant} \\
\hline
\textit{responses} = \textit{activeResponses} \cup \textit{inactiveResponses} \\
\textit{activeResponses} \cap \textit{inactiveResponses} = \emptyset \\
\operatorname{ran} \textit{reflexResponse} = \textit{responses} \\
\operatorname{ran} \textit{operantResponse} = \textit{responses} \\
\operatorname{ran} \textit{spontaneousResponse} = \textit{responses} \\
\hline
\end{array}
$$

4.2.3.2 Actions

To distinguish behavioural classes (i.e., reflexes and operants) from the actual behavioural responses, we introduce the concepts of *action* and *response*. Actions are what the organism actually does (e.g., the movement of a muscle is an action) and responses is how he does it (e.g., for how long, with what intensity). Actions are primitive concepts, the most fundamental things an organism can do.

$$[\textit{Action}]$$

Each concrete organism will have its own set of actions. But like for stimuli, it suffices for the purpose of this specification to have an abstract set of actions without their particularities.

Actions can either be conflicting or non-conflicting. For instance, if two actions require different movements from an organism's muscle at the same time, then they are conflicting. If, however, the execution of each action is independent, then they are non-conflicting.

$$\textit{Conflict} ::= \textit{conflicting} \mid \textit{nonconflicting}$$

Action conflicts are, of course, particularities of each organism. Therefore, we provide a structure to hold this information.

ActionConflict

$conflict : Action \times Action \rightarrow Conflict$

Though, in principle, an action can be part of any behavioural class, it is also necessary to restrict some of them to specific classes. For instance, pupil movements cannot figure in operant behaviour, since such movements are controlled completely by environmental light variations. On the other hand, some muscular movements can be triggered either by reflex (e.g., when one, reflexively, removes one's hands from a hot surface) or operants (e.g., all "voluntary" movements). At last, there is also the case in which only operant behaviour can be involved, as in speech. Therefore, we must provide definitions to account for two classes of actions.

Actions

$operantActions : \mathbb{P}\, Action$

$reflexActions : \mathbb{P}\, Action$

Notice that an action can belong to both classes at the same time.

As is detailed in Section 4.2.4, organisms can learn how their actions affect their environments. Once they know what to expect from a particular action, they can repeat such an action when the appropriate conditions arise. However, if an action has never being performed, organisms cannot know their consequences. The approach we employ to solve this issue is to assign a probability, called *base level*, of spontaneous occurrence to each possible action.[4] This base level probability is given by the *baseLevel* function of the *ActionBaselevel* schema.

ActionBaselevel

$Actions$

$baseLevel : Action \rightarrow Probability$

$\forall\, a : Action \mid a \in operantActions \bullet baseLevel(a) >_1 min_probability$

$\forall\, a : Action \mid a \in reflexActions \wedge a \notin operantActions \bullet$
$\quad baseLevel(a) =_1 min_probability$

Finally, we provide a record of all performed actions. This will be used when defining operant behaviour later on.

ActionHistory

$actionsHistory : Instant \nrightarrow \mathbb{P}\, Action$

[4] Note that such spontaneous occurrences can be seen as a form of curiosity, because the organism becomes inclined to explore new things for no particular reason other than chance.

4.2.3.3 Behavioural Responses

A response, captured by the *Response* schema, is an actual behavioural instance. It is the structure that actually interacts with the environment, a concrete action. Thus, besides the action to be performed, there must be also a related *duration* (i.e., for how long the action will be performed) and *magnitude* (i.e., the "vigor" of the response). Moreover, a *latency* (i.e., a temporal interval between the response emission and the performance of the corresponding action) is sometimes required.

Response

$action : Action$

$latency : Duration$

$duration : Duration$

$magnitude : Intensity$

4.2.3.4 Response Scheduling Operations

Before responses are actually performed, it is necessary to figure out which operants and reflexes have been triggered. Operant and reflex definitions provide schemas with the conditions for that, namely, *ReflexElicitationCond* and *OperantEmissionCond*. Moreover, response scheduling must also account for the spontaneous occurrences of actions, which are defined through the base level probability associated with each available action.

Operants, reflexes and spontaneous actions that fulfill the conditions are then scheduled to be realized as responses by putting them in the appropriate sets of the *CurrentBehaviors* schema we have seen. This is achieved, respectively, by the *OperantSchedulingOp*, *ReflexSchedulingOp* and *BaseLevelSchedulingOp* schemas (see Appendix A).

4.2.3.5 Conflict Resolution Operations

As we have seen, some actions conflict. Hence, when two such actions are scheduled for execution, a problem arises. To deal with this, a number of conflict resolution operations are defined. Each such operation provides a solution for the conflict between two classes of behaviour, and an associated condition is also defined in order to determine when there is a conflict in the first place.

Let us consider the particular case of an operant conflicting with another operant. The condition for this is given by the *OperantConflictCond* schema.

OperantConflictCond

$ActionConflict$

$o_1, o_2 : Operant$

$conflict(o_1.action, o_2.action) = conflicting$

If two operants conflict, the strategy to be adopted is quite clear: either the one with greater utility will be chosen or, if both have the same utility, the choice is arbitrary. This is specified by the *OperantConflictResolutionOp* schema, where the content of the *removeO* set defines the operants that are to be removed.

OperantConflictResolutionOp

$\Delta CurrentBehaviors$

$StimulationSubsystem$

$OperantUtility$

$ActionConflict$

$removeO : \mathbb{P}\ Operant$

$$\begin{aligned}
&\forall\, o_1, o_2 : emitted \mid OperantConflictCond \bullet \\
&\quad (oUtility(o_1, currentStimuli) >_1 oUtility(o_2, currentStimuli) \Rightarrow \\
&\qquad o_2 \in removeO) \wedge \\
&\quad (oUtility(o_1, currentStimuli) =_1 oUtility(o_2, currentStimuli) \Rightarrow \\
&\qquad (o_1 \in removeO) \vee (o_2 \in removeO))
\end{aligned}$$

Similarly, the following kinds of conflicts may take place:

- A reflex may conflict with another reflex. The condition for this is given in *ReflexConflictCond* and its solution in *ReflexConflictResolutionOp*. There are, however, many ways in which such a conflict could be solved, and thus this schema is supposed to be refined. We provide two possible refinements in *ReflexConflictResolutionOp_Ref*1 and *ReflexConflictResolutionOp_Ref*2 schemas, which use different attributes of the reflexes in order to determine which should have priority. (See Appendix A).
- An operant may conflict with a reflex. The condition for this is given by *OperantReflexConflicCond* and the solution is given by *OperantReflexConflictResolutionOp* (see Appendix A).
- A spontaneous action may conflict with another such action, or a reflex or an operant. The conditions and the solutions for this are all

given in the *BaseLevelConflictResolutionOp* schema (see Appendix A). We assume here that such spontaneous occurrences are as relevant as actions governed by operants. The reason is that both have to do with understanding the environment, since the former allows the organism to explore it, while the latter allows the organism to exploit information thus gained. Reflexes, however, always have priority over spontaneous actions.

An auxiliary schema, *AuxConflictResolutionOp*, is also provided to define how all of these conflict resolution operations work together to change *CurrentBehaviors* (see Appendix A).

We can then combine the conflict resolution operations.

$$
\begin{aligned}
&ConflictResolutionOp \mathrel{\widehat{=}} \\
&\qquad OperantConflictResolutionOp \wedge \\
&\qquad ReflexConflictResolutionOp \wedge \\
&\qquad OperantReflexConflictResolutionOp \wedge \\
&\qquad BaseLevelConflictResolutionOp \wedge \\
&\qquad AuxConflictResolutionOp
\end{aligned}
$$

If we use refinements for some of these operations, we also need to redefine this composed operation. Since we provide two possible refinements for the conflict resolution of reflexes, we also provide the corresponding composed operations *ConflictResolutionOp_Ref*1 and *ConflictResolutionOp_Ref*2 (see Appendix A).

4.2.3.6 Response Emission, Update and Termination Operations

Once behaviours have been selected, it is necessary to transform them into actual responses. Responses, in turn, are not instantaneous, they have duration. Thus, they must be updated for some time, until termination conditions are reached and they cease.

The *OperantEmissionOp* operation below defines how operants turn into responses. It states that there must not already be a response associated with the operant, and then defines that such a response must be created. It also records when the operant's action has been performed for future reference.

$$
\begin{array}{l}
\textit{OperantEmissionOp} \\
\hline
\Delta \mathit{ActionHistory} \\
\Delta \mathit{CurrentResponses} \\
\mathit{currentInstant?} : \mathit{Instant} \\
o? : \mathit{Operant} \\
\hline
\neg\, (\exists\, rp : \mathit{Response} \bullet \mathit{operantResponse}(o?) = rp) \\
\exists\, rp : \mathit{Response} \bullet \\
\quad rp.\mathit{action} = o?.\mathit{action} \wedge \\
\quad \mathit{inactiveResponses}' = \mathit{inactiveResponses} \cup \{rp\} \wedge \\
\quad \mathit{operantResponse}' = \mathit{operantResponse} \oplus \{o? \mapsto rp\} \\
\mathit{actionsHistory}'(\mathit{currentInstant?}) = \\
\quad \mathit{actionsHistory}(\mathit{currentInstant?}) \cup \{o?.\mathit{action}\}
\end{array}
$$

The elicitation of reflexes and the emission of spontaneous actions follow similar principles and are captured in *ReflexElicitationOp* and *BaseLevelEmissionOp* schemas, respectively (see Appendix A).

Notice that response emission operations do not constrain some parameters of the responses. This reflects the fact that there is no universally accepted computational theory capable of calculating the exact quantitative properties of operant emission (McDowell, 2004). Hence, our architecture does not enforce any particular view. In fact, one may extend it by providing suitable refinements to perform the initialization of these unconstrained variables according to specific theories.

Once responses are being performed, they must be updated over time. There are some cases to consider:

- A response might be inactive owing to its assigned latency. That is to say, the response is going to be performed, but only after its specified latency. In this circumstance, we just decrease the latency, to account for the fact that an instant has passed. This is done by the *InactiveResponseUpdateOp_1* schema, which we show below.
- Once the latency reaches zero, the response can be activated. This is accomplished in a similar fashion by the *InactiveResponseUpdateOp_2* operation (see Appendix A).
- If the response is active, its duration must be decreased, in order to account for the fact that an instant has passed. This is done by the *ActiveResponseUpdateOp* operation (see Appendix A).

- If none of the above cases hold, then nothing changes, as specified by *NeutralResponseUpdateOp* (see Appendix A).

$$
\begin{array}{|l}
\hline
InactiveResponseUpdateOp_1 \\
\Delta CurrentResponses \\
\Delta Response \\
\hline
\theta Response \in inactiveResponses \\
\theta Response \notin activeResponses \\
latency > 0 \\
latency' = latency - 1 \\
activeResponses' = activeResponses \\
inactiveResponses' = (inactiveResponses \setminus \{\theta Response\}) \cup \{\theta Response'\} \\
\hline
\end{array}
$$

Thus, we reach the following total operation for response update.

$$
\begin{aligned}
&T_ResponseUpdateOp \mathrel{\widehat{=}} \\
&\qquad InactiveResponseUpdateOp_1 \lor \\
&\qquad InactiveResponseUpdateOp_2 \lor \\
&\qquad ActiveResponseUpdateOp \lor \\
&\qquad NeutralResponseUpdateOp
\end{aligned}
$$

At last, we must consider how to terminate responses. There are three cases to consider:

- If the response's duration has reached zero, it means that it should cease. This is specified in the *ResponseTerminationOp*_1 operation below.

- If the behaviour that justified the response is no longer selected for execution, then the response must cease as well. This situation might arise during conflict resolutions, when a more important behaviour may take the place of another. This is specified by the *ResponseTerminationOp*_2 operation (see Appendix A).

- Finally, if none of these conditions hold, nothing changes, as specified by
  *ResponseTerminationOp*_3 (see Appendix A).

$$
\begin{array}{l}
\textit{ResponseTerminationOp_1} \\
\hline
\Delta\, \textit{CurrentResponses} \\
\Delta\, \textit{CurrentBehaviors} \\
rp? : \textit{Response} \\
\hline
rp? \in \textit{activeResponses} \\
rp?.\textit{duration} \leq 0 \\
\textit{activeResponses}' = \textit{activeResponses} \setminus \{rp?\} \\
\textit{inactiveResponses}' = \textit{inactiveResponses} \setminus \{rp?\} \\
\forall\, o : \textit{emitted} \bullet \textit{operantResponse}(o) = rp? \Rightarrow o \notin \textit{emitted}' \\
\forall\, r : \textit{elicited} \bullet \textit{reflexResponse}(r) = rp? \Rightarrow r \notin \textit{elicited}' \\
\forall\, a : \textit{spontaneous} \bullet \textit{spontaneousResponse}(a) = rp? \Rightarrow a \notin \textit{spontaneous}' \\
\hline
\end{array}
$$

Then, the total operation is as follows.

$$
\begin{array}{l}
\textit{T_ResponseTerminationOp} \mathrel{\widehat{=}} \\
\qquad \textit{ResponseTerminationOp_1} \vee \\
\qquad \textit{ResponseTerminationOp_2} \vee \\
\qquad \textit{ResponseTerminationOp_3}
\end{array}
$$

4.2.3.7 Integration

As the previous operations suggest, responding is constituted by four distinct stages, and for each one we provide an operation to allow its integration in the organism:

- First, operants, reflexes and spontaneous actions are selected. This is accomplished by the *Organism_BehaviorSelection* operation (see Appendix A).
- Then, possible conflicts are solved. This is done by the *Organism_ConflicResolution* operation (see Appendix A).
- Next, responses are generated, according to the *Organism_ResponseEmission* operation (see Appendix A).
- Finally, responses are updated, until they reach a termination condition. This is achieved by the *Organism_ResponseMaintenance* operation (see Appendix A).

4.2.4 Operant Behaviour

Organisms seek pleasure and avoid pain in an ever changing world. The consequences of their actions change constantly, in such a way that what used to be an applicable behaviour may no longer be appropriate, and useless actions may become interesting. Learning is, therefore, a necessary virtue. In Behaviour Analysis, operant behaviour is the kind of behaviour that accounts for this.

Because the consequences that organisms seek are always reinforcing, operants are also known as *contingencies of reinforcement*. However, by no means reinforcing stimuli are the only consequences that matter. It is equally important to know actions that lead to aversive stimuli in order to avoid them.

In this section we present operants, the relations that might be established among them, and the several operations that may be performed on them.

4.2.4.1 Basic Entities

An operant records the manner through which a specific stimulus may be reached. That is, how to *operate* in the environment in order to obtain some consequence. Unlike reflexes, operants are not predefined and static entities: they might be created, modified and destroyed. Each of these possibilities is given by a different procedure.

The *Operant* schema is a structure that links an *action* to a *consequence* (i.e., the *consequent stimulus*). This link models the belief that the action, when performed, causes the stimulus. Such a belief, however, varies in strength. And within the same operant, this strength varies, through the *consequenceContingency* function, depending on the stimuli present on the environment (i.e., the *antecedent stimuli*). Figure 4.4 depicts this tripartite structure.

Operant

$StimulusUtility$

$antecedents : \mathbb{P}(\mathbb{P}\, Stimulus)$

$action : Action$

$consequence : Stimulus$

$consequenceContingency : (\mathbb{P}\, Stimulus) \nrightarrow Correlation$

$sUtility(consequence) \neq neutral_utility$

$\mathrm{dom}\, consequenceContingency = antecedents$

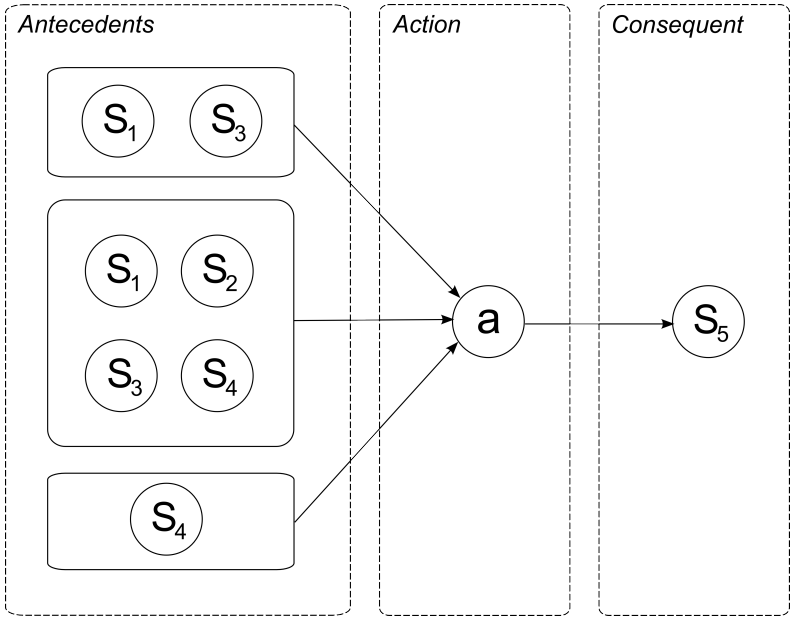


Figure 4.4: An operant is composed by possible antecedent stimuli, an action and a consequent stimulus. In this example, there are three sets of antecedent stimuli. Each set models a context that the organism encountered previously.

In Behaviour Analysis, an operant class of behaviours is defined by some shared consequence. To facilitate computation, though, our *Operant* schema associates only one action with a stimulus consequence. Two instances of this schema, then, could have the same consequent stimulus, but different actions. This allows the independent calculation concerning each particular action. For instance, it could be that an action has the same consequence that another at some moment, but this may not last forever. Hence, there must be a way to learn them separately.[5]

[5]The triple of antecedent stimuli, behaviour, and consequent stimulus is sometimes called a *three-term contingency* (e.g., by Catania, 1998), thus defining the second term as the *operant*. In this specification, however, we call this whole triple an *operant*. For our purposes, this formalizes the notion more correctly and succinctly, since: (i) the essence of operant behaviour is in the consequences of actions (i.e., a mere action with no effect on the organism's environment should not be considered an operant); and (ii) once we incorporate a consequence in the definition of what an operant is, there is no need to introduce another entity to model the contingency, and it is simpler to incorporate the antecedent stimuli (if any) in the definition of the operant as well. This gives us a single entity, the *Operant* schema, that captures the intuitive notion of what an operant is and can be easily used to compute related behavioural phenomena.

4.2.4.2 Operant Implication

A single operant holds information about how to obtain a particular stimulus. In order to link operants to several consequent stimuli, it is necessary to establish how operants relate to each other. We do this with the *operant implication oCauses* relation below. It specifies that an operant either directly causes a stimulus or sets the conditions for another operant to be executed, which, in turn, lead to other stimuli. The *oCorrelation* function, in turn, provides the correlation between an operant and its direct and indirect stimulus consequences.

OperantImplication

$$
\begin{array}{l}
StimulusImplication \\
Discrimination \\
oCauses : \mathbb{P}(Operant \times Stimulus) \\
oCorrelation : Operant \times Stimulus \nrightarrow Correlation \\
\hline
\forall\, o : Operant \bullet o \ \underline{oCauses}\ o.consequence \\
\forall\, o_1, o_2 : Operant;\ S : \mathbb{P}\, Stimulus \mid S\ \underline{discriminatesNonEmpty}\ o_2 \bullet \\
\quad (\forall\, s : S \bullet o_1.consequence\ \underline{sCauses}\ s) \Rightarrow \\
\qquad o_1\ \underline{oCauses}\ o_2.consequence \\
\mathrm{dom}\, oCorrelation = oCauses
\end{array}
$$

The crucial element that links different operants in the above schema is the discrimination relation given in schema *Discrimination* (see Appendix A). It establishes that a set of stimuli *discriminates* an operant if it is present among the operant's antecedents.

4.2.4.3 Operant Utility

Operants have utilities, since they lead to stimuli. That is to say, operants gain their usefulness owing to the stimuli they allow an organism to reach.

As it happens with stimulus utility, there are several ways to define operant utility. Thus, the *OperantUtility* schema below, merely defines that an utility function *oUtility* exists. This function takes an operant and a set of stimuli in order attribute an utility to the operant. This set of stimuli, as it is explained below, is actually the current stimuli that the organism is subject to.

$$
\begin{array}{|l}
\hline
\textit{OperantUtility} \\
\textit{StimulusUtility} \\
\\
\textit{OperantImplication} \\
\\
\textit{oUtility} : (\textit{Operant} \times \mathbb{P}\,\textit{Stimulus}) \rightarrow \textit{Utility} \\
\hline
\end{array}
$$

Refinements can now be provided. We give one such refinement, the *OperantUtility_Ref*1 schema (see Appendix A), which defines that the utility of an operant is calculated as:

- a neutral or positive utility given by the maximum stimulus utility that the operant can reach through the operant implication relation, provided that there are no reachable stimuli with negative utility;
- a negative utility given by the minimum stimulus utility that the operant can reach, if indeed there is at least one stimulus with negative utility;
- the neutral utility, if none of the previous cases hold.

In other words, the utility of an operant o is defined by considering which sequence of operants starting in o leads to the best stimulus, provided that no harmful stimuli can be reached in the same way. It is, therefore, a way of performing planning using the fact that one operant may set the necessary antecedents of another. However, it is a planning subject to constant re-evaluation, since the operants in which it depends may change at any instant.

If one has more knowledge about the organism being modeled, or if one merely wishes to experiment with different search strategies, one may define different refinements for this utility. In particular, it is interesting to note that our own refinement is an idealized one: the organism seeks the best solution. In practice, though, this is possibly an inefficient strategy. Hence, one may wonder what kind of approximations could be employed in order to improve this.

4.2.4.4 Fundamental Operations

Operants, being flexible units of learning, are subject to many operations. Most of these operations, however, share some characteristics, to be found in the *OperantFormationOp* or *OperantOp* schemas (see Appendix A). The former takes care of creating new operants, while the later accounts for operants that already exists.

Operant formation arises continuously. Every action that is followed by a stimulus presentation can, under certain timing restrictions, give birth to an operant, which records the contingency between the action and the stimulus.

At first, however, these are very weak contingencies. But an organism's operants are constantly being modified as well, and this will ensure that recently created operants evolve appropriately. If the recently detected contingencies never arise again, the organism interprets them as accidents and not as laws to be learned. On the other hand, if the contingencies keep coming up, the associated operants increase in strength.

When the correlations between an action and a consequent stimulus in an operant become too low, the operant loses its usefulness. Hence, it must be eliminated. This is accomplished by the *OperantEliminationOp_1* operation. A neutral complement to this is also provided in *OperantEliminationOp_2*. (See Appendix A).

It is then possible to form the total operation *TOperantEliminationOp*.

$$
\begin{aligned}
&T_OperantEliminationOp \mathrel{\widehat{=}} \\
&\qquad OperantEliminationOp_1 \lor \\
&\qquad OperantEliminationOp_2
\end{aligned}
$$

OperantOp can be further refined into four operations:

- *Discrimination.* Operant discrimination happens when a new set of antecedent stimuli is learned. That is, when a new environmental condition regarding an operant is found, a discrimination process takes place in order to incorporate this new knowledge. This process is formalized by the *DiscriminationOp* schema below.

- *Operant conditioning.* Operant conditioning takes place when a known environmental condition is met and the operant's stimulus consequence is reached. In this case, the contingency that links the action and the stimulus is strengthened. In other words, when the operant successfully leads to a stimulus, it becomes stronger. This is specified in the *OperantConditioningOp* schema (see Appendix A).

- *Operant extinction.* If some known environmental condition is found but the stimulus consequence is not reached, the relation of contingency is weakened, as specified in the *ExtinctionOp* schema (see Appendix A). This extinction operation also gives rise to an emotion called *frustration*. A frustrated organism becomes more likely to explore new possibilities, since his knowledge of the world has been proven to be incomplete. We examine this emotion in Section 4.2.7.

- As a technical matter, a neutral operation is also provided in schema *NeutralOp*, so that the fact that sometimes no change takes place becomes explicit (see Appendix A).

$$
\begin{array}{|l}
\hline
\textit{DiscriminationOp} \\
\textit{OperantOp} \\
\hline
\textit{discriminativeStimuli}? \notin \mathrm{dom}\ \textit{consequenceContingency} \\
\textit{consequence}?\ \underline{\textit{sCauses}}\ \textit{consequence} \\
\textit{consequence}? \notin \textit{discriminativeStimuli}? \\
\mathrm{dom}\ \textit{consequenceContingency}' = \\
\qquad \mathrm{dom}\ \textit{consequenceContingency} \cup \{\textit{discriminativeStimuli}?\} \\
\textit{consequenceContingency}'(\textit{discriminativeStimuli}?) >_1 \textit{min_correlation} \\
\hline
\end{array}
$$

At this point it should already be clear that operant behaviour endows an agent with learning capabilities. Similarly to the process of stimulus conditioning we saw in Section 4.2.2.4, the above operations allow the organism to modify its representation of how its environment work. Here, however, instead of relating stimuli to other stimuli, one is concerned with how actions relate to stimuli. This relation is subject to change based on how the environment responds to the organism's actions, and thus constitutes a way of learning about such an environment.

The above four operations are grouped as the *FundamentalOperantOp*.

$$
\begin{array}{l}
\textit{FundamentalOperantOp} \mathrel{\widehat{=}} \\
\qquad \textit{DiscriminationOp} \lor \\
\qquad \textit{OperantConditioningOp} \lor \\
\qquad \textit{ExtinctionOp} \lor \\
\qquad \textit{NeutralOp}
\end{array}
$$

As the name suggests, this combined operation will serve as the basis of more detailed operations. This basis accounts for the learning that takes place, but the behavioural modifications that happen go beyond this. For a complete mechanism, it is necessary to qualify the experience as a reinforcement or a punishment, because each case may bring different consequences, such as different emotional responses.

4.2.4.5 Reinforcement and Punishment Operations

Reinforcement and punishment play a large role in Behaviour Analysis, for they are the main behavioural modification mechanisms. It is therefore worth

to define such operations separately and in detail.

An operant is reinforced if the received stimulation is pleasant and associated with the operant's consequence. The purpose of reinforcement is to strengthen the relation between an action and a pleasant consequence.

Positive reinforcement accounts for the particular case in which pleasure comes from the provision of a pleasant stimulus.

$$
\begin{array}{|l}
\hline
\textit{PositiveReinforcement} \\
\textit{StimulusUtility} \\
\textit{consequence?} : \textit{Stimulus} \\
\hline
\textit{sUtility}(\textit{consequence?}) >_1 \textit{neutral_utility} \\
\textit{stimulus_status}(\textit{consequence?}) = \textit{Beginning} \\
\hline
\end{array}
$$

Notice that the above schema is not an operation. It merely states a condition for positive reinforcement. Recall that we have two types of operations concerning operants. The first deals with existing operants, and the other accounts for new operants. Hence, in order to turn positive reinforcement into actual operations, we group the condition above with both *FundamentalOperantOp* and *OperantFormationOp*.

$$
\begin{aligned}
\textit{PositiveReinforcementOp_1} \mathrel{\widehat{=}} & \\
& \textit{FundamentalOperantOp} \wedge \\
& \textit{PositiveReinforcement} \\
\textit{PositiveReinforcementOp_2} \mathrel{\widehat{=}} & \\
& \textit{OperantFormationOp} \wedge \\
& \textit{PositiveReinforcement}
\end{aligned}
$$

Reinforcement has a complementary negative form. Negative reinforcement takes place when pleasure arises from the removal of a painful stimulus, instead of the provision of a pleasant one.

$$
\begin{array}{|l}
\hline
\textit{NegativeReinforcement} \\
\textit{StimulusUtility} \\
\textit{consequence?} : \textit{Stimulus} \\
\hline
\textit{sUtility}(\textit{consequence?}) <_1 \textit{neutral_utility} \\
\textit{stimulus_status}(\textit{consequence?}) = \textit{Ending} \\
\hline
\end{array}
$$

Again, we provide two operations to account for negative reinforcement.

$$NegativeReinforcementOp_1 \mathrel{\widehat{=}} FundamentalOperantOp \wedge NegativeReinforcement$$

$$NegativeReinforcementOp_2 \mathrel{\widehat{=}} OperantFormationOp \wedge NegativeReinforcement$$

Let us now define punishment operations. An operant is punished when the received stimulation is undesirable. It teaches the organism that an action, which was previously neutral or beneficial, is becoming harmful.

Positive punishment accounts for the particular case in which pain comes from the provision of a painful stimulus.

$$\begin{array}{|l}
\hline
PositivePunishment \\
StimulusUtility \\
consequence? : Stimulus \\
\hline
sUtility(consequence?) <_1 neutral_utility \\
stimulus_status(consequence?) = Beginning \\
\hline
\end{array}$$

And the related operations are as follows.

$$PositivePunishmentOp_1 \mathrel{\widehat{=}} FundamentalOperantOp \wedge PositivePunishment \wedge StartAngerOp$$

$$PositivePunishmentOp_2 \mathrel{\widehat{=}} OperantFormationOp \wedge PositivePunishment \wedge StartAngerOp$$

Notice that besides the positive punishment condition, we also add *StartAngerOp*, which specifies that the *anger* emotion will be generated. Anger takes place as a mechanism of defense, putting the organism in an aggressive state. We explain what this means in Section 4.2.7.

Negative punishment, in turn, takes place when pain arises from the removal of a pleasant stimulus.

$$
\begin{array}{|l}
\hline
\textit{NegativePunishment} \\
\textit{StimulusUtility} \\
\textit{consequence?} : \textit{Stimulus} \\
\hline
\textit{sUtility}(\textit{consequence?}) >_1 \textit{neutral_utility} \\
\textit{stimulus_status}(\textit{consequence?}) = \textit{Ending} \\
\hline
\end{array}
$$

The related operations are below. Notice that, again, there is an emotional response associated. The organism becomes depressive if pleasant stimuli are removed. This will be explained in Section 4.2.7.

$$
\begin{aligned}
&\textit{NegativePunishmentOp_1} \mathrel{\widehat{=}} \\
&\qquad \textit{FundamentalOperantOp} \land \\
&\qquad \textit{NegativePunishment} \land \\
&\qquad \textit{StartDepressionOp} \\
&\textit{NegativePunishmentOp_2} \mathrel{\widehat{=}} \\
&\qquad \textit{OperantFormationOp} \land \\
&\qquad \textit{NegativePunishment} \land \\
&\qquad \textit{StartDepressionOp}
\end{aligned}
$$

In a number of occasions, neither reinforcement nor punishment takes place. The conditions for this are formalized in *NeutralReinforcementOp*_1 and *NeutralReinforcementOp*_2 (see Appendix A).

At last, we combine all previous operations in order to account for all possible cases of stimulus influence on an operant.

$$
\begin{aligned}
&\textit{T_OperantOp} \mathrel{\widehat{=}} \\
&\qquad \textit{PositiveReinforcementOp_1} \lor \textit{NegativeReinforcementOp_1} \lor \\
&\qquad \textit{PositivePunishmentOp_1} \lor \textit{NegativePunishmentOp_1} \lor \\
&\qquad \textit{NeutralReinforcementOp_1} \\
&\textit{T_OperantFormationOp} \mathrel{\widehat{=}} \\
&\qquad \textit{PositiveReinforcementOp_2} \lor \textit{NegativeReinforcementOp_2} \lor \\
&\qquad \textit{PositivePunishmentOp_2} \lor \textit{NegativePunishmentOp_2} \lor \\
&\qquad \textit{NeutralReinforcementOp_2}
\end{aligned}
$$

4.2.4.6 Emission Condition

An operant is to be emitted if it is relevant in the current state of affairs, if its utility is more than neutral and if other operants associated to the same action also have such an utility. The reason for this latter restriction is that

operants may have the same action, which at different times resulted in different consequences. Hence, it is necessary to make sure that an operant's action is not considered harmful in the context of another operant.

$$
\begin{array}{|l}
\hline
\textit{OperantEmissionCond} \\
\textit{RespondingSubsystem} \\
\textit{StimulationSubsystem} \\
\textit{OperantUtility} \\
\textit{Discrimination} \\
o : \textit{Operant} \\
\hline
\textit{currentStimuli}\ \underline{\textit{discriminates}}\ o \\
\textit{oUtility}(o, \textit{currentStimuli}) >_1 \textit{neutral_utility} \\
\forall\, x : \textit{operants} \mid \\
\quad x \neq o \wedge \textit{currentStimuli}\ \underline{\textit{discriminates}}\ x \wedge x.\textit{action} = o.\textit{action} \bullet \\
\qquad \textit{oUtility}(x, \textit{currentStimuli}) \geq_1 \textit{neutral_utility} \\
\hline
\end{array}
$$

4.2.4.7 Integration

Three operations are necessary in order to integrate operants to the organism:

- It is necessary to apply the *T_OperantFormationOp* operation considering the actions that have taken place in the recent past (defined by the *max_delay* constant). This ensures that an action that was performed previously has a chance of becoming an operant. This procedure is given in the *Organism_OperantFormationOp* schema (see Appendix A).
- Similarly, it is necessary to apply the *T_OperantOp* operation considering these same actions that took place in the recent past, so that the corresponding operants (if any) may me modified appropriately. This is achieved by the *Organism_OperantOp* schema (see Appendix A).
- Finally, it is necessary to apply *T_OperantEliminationOp* to each operant available to the organism in order to eliminate the useless ones (see Appendix A).

4.2.5 Respondent Behaviour

Respondent behaviour (also known as reflexes or reflexive behaviour) is the simplest kind of behaviour that an organism possess. A reflex is, essentially, a

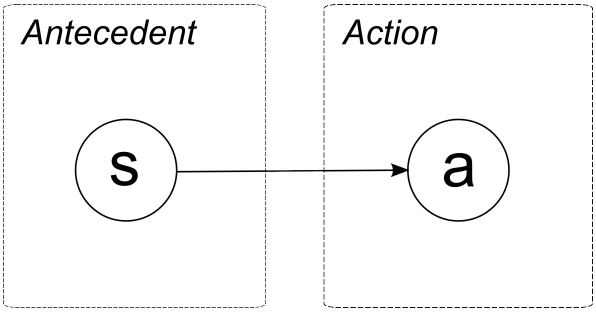


Figure 4.5: A reflex is composed by an antecedent stimulus and an action.

reliable causal relation between a stimulus and an action (see Figure 4.5). That is, the presence of the stimulus triggers, with high probability, the emission of the action.

While reflexes may adapt to account for, say, excessive stimulation, they are not learning structures. Organisms are born with predefined reflexes, which remain the same throughout their lives.

In this section we will see what constitutes a reflex, how it can be adjusted over time, and how it may be triggered.

4.2.5.1 Basic Entities

The *Reflex* schema defines a reflex as an *antecedent* stimulus which causes an *action* to be performed. The remaining variables account for the several properties of this causal relation:

- *threshold* defines the minimum intensity of the stimulation that causes the reflex to be triggered.
- *elicitation* defines the probability of the action to be actually performed after the stimulation threshold has been reached.
- *magnitude* specifies the intensity of the behavioural response when the action is performed.
- *duration* determines for how long the action will be performed.
- *latency* determines a duration prior to the action performance.

All of these variables change their values as time goes by, but they are always kept within lower and upper bounds.

Reflex

$Actions$

$ReflexParameters$

$antecedent : Stimulus$

$action : Action$

$threshold : Intensity$

$elicitation : Probability$

$magnitude : Intensity$

$duration : Duration$

$latency : Duration$

$action \in reflexActions$

$min_elicitation \leq_1 elicitation \leq_1 max_elicitation$

$min_magnitude \leq_1 magnitude \leq_1 max_magnitude$

$min_duration \leq duration \leq max_duration$

$min_latency \leq latency \leq max_latency$

$min_threshold \leq_1 threshold \leq_1 max_threshold$

These bounds, as well as the functions to modify the related variables, are given as parameters in the *ReflexParameters* schema (see Appendix A). This schema defines a number of functions, but do not specify their form. The reason is that each reflex may be adjusted differently (e.g., in the case of an animal model, because the underlying organs to realize the reflexes have different properties). Hence, one may experiment with many different functions (e.g., linear, exponential).

4.2.5.2 Operations

Although reflexes are innate to the organism, and therefore cannot be neither learned nor unlearned, it is still possible to modify them. This is useful, for instance, in order to account for the fact that a reflexive response uses resources, and thus successive responses may have different properties (e.g., the magnitude of the response may get increasingly weaker). The *ReflexAdjustmentOp* operation provides a way to adjust a given reflex according to the difference between the instant of the current reflex elicitation and the instant in which the reflex was used for the last time. That is to say, according to the time

that the organism did not employ that reflex (see Appendix A). To do so, this operation uses the functions found on the *ReflexParameter* schema of the reflex being adjusted, which assign a new value for the variables according to the elapsed time and their previous value.

4.2.5.3 Elicitation Condition

When a reflex is used, we say that it has been *elicited*. The *ReflexElicitationCond* schema gives the conditions for reflex elicitation. One of these conditions is that the intensity of the stimulation must be greater than or equal to the *threshold* parameter of the reflex. The other condition, which is somewhat more subtle, is that the stimulus that triggers the reflex must be related to the *antecedent* stimulus of the reflex by the stimulus implication relation. That is to say, any stimulus that the organism believes to cause the *antecedent*, including *antecedent* itself, may elicit the reflex.

ReflexElicitationCond

$StimulusImplication$

$r : Reflex$

$s : Stimulus$

$i : Intensity$

$s \ \underline{sCauses} \ (r.antecedent)$

$(r.threshold) \leq_1 i$

4.2.6 Drives

In order to stay alive, organisms constantly consume environmental resources. For instance, water, food, air, and so on. Clearly, the utility of these resources must vary over time. An animal that has just drank a lot of water most likely will not be thirsty. On the other hand, an animal that has not drank anything for a day or two will do anything for water.

The mechanisms that control these variations are called *drives*. A drive can be thought as an appetite for a particular stimulus. The longer one stays without this stimulus, the stronger the appetite for it will be. Conversely, the more one has of the stimulus, the less one will want it.

In this section we see what constitutes a drive, how it relates with the environmental stimuli, and how it affects the organism's behaviour.

4.2.6.1 Basic Entities

An organism has a set *activeDrives* of drives which form its Drive Subsystem. While individual drives will suffer alterations, none is ever added nor removed from the set. Each organism has a predefined set of drives that accompany him during his existence.

DriveSubsystem

$activeDrives : \mathbb{P}\, Drive$

Drives are defined by the schema *Drive*. A drive aims at motivating the organism to find the stimuli contained in its *desires* set. The intensity of this motivation is given by the drive's *importance*, which is a utility that vary over time within a minimum and a maximum value. The *deprivation* function modifies this importance when the organism is deprived from obtaining the stimuli in *desires*. Conversely, the *satiation* function modifies this importance when the organisms manages to reach such stimuli.

Drive

$importance : Utility$

$desires : \mathbb{P}\, Stimulus$

$deprivation : Utility \rightarrow Utility$

$satiation : Utility \rightarrow Utility$

$maxImportance, minImportance : Utility$

$importance \geq_1 minImportance$

$importance \leq_1 maxImportance$

$\forall\, u : Utility \bullet deprivation(u) \geq_1 u$

$\forall\, u : Utility \bullet satiation(u) \leq_1 u$

Drives depend strongly on the functions used to calculate the rate of satiation and deprivation. Therefore, one should consider which function is more suitable for each drive (e.g., linear, exponential). Our only restriction is the monotonicity requirement. Here, then, is another **extension point** of the model.

4.2.6.2 Stimulus Regulation

Drives work by modifying the way that the organisms perceive the utility of stimuli. The mechanism to do so is given by the *StimulusDriveRegulator*

schema, where the function *driveRegulator* is defined (see Appendix A). This function is used by the Stimulation Subsystem in order modify the organism's utility function. To do so, the function takes a stimulus and a initial utility as arguments. Then, it adds the influence of each drive to this initial utility. The specified stimulus is used to discover whether the drive is applicable (i.e., whether it is in its *desires* set). The resulting utility is then returned to the Stimulation Subsystem, which is then used as the utility to be currently attributed to the specified stimulus stimulus.

4.2.6.3 Operations

The *DriveOp* schema abstracts the common properties of satiation and deprivation operations (see Appendix A). It merely states that the stimuli to be considered must have a *Stable* status.

A drive's importance can be modified through operations of satiation and deprivation. Satiation happens when the organism is given a desired stimuli set.

$$
\begin{array}{|l}
\hline
\textit{SatiationOp} \\
\textit{DriveOp} \\
\hline
\textit{desires} \subseteq \textit{present} \\
\textit{importance}' = \textit{satiation}(\textit{importance}) \\
\hline
\end{array}
$$

On the other hand, deprivation takes place when the given stimulus is not desired.

$$
\begin{array}{|l}
\hline
\textit{DeprivationOp} \\
\textit{DriveOp} \\
\hline
\neg\ (\textit{desires} \subseteq \textit{present}) \\
\textit{importance}' = \textit{deprivation}(\textit{importance}) \\
\hline
\end{array}
$$

These two operations can then be combined into a total one.

$$T_DriveOp \mathrel{\widehat{=}} SatiationOp \lor DeprivationOp$$

4.2.6.4 Integration

At every instant, the organism is given a new set of stimulations. Then, for each drive, the *Organism_DrivesUpdate* operation (see Appendix A) applies

the *T_DriveOp* operation, which allows the drive to assess the current stimulations. That is, it makes sure every drive gets an opportunity to be satiated or deprived.

4.2.7 Emotions

Emotions are usually thought of as subjective and private events. Still, one can usually guess what a person is feeling by watching her behaviour. Aggressiveness, for instance, usually indicates a state of anger.

From a behaviourist point of view, though, private events are only relevant to the extent that they produce observable behaviour. So aggressiveness is not just a consequence of anger in a behaviourist theory; rather, it is taken to be anger itself.

In the present work, an emotion is defined as a temporary modification in operant behaviour that is not explained by the organism's drives. The purpose of emotions is to fine tune the organism's behaviour to match the needs of a given situation. "Pure" operant behaviour would only record the relations among actions and stimuli. However, the fact that sometimes actions must be, for example, specially vigorous (e.g., when fighting an opponent), would not be captured. "Pure" stimuli conditioning would be incapable of modifying the utility of primary reinforcers. And that might be exactly what is required sometimes, in order to explain certain kinds of behaviour (e.g., depression). Similarly to what we have seen for drives, there are clever ways to insert such fine tunning in the framework we have developed so far.

However, differently from what we did for drives, emotions are not defined in a very general manner. They encompass any behavioural modification, and therefore we cannot provide a single mechanism to account for all possible emotions. Hence, in our framework, an emotion must be defined mostly individually, although some general properties are established. For this reason, we have provided only three emotions, chosen mostly because they serve as good examples of what it means to formalize an emotion using our framework. Clearly, then, it would be possible to improve this subsystem by the addition of other emotions. To do so, it would suffice to create similar schemas to the ones we provide. This is another important **extension point** of our model.

4.2.7.1 Basic Entities

The Emotional Subsystem is given by the *EmotionSubsystem* schema, which holds information regarding the organism's current emotional state.

EmotionSubsystem

$anger : Anger$

$depression : Depression$

$frustration : Frustration$

Each emotion can be either active or inactive. This will determine whether the emotion influences or not the organism's behaviour.

$$EmotionStatus ::= Active \mid Inactive$$

The *Emotion* schema defines the general properties that every emotion must have. However, in itself this schema is not an emotion.

Emotion

$status : EmotionStatus$

$intensity : Intensity$

$duration : Duration$

The effect of an emotion is specified by how it affects stimuli processing and behavioural response emission. For each of these regulation mechanisms, we provide an appropriate regulation function, within the *UtilityRegulatorEmotion* and
ProbabilityRegulatorEmotion schemas.

UtilityRegulatorEmotion

$Emotion$

$utilityChange : Intensity \rightarrow Utility$

ProbabilityRegulatorEmotion

$Emotion$

$probabilityChange : Intensity \rightarrow Probability$

We can now define the three emotions we provide, *Anger*, *Depression* and *Frustration*.

Anger

$UtilityRegulatorEmotion$

Depression

UtilityRegulatorEmotion

Frustration

ProbabilityRegulatorEmotion

4.2.7.2 Stimulus Regulation

Much like drives, some emotions exert their influence by modifying how the organism perceives the utility of stimuli. This regulation mechanism is defined by the
StimulusEmotionalRegulator schema, which provides the *emotionalRegulator* function (see Appendix A).

This regulation mechanism depends on two emotions, namely, depression and anger. Thus, for each one, a regulator is provided in the *DepressionRegulator*_1 and *AngerRegulator*_1 schemas. When the emotions are inactive, their effect is neutral, as specified in *DepressionRegulator*_2 and *AngerRegulator*_2 schemas. (See Appendix A).

These two possibilities for depression and anger are put together in the *DepressionRegulator* and *AngerRegulator* schemas, respectively.

$$DepressionRegulator \mathrel{\widehat{=}} DepressionRegulator_1 \lor DepressionRegulator_2$$

$$AngerRegulator \mathrel{\widehat{=}} AngerRegulator_1 \lor AngerRegulator_2$$

Depression regulation causes any stimulus utility to be reduced by the intensity of the depression. This implies that the organism will behave less, because stimuli that used to be desirable become either less desirable or even aversive.

Anger regulation, in turn, increases the utility of stimuli which indicates that harm has been caused to either the environment or another agent. The stimuli that indicate this can be found on the *StimulationHints* schema we saw in Section 4.2.2. For example, the sight of blood could be defined as one such stimulus. By this method, the organism becomes more inclined in behave in a way that brings such stimuli (i.e., by performing operants whose consequences are among these stimuli).

4.2.7.3 Response Regulation

We have just seen that some emotions can be defined according to the effects that they have on the perception of stimuli. However, some emotions cannot

be defined in this way, because they modify behaviour more directly. The *ResponseEmotionalRegulator* schema defines the *responseRegulator* function, which take as input an action and a the probability of spontaneously emitting such an action (see Appendix A). By modifying this probability, it is possible to interfere directly with an organism's behaviour.

As an example of such an emotion, our specification defines frustration as a generalized increase of spontaneous behaviour. This captures the ordinary notion of frustration, which is a response to a situation in which actions do not produce their expected outcome. As a result, the organism becomes more inclined to perform arbitrary actions in order to check whether any of them is useful.

Again, the formal definition of the emotion is divided in a schema that defines these effects, *FrustrationRegulator*_1, and another schema, *FrustrationRegulator*_2, which accounts for the case in which the emotion is inactive (see Appendix A). We can then compose the complete frustration regulator.

$$FrustrationRegulator \mathrel{\widehat{=}} FrustrationRegulator_1 \lor FrustrationRegulator_2$$

4.2.7.4 Operations

The operations concerning emotions are very simple and deal only with their start, maintenance and termination. For each emotion, thus, three operations are defined[6]:

- *Start operations.* When an emotion starts, it is necessary to set its status to *Active*, and attribute it a duration and an intensity. These operations are provided by *StartDepressionOp*, *StartAngerOp* and *StartFrustrationOp* schemas (see Appendix A).
- *Maintenance operations.* Once an emotion is active, it may be updated. This update consists in reducing its remaining duration, to account for the time that has passed. These operations are provided by *UpdateDepressionOp*, *UpdateAngerOp* and *UpdateFrustrationOp* schemas (see Appendix A).
- *Termination operations.* Finally, when an emotion reaches a duration less than or equal to zero, it must be terminated by setting its status to *Inactive*. This is achieved by *EndDepressionOp*, *EndAngerOp* and *EndFrustrationOp* schemas (see Appendix A).

[6]It would be better to define only three operations that could work for all the emotions. However, this is not possible because of limitations on the Z Notation (i.e., there is no polymorphism). We are thus forced to define new operations for each particular emotion.

Notice that while our maintenance operations merely decrement the remaining duration, one can think of other modifications they could perform. For example, they could decrease the emotion's intensity according to some function. This, then, is another way in which our model can be customized.

4.2.7.5 Integration

The Emotion Subsystem is integrated in the organism by the *Organism_EmotionUpdate* operation, which merely applies update or termination operations (see Appendix A). To start an emotion, however, the appropriate start operation has to be called directly, which we did in Section 4.2.4 when defining punishment and reinforcement.

Notice that since each emotion is rather unique, it follows that its starting points are also idiosyncratic. Hence, if one wishes to add new emotions to our framework, it would also be necessary to add starting points elsewhere in the organism.

4.2.8 Subsystems Integration

We have seen that each subsystem provides a number of operations to allow their integration with the rest of the organism. These operations assume that time advances in a discrete manner, and often require the specification of a current instant. To group them together, then, we specify a data structure, *Simulator*, and an overall operation called *SimulatorIterationOp*. This operation advances time, applies the integration operations, deliver stimuli and collect responses.

Simulator

Organism

currentInstant : *Instant*

$$
\begin{array}{l}
\textit{SimulatorIterationOp} \\
\hline
\Delta \textit{Simulator} \\
\textit{Organism_ConflicResolution} \\
\textit{Organism_ResponseMaintenance} \\
\textit{Organism_OperantEliminationOp} \\
\textit{Organism_DrivesUpdate} \\
\textit{Organism_EmotionUpdate} \\
\textit{stimulations}? : \mathbb{P}\, \textit{Stimulation} \\
\textit{responses}! : \mathbb{P}\, \textit{Response} \\
\hline
\textit{currentInstant}' = \textit{currentInstant} + 1 \\
\exists\, \textit{Organism_StimulusProcessing} \bullet \textit{currentInstant}? = \textit{currentInstant} \\
\exists\, \textit{Organism_BehaviorSelection} \bullet \\
\quad \textit{currentInstant}? = \textit{currentInstant} \land \textit{responses}! = \textit{activeResponses} \\
\exists\, \textit{Organism_ResponseEmission} \bullet \textit{currentInstant}? = \textit{currentInstant} \\
\exists\, \textit{Organism_OperantOp} \bullet \textit{currentInstant}? = \textit{currentInstant} \\
\exists\, \textit{Organism_OperantFormationOp} \bullet \textit{currentInstant}? = \textit{currentInstant} \\
\hline
\end{array}
$$

These schemas *are not* part of what constitutes an agent. Rather, they specify how an actual simulator should interact with an agent. The environment is represented in this interaction by the variables *stimulations*? and *responses*!. The former collects the stimulations coming from the environment to a particular organism, and the latter specifies the behavioural responses of this organism being delivered to the environment, both in the current instant. There is, therefore, merely an interface to the surrounding environment – which is all that is required to simulate an environment with its several agents, as we shall see in Chapter 5.

Moreover, the passage of time as perceived by the organism is subjective. From the point of view of the organism, time is perceived to advance at each interaction with the environment, and it is this *subjective time counting* that is taken in account. This provision makes it possible for the organism to make internal calculations based on the passage of time without having to be told the actual, universal and absolute time, which is not available from the **EMMAS** environments of Chapter 5. Of course, if a global clock was available, then the internal time counting of the organism could be made to match such a clock, but that is not a requirement.

4.3 Conclusion

Any mathematical formalization of a domain not strictly mathematical is bound to make certain choices (e.g., to solve textual ambiguities) and introduce new technicalities in order to mechanize as much as possible of the informal and original meaning. It is also difficult to capture all relevant phenomena under one unified mathematical theory, since the details necessary for such an unification might not be present in the original informal theory. Our formalization, then, is subject to similar problems. Nevertheless, we have tried to minimize any such idiosyncrasies so that the final result can be regarded as a sensible interpretation of Behaviour Analysis. In particular, though it cannot account for all possible behavioural phenomena found on the literature, it is capable of modelling many of them, and in such a way that they relate to each other in a coherent whole. Indeed, it is largely because of this coherence that our formalization is suitable as an agent architecture, since it allows different mechanisms to operate together in creating several aspects that contribute to an interesting agent (e.g., a certain autonomy, learning capabilities, interaction with the environment).

Despite the complexities of our architecture, its elements are all ultimately given in terms of stimuli and behavioural responses. As a consequence, an agent's behaviour is always either controllable or at least observable in this approach. Hence, much more power is available to the environment where the agent is located (e.g., a laboratory), for the agent's state can be easily characterized by external events alone. This is a distinctive feature of behaviourist approaches, and should be true in any such behaviourist architecture.

The behaviourist view of agency that we presented brings a reversed perspective on agents – let us see why. Usually, agents are defined by their internal elements and their relations. Thus, for instance, the question of whether the agent performs correct deductions is important in such cases. But by making stimuli and behaviour prominent, and defining everything else in their terms, we effectively shift the questions that can be asked about such agents. We are not worried about, say, knowledge and correct reasoning, but by predicting and controlling behaviour. This emphasizes the relation that the agent has with its environment, and creates new possibilities therein. For example, one may consider sophisticated ways in which the environment may influence its agents in order to achieve certain results of interest. This shall become clear in Chapter 5.

CHAPTER 5

Environment Model for Multi-Agent Systems

Environments account for the medium through which agents may interact. In this chapter we develop an environment model that has a social network structure in which nodes are agents, and the links between them are defined by the capabilities that agents have to act upon each other. Furthermore, these environments are more than a network structure, as they may change dynamically, either spontaneously or as a reaction to an agent's actions. These design choices arise from the agent model given in Chapter 4, which suggests a number of desirable features from an environment that brings them together. For instance, we place great importance on the possibility of performing experiments of different kinds, and of responding to agent's actions in appropriate ways. Our approach achieves this by the **environment behaviours** it defines. Moreover, interaction can be treated by abstracting physical properties (e.g., spacial position) away and dealing only with relationships, which we do by adopting a social network structure and **environment operations** to modify it.

We provide a simple formal framework in which to define such environments so that they can be subject to automated analyses procedures. A mathematical model is provided, which we call the ***Environment Model for Multi-Agent Systems (EMMAS)***, and its semantics is given in terms of the π-calculus process algebra (see Appendix E for an overview of π-calculus).

Environment Model for Multi-Agent Systems (EMMAS)

Process algebras are typically employed to describe concurrent systems. They are good at succinctly describing behaviours relevant to inter-process communication. The particular choice of π-calculus as a theoretical foundation is motivated by some of its features, which together make it a distinguished for-

malism among existing such algebras. First, it takes communication through channels as a primitive notion, which makes it a natural choice for representing networks. Second, it allows for dynamic modification, which makes the creation and destruction of connections between agents possible. Third, it provides a convenient representation for broadcast behaviour through its replication operator. Finally, it has few operators and a simple operational semantics.

The semantics of **EMMAS** is actually given in two stages, by considering: (i) a syntactical translation of **EMMAS** into π-calculus expressions; (ii) a mathematical foundation which relates π-calculus events to the stimuli and actions of agents in a transition system. The π-calculus translation of (i), through its operational semantics (Definition E.5), provides an over-approximation of the desired behaviour[1], which is then made precise using the restrictions provided by (ii). By this method, we are able to build a transition system that defines the possible states and transitions for any particular environment specification. In the present chapter, however, we only provide (i). Stage (ii) is left for Chapter 6, because it requires a number of new definitions concerning transition systems, which are better understood if isolated in a chapter of their own.

The semantics thus achieved is general and is not tied to any particular application, not even simulation. For the purposes of the verification technique, however, it will be necessary to carry out stage (ii) in a slightly more specific manner, so that the result can be used in a simulator. This is explained in more detail and accomplished in Chapter 7, which also presents the verification technique itself.

We purposefully treat agents as black-boxes in **EMMAS**, because it is not necessary to expose their internal structure in order to manipulate them from an environmental perspective. As we saw in Section 4.1, this arises directly from the behaviourist tradition we use as inspiration. However, there must be a way to interface the agents with their environment. This is achieved through the assumption that agents receive **stimuli** as input and that they output **actions**, as explained in Chapter 4. Moreover, communication between agents is also mediated by the environment.

This view of the agents as black-boxes does not mean that the **Behaviourist Agent Architecture** given in Chapter 4 is irrelevant. Rather, it only means that its purpose is to allow the simulated agents to behave in ways such that the environment defined in the present chapter can fruitfully interact with them. That is to say, while the internal mechanisms of agents are not represented explicitly in **EMMAS**, they are necessary to actually simulate and

[1] That is to say, an approximation that contains all the desired behaviours, but also some undesired ones.

verify it. Indeed, as argued in Section 1.1, it is because this separation of environments and agents can be done that the technique presented in this thesis is practical. The role of environments is to be amenable to systematic analyses, whereas the purpose of agents is to implement, as completely as possible, individual behavioural phenomena, with no particular commitment to being amenable to *internal* formal analyses. It is only the *external*, observable, actions of agents, as reflected in an environment, that one may analyse by this method.

A few remarks on notation are in order before proceeding:

- We have omitted π-calculus input and output parameters when such parameters are not relevant (e.g., we write $\overline{a}$ instead of $\overline{a}\langle x\rangle$ if x is not used later).

- For the sake of readability, some elements are coloured in a different manner. This will be clear from their use, but let us quickly summarize what these colours are for each kind of element: semantic definitions; expressions, sets and logical formulas used by **EMMAS**; and the $[\,]_\pi$ translation function used to convert **EMMAS** expressions in π-calculus introduced by Definition 5.1.

This chapter is organized as follows. In Section 5.1 we explain in more detail the role of environments in the overall approach. The environment model itself is presented in Section 5.2. It is designed to be small, which implies that convenience constructs are left out. Yet, it provides the basic elements with which such conveniences can be built, and thus in Section 5.3 we provide some such conveniences. The reason for this twofold division is to facilitate both the mathematical treatment of the model (i.e., because it is kept small) and the addition of new convenience constructs beyond those we provide here (i.e., because the fundamental rules to respect are few). At last, in Section 5.4 we make some concluding remarks. As already remarked above, an overview of π-calculus is given in Appendix E.

5.1 The Role of Environments

Though environments are often ignored in the design of MAS (Weyns *et al.*, 2005), they play a crucial role in our approach. In Section 1.1 of Chapter 1, we saw that it can be useful to see the environment as a simpler, more tractable, entity than the agents that inhabit it. In our approach, this intuition is made concrete by the following main points:

- The **Behaviourist Agent Architecture** presented in Chapter 4 is only meaningful if there is an environment to provide it with stimuli and receive its actions. In particular, an agent can only interact with another agent if there is an environment to transform the actions of one into stimuli for the other.
- The environments can be translated into a representation of the MAS in terms of a transition system, which is required by the verification technique presented in Chapter 7. That is to say, the environment defines the state-space that the verification algorithms shall explore.

These characteristics are, of course, very related. It is precisely because the environment defines the *possible* communications with and between agents that it can provide a representation of all possible behaviours of the MAS.

This does not mean that one may know *a priori*, just by the structure of the environment, exactly how a simulation of an MAS will progress. While all possibilities are known, the actual sequence of states to be produced by a simulation will depend on the internal mechanisms of the agents, which are not available to the environment. In other words, the interaction between agents and their environment is essential in order to have an actual simulation.

It is also worth to consider the methodological implications of having the environment in such a prominent position. As we remarked in Chapter 4, the agent architecture follows a behaviourist approach, which puts great emphasis on defining behaviours in terms of their effects and dependencies upon an environment. This means that problems concerning such agents are invariably formulated in terms of environmental properties. For example, if one is interested in teaching a certain behaviour to an agent, the solution will be given in environmental terms: how the environment should reinforce the agent, or how the environment should connect agents so that one may interfere in the behaviours of the other. Therefore, the environment is a fundamental part of the very questions which our theory and technology address.

5.2 Environment Model

EMMAS is a mathematical framework that can be used to specify environments for multi-agent systems. In order to give its semantics, we have chosen to translate its constructs to the π-calculus process algebra, which provide simpler elements, with an already established semantics. Such a translation is achieved by using a translation function to map constructs of **EMMAS** into π-calculus (i.e., a construct C is translated to $[C]_\pi$).

TRANSLATION FUNCTION

Definition 5.1 (Translation function)**.** *The **translation function** $[]_\pi$ maps*

constructs of **EMMAS** *into π-calculus expressions.*

The full definition of this function is given as new constructs are introduced.

The constructs of **EMMAS** can be divided into *structures* and *environment operations*. The former define the elements that exist and how they interact. The later account for the manipulation of these structures.

The text below is organized as follows. Section 5.2.1 defines the fundamental π-calculus events upon which the formalization is built. Section 5.2.2 establishes what is an operation, which is an important concept used to define the model. Finally, Section 5.2.3 describes the structures of the model itself.

5.2.1 Underlying Elementary π-Calculus Events

A π-calculus specification can be divided into two parts. First, and most fundamentally, it is necessary to specify the set of events that are particular to that specification. Second, it is necessary to specify processes built using those events. In this section we account for this first part.

Input and output events are all made from basic names. Hence, we first formally define a set of names in order to have the corresponding events. The definition below establishes such names, and Table 5.1 presents an informal description of the events that arise. The formal description of their meaning, however, shall be given later on, in Sections 6.2.2 and 7.1, by defining the possible transitions associated with each name.

Definition 5.2 (Environment Names)**.** *The* ***environment names*** *are defined by the following set:* ENVIRONMENT NAMES

$$\begin{aligned} ENames = \; & \{emit_a^n, stop_a^n, beginning_s^n, stable_s^n, absent_s^n, \\ & destroy_{a,n}^{s,m}, ccn, done \mid \\ & a \in Actions, s \in Stimuli, m, n \in AgentIDs\} \end{aligned}$$

The sets Actions, Stimuli and AgentIDs shall be introduced in Definition 6.5. For the moment, it suffices to note that they represent all possible actions, stimuli and agents in an environment, respectively. In this way, these **environment names** are tied to particular actions, stimuli and agents. Nevertheless, they are atomic entities from the point of view of π-calculus, even though they are denoted here with subscripts and superscripts. This writing style is merely for readability's sake.

With these names, we now establish the set of events relevant to **EMMAS**.

Definition 5.3 (Environment Events)**.** *The* ***environment events*** *are defined by the following set:* ENVIRONMENT EVENTS

$$EEvents = \{\overline{e}\langle x\rangle, e(x) \mid e, x \in ENames\} \cup \{\tau\}$$

As a technicality, it is sometimes convenient to be able to translate π-calculus processes and events using the $[\,]_\pi$ function. The result of such a translation is, of course, the process or event itself. Thus we extend the domain of $[\,]_\pi$ to include π-calculus and give the following definition.

Definition 5.4. *Let P be an arbitrary π-calculus process or prefix. Then,*

$$[P]_\pi = P$$

A corollary of this definition is that the $[\,]_\pi$ function is idempotent (i.e., $[[C]_\pi]_\pi = [C]_\pi$).

Event	**Informal description**
Agent to environment	
$\overline{emit_a^n}$	Agent identified by n performs action a.
$\overline{stop_a^n}$	Agent identified by n stops performing action a.
Environment to agent	
$beginning_s^n$	Delivery of stimulus s to the agent identified by n is beginning.
$stable_s^n$	Delivery of stimulus s to the agent identified by n is stable.
$ending_s^n$	Delivery of stimulus s to the agent identified by n is ending.
$absent_s^n$	Delivery of stimulus s to the agent identified by n becomes absent.
Environment to environment	
$\overline{destroy_{a,n}^{s,m}}$	Requests the destruction of an action transformer that converts action a from agent identified by n into stimulus s accepted by the agent identified by m.
$\overline{ccn}$	Requests the creation of a new action transformer.
$\overline{done}$	Signals that an operation has terminated.

Table 5.1: Informal description of events, divided in three categories according to their origin and destination. The corresponding output or input events not shown merely allow the ones described to work properly.

5.2.2 Environment Operations

In order to exhibit dynamic behaviour, the environment depends on **environment operations** to modify its structures.

ENVIRONMENT OPERATION

Definition 5.5 (Environment Operation)**.** *An **environment operation** is any π-calculus process such that:*

- *its names belong to the set ENames;*
- *in the corresponding LTS, for all transitions of the form* $P \overset{l}{\rightarrowtail} \mathbf{0}$*, it must be the case that* $l = \overline{done}$ *(i.e., the operation signals its termination using the* $\overline{done}$ *prefix).*

The second condition is particularly important because it allows the sequential composition of operations, as described in Section 5.3.1 later on. It states that whenever the processes is reduced to the primitive Nil process (denoted by $\mathbf{0}$), it must be the case that the $\overline{done}$ prefix preceded it. In this manner, other **operations** can detect the termination (i.e., by specifying a *done* prefix to synchronize with $\overline{done}$).

Of course such an abstract definition of **environment operations** cannot be used directly. Nevertheless, it suffices to define the basic model for environments. Concrete **environment operations** are given in Section 5.3.2.

Because the Z Notation used in Chapter 4 has its own notion of operation, for consistency we must name the operations of **EMMAS** differently, and we have opted for calling them **environment operations**. Nonetheless, for convenience, we refer to these **environment operations** merely as **operations** in the remainder of the present chapter, as well as in the rest of the thesis whenever it is clear from context that the subject is **EMMAS**.

5.2.3 Environment Structures

The **environment** is the central structure of specifications. It defines which agents are present, how they are initially connected, and what dynamic behaviours exist in the environment itself.

Definition 5.6 (Environment)**.** *An* ***environment*** *is a tuple* $\langle AG, AT, EB \rangle$ *such that:* ENVIRONMENT

- $AG = \{ag_1 \ldots ag_l\}$ *is a set of* **agent profiles***;*
- $AT = \{t_1 \ldots t_m\}$ *is a set of* **action transformers***;*
- $EB = \{eb_1 \ldots eb_n\}$ *is a set of* **operations***, referred to as* ***environment behaviours****.* ENVIRONMENT BEHAVIOURS

Moreover, let $\{en_1, \ldots, en_o\} = ENames$*. Then the corresponding* π*-calculus expression for the environment is defined as:*

$$
\begin{aligned}
[\langle AG, AT, EB\rangle]_\pi = \quad & (\nu\, en_1, \ldots, en_o) \\
& ([ag_1]_\pi \mid [ag_2]_\pi \mid \ldots \mid [ag_l]_\pi \mid \\
& [t_1]_\pi \mid [t_2]_\pi \mid \ldots \mid [t_m]_\pi \mid \\
& [eb_1]_\pi \mid [eb_2]_\pi \mid \ldots \mid [eb_n]_\pi \mid \\
& !NewAT
\end{aligned}
$$

where

$$
\begin{aligned}
NewAT = \quad & ccn(emit, stop, absent, beginning, stable, ending, destroy). \\
& T(emit, stop, absent, beginning, stable, ending, destroy)
\end{aligned}
$$

and T is given in Definition 5.8.

This definition merits a few comments. First, all names from *ENames* are restricted to the environment. Second, the set of action transformers provide the network structure that connects the agents. Third, the environment behaviours, as the name implies, specifies behaviours that belong to the environment itself. This is useful to model reactions to agent's actions, as well as to capture ways in which the environment may evolve. In the first case the behaviour is specified as an *environment response* (Definition 5.19 below), while in the second case the behaviour is simply an **EMMAS** operation. Finally, the component *NewAT* allows the creation of new action transformers. In order to do so, it receives a message $\overline{ccn}$ ("create connection"), whose parameters initialize the rest of the expression. To see this more clearly, suppose that *NewAT* is in parallel composition as follows:

$$\overline{ccn}\langle emit_a^n, stop_a^n, absent_s^m, beginning_s^m, stable_s^m, ending_s^m, destroy_{a,n}^{s,m}\rangle \mid NewAT$$

Then $\overline{ccn}$ will react with *ccn* in *NewAT*, and the resulting expression will be the following:

$$T(emit_a^n, stop_a^n, absent_s^m, beginning_s^m, stable_s^m, ending_s^m, destroy_{a,n}^{s,m})$$

This expression corresponds to the definition of an action transformer, which is introduced in Definition 5.8, and specify how an action emitted by an agent is received as a stimulus by another agent. Furthermore, in the environment definition there is a parallel replication operator on $!NewAT$ to ensure that the creation of action transformers can happen as many times as needed to produce reactions[2], owing to the following structural congruence rule:

[2]It can be observed from the π-calculus operational semantics given in Appendix E that because all **environment names** are restricted, the only way for the system to progress is by performing reactions by the application of the *COM* rule. Moreover, the rule *STRUCT* together with the structural congruence relation ensures that *COM* will be applied as long as there are $\overline{ccn}$ events to react with *NewAT*.

$$!NewAT \equiv NewAT \mid !NewAT$$

Environments exist in order to allow agents to interact. As we remarked earlier, the internal structure of these agents, as complex as it may be, is mostly irrelevant to their interaction model. Thus, we have abstracted it away as much as possible. What is left are the interfaces that allow agents to interact with each other and with the environment itself, which we call **agent profiles**. Hence, we have the following definition.

Definition 5.7 (Agent Profile)**.** *An **agent profile** is a triple $\langle n, S, A\rangle$ such that:* AGENT PROFILE

- $n \in AgentIDs$ *is a unique identifier for the agent;*
- $A = \{a_1 \ldots a_i\} \subseteq Actions$ *is a set of actions;*
- $S = \{s_1 \ldots s_j\} \subseteq Stimuli$ *is a set of stimuli.*

Moreover,

$$\begin{aligned}[\langle n, S, A\rangle]_\pi = \quad & ([Act(a_1, n)]_\pi \mid [Act(a_2, n)]_\pi \mid \ldots \mid [Act(a_i, n)]_\pi) \mid \\ & ([Stim(s_1, n)]_\pi \mid [Stim(s_2, n)]_\pi \mid \ldots \mid [Stim(s_j, n)]_\pi)\end{aligned}$$

such that, for all $a \in A$ and $s \in S$, we have:

$$[Act(a, n)]_\pi = !(\overline{emit_a^n}.\overline{stop_a^n})$$

$$[Stim(s, n)]_\pi = \quad piStim(beginning_s^n, stable_s^n, ending_s^n, absent_s^n)$$

where

$$\begin{aligned}& piStim(beginning, stable, ending, absent) = \\ & \quad beginning.stable.ending.absent.piStim(beginning, stable, ending, absent)\end{aligned}$$

This definition states that agents have several components, each responsible for controlling one particular action or stimulus. $Act(a, n)$ defines that the agent identified by n can start emitting an action a and can then stop such emission. The replication operator (!) ensures that this sequence can be carried out an unbounded number of times. $Stim(s, n)$, in turn, defines that the agent identified by n can be stimulated by s, and that this stimulation follows four steps. The recursive call ensures that this stimulation sequence can start again as soon as it finishes the last step. These definitions reflect the assumptions about the agent model we consider (Chapter 4).

The relations among agents in **EMMAS** are given in the form of a social network. This means that the physical positions of agents are not taken into account; rather, only the relationships between agents are represented, thus inducing a graph in which the vertices are agents and the edges denote possible interactions between them. In this manner, modelling and analysis can be focused on the logical properties of their interaction, and ignore physical details (e.g., it does not matter that agent ag_1 is 3 meters away from ag_2 if one is concerned merely about specifying that ag_1 can hear what ag_2 says).

Given the behaviourist point of view that we adopt, these relationships are modelled by defining how the actions of an agent are transformed in stimuli for other agents. Their interaction, thus, is based on stimulation. Formally, this is represented by **action transformers**, which define how a particular action of an agent is perceived as a particular stimulus by another agent. **Action transformers** are not static: they can be created and destroyed dynamically. The importance of this is twofold. First, it allows the specification of phenomena in which the relation among agents change as they age. Second, it allows specification of several possible network structures for the same environment (i.e., the description of a class of social networks, and not one particular social network). This latter possibility can be used to determine, through the verification algorithms we shall introduce in Chapter 7, whether any of these possible network structures satisfy some property of interest.

ACTION TRANSFORMER

Definition 5.8 (Action Transformer)**.** *An* ***action transformer*** *is a tuple* $\langle ag_1, a, s, ag_2\rangle$ *such that:*

- ag_1 *is an* **agent profile** $\langle n, S_1, A_1\rangle$*;*
- ag_2 *is an* **agent profile** $\langle m, S_2, A_2\rangle$*;*
- a *is an action such that* $a \in A_1$*;*
- s *is a stimulus such that* $s \in S_2$*;*

Moreover, the corresponding π*-calculus expression for the action transformer is defined as:*

$$[\langle ag_1, a, s, ag_2\rangle]_\pi = T(emit_a^n, stop_a^n, absent_s^m, beginning_s^m, stable_s^m, ending_s^m, destroy_{a,n}^{s,m})$$

where

$$T(emit, stop, absent, beginning, stable, ending, destroy) = \overbrace{(emit.\overline{beginning}.\overline{stable}.stop.\overline{ending}.\overline{absent}}^{Normal\ behaviour}.T(emit, stop, absent, beginning, stable, ending, destroy)) + \underbrace{destroy}_{To\ destroy}$$

The above definition can be divided in two parts. First, there is its normal behaviour, which merely defines the correct sequence through which an action is transformed in a stimulus. Once such a sequence is completed, a recursive call to the process definition restarts the action transformer. Second, there is the part that allows the transformer to be destroyed. By performing *destroy*, the action transformer disappears, since this event is not followed by anything.

Providing an intermediate structure such as the action transformer between the agents instead of allowing a direct communication is useful because an agent's actions may have other effects besides stimulation. In particular, the environment can also respond to such actions in custom ways. This can be done by specifying **environment response operations** as part of the environment behaviours (see Section 5.3.2.3).

5.3 Convenience Elements and Operations

So far we have defined the bare minimum for describing environments so that they can be formally analysed. Clearly, though, more constructs are necessary in order to make such specifications. For example, we defined what is an operation in general, but we have not presented any particular operation. In the present section, then, we provide a number of convenience elements that can be used to build concrete **EMMAS** models. Section 5.3.1 gives operators that can be used to build more complex **operations** from simpler ones. Section 5.3.2 presents core **operations** that accomplish basic tasks. Section 5.3.3 and Section 5.3.4 define, respectively, some core sets and predicates. Section 5.3.5 provides some useful quantifiers. Finally, Section 5.3.6 employs all of these elements in order to define some complex operations.

5.3.1 Composition Operators

In order to build complex **operations** on top of the basic ones, it is useful to define composition operators. Some of these can be mapped directly to π-calculus operators, but others require more sophistication.

Definition 5.9 (Sequential Composition)**.** *Let* Op_1 *and* Op_2 *be* **operations**. *Then their* ***sequential composition*** *is also an* **operation** *and is written as:*

SEQUENTIAL COMPOSITION

$$Op_1;\ Op_2$$

Moreover,

$$[Op_1;\ Op_2]_\pi = (\nu\, start)[Op_1]_\pi\{start/done\} \mid start.[Op_2]_\pi$$

The above translation aims at accounting for the intuition that Op_1 must take place before Op_2. However, we cannot translate $Op_1;\ Op_2$ immediatly as $[Op_1]_\pi.[Op_2]_\pi$, because in general π-calculus would not allow the resulting syntax (e.g., $(P+Q).R$ would not be a valid expression). Therefore, we adapt the suggestion offered by Milner (1999) (in Example 5.27), which works as follows. We assume that every operation signals its own termination using the $\overline{done}$ event. Then, when composing Op_1 and Op_2, we: (i) create a new event, *start*; (ii) rename the *done* event in Op_1 to *start*; (iii) make *start* guard Op_2; (iv) put the two resulting processes in parallel. By this construction, the only way that Op_2 can be performed is after $\overline{start}$ is performed, which can only happen when Op_1 terminates.

SEQUENCE

Definition 5.10 (Sequence)**.** *Let Op be an* **operation** *and n be an integer such that $n \geq 1$. Then a* ***sequence*** *of n compositions of Op is defined as:*

$$Seq(Op, n) = \begin{cases} Op;\ Seq(Op, n-1) & n > 1 \\ Op & n = 1 \end{cases}$$

UNBOUNDED SEQUENCE

Definition 5.11 (Unbounded Sequence)**.** *Let Op an* **operation**. *Then an* ***unbounded sequence*** *of compositions of Op is defined as:*

$$Forever(Op) = Op;\ Forever(Op)$$

The translation of these two kinds of sequences to π-calculus follows, of course, from the translation of the sequential composition operator.

CHOICE

Definition 5.12 (Choice)**.** *Let Op_1 and Op_2 be operations. Then their composition as a* ***choice*** *is also an operation and is written as:*

$$Op_1 + Op_2$$

Moreover,

$$[Op_1 + Op_2]_\pi = [Op_1]_\pi + [Op_2]_\pi$$

PARALLEL COMPOSITION

Definition 5.13 (Parallel Composition)**.** *Let Op_1 and Op_2 be* **operations**. *Then their* ***parallel composition*** *is also an* **operation** *and is written as:*

$$Op_1 \parallel Op_2$$

Moreover,

$$\begin{aligned}[Op_1 \parallel Op_2]_\pi = \quad & (\nu\, start)[Op_1]_\pi\{start/done\} \mid \\ & [Op_2]_\pi\{start/done\} \mid \\ & start.start.\overline{done}\end{aligned}$$

The translation for the **parallel composition** is not straightforward because it is necessary to ensure that $\overline{done}$ is sent only once in the composed operation. That is to say, the parallel composition of 2 **operations** is an **operation** itself, and it only terminates when each of its components terminates. If this care is not taken, later sequential compositions will not work as expected. This definition ensures the correct translation by: (i) creating a new name, *start*, restricted to the composition; (ii) renaming *done* to *start* in Op_1 and Op_2; (iii) creating a new component that waits for 2 *start* events before sending one $\overline{done}$. By this construction, the only way that a $\overline{done}$ event can be sent is by first producing 2 *start* events, which can only happen if each operation terminates individually.

5.3.2 Core Operations

We can now provide a core of **operations** upon which others can be built. Below we present them according to their purpose.

5.3.2.1 Agent Stimulation Operations

The following **operations** are provided to control the stimulation of agents.

Definition 5.14 (Begin stimulation operation)**.** *Let* $ag = \langle n, S, A \rangle$ *be an agent profile, and* $s \in S$ *be a stimulus. Then the* ***begin stimulation*** **operation** *is writen as:* BEGIN STIMULATION

$$BeginStimulation(s, ag)$$

Moreover,

$$[BeginStimulation(s, ag)]_\pi = \overline{beginning_s^n}.\overline{stable_s^n}.\overline{done}$$

Definition 5.15 (End stimulation operation)**.** *Let* $ag = \langle n, S, A \rangle$ *be an agent profile, and* $s \in S$ *be a stimulus. Then the* ***end stimulation*** **operation** *is writen as:* END STIMULATION

$$EndStimulation(s, ag)$$

Moreover,

$$[EndStimulation(s, ag)]_\pi = \overline{ending_s^n}.\overline{absent_s^n}.\overline{done}$$

Definition 5.16 (Stimulate operation)**.** *Let* $ag = \langle n, S, A \rangle$ *be an agent profile, and* $s \in S$ *be a stimulus. Then the* ***stimulate*** **operation** *is defined as:* STIMULATE

$$Stimulate(s, ag) = BeginStimulation(s, ag);\ EndStimulation(s, ag)$$

5.3.2.2 Action Transformers Operations

The following **operations** are provided to manipulate action transformers.

CREATE ACTION TRANSFORMER

Definition 5.17 (Create action transformer operation)**.** *Let* $ag_1 = \langle n, S_1, A_1 \rangle$ *be an agent profile,* $ag_2 = \langle m, S_2, A_2 \rangle$ *be another agent profile,* $a \in A_1$ *be an action, and* $s \in S_2$ *be a stimulus. Then the* ***create action transformer*** *operation is writen as:*

$$Create(ag_1, a, s, ag_2)$$

Moreover,

$$[Create(ag_1, a, s, ag_2)]_\pi = \begin{array}{l} \overline{ccn}\langle emit_a^n, stop_a^n, absent_s^m, beginning_s^m, \\ stable_s^m, ending_s^m, destroy_{a,n}^{s,m} \rangle . \overline{done} \end{array}$$

In the above definition, $\overline{ccn}$ is crafted to react with the component $NewAT$ given in Definition 5.6. Since **operations** will ultimately be put together with parallel composition in the environment, it follows that the $Create(ag_1, a, s, ag_2)$ operation will be able to react with $NewAT$ and originate a new action transformer.

DESTROY ACTION TRANSFORMER

Definition 5.18 (Destroy action transformer operation)**.** *Let* $ag_1 = \langle n, S_1, A_1 \rangle$ *be an agent profile,* $ag_2 = \langle m, S_2, A_2 \rangle$ *be another agent profile,* $a \in A_1$ *be an action, and* $s \in S_2$ *be a stimulus. Then the* ***destroy action transformer*** *operation is written as:*

$$Destroy(n, a, s, m)$$

Moreover,

$$[Destroy(n, a, s, m)]_\pi = \overline{destroy_{a,n}^{s,m}} . \overline{done}$$

5.3.2.3 Environment Response Operations

As we remarked earlier, besides transforming an action of an agent into stimuli for other agents, the **environment** itself can also react to such actions. This is achieved by **environment response** operations, which may define a custom **operation** for each action of each agent.

ENVIRONMENT RESPONSE

Definition 5.19 (Environment Response)**.** *Let* $\langle n, S, A \rangle$ *be an* **agent profile**, $a \in A$ *an action and* Op *an* **operation**. *Then the* ***environment response*** *function* $ER()$ *for these elements is defined as follows:*

$$ER(a, ag, Op) = Forever(Emit(a, ag);\ Op;\ Stop(a, ag))$$

Where:

$$\begin{aligned}[Emit(a, ag)]_\pi &= emit_a^n.\overline{done} \\ [Stop(a, ag)]_\pi &= stop_a^n.\overline{done}\end{aligned}$$

As an example of such an environment response, we may cite the classical notion of reinforcement from behaviourist psychology. When an agent performs a desirable action, the environment may be designed so that the agent receives a reward in order to reinforce this behaviour. This relation between the agent's action and an associate reward can be elegantly modelled in a process algebraic way according to the above definition of environment response.

5.3.2.4 Do Nothing Operation

At last, it is also convenient to define a standard **operation** to state that nothing should be performed. This can be used in a number of ways, such as delaying (in a **sequential composition**) the performance of another **operation**, serving as place holders in an incomplete model, or stating conditions in the form of **environment responses** without defining any particular effects. This last possibility has a particularly important technical purpose, since only the actions that are used somehow in the environment are taken in account in the final semantic model. The reason is that in this way only the actions relevant to an environment are taken in account, thereby making its analysis more efficient.

Definition 5.20 (Do Nothing Operation)**.** *The **do nothing operation** is denoted by* DO NOTHING OPERATION

$$NOP$$

Moreover, the corresponding π-calculus expression is as follows:

$$[NOP]_\pi = \overline{done}$$

5.3.3 Sets

Certain sets of elements are particularly useful for modelling.

Definition 5.21. *Let X be any set, $S \subseteq Stimuli$, $A \subseteq Actions$, $ag = \langle n, S, A\rangle$ be an agent profile, i, j be natural numbers and $I \subseteq AgentIDs$. Then we have the following special sets:*

- $\emptyset$*: The empty set.*

- $\mathbb{P}(X)$: *The set of all subsets of* X *(i.e., its power set).*
- $canReceive(n) = S$
- $canEmit(n) = A$
- $i..j = \{k \mid i \leq k \leq j\}$.
- $\langle I, S, A\rangle = \{\langle id, S, A\rangle \mid id \in I\}$

The $\langle I, S, A\rangle$ construction allows the concise specification of large sets of similar agents. It is especially useful if the agent identifiers are natural numbers, because in this case it can be used in association with the $i..j$ construction. For example, if we know that agent identified by 1 up to 100 are all similar, we can specify all of their profiles at once by writing $\langle 1..100, S, A\rangle$.

Composite sets can be obtained by the usual operators of $\cup$ (union), $\cap$ (intersection) and $\setminus$ (subtraction).

5.3.4 Predicates and Logical Formulas

Primitive predicates are necessary to specify conditions. Below we define relevant predicates for **EMMAS**.

Definition 5.22. *Let* X *be a set,* $ag_1 = \langle n, S_1, A_1\rangle$ *and* $ag_2 = \langle m, S_2, A_2\rangle$ *be* **agent profiles**, $a \in A_1$ *be an action and* $s \in S_1$ *be a stimulus. Then we have the following predicates:*

- $isConnected(ag_1, a, s, ag_2)$: *True if, and only if, there exists an action transformer that takes action* a *from agent* ag_1 *and transforms it in stimulus* s *delivered to agent* ag_2.
- $ag_1 = ag_2$: *True if, and only if,* $n = m$.
- $ag_1 \neq ag_2$: *True if, and only if,* $n \neq m$.

Formulas can be obtained by using the usual logical connectives $\neg$ (negation), $\wedge$ (conjunction), $\vee$ (disjunction) and $\rightarrow$ (implication).

5.3.5 Quantification

In order to succinctly express arbitrary number either of choices or of concurrent execution, it is convenient to define two special quantification operators.

Given a set of possible parameters and a parameterized expression, these operators generate a new expression that corresponds to a composition of the several instantiations that the given expression might have with respect to the specified set of possible parameters.

Definition 5.23 (Universal quantification with sum)**.** *Let Y be a finite set, $Exp()$ be an arbitrary expression, and $Formula$ be a logic formula that is obeyed by the elements $y_1, y_2, \ldots y_n \in Y$. Then the* ***universal quantification with sum*** *is defined as:*

UNIVERSAL QUANTIFICATION WITH SUM

$$\forall_{+}\, y : Y \mid Formula \bullet Exp(y) = Exp(y_1) + Exp(y_2) + \ldots + Exp(y_n)$$

Definition 5.24 (Universal quantification with parallel composition)**.** *Let Y be a finite set, $Exp()$ be an arbitrary expression, and $Formula$ be a logic formula that is obeyed by the elements $y_1, y_2, \ldots y_n \in Y$. Then the* ***universal quantification with parallel composition*** *is defined as:*

UNIVERSAL QUANTIFICATION WITH PARALLEL COMPOSITION

$$\forall_{|}\, y : Y \mid Formula \bullet Exp(y) = Exp(y_1) \parallel Exp(y_2) \parallel \ldots \parallel Exp(y_n)$$

5.3.6 Complex Operations

Using the elements defined above, it is possible to create a number of other convenience operations. There are many possibilities for such operations. Below we give some examples that seem useful. We employ polymorphism where appropriate to avoid creating new names and to show possible variations of an operation.

Let $S \subseteq Stimuli$ be a set of stimuli, $s \in Stimuli$ be a stimulus, $A \subseteq Actions$ be a set of actions, and AG, AG_1 and AG_2 be sets of **agent profiles**. Then we have the following operations.

Stimulate several agents. A stimulus is delivered to the agents.

$$Stimulate(s, AG) = \forall_{|}\, ag : AG \mid s \in canReceive(ag) \bullet Stimulate(s, ag)$$

Stimulate several agents with several stimuli. Several stimuli are delivered to the agents.

$$Stimulate(S, AG) = \forall_{|}\, s : S \bullet Stimulate(s, A)$$

Connect two sets of agents. Allows the creation of action transformers between two specified sets of agents using the specified sets of actions and stimuli. This does not mandate that the action transformers should actually be created. Rather, it specifies that it is possible for them to be

created. This allows one to consider all the possibilities of connections between the two sets.

$$
\begin{aligned}
Connect(AG_1, AG_2, A, S) = \; & \forall_{|}\, ag_1 : AG_1 \bullet \forall_{|}\, ag_2 : AG_2 \bullet \\
& \forall_{|}\, a : A \bullet \forall_{|}\, s : S \mid \\
& AG_1 \cap AG_2 = \emptyset \wedge \\
& a \in canEmit(ag_1) \wedge \\
& s \in canReceive(ag_2) \bullet \\
& Create(ag_1, a, s, ag_2)
\end{aligned}
$$

Connect agents in set. Similarly, allows the creation of action transformers between the agents of a specified set using the specified sets of actions and stimuli.

$$
\begin{aligned}
Connect(AG, A, S) = \; & \forall_{|}\, ag_1 : AG \bullet \forall_{|}\, ag_2 : AG \bullet \\
& \forall_{|}\, a : A \bullet \forall_{|}\, s : S \mid \\
& ag_1 \neq ag_2 \wedge \\
& a \in canEmit(ag_1) \wedge \\
& s \in canReceive(ag_2) \bullet \\
& Create(ag_1, a, s, ag_2)
\end{aligned}
$$

Disconnect agent in a set. Destroys the action transformers between the agents in the specified set.

$$
\begin{aligned}
Disconnect(AG) = \; & \forall_{|}\, ag_1 : AG \bullet \forall_{|}\, ag_2 : AG \bullet \\
& \forall_{|}\, a : canEmit(ag_1) \bullet \\
& \forall_{|}\, s : canReceive(ag_2) \mid \\
& ag_1 \neq ag_2 \wedge isConnected(ag_1, a, s, ag_2) \bullet \\
& Destroy(ag_1, a, s, ag_2)
\end{aligned}
$$

5.4 Conclusion

In this chapter we presented **EMMAS**, a model of environments for multi-agent systems. The proposed environments have both structural and operational aspects. That is to say, they represent certain structures, which can then be changed by certain operations. These **operations** serve to two purposes. First, they provide a way to specify behaviours of the environments themselves (e.g., environment responses to the actions of agents). Second, they allow the succinct specification of several possible scenarios for an environment (e.g., several possible ways of stimulating agents). This latter possibility is one of the great advantages offered by the use of a process algebra as a semantic basis (e.g., an algebraic expression $a + b$ defines the non-deterministic *possibility* of

either a or b), and to the best of our knowledge renders our approach unique insofar as environments for MASs are concerned.

EMMAS is also distinctive in that it is designed to work with the **Behaviourist Agent Architecture** developed in Chapter 4. As seen in that chapter, the agents strongly depend on an external environment, since problems dealing with them must be specified in terms of the stimulation they receive (from an environment) and the actions they produce (to an environment). In the present chapter we have seen how this can be accomplished, for instance, with the **operations** given in Section 5.3.2 which provide ways to manipulate stimulation. In Chapter 9 we shall see concrete application examples.

In our implementation, an **EMMAS** specification is provided as a XML description. This practical aspect is presented in Chapter 8, and a reference of the input format is given in Appendix C.

The semantics of **EMMAS** is given in two stages. First, its elements are translated to π-calculus expressions. This was accomplished in this chapter. The second stage consists in computing the semantics of such π-calculus expressions in terms of transition systems. This is crucial, because the verification algorithms we develop later on will operate on such transition systems, and not on π-calculus expressions. However, this second stage is described in the next chapter. The reason for this is simple: transition systems constitute a formalism in their own right, and at a different level of abstraction. We have therefore put them in a chapter of their own, in which we present these structures in their general form and then use them to produce the final semantics of **EMMAS**.

Part III

Formal Analysis and Verification

CHAPTER 6

Transition Systems and Semantics

Formal verification requires formal structures to operate on. This chapter introduces such structures as the underlying semantics of the MASs to be investigated. We first present, in Section 6.1, the notion of **annotated transition systems (ATSs)**, the formal structures to be operated on. In Section 6.2, then, we employ these transition systems to give the semantics of **EMMAS**, the environment model presented in Chapter 5. This is achieved by considering the π-calculus translation provided in Chapter 5, and employing the π-calculus operational semantics and certain constraints to build an **ATS** that defines the possible evolutions of an environment. Although an **EMMAS** specification is syntactically finite, the corresponding **ATS** that gives its semantics possibly has infinitely many states. This semantics is independent of any particular application, and in particular is not restricted to simulations – it is a general semantics. For the purpose of simulation and the related verification technique, it will have to be modified. The main reason is that to perform simulations efficiently it will be necessary to make the semantics more concrete. But since this is an implementation concern, this provision is left for Chapter 7. Finally, Section 6.3 concludes the chapter.

6.1 Annotated Transition Systems

While there may be many ways to specify the systems and their properties (e.g., programming languages, process algebras, logic), it is convenient to have a simple and canonical representation to serve as their common underlying semantic model. Here, we define and employ **annotated transition systems**

(ATSs) to this end, which are nothing but transition systems with labels given to both states and transitions.

The **ATS** definition is very similar to what is merely called a transition system by Baier and Katoen (2008). We think, however, that it is worth to emphasize that it is a special kind of transition system, in order to avoid confusion. In particular, an **ATS** is not what is usually called a labelled transition system (LTS). In an LTS, states are not labelled, the set of events may be infinite (Milner, 1999, p. 16). In the **ATSs**, by contrast, states may be labelled (i.e., "annotated", as we say, to avoid confusion) and the set of events is finite.

In an **ATS**, events play a central role, and are further divided into **input events** and **output events**. The former represent events that may be controlled by the verification procedure (i.e., may be given as an input to the simulator), and the latter events that cannot (e.g., because they are the output of some internal – and uncontrollable – behaviour of the simulator). Two special events are also provided. First, the **internal event** (τ) denotes an event that takes place but whose precise identity is not known.[1] Second, the **other event** ($\circledast$) represents an event that, given a state s, matches any **input** or **output event** e, provided that e is not part of a transition leaving s. That is to say, the **other event** is a convenience to allow the specification of a default transition for the events that are not explicitly mentioned in any given state. Such a default transition, moreover, simplifies calculations during verification, since only one event must be considered instead of a set of several events. The following definition establishes all these possible kinds of events.

EVENT

Definition 6.1 (Events)**.** *Let $\mathcal{N}$ be a primitive set of names. An **event** is one of the following:*

INPUT EVENT

- *an **input event**, denoted by $?n$ for some $n \in \mathcal{N}$.*

OUTPUT EVENT

- *an **output event**, denoted[2] by $!n$ for some $n \in \mathcal{N}$.*

INTERNAL EVENT

- *the **internal event**, denoted by τ, such that $\tau \notin \mathcal{N}$.*

OTHER EVENT

- *the **other event**, denoted by $\circledast$, such that $\circledast \notin \mathcal{N}$.*

An **ATS** is then defined as follows.

ANNOTATED TRANSITION SYSTEM (ATS)

Definition 6.2 (Annotated Transition System)**.** *An **annotated transition system (ATS)** is a tuple $\langle S, E, P, \rightarrow, L, s_0 \rangle$ such that:*

STATES

- *S is the set of primitive **states**.*

[1] This concerns our **ATSs**, but note that the π-calculus itself defines such an internal event as well.

[2] Not to be confused with the replication operator of the π-calculus.

- *E is the finite set of* **events**.
- *P is the finite set of primitive* propositions.
- *$\rightarrow$: $S \times E \times S$ is the* transition relation.
- *For any $s \in S$ and $e \in E$, there are only finitely many $s' \in S$ such that $s \xrightarrow{e} s'$ (i.e., finite branching).*
- *$L : S \mapsto \mathbb{P}(P \cup \neg P)$ is the* labelling function of states.[3]
- *For all $s \in S$ and all $p \in P$, if $p \in L(s)$, then $\neg p \notin L(s)$ (i.e., the labelling function is consistent).*
- *$s_0 \in S$ is the* initial state.

The labelling function associates literals[4], and not merely propositions, to the **states**. This allows the specification that some propositions are known to be false in a **state** (i.e., $\neg p$), but also that other propositions are not known (i.e., in case neither p nor $\neg p$ are assigned to the state). This last possibility is convenient for modelling situations in which the truth value of a proposition cannot be assessed, as it may happen in experimental situations.

Thus, an **ATS** represents some system that has several **states**, each one possessing a number of attributes, and a number of transition choices. The system progresses by choosing, at every **state**, a transition that leads to another **state** through some **event**. Given an **ATS**, any such particular sequence of its **events** and **states** is called a **run**.

Definition 6.3 (Run)**.** *Let $\langle S, E, P, \rightarrow, L, s_0 \rangle$ be an* **ATS***, $e_0, e_1, \ldots, e_n \in E$ and $s_0, s_1, \ldots, s_n \in S$. Then the sequence*

$$(s_0, e_0, s_1, e_1, \ldots, s_{n-1}, e_{n-1}, s_n)$$

*is a a **run** of the* **ATS***. Let us denote this sequence by σ. Then its length, denoted by $\mid \sigma \mid$, is $n+1$. Moreover, we also denote σ by $\sigma'.e_{n-1}.s_n$, where σ' corresponds to the **subrun** $(s_0, \ldots, s_{n-1})$.* RUN SUBRUN

The set of all possible **runs** of an **ATS** can also be defined.

Definition 6.4 (*runs*() Function)**.** *Let $\mathcal{M}$ be an* **ATS***. Then the set of all* **runs** *of $\mathcal{M}$ is denoted by:*

$$runs(\mathcal{M})$$

[3]As indicated in Section 1.2, by $\mathbb{P}(P \cup \neg P)$ we mean the power set of $(P \cup \neg P)$ (i.e., the set of all subsets of $(P \cup \neg P)$), and by $\neg P$ we mean the set $\{\neg p \mid p \in P\}$.

[4]For any proposition p, its associate literal l is defined either by $l = p$ or $l = \neg p$. In the former case, we say it is a *positive literal*, whereas in the latter we say it is a *negative literal*.

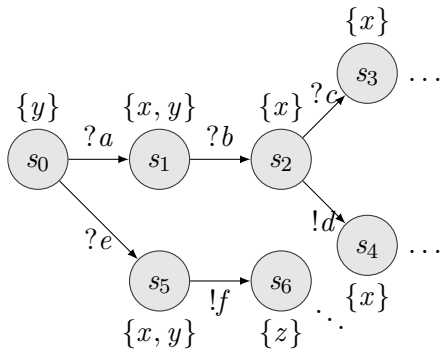


Figure 6.1: Examples of an **ATS**. Transitions are annotated with **events** (i.e., $?a$, $?b$, $?c$, $?e$, $!f$, $!d$) and **states** are annotated with literals (i.e., x, y, z). The dots (...) denote that the **ATS** continues beyond the **states** shown (it may have infinitely many states).

6.2 EMMAS Semantics

As seen in Chapter 5, the semantics of **EMMAS** is given in two main steps. First, a translation from the elements of **EMMAS** to π-calculus expressions is provided, and this was done in that chapter. The second step consists in using the π-calculus operational semantics, as well as some other restrictions we introduce, in order to transform these π-calculus expressions into transition systems – more precisely, into **ATSs**. In this section we accomplish this latter step. Section 6.2.1 presents some preliminary structures. Building on these elements, Section 6.2.2 presents the actual construction of the transition systems.

6.2.1 Preliminary Definitions

The model must have a way to effectively interact with the agents of an MAS. Agents may trigger **events** that have a meaning in the environment specification (e.g., the performance of an action). Conversely, the environment specification may request the performance of an operation (e.g., to stimulate an agent). We fulfil such requirements by providing both a **vocabulary** in which a few primitives are defined and a definition for what constitutes an **environment status** with respect to these primitives. These definitions emanate from the agent model provided in Chapter 4, and can be seen as interfaces that allow an environment to communicate with its agents.

VOCABULARY

Definition 6.5 (Vocabulary)**.** *A **vocabulary** is a tuple*

$$\langle \mathit{Stimuli}, \mathit{Actions}, \mathit{AgentIDs}, \mathit{Propositions} \rangle$$

such that:

- *Stimuli is a finite set of* stimuli*;*
- *Actions is a finite set of* actions*;*
- *AgentIDs is a finite set of* agent identifiers*;*
- *Propositions is a finite set of* atomic propositions*.*

The sets *Stimuli*, *Actions*, *AgentIDs* and *Propositions* define, respectively, all available stimuli, actions, agent identifiers and atomic propositions. These are sets containing primitive, unstructured, elements.

Moreover, the sets *Stimuli* and *Actions* must reflect the actual agents being considered. In Chapter 4 we saw that the agents interact with their environment by means of stimuli (see p. 65) and actions (see p. 75) – but each agent can, in principle, adopt different stimuli and actions. Therefore, the sets *Stimuli* and *Actions* introduced here must contain the stimuli and actions adopted by each agent.

The **environment status**, in turn, describes the dynamic connection between the agents of Chapter 4 and the environment described in the present chapter: the actual values of the functions defined therein reflect the state of the agents, which can (and normally will) change as the MAS evolves.[5]

Definition 6.6 (Environment Status)**.** *An **environment status** is a tuple* ENVIRONMENT STATUS

$$\langle Stimulation, Response, Literals \rangle$$

such that:

- $Stimulation : AgentIDs \times Stimuli \rightarrow \{Beginning, Stable, Ending, Absent\}$*;*
- $Response : AgentIDs \times Actions \rightarrow \{Emitting, NotEmitting\}$*;*
- $Literals \subseteq Propositions \cup \neg\, Propositions$*.*

The *Stimulation* function gives the stimulation of a particular agent by a particular stimulus. Agent stimulation is not an instantaneous operation. As defined in Chapter 4, the agents differentiate the beginning, the stable phase,

[5]From the semantic point of view considered in the present chapter, these functions are merely given (i.e., they are assumed to exist). How they are actually computed is a topic pertaining to the implementation of the simulator, to be seen in Chapter 8.

the ending, and the absence of a particular stimulation. Hence, we provide the appropriate elements in the function's range.

The *Response* function keeps track of the actions being emitted by the agents. In accordance with the agent architecture, we assume that actions begin and end instantaneously, and therefore we define only two elements in the function's range.

Finally, the *Literals* set contains propositions and their negations. This allows the specification of more general constraints that are not immediately related to stimulation or behavioural responses.

6.2.2 Building the Transition System

Given an **environment** *Env*, we build an **environment ATS** in two steps. First, we consider the transition system induced by $[Env]_\pi$ and show how to transform it into an **ATS** whose **states** are each annotated with an environment status (Definition 6.6). Then, we subject the resulting **ATS** to some restrictions concerning its possible runs, thereby obtaining the **environment ATS**, which describes all the legal evolutions of the MAS.

6.2.2.1 Step 1: From the π-calculus LTS to the Unrestricted Environment ATS

In order to obtain the desired behaviour, we had to restrict most π-calculus names on the **environment** (Definition 5.6). For instance, this ensured that $\overline{emit_a^n}$ prefix would only take place if its counterpart $emit_a^n$ was available elsewhere in the **environment**, by means of the *COM* rule of the π-calculus operational semantics (Definition E.5). However, in the corresponding π-calculus LTS, this reaction, like any other, appears merely as an internal prefix (i.e., the τ prefix). While this provides the correct structure to the LTS, it also hides the causes of such transitions. This poses a problem, since in most situations we would like to know which **events** led to the transitions that took place.

A solution to this issue is to merely transform each such τ prefix into an appropriate **input** or **output event** for the **ATSs**, as described in Chapter 7. To differentiate these **events** from the π-calculus prefixes, we denote them by $?n$ (**input**) and $!n$ (**output**), where n is some **name**. Sometimes, however, the underlying input and output prefixes are not useful, because they pertain only to the internal machinery of the environment, and in such cases we leave the τ prefix in place. All of this is formalized by the following *econv* function, which takes π-calculus prefixes and map them to the appropriate **events**.

Definition 6.7 (Event Conversion Function)**.** *Let $Proc_1$ and $Proc_2$ be π-calculus processes, l a π-calculus event, a, x and y arbitrary names, and $\rightarrowtail$ the transition relation induced by the π-calculus operational semantics (Definition E.5) such that $Proc_1 \overset{l}{\rightarrowtail} Proc_2$. Then the* ***event conversion function*** *$econv$ is defined by the following rules:*

EVENT CONVERSION FUNCTION

- *If $l = \tau$, and it was obtained in $Proc_1$ by the internal reaction of some $a_i^j(x)$ and $\overline{a}_i^j\langle y\rangle$ such that $a \in \{emit, stop\}$, then $econv(Proc_1, l, Proc_2) = ?a_i^j$;*
- *If $l = \tau$, and it was obtained in $Proc_1$ by the internal reaction of some $a_i^j(x)$ and $\overline{a}_i^j\langle y\rangle$ such that $a \in \{absent, beginning, stable, ending\}$, then $econv(Proc_1, l, Proc_2) = !a_i^j$;*
- *If $l = \tau$ and none of the previous cases hold, then $econv(Proc_1, l, Proc_2) = \tau$ as well;*

In the above definition, the first rule defines that prefixes pertaining to agent action shall be **input events** in the **ATS**. This means that the **event** shall be given *by* some external source as an input. The second rule, in turn, defines that all prefixes concerning stimulation are transformed in **output events**. That is to say, such **events** are to be given *to* some external receptor. We shall see in Chapter 7 that the external source and receptor, in this thesis, is the simulator which controls the agents. At that point it will be clear that this difference between output and input is fundamental, since in one case the simulation may always proceed, whereas in the other case it depends on a condition which is not certain to be fulfilled.

The states of the original π-calculus LTS must also be augmented with contextual information relevant to the **ATS**. Thus, besides the original π-calculus process, the **state** will also contain an **environment status** tuple that we saw earlier, resulting in the following form.

Definition 6.8 (Environment State)**.** *Let Env be an environment and $Proc$ be a π-calculus process obtained by applying π-calculus operational semantics rules to $[Env]_\pi$. Moreover, let $\langle Stimulation, Response, Literals\rangle$ be an* **environment status**. *Then an* ***environment state*** *is defined as the following pair:*

ENVIRONMENT STATE

$$(Proc, \langle Stimulation, Response, Literals\rangle)$$

By this construction, at any point of the **ATS** we shall be able to know both what is the current situation of the agents in so far as the environment is concerned (because of the added environment status) and what are the possible changes from that point (because of the π-calculus operational semantics).

At last, given a method of obtaining the relevant **events**, and the form of the **environment states**, we now define the **unrestricted environment ATS** inductively.

UNRESTRICTED ENVIRONMENT ATS

Definition 6.9 (Unrestricted Environment ATS)**.** *Let Env be an environment (Definition 5.6), and let* $\rightarrowtail$ *be the transition relation induced by the* π*-calculus operational semantics (Definition E.5). Then the* ***unrestricted environment ATS*** $\langle S, E, P, \rightarrow, L, s_0\rangle$ *is such that:*

- $P = \mathit{Propositions}$;
- S *and* $\rightarrow$ *are constructed inductively as follows:*
  - ***Initial state.*** $s_0 = ([\mathit{Env}]_\pi, es) \in S$, *where* $es = \langle \mathit{Stimulation}, \mathit{Response}, \mathit{Literals}\rangle$ *such that* $L(s_0) = \mathit{Literals}$ *and for all* $a \in \mathit{Actions}$, $s \in \mathit{Stimuli}$, *and* $n \in \mathit{AgentIDs}$ *we have* $\mathit{Stimulation}(n, s) = \mathit{Absent}$ *and* $\mathit{Response}(n, a) = \mathit{NotEmitting}$.
  - ***Other states and transitions.*** *If* $s_1 = (\mathit{Proc}_1, \langle \mathit{Stimulation}_1, \mathit{Response}_1, \mathit{Literals}_1\rangle) \in S$, *then* $s_2 = (\mathit{Proc}_2, \langle \mathit{Stimulation}_2, \mathit{Response}_2, \mathit{Literals}_2\rangle) \in S$, $s_1 \xrightarrow{e} s_2$, $e \in E$ *and* $L(s_2) = \mathit{Literals}_2$ *if and only if:*
    - *There exists a* π*-calculus event* l *such that* $\mathit{Proc}_1 \overset{l}{\rightarrowtail} \mathit{Proc}_2$ *and* $e = \mathit{econv}(\mathit{Proc}_1, l, \mathit{Proc}_2)$;
    - $\mathit{Stimulation}_2$ *is defined with respect to* Proc_2 *according to Definition 6.11.*

This definition can be summarized as follows. The **ATS** has an initial **state**, which is made of the π-calculus process of some **environment**, as well as an **environment status** that says that all actions are not being emitted, and that all stimuli are absent in every agent. From this initial **state** we begin the construction of the remaining (reachable) **states** and of the transition relation. This is accomplished by using the π-calculus operational semantics to know the available transitions at any given **state**, and augment the reachable **states** with environment status. This procedure is repeated to every new **state** introduced until there are no new transitions possible.

To proceed with this construction, we need a number of definitions. Let us begin by providing a way to observe the internal transitions of an environment, which is a fundamental capability that we need before proceeding. As seen in Definition 5.6, an environment's π-calculus process has a number of restrictions that would prevent such observations (i.e., the transitions would be internal to the process and not discernible in the LTS). It is, however, possible to characterize these restrictions syntactically, and thus we may provide a simple

method to remove them when needed. This is accomplished by the following **environment unrestriction function** *unr*.

Definition 6.10 (Environment Unrestriction Function)**.** *Let P and Q be π-calculus processes such that*

$$P = (\nu\, en_1, \ldots, en_o)\, Q$$

where $\{en_1, \ldots, en_o\} = ENames$. Then the ***environment unrestriction function*** *is defined as $unr(P) = Q$.*

ENVIRONMENT UNRESTRICTION FUNCTION

We may now define the *Stimulation* function present in each **state** as follows.

Definition 6.11 (Stimulation)**.** *Let $(Proc, \langle Stimulation, Response, Literals\rangle)$ be an* **environment state**. *Moreover, let $\rightarrow$ be the transition relation induced by the π-calculus operational semantics. Then, for all $s \in Stimuli$ and $n \in AgentIDs$, we have:*

$$Stimulation(n, s) = \begin{cases} Absent & \text{if } \exists\, P' \text{ such that } unr(P) \overset{beginning_s^n}{\rightarrow} P' \\ Beginning & \text{if } \exists\, P' \text{ such that } unr(P) \overset{stable_s^n}{\rightarrow} P' \\ Stable & \text{if } \exists\, P' \text{ such that } unr(P) \overset{ending_s^n}{\rightarrow} P' \\ Ending & \text{if } \exists\, P' \text{ such that } unr(P) \overset{absent_s^n}{\rightarrow} P' \end{cases}$$

The *Stimulation* definition establishes the status of a particular stimulation based on the order that stimulations must change (see Definition 5.7). For instance, if a process is capable of receiving a $beginning_s^n$ **event**, it must be the case that stimulus s is currently absent in agent identified by n. The *Stimulation* function, therefore, merely gives a way of reading the π-calculus LTS in order to have this information explicitly for every agent and stimulus in any given process.

On the other hand, both the *Response* function and the *Literals* set are assumed as given. In Chapter 7 we will see that the simulator provides their values according to the current simulation state. Thus, we do not need to formally define them here. However, *Response* imposes some constraints on the **ATS**, which we must specify and take into account.

6.2.2.2 Step 2: From the Unrestricted Environment ATS to the Environment ATS

The **unrestricted environment ATS** we have obtained so far is an over-approximation of the desired **ATS**. It contains runs which are not supposed to be part of the model. For instance, if an agent identified by n is still emitting

an action a, it cannot be the case that the **event** $?stop_a^n$ takes place, since this would indicate that the agent is not emitting the action (a contradiction). Such problems arise because the relation between the π-calculus specification and the contextual information about the agents (i.e., the functions in an **environment status**) has not yet been considered. To handle this issue, the following constraints are used to prune the **unrestricted environment ATS**.

Definition 6.12 (Transitions Constraints)**.** *Let* $s_1 = (P_1, \langle Stimulation_1, Response_1, Literals_1 \rangle)$ *and* s_2 *be states of an* **ATS** $\langle S, E, P, \rightarrow, L, s_0 \rangle$*. Then the transition* $s_1 \xrightarrow{e} s_2$ *is forbidden if one of the cases hold:*

- *There exists* $a \in Actions$ *and* $n \in AgentIDs$ *such that:*
  - $Response_1(n, a) = Emitting$;
  - $e = ?stop_a^n$.
- *There exists* $a \in Actions$ *and* $n \in AgentIDs$ *such that:*
  - $Response_1(n, a) = NotEmitting$;
  - $e = ?emit_a^n$.
- *There exists* $a \in Actions$ *and* $n \in AgentIDs$ *such that:*
  - $Response_1(n, a) = Emitting$;
  - $Response_2(n, a) = NotEmitting$;
  - *there exists an* $s' \in S$ *such that* $s_1 \xrightarrow{?emit_a^n} s'$.
- *There exists* $a \in Actions$ *and* $n \in AgentIDs$ *such that:*
  - $Response_1(n, a) = NotEmitting$;
  - $Response_2(n, a) = Emitting$;
  - *there exists an* $s' \in S$ *such that* $s_1 \xrightarrow{?stop_a^n} s'$.

The first constraint asserts that if an agent identified by n is emitting an action a, then it cannot produce the $?stop_a^n$ **event** to proceed to a new state. Conversely, the second constraint states that if the agent is not emitting such an action, then it cannot produce the $?emit_a^n$ **event**. The third constraint asserts that if the agent is emitting the action in a given **state**, and it proceeds to a **state** in which it might no longer emitting such an action, then it must not be the case that some process was still ready to receive that action (i.e., by producing the **input event** $?emit_a^n$). This means that it can only stop

emitting an action when the action has already produced all of its effects. The final constraint is the counterpart for stopping an emission. Hence, if an agent is not emitting some action, and then it start emitting it, it must not be the case that some process was still ready to receive the stop signal (i.e., by producing the **input event** $?stop_a^n$).

With such restrictions in place, we may now proceed to the definition of the final **Environment ATS**.

Definition 6.13 (Environment ATS)**.** *Let $\mathcal{M}$ be an* **unrestricted environment ATS** *(Definition 6.9). Then the* ***environment ATS*** *$\mathcal{M}'$ is equal to $\mathcal{M}$ pruned according to transition constraints (Definition 6.12).* ENVIRONMENT ATS

This **environment ATS** possibly has infinitely many **states**, since there could be evolutions of the MAS that always result in new **states**. This arises because the underlying π-calculus process may contain recursive definitions and the use of the replication operator that ensure that the transition system can always move into a new state.

6.3 Conclusion

We have seen in this chapter a special kind of transition system which we have called **ATS**. Using such structures, we managed to provide the semantics of **EMMAS**. We have therefore reduced the problem of analysing the environment of an MAS to the one of analysing an **ATS**.

The semantics given, however, is not geared towards any particular application of the MAS being modelled – it is a general semantics. In particular, details necessary for the simulation of an **EMMAS** environment are not present. This shows that **EMMAS** is capable of representing MASs independently of their implementation (e.g., as a simulation), which is a desirable feature, and for this reason we have proceeded in this way. Nevertheless, for the purposes of the verification technique, it will be necessary to introduce new characteristics in the semantics. Accordingly, we have left this provision for Chapter 7, which is also where the verification technique is presented.

CHAPTER 7

Verification Technique

In this chapter we present our approach to the formal verification of the multi-agent systems composed by the agents and environments described in previous chapters. In Section 1.1 of Chapter 1 we saw that in this thesis we view verification as a means of performing experiments in an automated way. This means that given a system model $\mathcal{M}$ and a property $\mathcal{SP}$, we determine whether $\mathcal{M}$ satisfies $\mathcal{SP}$ in a number of precise senses that we introduce. Notably, there is a sense in which the satisfaction of $\mathcal{SP}$ provides the instructions of how to bring it about, in the spirit of the experimental perspective we take (i.e., by showing how to construct a successful experiment out of several possibilities). All this is accomplished by algorithms that perform on-the-fly explorations in $\mathcal{M}$.

Formally, $\mathcal{M}$ is an **annotated transition system (ATS)** (possibly with infinitely many **states**) and $\mathcal{SP}$ is a **simulation purpose** . The former represents an **EMMAS environment**, while the latter is introduced in the present chapter – but for the moment it suffices noting that it is a finite **ATS** subject to certain extra restrictions. Verification is achieved by considering the **synchronous product** of these two transition systems. The algorithms perform depth-first searches on this **synchronous product**, which is built on-the-fly. These searches are limited to a maximum depth $depth_{max}$, since there might be branches of infinite length in the search tree.

These characteristics lead to many parameters to be accounted for in the statement of the complexities. In a few words, the complexity in space is polynomial with respect to the size of the **environment** and other parameters, and the complexity in time is exponential with respect to $depth_{max}$. The complete development of these calculations is provided in Section 7.6 of this chapter.

The technique described here is designed to work with the **EMMAS** environments, whose semantics was given in Chapter 6. However, in order to simulate (a precondition for verification) such environments, it is necessary to introduce certain implementation considerations in their semantics. Since this concerns the particular application of **EMMAS** to simulation, and not its general role with respect to MASs, we address this issue in this chapter as well.

We divide the presentation as follows. First, in Section 7.1 we explain the necessity and make the required adjustments in the semantics of **EMMAS** so that the resulting **ATS** can be used for simulations. In Section 7.2, we define precisely what **simulation purposes** are. In Section 7.3, we present a **synchronous product** that provides the basis for verification. Then, in Section 7.4 we define the satisfiability relations of interest. Based on these, in Section 7.5 we provide the verification algorithms themselves, and explain informally how they work. More rigorous analyses concerning soundness, completeness and worst-case complexities are given in Section 7.6. We finish with some concluding remarks in Section 7.7. Actual execution of these algorithms is postponed until Chapter 9.

7.1 Making the Environment ATS Suitable for Simulation and Verification

The semantics we provided to **EMMAS** in Chapter 6 is sufficient to describe all the relevant evolutions of any given environment, in the form of an **ATS**, without making reference to implementation details. In particular, this abstract model does not define precisely how a simulation based on **EMMAS** should be carried out. However, since the verification technique is based on the possibility of simulating an MAS, it is necessary, before proceeding, to make the semantics provided there more concrete so that each **run** in the final **ATS** corresponds to something that can be directly and efficiently simulated.

The problem lies in how the simulator is supposed to interact with the agents while obeying the restrictions imposed by Definition 6.12. These restrictions forbid certain transitions from happening by employing both preconditions (i.e., what must be true in the current **state**) and postconditions (i.e., what must be true in the next **state**). During simulations, the preconditions can be assessed merely by examining the current simulation state. But the postconditions can only be known after the transition is simulated. If after this it is found that the postconditions are violated, then it is necessary to backtrack to the previous state and try another transition. Clearly, it would be more efficient to have a way to be sure *a priori* that the postconditions will hold, instead of having to test them and backtrack if needed.

The restrictions given by Definition 6.12 are closely related to the behaviour of the agents. In particular, the postconditions are given in terms of what actions are being emitted or not emitted by agents (i.e., the value of $Response_2(n, a)$ for a given agent identifier n and an action a). Hence, the satisfaction of such postconditions is related to how the agents emit and stop emitting actions.

In Chapter 4 we saw that the agents must be periodically updated so that: (i) they may receive new stimulation from the environment; and (ii) they may provide new behavioural responses to the environment. Crucially, (ii) implies that any change with respect to the actions an agent is emitting depends on the execution of such an update. Therefore, it is possible to anticipate whether postconditions will hold or not, provided that one knows the appropriate moments to perform the updates.

However, in the **unrestricted environment ATS** given in Chapter 6, this information is not present. Thus, to prune it according to the constraints of Definition 6.12, it is necessary to employ the inefficient strategy of simulating a next **state** and backtracking if necessary. To see this more clearly, consider the following possible **run** in such an **unrestricted environment ATS**:

$$s_1 \overset{!beginning_u^0}{\rightarrow} s_2 \overset{!beginning_v^0}{\rightarrow} s_3 \overset{?emit_a^0}{\rightarrow} s_4$$

where s_1, s_2, s_3 and s_4 are **states** , and $!beginning_u^0$, $!beginning_v^0$ and $?emit_a^0$ denote **events** relative to an agent identified by 0, stimuli u and v and an action a (i.e., stimulation and action **events** concerning an agent). Let us further suppose that the current **state** is $s_3 = (P_3, \langle Stimulation_3, Response_3, Literals_3 \rangle)$ and that $Response_3(0, a) = Emitting$. The question is whether $s_4 = (P_4, \langle Stimulation_4, Response_3, Literals_4 \rangle)$ is a legal **state** . According to Definition 6.12, it will be an illegal **state** if $Response_4(0, a) = NotEmitting$ (i.e., if the action was being emitted and suddenly it is no longer being emitted). However, since s_4 has not yet been reached in the simulation, it is not possible to know the valuation of $Response_4(0, a)$. Therefore, it is necessary to simulate the transition $s_3 \overset{?emit_a^0}{\rightarrow} s_4$ to obtain this valuation and if indeed $Response_4(0, a) = NotEmitting$, then the transition is considered illegal and it is necessary to backtrack to **state** s_3.

To address this issue, here we introduce a new **event** (called *commit*) to signal when agents should both observe environmental **events** and provide behaviours to the environment (i.e., when agents perform their update). Every time the simulator encounters this **event** on a **run**, it executes the update cycles of all agents. Furthermore, these updates can only happen in virtue of such a commit **event**. If we apply this idea to the **run** given above, we would get one or more **runs** containing the new commit **event**. For instance, the following **run** defines that all **events** take place and only then the agent is updated:

$$s_1 \xrightarrow{!beginning_u^0} s_2 \xrightarrow{!beginning_v^0} s_3 \xrightarrow{?emit_a^0} s_4 \xrightarrow{!commit} s'$$

In this case, we can be sure that $Response_4(0, a) = Emitting$ (and therefore that s_4 is a legal **state**) simply because the agent can only change the actions it is emitting after a commit **event**. Thus, since $Response_3(0, a) = Emitting$, and there is no commit between s_3 and s_4, it must be the case that $Response_4(0, a) = Emitting$.

Similarly, the following **run** would also be legal:

$$s_1 \xrightarrow{!beginning_u^0} s_2 \xrightarrow{!commit} s' \xrightarrow{!beginning_v^0} s_3 \xrightarrow{?emit_a^0} s_4$$

In this **run**, like in the previous one, the agent cannot stop emitting the action when reaching **state** s_4. However, in contrast with the previous **run**, stimulus u and v are not taken in account by the agent at the same time. Rather, first u is delivered, then the agent process it (because of the commit event), and then v is delivered. This shows that besides facilitating the enforcement of constraints, the presence of the commit **event** also makes it explicit the difference between stimuli that are perceived simultaneously from those which are not.

The position of the commit **event** in **runs** must obey restrictions. So, contrary to the previous **runs**, the following one would be illegal:

$$s_1 \xrightarrow{!beginning_u^0} s_2 \xrightarrow{!beginning_v^0} s_3 \xrightarrow{!commit} s' \xrightarrow{?emit_a^0} s_4$$

In this case, it would be possible that the agent stops emitting a in s' (since a commit allows such a change), but an $?emit_a^0$ **event** still takes place immediately after, which would be a clear inconsistency. Therefore, this **run** is not allowed.

The important point here is that it is possible, from any **state** , to determine whether the commit **event** may happen next or not, without simulating the transition. In this manner, it is also possible to avoid the problem of having to simulate a successor **state** in order to determine its legality in a **run**. In this section we show how to do this. Essentially, we need to constrain what may happen between two consecutive commits, which can be done by examining the **events** that have already taken place.

The **events** that happen between any such pair of consecutive commits can be regarded as simultaneous as far as the agents are concerned, since they are indeed perceived simultaneously. Nonetheless, this does not imply that the **events** actually happened at the same time, nor would such an implication be important: what matters is the order of **events** and how they are observed.

This strategy works under two assumptions: (i) the existence of a global entity capable of enforcing the commit **event** upon all agents (i.e., request

their updates as described above); and (ii) that the time between any two consecutive commit **events** is greater than the time that it takes to execute the **events** between them. The first assumption is met easily since the simulator is such a global entity. The second assumption, however, imposes a limitation on the kinds of MASs that can be considered, but allows the application of the technique developed in this thesis. In making such an assumption we are able to ignore time as an absolute and numeric entity, and regard merely the order of **events** . It would be possible, of course, to consider more complex temporal representations (e.g. continuous time, events of variable and long duration), but this would introduce further problems that would be out of the scope of the present work.

Let us now see precisely how this new commit **event** must be introduced and how it leads to a new and more concrete (i.e., more specific to an application) **ATS**. We follow a similar procedure to that which we used to get from an **environment** to an **environment ATS** in Section 6.2.2. The difference here is that we define a new π-calculus prefix, $\overline{commit}$, and take it into account when building the relevant transition systems. Given some **environment**, the introduction of this new prefix is accomplished by defining a corresponding **concrete environment process** as follows.

Definition 7.1 (Concrete Environment Process)**.** *Let Env be an* **environment** *(Definition 5.6). Then the following π-calculus expression*

$$[Env]_{\pi} \mid !\overline{commit}$$

is a ***concrete environment process****.*

CONCRETE ENVIRONMENT PROCESS

The remaining transformations are also divided in two steps, analogous to those of Section 6.2.2.

7.1.1 Step 1: From the Concrete Environment Process LTS to the Unrestricted Environment ATS

First, we must adapt the **event conversion function** (Definition 6.7) to account for the newly introduced $\overline{commit}$ π-calculus prefix. To do so, we merely add a fourth rule to the three that we already had.

Definition 7.2 (Concrete Event Conversion Function)**.** *Let $Proc_1$ and $Proc_2$ be π-calculus processes, l a π-calculus event, a, x and y arbitrary names, and $\rightarrowtail$ the transition relation induced by the π-calculus operational semantics (Definition E.5) such that $Proc_1 \overset{l}{\rightarrowtail} Proc_2$. Then the* ***concrete event conversion function*** *$econv_c$ is defined by the following rules:*

CONCRETE EVENT CONVERSION FUNCTION

- *If $l = \tau$, and it was obtained in $Proc_1$ by the internal reaction of some $a(x)_i^j$ and $\overline{a}_i^j\langle y\rangle$ such that $a \in \{emit, stop\}$, then $econv_c(Proc_1, l, Proc_2) = ?a_i^j$;*
- *If $l = \tau$, and it was obtained in $Proc_1$ by the internal reaction of some $a_i^j(x)$ and $\overline{a}_i^j\langle y\rangle$ such that $a \in \{absent, beginning, stable, ending\}$, then $econv_c(Proc_1, l, Proc_2) = !a_i^j$;*
- *If $l = \tau$ and none of the previous cases hold, then $econv_c(Proc_1, l, Proc_2) = \tau$ as well;*
- *If $l = \overline{commit}\langle x\rangle$, then $econv_c(Proc_1, l, Proc_2) = !commit$.*

The **states** of the final **ATS** will be very similar as well, the only difference is that they are induced from a **concrete environment process**.

CONCRETE ENVIRONMENT STATE

Definition 7.3 (Concrete Environment State)**.** *Let Env be an environment and Proc be a π-calculus process obtained by applying π-calculus operational semantics rules to the corresponding* **concrete environment process**. *Moreover, let $\langle Stimulation, Response, Literals\rangle$ be an* **environment status**. *Then an* ***concrete environment state*** *is defined as the following pair:*

$$(Proc, \langle Stimulation, Response, Literals\rangle)$$

The **Concrete Unrestricted Environment ATS** is obtained in almost the same manner of **Unrestricted Environment ATS** (Definition 7.4), the difference being that now the initial **state** is calculated using a **synchronous environment process**.

CONCRETE UNRESTRICTED ENVIRONMENT ATS

Definition 7.4 (Concrete Unrestricted Environment ATS)**.** *Let Env be an environment (Definition 5.6), and let $\rightarrowtail$ be the transition relation induced by the π-calculus operational semantics (Definition E.5). Then the* ***concrete unrestricted environment ATS*** *$\langle S, E, P, \rightarrow, L, s_0\rangle$ is such that:*

- *$P = Propositions$;*
- *S and $\rightarrow$ are constructed inductively as follows:*
  - ***Initial state***. *$s_0 = ([Env]_\pi \mid !\overline{commit}, es) \in S$, where $es = \langle Stimulation, Response, Literals\rangle$ such that $L(s_0) = Literals$ and for all $a \in Actions$, $s \in Stimuli$, and $n \in AgentIDs$ we have $Stimulation(n, s) = Absent$ and $Response(n, a) = NotEmitting$.*
  - ***Other states and transitions***. *If $s_1 = (Proc_1, \langle Stimulation_1, Response_1, Literals_1\rangle) \in S$, then $s_2 = (Proc_2, \langle Stimulation_2, Response_2, Literals_2\rangle) \in S$, $s_1 \xrightarrow{e} s_2$, $e \in E$ and $L(s_2) = Literals_2$ if and only if:*

* *There exists a π-calculus event l such that $Proc_1 \overset{l}{\rightarrowtail} Proc_2$ and $e = econv_c(Proc_1, l, Proc_2)$;*
* *$Stimulation_2$ is defined with respect to $Proc_2$ according to Definition 6.11.*

7.1.2 Step 2: From the Concrete Unrestricted Environment ATS to the Concrete Environment ATS

This second step brings some more substantial differences with respect to the construction of Section 6.2.2. The reason is that the newly introduced *commit* **event** requires special constraints to be imposed on the **ATS**.

The first set of special constraints is similar to those of Definition 6.12. However, here the presence of the !*commit* **event** is used to calculate the third and the fourth constraint.

Definition 7.5 (Local Transition Constraints)**.** *Let $s_1 = (P_1, \langle Stimulation_1, Response_1, Literals_1 \rangle)$ and s_2 be* **states** *of an* **ATS** *$\langle S, E, P, \rightarrow, L, s_0 \rangle$. Then the transition $s_1 \overset{e}{\rightarrow} s_2$ is forbidden if one of the cases hold:*

- *There exists $a \in Actions$ and $n \in AgentIDs$ such that:*
  - *$Response_1(n, a) = Emitting$;*
  - *$e = ?stop_a^n$.*
- *There exists $a \in Actions$ and $n \in AgentIDs$ such that:*
  - *$Response_1(n, a) = NotEmitting$;*
  - *$e = ?emit_a^n$.*
- *There exists $a \in Actions$ and $n \in AgentIDs$ such that:*
  - *$Response_1(n, a) = Emitting$;*
  - *$e = !commit$.*
  - *there exists an $s' \in S$ such that $s_1 \overset{?emit_a^n}{\rightarrow} s'$.*
- *There exists $a \in Actions$ and $n \in AgentIDs$ such that:*
  - *$Response_1(n, a) = NotEmitting$;*
  - *$e = !commit$.*
  - *there exists an $s' \in S$ such that $s_1 \overset{?stop_a^n}{\rightarrow} s'$.*

In addition to these local restrictions, we must also introduce some **run** constraints in order to limit what may happen between two !*commit* **events** . By this provision, we ensure that **events** that must be processed separately by the agents are properly handled (i.e., because agents only update their internal state after the !*commit* event). These constraints are actually new, and had nothing equivalent in Section 6.2.2.

Definition 7.6 (Run Constraints)**.** *Let* $\mathcal{M} = \langle S, E, P, \rightarrow, L, s_0 \rangle$ *be an* **ATS**, $\{s_0, s_1, s_2, \ldots, s_n\} \subseteq S$ *a subset of its* **states** *, and* $\{e_1, e_2, \ldots, e_{n-2}\} \subseteq E$ *a subset of its* **events** *such that the run*

$$s_0 \overset{!commit}{\rightarrow} s_1 \overset{e_1}{\rightarrow} s_2 \overset{e_2}{\rightarrow} \ldots \overset{e_{n-2}}{\rightarrow} s_{n-1} \overset{!commit}{\rightarrow} s_n$$

is defined and for all $1 \leq i \leq n-2$*, we have that* $e_i \neq !commit$*. Moreover, for all* $0 \leq i \leq n$*, let* $s_i = (P_i, \langle Stimulation_i, Response_i, Literals_i \rangle)$ *be an* **environment state**. *Then, this* **run** *is forbidden if one of the following cases hold:*

- *There exist* $1 \leq i, j \leq n-2$*,* $s \in Stimuli$ *and* $id \in AgentIDs$ *such that:*
  - $i \neq j$*;*
  - $e_i \in \{!beginning_s^{id}, !stable_s^{id}, !ending_s^{id}, !absent_{crs}^{id}\}$*;*
  - $e_j \in \{!beginning_s^{id}, !stable_s^{id}, !ending_s^{id}, !absent_s^{id}\}$.
- *There exist* $1 \leq i, j \leq n-2$*,* $a \in Actions$ *and* $id \in AgentIDs$ *such that:*
  - $i \neq j$*;*
  - $e_i \in \{?emit_a^{id}, ?stop_a^{id}\}$*;*
  - $e_j \in \{?emit_a^{id}, ?stop_a^{id}\}$.
- *There exist* $1 \leq i, j \leq n-1$ *such that* $Response_i \neq Response_j$.

The first constraint ensures that only one stimulation **event** concerning a particular agent and stimulus can take place between commit **events** . This allows the agent to process such **events** correctly, that is to say, one per commit signal. The second constraint is analogous, but concerns an agent's actions. Finally, the third constraint defines that the agents must not change their choice of actions before the next !*commit* **event**.

Finally, we arrive at the **ATS** that is subject to simulation and verification.

Concrete Environment ATS

Definition 7.7 (Concrete Environment ATS)**.** *Let* $\mathcal{M}$ *be a* **concrete unrestricted environment ATS** *(Definition 7.4). Then the* ***Concrete Environment ATS*** $\mathcal{M}'$ *is equal to* $\mathcal{M}$ *subject to local (Definition 7.5) and* **run** *(Definition 7.6) constraints.*

Like the **environment ATS** of Definition 6.13, and for the same reasons, this **concrete environment ATS** possibly has infinitely many **states**.

7.2 Simulation Purposes

The **simulation purposes** developed in this thesis are adapted from the notion of formal test purposes of Jard and Jéron (2005). As we saw in Section 3.4.1, what we reuse here is mainly the idea of employing a transition system to specify the relevant paths in another transition system, which allows the partial exploration of the latter. However, whereas Jard and Jéron (2005) are interested in using this partial exploration to generate test cases, a **simulation purpose** is used to select the simulation paths and check the relevant satisfiability criteria during the exploration itself. It thus provides a way to specify the relevant simulation runs.

A **simulation purpose** is an **ATS** subject to a number of restrictions. In particular, it defines **states** to indicate either success or failure of the verification procedure, and requires that, from every other **state**, one of these must be reachable. Intuitively, it can be seen as a specification of desirable and undesirable simulation runs. Each possible path in a **simulation purpose** terminating in either success or failure defines one such run. Formally, we have the following.

Definition 7.8 (Simulation Purpose)**.** *A* ***simulation purpose*** *(SP) is an* **ATS** $\langle Q, E, P, \rightsquigarrow, L, q_0 \rangle$ *such that:* SIMULATION PURPOSE

(i) Q *is finite.*

(ii) $Success \in Q$ *is the verdict* **state** *to indicate success.*

(iii) $Failure \in Q$ *is the verdict* **state** *to indicate failure.*

(iv) $L(q_0) = L(Success) = L(Failure) = \emptyset$.

(v) For every $q \in Q$*, if there are* $q', q'' \in Q$ *and* $e \in E$ *such that* $q \overset{e}{\rightsquigarrow} q'$ *and* $q \overset{e}{\rightsquigarrow} q''$*, then* $q' = q''$ *(i.e., the system is* deterministic*).*

(vi) For every $q \in Q$*, there exists a* **run** *from* q *to either Success or Failure.*

These restrictions merit a few comments. First, requirement (iv) specifies that $L(q_0) = \emptyset$, which ensures that the initial **state** can always synchronize with the initial **state** of another **ATS**. This is a way to guide the simulation from the start. Second, condition (v) ensures that there are no two identical **event** paths such that one leads to *Success* and another to *Failure*. For

example, if non-deterministic transitions were allowed, we could have both the **run** $(s_0, e_0, s_1, e_1, \mathit{Success})$ and $(s_0, e_0, s_1, e_1, \mathit{Failure})$, which is inconsistent (i.e., because identical situations must have identical verdicts). That this works is shown in Proposition 7.1. Finally, restriction (vi) ensures that every **state** can, in principle, lead to a verdict. Nevertheless, an inconclusive verdict might be issued owing to the presence of cycles and the finite nature of simulation runs.

A visual depiction of both a general **ATS** and a **simulation purpose** is given in Figure 7.1.

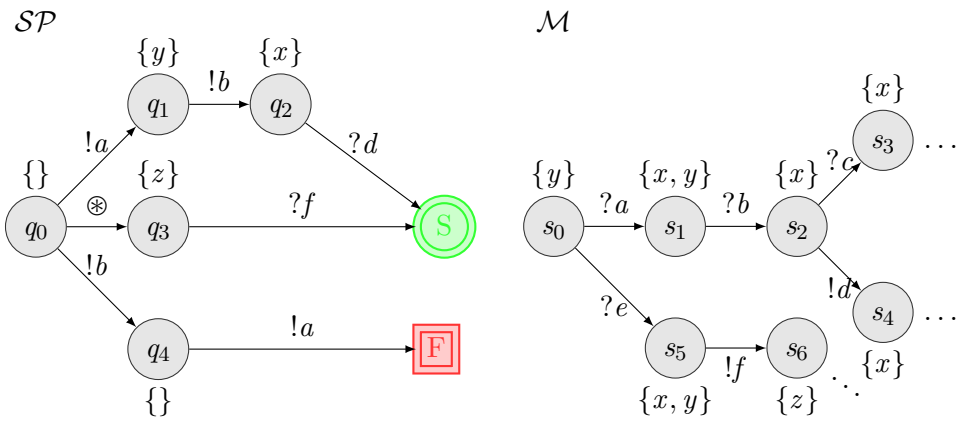


Figure 7.1: Examples of: $\mathcal{M}$, an **ATS**; and $\mathcal{SP}$, a **simulation purpose**. Transitions are annotated with **events** (i.e., $?a$, $?b$, $?c$, $?d$, $?e$, $?f$, $!a$, $!b$, $!c$, $!d$, $!e$, $!f$) and **states** are annotated with literals (i.e., x, y, z). The *Success* state is denoted by the green double circle, while the *Failure* **state** is denoted by the red double square. The dots $(\ldots)$ denote that $\mathcal{M}$ continues beyond the **states** shown (it may have infinitely many **states**).

7.3 Synchronous Product of an ATS and a SP

The idea that a **simulation purpose** can guide what runs to generate in another **ATS** is formalized by the notion of synchronization . Since both **events** and **states** contain relevant information for this guidance, a particular definition for each case is first required. With respect to **events** , it is also necessary to introduce a notion of complementarity between **events** , which allows their synchronization.

COMPLEMENTARY EVENT

Definition 7.9 (Complementary Event)**.** *Let e be an* **event**. *Then its* ***complementary event****, denoted by e^C, is defined as*

$$e^C = \begin{cases} !n & \textit{if } e = ?n \\ ?n & \textit{if } e = !n \\ \tau & \textit{if } e = \tau \\ \circledast & \textit{if } e = \circledast \end{cases}$$

Definition 7.10 (Event Synchronization)**.** *Let $\mathcal{SP} = \langle Q, E_{sp}, P_{sp}, \rightsquigarrow, L_{sp}, q_0 \rangle$ be a* **simulation purpose***, and $\mathcal{M} = \langle S, E, P, \rightarrow, L, s_0 \rangle$ be an* **ATS***. Moreover, let*

$q_1 \overset{e_1}{\rightsquigarrow} q_2$ *be a transition from $\mathcal{SP}$; and*

$s_1 \overset{e_2}{\rightarrow} s_2$ *be a transition from $\mathcal{M}$.*

Then we define that **events** *e_1 and e_2* ***synchronize*** *if, and only if, one of the following cases holds:* SYNCHRONIZE

- *$e_1 = ?n$ and $e_2 = !n$ for some name n; or*
- *$e_1 = !n$ and $e_2 = ?n$ for some name n; or*
- *$e_1 = \circledast$ and there is no $q' \in Q$ such that $q_1 \overset{e_2^C}{\rightsquigarrow} q'$; or*
- *$e_1 = e_2 = \tau$.*

Moreover, we denote the fact that e_1 and e_2 **synchronize** *by*

$$e_1 \bowtie e_2$$

The synchronization of **states** does not require a similar notion of complementarity. Its sole purpose is to check whether the **state** being considered has, or has not, a required set of propositions. The appropriate notion, therefore, is that of set inclusion.

Definition 7.11 (State Synchronization)**.** *Let $\mathcal{SP} = \langle Q, E_{sp}, P_{sp}, \rightsquigarrow, L_{sp}, q_0 \rangle$ be a* **simulation purpose***, and $\mathcal{M} = \langle S, E, P, \rightarrow, L, s_0 \rangle$ be an* **ATS***. Moreover, let $q \in Q$ be a* **state** *from $\mathcal{SP}$ and $s \in S$ be a* **state** *from $\mathcal{M}$. Then we define that q and s* ***synchronize*** *if, and only if,* SYNCHRONIZE

$$L_{sp}(q) \subseteq L(s)$$

Moreover, we denote the fact that q and s **synchronize** *by*

$$q \bowtie s$$

We may then specify the overall **synchronous product** , which, by synchronizing **events** and **states** , selects only the runs relevant for the **simulation purpose**. The result of such a product, then, is an **ATS** that contains only the relevant runs.

Definition 7.12 (Synchronous Product of a Simulation Purpose and an ATS)**.** *Let* $\mathcal{SP} = \langle Q, E_{sp}, P_{sp}, \rightsquigarrow, L_{sp}, q_0 \rangle$ *be a simulation purpose, and* $\mathcal{M} = \langle S, E, P, \rightarrow, L, s_0 \rangle$ *be an Environment ATS. Then their* ***synchronous product****, denoted as*

SYNCHRONOUS PRODUCT

$$\mathcal{SP} \otimes \mathcal{M}$$

is an **ATS** $\mathcal{M}' = \langle S', E', P', \rightarrow', L', s'_0 \rangle$ *such that:*

- $E' = E$;
- $P' = P$;
- S' *and* $\rightarrow'$ *are constructed inductively as follows:*
  - **Initial state.** $s'_0 = (q_0, s_0) \in S'$ *and* $L'(s'_0) = L(s_0)$.
  - **Other states and transitions.** *Built using the following rule*[1]*:*

$$\frac{q \overset{e_1}{\rightsquigarrow} q' \quad s \overset{e_2}{\rightarrow} s' \quad (q, s) \in S' \quad e_1 \bowtie e_2 \quad s' \bowtie q'}{(q, s) \overset{e_2}{\rightarrow'} (q', s')} \text{ SYNCH}$$

- *If* $(q', s') \in S'$*, then* $L'((q', s')) = L(s')$

This product defines the search space relevant for the algorithms. For this reason, we may refer to it as if it was completely computed. Nevertheless, algorithmically it can be built on-the-fly , and we profit from this in order to perform verification.

We refer to **runs** of **synchronous products** as **synchronous runs**.

Definition 7.13 (Synchronous Run)**.** *A* **run** *in a* **synchronous product** *is called a* ***synchronous run****.*

SYNCHRONOUS RUN

In the example of Figure 7.1, the only such **synchronous run** is the following:

$$((q_0, s_0), ?a, (q_1, s_1), ?b, (q_2, s_2), !d, (Success, s4))$$

This can be seen by examining Figure 7.2.

[1] As seen in Definitions 7.10 and 7.11, $a \bowtie b$ means that a and b **synchronize**.

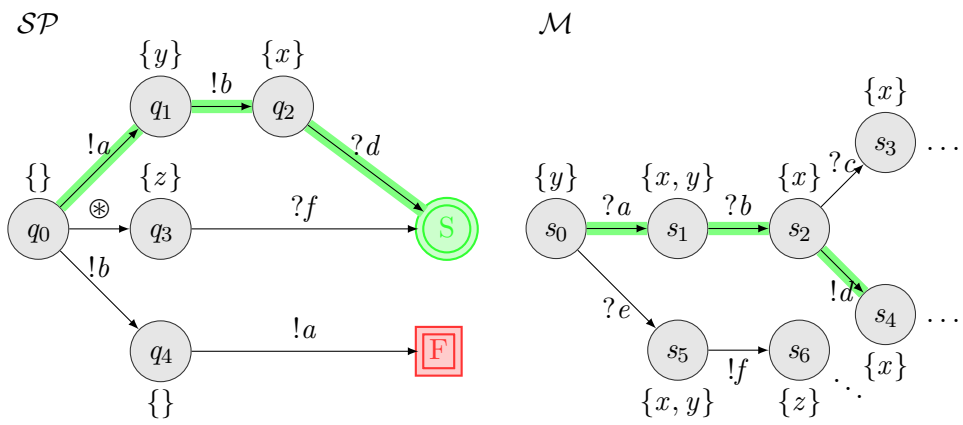


Figure 7.2: Only the shaded **runs** in the **ATS** $\mathcal{M}$ and the **simulation purpose** $\mathcal{SP}$ can **synchronize**.

7.4 Satisfiability Relations

Given an **ATS** $\mathcal{M}$ and a **simulation purpose** $\mathcal{SP}$, one is interested in whether $\mathcal{M}$ satisfies $\mathcal{SP}$. There are a number of ways in which such satisfaction can be defined, and we call each one a *satisfiability relation*.

To begin with, we may be interested in whether the **simulation purpose** is capable of conducting to a **state** of either success or failure. This is to be interpreted as the possibility of constructing an experiment, which can be used as evidence either in favour or against some hypothesis. This can be done in either a weak manner or a strong manner. In the weak manner, at each step in the experiment one is concerned only with the possibility of proceeding to a next desirable step, without forbidding other courses of action. In the strong manner, on the other hand, at each step in the experiment one may forbid certain actions, so that if they are possible the step is considered useless. These notions are formalized as follows.

Definition 7.14 (Feasibility)**.** *Let $\mathcal{SP}$ be a* **simulation purpose***, $\mathcal{M}$ be an* **ATS** *and $R \subseteq runs(\mathcal{SP} \otimes \mathcal{M})$. Then we define that:*

- *$\mathcal{SP}$ is* ***weakly feasible*** *with respect to $\mathcal{M}$ and R if, and only if, there exists a* **synchronous run** *(called* ***weakly feasible run****) $r \in R$ such that its last state (q, s) is such that $q = Success$. Otherwise, we call it* ***weakly unfeasible****;* WEAKLY FEASIBLE WEAKLY FEASIBLE RUN WEAKLY UNFEASIBLE
- *$\mathcal{SP}$ is* ***strongly feasible*** *with respect to $\mathcal{M}$ and R if, and only if, there exists a* **synchronous run** *(called* ***strongly feasible run****) $r \in R$ such that: (i) its last state (q, s) is such that $q = Success$; and (ii) there is* STRONGLY FEASIBLE STRONGLY FEASIBLE RUN

STRONGLY UNFEASIBLE

no $r' \in R$ such that it has a common **state** (q', s') *with r and $(q', s') \overset{e}{\rightsquigarrow} (Failure, s'')$ is in r', for some e and s''. Otherwise, we call it* ***strongly unfeasible****.*

FEASIBLE

FEASIBLE RUN

Moreover, if one of the two cases above hold, we say that $\mathcal{SP}$ is ***feasible*** *with respect to $\mathcal{M}$, and the corresponding* **run** *is called* ***feasible run****.*

Definition 7.15 (Refutability)**.** *Let $\mathcal{SP}$ be a* **simulation purpose***, $\mathcal{M}$ be an* **ATS** *and $R \subseteq runs(\mathcal{SP} \otimes \mathcal{M})$. Then we define that:*

WEAKLY REFUTABLE

WEAKLY REFUTABLE RUN

WEAKLY IRREFUTABLE

STRONGLY REFUTABLE

STRONGLY REFUTABLE RUN

- *$\mathcal{SP}$ is* ***weakly refutable*** *with respect to $\mathcal{M}$ and R if, and only if, there exists a* **synchronous run** *(called* ***weakly refutable run****) $r \in R$ such that its last state (q, s) is such that $q = Failure$. Otherwise, we call it* ***weakly irrefutable****;*
- *$\mathcal{SP}$ is* ***strongly refutable*** *with respect to $\mathcal{M}$ if, and only if, there exists a* **synchronous run** *(called* ***strongly refutable run****) $r \in R$ such that: (i) its last state (q, s) is such that $q = Failure$; and (ii) ii) there is no $r' \in R$ such that it has a common* **state** *(q', s') with r and $(q', s') \overset{e}{\rightsquigarrow} (Success, s'')$ is in r', for some e and s''. Otherwise, we call it* ***strongly irrefutable****.*

STRONGLY IRREFUTABLE

REFUTABLE

REFUTABLE RUN

Moreover, if one of the two cases above hold, we say that $\mathcal{SP}$ is ***refutable*** *with respect to $\mathcal{M}$, and the corresponding* **run** *is called* ***refutable run****.*

In the above definitions, the set R of **runs** to consider is left as a parameter because the notions of **feasibility** and **refutability** are applicable even in the case where not all possible **runs** (i.e., $runs(\mathcal{SP} \otimes \mathcal{M})$) can be known. Later in this chapter we show how this can be used to perform verifications with respect to the incomplete observations obtained through a simulator.

A **simulation purpose** can be both **feasible** and **refutable** with respect to the same **ATS**. In such a case, it merely means that there are experiments that lead to different verdicts, which is not a contradiction[2].

It might be interesting, though, to know whether all the experiments that follow from a **simulation purpose** lead to the same **verdict** . In this case, we are interested in establishing that all courses of action are either acceptable or unacceptable. This can be useful, for instance, if the **simulation purpose** is to model a protocol to be followed, and which, therefore, should always lead to some desirable **state**.

[2]In contrast, as remarked earlier, a contradiction would follow if *the same* experiment was to lead to different verdicts. But owing to the definition of **simulation purposes**, this can never happen, as we show in Proposition 7.1 below.

Definition 7.16 (Certainty)**.** *Let $\mathcal{SP}$ be a* **simulation purpose***, $\mathcal{M}$ be an* **ATS** *and $R \subseteq runs(\mathcal{SP} \otimes \mathcal{M})$. Then we define that $\mathcal{SP}$ is* ***certain*** *with respect to $\mathcal{M}$ if, and only if, every* **run** *in R terminates in a* **state** *(q, s) such that $q = Success$.* CERTAIN

Definition 7.17 (Impossibility)**.** *Let $\mathcal{SP}$ be a* **simulation purpose***, $\mathcal{M}$ be an* **ATS** *and $R \subseteq runs(\mathcal{SP} \otimes \mathcal{M})$. Then we define that $\mathcal{SP}$ is* ***impossible*** *with respect to $\mathcal{M}$ if, and only if, every* **run** *in R terminates in a* **state** *(q, s) such that $q = Failure$.* IMPOSSIBLE

Clearly, a **simulation purpose** only has value if it is capable of reaching some of its terminal **states** . Depending on the structure of the **ATS** to be verified, this might not happen in their **synchronous product**. In such a case the **simulation purpose** is incapable of providing information about the **ATS**. Therefore, there is an important notion of **informativeness** that we wish to attain.

Definition 7.18 (Informativeness)**.** *Let $\mathcal{SP}$ be a* **simulation purpose** *and $\mathcal{M}$ be an* **ATS***. Then we define that $\mathcal{SP}$ is* ***informative*** *with respect to $\mathcal{M}$ if, and only if, it is either* **feasible** *or* **refutable***. Otherwise, $\mathcal{SP}$ is said to be* ***uninformative***. INFORMATIVE UNINFORMATIVE

We claimed in Section 7.2 that **simulation purposes** avoid non-determinism in order to provide consistent verdicts. We now carefully justify this.

Proposition 7.1 (Consistency)**.** *Let $\mathcal{SP}$ be a* **simulation purpose***, $\mathcal{M}$ be an* **ATS** *and $Prod = \mathcal{SP} \otimes \mathcal{M}$. Moreover, let t_1 and t_2 be* **runs** *of $Prod$ which share a* **subrun** *t and an* **event** *e such that*

- $t_1 = t.e.(q_n^1, s_n^1)$
- $t_2 = t.e.(q_n^2, s_n^2)$

Then, we have that $q_n^1 = q_n^2$.

Proof. Let $n - 1$ be the length of **subrun** t, and (q_{n-1}, s_{n-1}) be its last **state**. By hypothesis, $(q_{n-1}, s_{n-1}) \xrightarrow{e} (q_n^1, s_n^1)$ is a transition in *Prod*. Then, because of the determinism requirement on **simulation purposes**, it follows that there exists exactly one transition in $\mathcal{SP}$ from q_{n-1} using the **event** e, namely, $q_{n-1} \overset{e}{\rightsquigarrow} q_n^1$. Therefore, if $(q_{n-1}, s_{n-1}) \xrightarrow{e} (q_n^2, s_n^2)$ is also a transition in *Prod*, it must be the case that it arises from the **synchronization** with same transition in $\mathcal{SP}$, which implies that $q_n^1 = q_n^2$. □

This result does not depend on determinism concerning $\mathcal{M}$, but only $\mathcal{SP}$.

7.5 Verification Algorithms

The satisfiability relations presented in the previous section can be verified by analysing the **synchronous product** given by Definition 7.12. The required algorithms, moreover, are all very similar: they all consist in a depth-first search on this product. For this reason, we first present and analyse extensively the algorithm for checking **feasibility** (Algorithm 1). By a trivial modification of the input, the same algorithm can be used to check **refutability**. We then define the algorithm for checking **certainty** (Algorithm 2), which is very similar to Algorithm 1, but do require some subtle adjustments. Again, by a trivial modification of the input, Algorithm 2 can also be used to check the remaining **impossibility** relation. It is therefore only necessary to provide these two algorithms to verify all the satisfiability relations defined previously. Both algorithms require the existence of a simulator interface to interact with, so we begin by introducing it.

7.5.1 Simulator Interface

Before we proceed to the verification algorithms themselves, it is necessary to introduce a way for them to access the simulation infrastructure. We do this here by specifying a number of operations that the simulator must make available to the verification algorithms. This means that any simulator that provides this interface can be used to implement the verification technique presented in this chapter.

The required simulator interface is composed by the following operations:

- `GoToState(`*sim*`)`: Makes the simulation execution return to the specified simulation **state** *sim*, which must have taken place previously.
- `CurrentState()`: Returns the current simulation state.
- `ScheduleStep(`*e*`)`: Schedules the specified **event** *e* for simulation.
- `Step()`: Requests that all scheduled **events** get simulated.
- `isCommitEvent(`*e*`)`: Checks whether *e* is an **event** that serves as a signal of when it is appropriate to call the `Step()` operation. Such an **event** can be thought of as a clock used by the simulator. If the simulator does not employ any such special **event**, then this operation always returns **true**.
- `Successors(`*ATS, s*`)`: Calculates the finite set of all transitions in the specified `ATS` that can be simulated and that have the **state** `s` as their

origin. This operation is necessary to allow the on-the-fly construction of `ATS`.

7.5.2 Feasibility Verification

Algorithm 1 implements **feasibility** verification. Besides the simulator operations described above, it also assumes that the following simple elements are available:

- `CanSynch(`$q \overset{f}{\rightsquigarrow} q'$, $s \overset{g}{\rightarrow} s'$`)`: Checks whether the two specified transitions can **synchronize** according to Definition 7.12.
- $depth_{max}$: The maximum depth allowed in the search tree. Note that since simulations are always finite (i.e., they must stop at some point), we can assume that $depth_{max}$ is finite.
- **max_int**: A very large integer (in most common programming languages, this can be the maximum integer available, thus the name), which is used to indicate distances that can be considered as infinite. It is assumed that in a **simulation purpose** all shortest paths from any **state** to **verdict states** are less than **max_int**.

The remaining procedures required for the algorithm are given explicitly after it.

A careful and detailed investigation of the correctness and the complexities of Algorithm 1 is provided in Section 7.6. Before proceeding to this, however, let us explore more informally how the algorithm works and the related issues.

How the Algorithm Works First of all, a preprocessing of the **simulation purpose** is required. This consists in calculating how far from *Success*, the desired **verdict state**, each of the **states** in the **simulation purpose** is. By this provision, we are able to take the shortest route from any given **simulation purpose state** towards *Success*. The importance of such a route is that it avoids cycles whenever possible, which is crucial to prevent the algorithm from entering in infinite loops later on. For every **simulation purpose state** q, then, its distance to the desired **verdict state** is stored in $dist[q]$. However, if one is checking **strong feasibility**, $dist[Failure]$ is set to -1, so that later the algorithm will always find a successor that leads to *Failure* before any other successor (in order to discard it promptly, and thereby respect the **strong** variant of **feasibility**).

Once this preprocessing is complete, the algorithm performs a depth-first search on the **synchronous product** $\mathcal{SP} \otimes \mathcal{M}$. The central structure to

Algorithm 1: On-the-fly verification of feasibility

Input: A **simulation purpose** $\mathcal{SP} = \langle Q, E_{sp}, P_{sp}, \rightsquigarrow, L_{sp}, q_0 \rangle$, the initial **state** s_0 of an **ATS** $\mathcal{M}$ and a *variant* $\in \{weak, strong\}$.

Output: `SUCCESS` and a **feasible run**; `FAILURE`; or `INCONCLUSIVE`.

```
1  dist[] := Preprocess(SP, Success);
2  if variant = strong then dist[Failure] = −1;
3  let SynchStack be an empty stack;
4  let sim_0 := CurrentState() ;
5  let Unexplored_0 := Successors(SP, q_0);
6  Push (q_0, s_0, nil, nil, sim_0, Unexplored_0, 0) on SynchStack;
7  let verdict := FAILURE;
8  while SynchStack ≠ ∅ do
9      Peek (q, s, e, p, sim, Unexplored, depth) from SynchStack;
10     let progress := false;
11     while Unexplored ≠ ∅ ∧ progress = false ∧ depth < depth_max do
12         q ⇝^f q' := RemoveBest(Unexplored, dist[]);
13         let depth' := depth + 1;
14         let Succs := Successors(M, s);
15         while Succs ≠ ∅ do
16             Remove some s -g-> s' from Succs;
17             GoToState(sim);
18             ScheduleStep(g);
19             if isCommitEvent(g) then
20                 Step();
21             if CanSynch(q ⇝^f q', s -g-> s') then
22                 if q' = Failure ∧ variant = strong then
23                     Pop from SynchStack;
24                     progress := true;
25                     Succs := ∅;
26                 else
27                     sim' := CurrentState();
28                     unexplored' := Successors(SP, q');
29                     Push (q', s', g, q, sim', unexplored', depth') on
                       SynchStack;
30                     progress := true;
31                 if q' = Success then
32                     return SUCCESS and BuildRun(SynchStack, q',
                       depth');
33     if depth ≥ depth_max then verdict := INCONCLUSIVE;
34     if progress = false then Pop from SynchStack;
35 return verdict;
```

Procedure Preprocess($\mathcal{SP}$, v)

Input: A **simulation purpose** $\mathcal{SP} = \langle Q, E, P, \rightsquigarrow, L, q_0 \rangle$ and a **verdict state** v.

Output: A function $dist : Q \longrightarrow \mathbb{Z}$ such that, for every $q \in Q$, $dist[q]$ is the minimal distance between q and v.

```
1  let visited[] be a map from states to boolean values;
2  let dist[] be a map from states to either integers or nil;
3  foreach q ∈ Q do
4      visited[q] := false;
5      dist[q] := nil;
6  PreprocessAux(SP, q0, v, dist[], visited[]);
7  foreach q ∈ Q do
8      visited[q] := false;
9  PreprocessAux(SP, q0, v, dist[], visited[]);
10 return dist[];
```

achieve this is the stack *SynchStack*. Every time a successful **synchronization** between a transition in $\mathcal{SP}$ and one in $\mathcal{M}$ is reached, information about it is pushed on this stack. The pushed information is a tuple containing the following items:

- The **state** q of $\mathcal{SP}$ that **synchronized**.
- The **state** s of $\mathcal{M}$ that **synchronized**.
- The **event** e of $\mathcal{M}$ that **synchronized**. This will be used later to calculate a **synchronous run**.
- The **state** p of $\mathcal{SP}$ that came immediately before the one that has been **synchronized** (i.e., $p \overset{e}{\rightsquigarrow} q$). Again, this will be used later to calculate a **run**.
- The state of the simulation, *sim*, which can be extracted from the simulator using the `CurrentState()` function.
- The set of transitions starting at q (i.e., *Unexplored* = `Successors(`$\mathcal{SP}$, q`)`) which have not been explored yet.
- The depth in the search tree.

In the beginning, we assume that the **initial states** of both transition systems synchronize and push the relevant initial information on *SynchStack*. Thereafter, while there is any tuple on the stack, the algorithm will systematically

Procedure PreprocessAux($\mathcal{SP}$, *source*, v, *dist*, *visited*)

Input: A **simulation purpose** $\mathcal{SP} = \langle Q, E, P, \rightsquigarrow, L, q_0 \rangle$, a *source* $\in Q$, a map *dist*[], a map *visited*[] and a **verdict state** v.

```
1  visited[source] = true;
2  if source = v then
3      dist[source] := 0;
4  else
5      let min := nil;
6      if Successors(SP, source) = ∅ then
7          min := max_int ;
8      else
9          foreach source ~f~> q' do
10             if visited[q'] = false then
11                 PreprocessAux(SP, q', v, dist[], visited[]);
12             if dist[q'] ≠ nil then
13                 if min = nil then
14                     min := dist[q']
15                 else if dist[q'] < min then
16                     min := dist[q'];
17         if min ≠ nil ∧ min ≠ max_int then
18             min := min + 1;
19     dist[source] := min;
```

Procedure RemoveBest(*Unexplored*, *dist*)

Input: A set *Unexplored* of transitions of a **simulation purpose** and a map *dist*[] from of **states** to integers.

Output: A transition in *Unexplored*.

```
1  let q ~e~> q' ∈ Unexplored such that there is no q ~e~> q'' ∈ Unexplored
   with dist[q''] < dist[q'];
2  return q ~e~> q'
```

Procedure BuildRun(SynchStack, q^*, $depth^*$)

Input: A search stack *SynchStack*, the last **simulation purpose state** q^* to include in the **run** to be built, and the depth $depth^*$ of this **state**.

Output: A **synchronous run**.

```
1 let Run be a list initially empty;
2 while SynchStack ≠ ∅ do
3     Pop (q', s', f, q, sim, Unexplored, depth) from SynchStack;
4     if q' = q* ∧ depth* = depth then
5         Put (q', s') at the beginning of Run;
6         Put f at the beginning of Run;
7         q* := q;
8         depth* := depth* − 1;
9 return Run;
```

examine it. It peeks the topmost tuple, $(q, s, e, p, sim, Unexplored, depth)$, and access its *Unexplored* set. These are **simulation purpose** transitions beginning in q which have not yet been considered at this point. The algorithm will examine each of these transitions while: (i) no **synchronization** is possible (i.e., the variable *progress* is **false**); and (ii) the search depth is below some maximum limit $depth_{max}$. In case (i), the rationale is that we wish to proceed with the next **synchronized state** as soon as possible, so once we find a **synchronization**, we move towards it. If that turns out to be unsuccessful, the set *Unexplored* will still hold further options for later use. In case (ii), we are merely taking into account the fact that there are situations in which the algorithm could potentially go into an infinite depth. For instance, $\mathcal{SP}$ could contain a cycle that is always taken because no other **synchronizations** are possible starting from the beginning of this cycle. The depth limit provides an upper-bound in such cases, and forces the search to try other paths which might lead to a **feasible run**, instead an infinite path.

In each iteration of this while loop, the algorithm selects the best transition $q \overset{f}{\rightsquigarrow} q'$ available in *Unexplored*. This selection employs the preprocessing of the **simulation purpose**, and merely selects the transition that is closer to the goal. That is to say, q' is such that there is no $q \overset{f}{\rightsquigarrow} q''$ such that $dist[q''] < dist[q']$. As we remarked above, this is intended to guide the search through the shortest path in order to avoid cycles whenever possible. Once such a transition is chosen, we may examine all possible transitions of $\mathcal{M}$ starting at s, the current **synchronized state**.

At this point, the simulator interface will be of importance. For each such transition $s \overset{g}{\rightarrow} s'$, we have to instruct the simulator to go to the simulation

state *sim* in the peeked tuple. This simulation state holds the configuration of the structures internal to the simulator that correspond to the transition system **state** s. The algorithm may then request that the **event** g be scheduled. Then, if the **event** turns out to be a commit **event**, the simulator is instructed to perform one simulation step, which implies in delivering all the scheduled **events** . This will put the simulator into a new state, which will correspond to s' in $\mathcal{M}$. Then we may check whether $q \overset{f}{\rightsquigarrow} q'$ and $s \overset{g}{\rightarrow} s'$ can **synchronize**. If it is possible, then q' is *Success*, *Failure* or another **state** in Q. In the first case we have found the **feasible run** we were looking for and we are done. The second case only matters if we are checking the **strong** variant of **feasibility**, in which case we must discard the current synchronization **state** (by popping it from *SynchStack* and going to the next one) because it has led to a *Failure* and thus according to the definition of **strong feasibility** cannot be part of the **strong feasible run**. In the the third case, we merely push the current $(q', s', g, q, sim', unexplored', depth')$ tuple on *SynchStack* for later analysis and signal that the search can move on by setting *progress* to **true**.

If the algorithm abandons a search branch because its depth is greater or equal to $depth_{max}$, the verdict in case of failure is set to be `INCONCLUSIVE`, since it is possible that there was a **feasible run** with length greater than $depth_{max} + 1$ that was not explored. Moreover, it is possible that after examining all transitions in *Unexplored*, none **synchronized** (i.e., the variable *progress* is still set to **false**). If this happens, the tuple is popped from *SynchStack* because by then we are sure that no **feasible run** will require it.

At last, if *SynchStack* becomes empty, it means that no **run** in $\mathcal{SP}$ up to $depth_{max}$ leads to a **feasible run**. So we return a verdict which will be `FAILURE` if $depth_{max}$ was never reached, or `INCONCLUSIVE` otherwise.

How the Algorithm Handles Cycles We have just said that a preprocessing of $\mathcal{SP}$ is required in order to deal with its cyclic paths. By this provision, we are able to determine at any **state** of $\mathcal{SP}$ which successor is closer to *Success*, the desired **verdict state**. Since any cyclic path from a **state** is longer than an acyclic one from the same **state**, this suffices to avoid cycles whenever possible. That said, let us see how this minimum distance is calculated by `Preprocess()`.

The calculation is divided between the main `Preprocess()` procedure and the auxiliary `PreprocessAux()` procedure. Indeed, `Preprocess()` merely: (i) initializes two maps, *visited*[] and *dist*[], which stores whether a **state** has been visited and the distance from a **state** to the desired verdict, respectively; and (ii) call `PreprocessAux()` twice. In the first call, all the acyclic paths are examined and have the corresponding *dist*[] values set. In the second call,

using this partial $dist[\,]$, `PreprocessAux()` is then capable of computing the distances for the **states** in cyclic paths as well.

`PreprocessAux()` is a function that for a given *source* **state** recursively examines all of its successor **states** q' to determine which one is closer to the desired verdict **state** v. Once the closer successor q^* is found, the function merely sets $dist[source] := dist[q^*]+1$. The recursion base takes place in three situations. First, when the *source* being examined is actually the verdict **state** v, and therefore its distance is 0. Second, if the *source* being considered has no successors, and thus cannot get to v, which implies that $dist[source]$ is infinite. And third, when all successors of *source* have already being visited.

In the latter case it means that the procedure has found a cycle. Moreover, because of the recursive nature of the procedure, none of these successors q' will have their $dist[q']$ set yet, so that $dist[source]$ remains **nil**, which indicates that *source* is in a cycle. However, when `Preprocess()` calls `PreprocessAux()` a second time, these $dist[q']$ will be set, so that even in the case in which *source* is in a cycle, we are able to assign it a distance. This distance, indeed, is nothing but the sum of an acyclic path and the length of the corresponding cycle. That is to say, for any *source* located in a cyclic path, the procedure assign it the shortest distance considering a way to get out of the cycle. Since by Definition 7.8 there is always an acyclic path towards a **verdict state**, it is always possible to calculate this distance. This does not prevent Algorithm 1 from taking such a cycle infinitely, but merely provides guidance to avoid it whenever possible.

Hints on Termination The complete treatment concerning termination is given in Section 7.6.3. Let us nonetheless give some hints on this matter here. The termination of Algorithm 1 depends on whether $\mathcal{SP}$ is cyclic. If it is not, termination is always guaranteed because only finitely many **states** of $\mathcal{SP}$ are visited during the depth-first search (i.e., no previously visited **state** of $\mathcal{SP}$ would be revisited). However, if there are cycles in $\mathcal{SP}$, it would in principle be possible that infinitely many **states** were to be visited (since the same **state** in $\mathcal{SP}$ could **synchronize** infinitely many times with **states** in $\mathcal{M}$), which of course would compromise termination.

In such a cyclic case, the crucial factor that determines termination is the value of $depth_{max}$. Since $depth_{max}$ is assumed to be finite (i.e., the search is bounded), termination is guaranteed, because: (i) there are only finite many paths in $\mathcal{SP}$ of length $depth_{max}$; and (ii) each such path is used only once. If the search was not bounded in this manner, such a guarantee could not be given. But as explained previously, by their very nature simulations must be finite, so the assumption of a bounded search is actually necessary.

Hints on Complexity The exact analysis of the complexities is provided in Section 7.6.5. Here it suffices to say that the complexity in space is polynomial with respect to the size of the **environment**, $depth_{max}$ and other parameters, and the complexity in time is exponential with respect to $depth_{max}$. This exponential complexity in time arises from the fact that the algorithm do not keep track of the visited **states** of $\mathcal{M}$.

7.5.3 Refutability Verification

To verify **refutability**, one must look for a **run** leading to *Failure* instead of a **run** leading to *Success*. Therefore, it suffices to swap the *Success* and *Failure* **states** in the **simulation purpose** and then apply Algorithm 1.

7.5.4 Certainty Verification

The verification of **certainty** can be achieved by a slightly modified version of Algorithm 1. Like for **refutability**, it should search for the *Failure* **state**. However, when it does find it, the final verdict is `FAILURE`, because by Definition 7.16 there should be no **run** leading to *Failiure*. Moreover, for every visited transition $q \overset{f}{\rightsquigarrow} q'$ of $\mathcal{SP}$, there must be a **synchronizable** $s \overset{g}{\rightarrow} s'$. Otherwise, there would be a terminal **state** (q', s') such that $q' \neq Success$, which is forbidden by Definition 7.16. Finally, when the outer loop terminates, the verdict to be returned is either `SUCCESS` or `INCONCLUSIVE` (in case the search depth reached the maximum allowed), because no counter-examples have been found. Algorithm 2 below incorporates these changes. The complexities remain the same, because these modifications do not change the arguments used to calculate them.

7.5.5 Impossibility Verification

To verify **impossibility**, one must find that all runs in the **synchronous product** lead to *Failure*. It suffices then to swap the *Success* and *Failure* **states** in the **simulation purpose** and then apply Algorithm 2. This is similar to the verification of **refutability**, which can be accomplished by Algorithm 1 through such a swap.

7.6 Analysis of the Algorithms

Let us now provide a more rigorous account of whether the algorithms are correct, and of how much resources they employ. To this end, we investigate their

Algorithm 2: On-the-fly verification of certainty

Input: A **simulation purpose** $\mathcal{SP} = \langle Q, E_{sp}, P_{sp}, \rightsquigarrow, L_{sp}, q_0 \rangle$ and the initial **state** s_0 of an **ATS** $\mathcal{M}$.

Output: `SUCCESS`; `FAILURE` and a **refutable run**; or `INCONCLUSIVE`.

```
1  dist[] := Preprocess(SP, Success);
2  let SynchStack be an empty stack;
3  let sim_0 := CurrentState() ;
4  let Unexplored_0 := Successors(SP, q_0);
5  Push (q_0, s_0, nil, nil, sim_0, Unexplored_0, 0) on SynchStack;
6  let verdict := SUCCESS;
7  while SynchStack ≠ ∅ do
8      Peek (q, s, e, p, sim, Unexplored, depth) from SynchStack;
9      while Unexplored ≠ ∅ ∧ depth < depth_max do
10         q ~f~> q' := RemoveBest(Unexplored, dist[]);
11         let depth' := depth + 1;
12         let progress := false;
13         let Succs := Successors(M, s);
14         while Succs ≠ ∅ do
15             Remove some s -g-> s' from Succs;
16             GoToState(sim);
17             ScheduleStep(g);
18             if isCommitEvent(g) then
19                 Step();
20             if CanSynch(q ~f~> q', s -g-> s') then
21                 sim' := CurrentState();
22                 unexplored' := Successors(SP, q');
23                 Push (q', s', g, q, sim', unexplored', depth') on SynchStack;
24                 progress := true;
25                 if q' = Failure then
26                     return FAILURE and BuildRun(SynchStack, q',
                       depth');
27         if progress = false then
28             return FAILURE and BuildRun(SynchStack, q, depth);
29     if depth ≥ depth_max then  verdict := INCONCLUSIVE;
30     Peek (q, s, e, p, sim, Unexplored, depth) from SynchStack;
31     if Unexplored = ∅ then  Pop from SynchStack;
32 return verdict;
```

soundness, completeness (notions which must be defined) and worst-case complexities. Although the detailed arguments presented in this section are not *formal* proofs, we have nonetheless organized them in a typical mathematical fashion, by separating the arguments in definitions, propositions and auxiliary lemmas. The proofs given should be understood as careful justifications.

As noted in the previous section, the presented algorithms are very similar to each other. Owing to this, here we provide a detailed analysis only of Algorithm 1, and then we explain how Algorithm 2 differs.

To reason about soundness and completeness, it is first necessary to establish two things: what entities are to be evaluated; and what is the standard against which this evaluation is to be made. In this thesis, the entities to be evaluated are Algorithms 1 and 2, and ideally the standard to be used would be the **synchronous product** $\mathcal{SP} \otimes \mathcal{M}$ that arises from a **simulation purpose** $\mathcal{SP}$ and a **concrete environment ATS** $\mathcal{M}$. That is to say, an algorithm would be considered: *sound* if whenever it issues a `SUCCESS` or `FAILURE` verdict, the desired satisfiability relation holds or does not hold, respectively, in relation to $runs(\mathcal{SP} \otimes \mathcal{M})$; and *complete* if whenever the desired satisfiability relation holds or does not hold with respect to $runs(\mathcal{SP} \otimes \mathcal{M})$, it issues a `SUCCESS` or `FAILURE` verdict.

However, this ideal standard is very strong for the problems considered in this thesis, which are based on finite simulations on the presence of evolving and autonomous agent behaviour. Thus, this ideal notion of completeness is out of the scope of the technique considered here. The same is true for soundness, for the fact that certain outcomes cannot be observed influences the evaluation of satisfiability relations of interest. These issues are inherent to the verification of the satisfiability relations with respect to **EMMAS environments** by means of simulations, and not particular to the algorithmic solutions given in the present chapter.

This difficulty can be addressed in three ways: (i) by removing the autonomy and adaptation capabilities of the agents; (ii) by making the properties to be verified independent of the actions of the agents; or (iii) by using a weaker standard for soundness and completeness. Option (i) is discarded, because autonomy and adaptation, even though they bring a number of problems, are fundamental characteristics of the agents that this thesis is concerned with. Option (ii) is also discarded, because questions about the actions of agents are essential to the verification of MASs – otherwise, the agents would be irrelevant, and there would be no MAS in the first place. Thus we are left with option (iii).

Let us then introduce the notions of **observational soundness** and **observational completeness**. To do so, it is first necessary to define the notion of **observed runs**.

Definition 7.19 ($runs^{\mathcal{A}}_{obs}()$ Function)**.** *Let $\mathcal{TS}$ be an* **ATS***, and $\mathcal{A}$ an algorithm. Then the set of* ***observed runs*** *of $\mathcal{A}$ is denoted by $runs^{\mathcal{A}}_{obs}(\mathcal{TS})$ and such that:*

OBSERVED RUNS

$$runs^{\mathcal{A}}_{obs}(\mathcal{TS}) \subseteq runs(\mathcal{TS})$$

This denotes the set of the actually encountered **runs**, among all possible **runs**.

Definition 7.20 (Observational Soundness)**.** *Let $\mathcal{SP}$ be a* **simulation purpose** *and $\mathcal{M}$ be a* **concrete environment ATS***. Then an algorithm is* ***observationally sound*** *if whenever it issues:*

OBSERVATIONALLY SOUND

- *a* `SUCCESS` *verdict, the desired satisfiability relation holds in relation to $runs^{\mathcal{A}}_{obs}(\mathcal{SP} \otimes \mathcal{M})$;*
- *a* `FAILURE` *verdict, the desired satisfiability relation does not hold in relation to $runs^{\mathcal{A}}_{obs}(\mathcal{SP} \otimes \mathcal{M})$, even if the* **observed runs** *could be made longer;*
- *an* `INCONCLUSIVE` *verdict, the desired satisfiability relation does not hold in relation to $runs^{\mathcal{A}}_{obs}(\mathcal{SP} \otimes \mathcal{M})$, but it could perhaps hold if the* **observed runs** *could be made longer.*

Definition 7.21 (Observational Completeness)**.** *Let $\mathcal{SP}$ be a* **simulation purpose***, $\mathcal{M}$ be a* **concrete environment ATS** *and l the maximum length of* **runs** *in $runs^{\mathcal{A}}_{obs}(\mathcal{SP} \otimes \mathcal{M})$. Then an algorithm is* ***observationally complete up to*** *l if whenever the desired satisfiability relation holds or does not hold with respect to $runs^{\mathcal{A}}_{obs}(\mathcal{SP} \otimes \mathcal{M})$, it issues a* `SUCCESS`*,* `FAILURE` *or* `INCONCLUSIVE` *verdict.*

OBSERVATIONALLY COMPLETE UP TO

Each particular simulation execution provides a certain number of observations concerning the actions of the agents. Since one cannot know what all the possible observations are, the next best thing is to be sound and complete with respect to the observations made. The `INCONCLUSIVE` verdict is added to account for the case in which a mere extension of the observations could have sufficed to find a **run** of interest.

Algorithms 1 and 2 are designed with this goal in mind. They systematically investigate all the possible **runs** that arise from the observations made in the course of exploring the **concrete environment ATS** up to a certain length. In fact, both Algorithms 1 and 2 are **observationally sound**, are **observationally complete**, and terminate. It turns out, moreover, that when Algorithm 1 outputs a `SUCCESS` verdict, this verdict is also sound in the ideal sense, not only the **observational** one. The same is true with respect to the `FAILURE` verdict for Algorithm 2.

These results are shown to be true in Section 7.6.1 (completeness), Section 7.6.2 (soundness) and Section 7.6.3 (termination). In Section 7.6.4 these several analyses are grouped in order to establish the correctness of the algorithms. The worst-case complexities, in turn, are provided in Section 7.6.5. In all of the analyses below, definitions, lemmas and propositions are given in the order that they are needed (i.e., bottom-up).

7.6.1 Justification of Completeness

There are two problems that hinders the completeness of the algorithms. The first is that the search tree may have branches of infinite length, which forces the imposition of a maximum depth to guarantee termination, and that is why the `INCONCLUSIVE` verdict exists. The second is that the search can only be made with respect to **observed synchronous runs**, which may not contain all the relevant **synchronous runs**. Nevertheless, both algorithms are still **observationally complete up to** $depth_{max} + 1$. This is carefully stated and proved below.

7.6.1.1 Justification of Completeness of Algorithm 1

Proposition 7.2 (Observational Completeness of Algorithm 1 up to $depth_{max}+1$)**.** *If there are* **feasible runs** *of length less than or equal to* $depth_{max} + 1$ *in* $runs^1_{obs}(\mathcal{SP} \otimes \mathcal{M})$ *(i.e.,* **feasibility** *holds with respect to the* **observed runs***), Algorithm 1 will find a minimal length one among them and return the verdict* `SUCCESS`*.*

Proof. Let n be the length of an arbitrary minimal length **feasible run** such that $n \leq depth_{max} + 1$. We prove by induction on n that the algorithm will find some **feasible run** t such that $\mid t \mid= n$.

Base ($n = 2$): Since the initialization phase of the algorithm pushes one tuple on *SynchStack*, and $q_0 \overset{f}{\rightsquigarrow} Success \in$ `Successors(`$\mathcal{SP}$, q_0`)` for some **event** f (because, by hypothesis, t is feasible), it follows that line 12 is reached. By the construction of `RemoveBest`, it always removes the transition which composes the shortest path to *Success*, or a transition that leads to *Failure* (in the case of **strong feasibility**). But in this case we are assuming that the **feasible run** exists, so the transition chosen is $q_0 \overset{f}{\rightsquigarrow} Success$. Finally, since by Definition 7.11 *Success* synchronizes with any s', it follows that the if block of line 31 is reached. Therefore, the **run** $t = ((q_0, s_0), f, (Success, s'))$ is found and returned together with the `SUCCESS` verdict in line 32. And t is minimal because there is no **feasible run** of shorter length, as any **run** must contain at least a_0 and *Success*.

Induction Hypothesis: Let σ be a **subrun**.
If $t = \sigma.(p,s).e.(q,s').f.(Success, s'')$ is a **feasible run** of minimal length, then $r = \sigma.(p,s).e.(Success, s')$ *would* have been a minimal length **feasible run** if the transition $p \overset{e^C}{\rightsquigarrow} Success$ was added to $\mathcal{SP}$.

Inductive Step ($n > 2$): By the inductive hypothesis, there could be a minimal length **feasible run** $r = \sigma.(p,s).e.(Success, s')$ such that $t = \sigma.(p,s).e.(q,s').f.(Success, s'')$ for **events** e and f, and **states** (p,s) and (q,s'), if $\mathcal{SP}$ was properly modified. This r could be obtained when the if block of line 31 was reached. In the present case, however, $q' \neq Success$, and as a consequence the algorithm continues running from that point on. Since *progress* = **true**, the while loop (lines 11 – 32) will not iterate again, and execution continues from line 8. The algorithm has just pushed a tuple on *SynchStack*, so it is not empty. Indeed, because of line 9, this tuple will be immediately analysed by the while loop (lines 11 – 32). As in the induction basis, since `RemoveBest` picks the transition that leads to the shortest path to *Success*, it follows that there is an **event** f such that $q \overset{f}{\rightsquigarrow} Success$ will be chosen, and therefore the if block of line 31 will execute and return a **feasible run** together with the `SUCCESS` verdict. Moreover, since $\mid t \mid = \mid r \mid +1$ and by hypothesis r was of minimal length, it follows that t is of minimal length.

□

7.6.1.2 Justification of Completeness of Algorithm 2

Proposition 7.3 (Observational Completeness of Algorithm 2 up to $depth_{max}+1$)**.** *If there are* **synchronous runs** *of maximal length less than or equal to* $depth_{max}+1$ *in* $runs^1_{obs}(\mathcal{SP} \otimes \mathcal{M})$ *such that in their last* **state** (q,s), $q \neq Success$ *(i.e.,* **certainty** *does not hold with respect to the* **observed runs***), then Algorithm 2 will find the shortest among them and return the verdict* `FAILURE`*.*

Proof. Follows by a proof similar to that of Proposition 7.2. There are two differences: (i) Algorithm 2 is searching for the *Failure* verdict, and not *Success*; (ii) if no further **synchronization** is possible at some point, this suffices to provide a **run** whose last **state** (q,s) is such that $q \neq Success$. □

7.6.2 Justification of Soundness

In what follows we investigate the soundness properties of Algorithms 1 and 2. This requires first the consideration of the auxiliary procedures that they employ.

7.6.2.1 Justification of Correctness of Auxiliary Procedures

The verification algorithms depend on an important auxiliary procedure for their initialization, `Preprocess()`, and therefore we investigate its correctness[3] first.

To begin, some auxiliary lemmas are required.

Lemma 7.1 (Correctness of `PreprocessAux()` for Acyclic Paths)**.** *Let* $\mathcal{SP} = \langle Q, E, P, \rightsquigarrow, L, q_0 \rangle$ *be a* **simulation purpose**, *source* $\in Q$, *and* v *a* **verdict state**. *Moreover, for every* $q \in Q$, *let the following initial conditions hold:* $dist[q] = \mathbf{nil}$ *and* $visited[q] = \mathbf{false}$. *Then, after the execution of* `PreprocessAux(`$\mathcal{SP}$, *source*, v, $dist[\,]$, $visited[\,]$`)` *we have that for every* $p \in Q$ *that is part of an acyclic path of* $\mathcal{SP}$, $dist[p]$ *will be the minimal distance between* p *and* v.

Proof. The proof is by induction on the possible distances towards v, denoted by n.

Base ($n = 0$)*:* This can only be the case if $source = v$, which is a (trivial) acyclic path, and in which case the algorithm will correctly attribute 0 to $dist[source]$ in line 3.

Induction Hypothesis: Let n be the minimal distance from *source* to v. Then, if *source* is in an acyclic path, there exists a successor q' such that $n > dist[q'] \in \mathbb{N}$ after executing `PreprocessAux(`$\mathcal{SP}$, q', v, $dist[\,]$, $visited[\,]$`)`.

Inductive Step($n > 0$)*:* If *source* has no successors, it cannot possibly reach v. Hence, $dist[source]$ is made infinite in lines 7 and 19. On the other hand, if there are successors, then each such successor q' that has not yet been visited will be recursively subject to `PreprocessAux()` in line 11. By the induction hypothesis, this implies that if $dist[q'] \in \mathbb{N}$, then $dist[q']$ is the minimal distance from q' to v. Lines 12 – 16, in turn, select the minimal distance among all such successors and store it in *min*. Since the distance from *source* to any of its successors q' is 1, $dist[source]$ clearly must be $min + 1$, provided that *min* could be calculated at all (i.e., $min \neq \mathbf{nil}$). Thus, line 19 correctly attributes the minimal distance to $dist[source]$, if it can be calculated. If, however, $min = \mathbf{nil}$, it means, again by the induction hypothesis, that no successor of *source* is in an acyclic path, and therefore neither is *source*. So there is no need to attribute a number to $dist[source]$.

□

[3] The division of correctness in soundness and completeness only makes sense if there is something external to the algorithms to used as a standard. The auxiliary procedures require no such standard, and therefore we refer to the fact that they work as specified merely as correctness.

Lemma 7.2 (Correctness of `PreprocessAux()` for Cyclic Paths)**.** *Let* $\mathcal{SP} = \langle Q, E, P, \rightsquigarrow, L, q_0 \rangle$ *be a* **simulation purpose**, *source* $\in Q$, *and* v *a* **verdict state**. *Moreover, for every* $p \in Q$ *such that* p *is in an acyclic path of* $\mathcal{SP}$, *let the following initial conditions hold:* $dist[p] \neq$ **nil** *and* $visited[p] =$ **false**. *Then, after the execution of* `PreprocessAux(`$\mathcal{SP}$, *source*, v, $dist[\,]$, $visited[\,]$`)`, *we have that for every* $q \in Q$, $dist[q]$ *will be the minimal distance between* q *and* v.

Proof. The proof is similar to that of Lemma 7.1. The only difference is that, in the inductive step, we know by hypothesis that for all $q \in Q$ in acyclic paths, $dist[q] \in \mathbb{Z}$. Thus, the only **states** that remain without an assigned distance are in a cycle. But every cycle eventually reaches a **state** q' of an acyclic path, so that now there is always a successor q' to *source* such that $dist[q'] \neq$ **nil** in line 12. Thus, even in the cyclic case, a distance is now assigned to *source*. □

We thus have the following proposition.

Proposition 7.4 (Correctness of `Preprocess()`)**.** *Let* $\mathcal{SP} = \langle Q, E, P, \rightsquigarrow, L, q_0 \rangle$ *be a* **simulation purpose** *and* v *a* **verdict state**. *Then* `Preprocess(`$\mathcal{SP}$, v`)` *returns a function*

$$dist : Q \longrightarrow \mathbb{Z}$$

such that, for every $q \in Q$, $dist[q]$ *is the minimal distance between* q *and* v.

Proof. `Preprocess()` calls `PreprocessAux()` twice. In the first time, by Lemma 7.1, $dist[q]$ is correctly defined to all $q \in Q$ that belong to an acyclic path of $\mathcal{SP}$. In the second time, by Lemma 7.2, $dist[p]$ is correctly defined for the remaining $p \in Q$ that belong to cyclic paths of $\mathcal{SP}$. □

The correctness of the `BuildRun()` auxiliary procedure will be assessed later, when it becomes necessary. Finally, we note that there is nothing to prove with respect to the `RemoveBest()` auxiliary procedure, since its very definition has the property that is used in later proofs (i.e., the capability of selecting the successor marked with a minimal distance).

7.6.2.2 Justification of Soundness of Algorithm 1

Observational Soundness of Algorithm 1 (for Weak Feasibility) The contents of the *SynchStack* stack of the algorithm is central to this analysis, so let us introduce a related concept and invariant now.

Definition 7.22 (run_{max})**.** *Let SynchStack be non-empty and as built by Algorithms 1. Then,* $run_{max}(SynchStack)$ *is a non-empty* **synchronous run** *of maximal length stored in SynchStack.*

Definition 7.23 (Main Invariant)**.** *If SynchStack is non-empty, then a non-empty* **synchronous run** $run_{max}(SynchStack)$ *can always be extracted from it.*

To show the **observational soundness** of Algorithm 1 for **weak feasibility**, a number of auxiliary lemmas must be given first. Let us begin by those concerning the invariants of the algorithm.

Lemma 7.3 (Inner While Loop Invariant Maintenance)**.** *Let* $t = run_{max}(SynchStack)$ *immediately before the inner while loop of lines 15 – 32,* (q, s) *its last element,* $q \overset{f}{\rightsquigarrow} q'$ *a transition in* $\mathcal{SP}$ *and* $t' = run_{max}(SynchStack)$ *during the loop. Then this loops maintains the following invariant:* t' *exists and either* $\mid t' \mid = \mid t \mid$ *or* $\mid t' \mid = \mid t \mid +1$.

Proof. The only place within the inner while loop in which $run_{max}(SynchStack)$ is modified is in the if block of lines 21 – 32. This block is only executed if $q \overset{f}{\rightsquigarrow} q'$ synchronize $s \overset{g}{\rightarrow} s'$. By hypothesis, (q, s) is the last element of t. Therefore, the synchronization of these transitions imply that $t' = t.f.(q', s')$, which clearly is a **synchronous run**. Since the pushing of this synchronization is the only modification performed to *SynchStack*, it follows indeed that $t' = run_{max}(SynchStack)$ exists and that $\mid t' \mid = \mid t \mid$ (when the if block is not executed) or $\mid t' \mid = \mid t \mid +1$ (when the if block is executed). □

Lemma 7.4 (Middle While Loop Invariant Maintenance)**.** *Let* $t = run_{max}(SynchStack)$ *immediately before the while loop of lines 11 – 32,* (q, s) *its last* **state**, *and* $t' = run_{max}(SynchStack)$ *during the loop. Then this loop maintains one of the following statements true:*

- $\mid t' \mid = \mid t \mid +1$ *and progress* = **true**.
- $\mid t' \mid = \mid t \mid$, *progress* = **false** *and SynchStack is not modified.*

Proof. SynchStack is only modified in the inner while loop of lines 15 – 32, and therefore by Lemma 7.3 we immediately have the desired result. □

Lemma 7.5 (Main Invariant Maintenance)**.** *The outer-most while loop of lines 8 – 34 maintains Main Invariant.*

Proof. By hypothesis, Main Invariant holds in the beginning of the loop. *SynchStack* is not changed (only examined) until the while loop of lines 11

– 32. By Lemma 7.4, this loop guarantees that after its execution $t = run_{max}(SynchStack)$ will either have its length increased by 1 or remain the same. In the first case, *SynchStack* is not changed further after this loop, since Lemma 7.4 guarantees that *progress* = **true**. Therefore, $run_{max}(SynchStack)$ is the same as before. In the latter case, though, the same Lemma guarantees that *progress* = **false** and thus *SynchStack* has its top-most element popped in line 34. This implies in $run_{max}(SynchStack)$ being a different run, t', such that $\mid t' \mid = \mid t \mid$ or $\mid t' \mid = \mid t \mid -1$. But in both cases it exists, and therefore Main Invariant is maintained. □

It is also necessary to show that the algorithm not only maintains the Main Invariant, but also establishes it in its initialization.

Lemma 7.6 (Initialization)**.** *The algorithm initialization (lines 1 – 7) establishes Main Invariant.*

Proof. *SynchStack* begins as an empty stack. Since q_0, by Definition 7.8, is not labelled, it follows by Definition 7.11 that it synchronizes with s_0. Indeed, q_0 and s_0 are synchronized and pushed on the previously empty *SynchStack*. Hence, after the initialization we have that $run_{max}(SynchStack) = ((q_0, s_0))$. □

The last two lemmas guarantee correct results, provided that the algorithm terminates.

Lemma 7.7 (Construction of run_{max} by `BuildRun()`)**.** *Let SynchStack be non-empty and as built by Algorithms 1 or 2. Then procedure* `BuildRun()` *can build* $run_{max}(SynchStack)$*.*

Proof. `BuildRun()` merely starts at the top of *SynchStack* and proceeds downwards in such a way that at every iteration a new element is added to the beginning of the **run**. This new element is chosen to be one whose *depth* is exactly one less than than the previously examined element, and which is, by construction of *SynchStack* (line 29 of Algorithm 1, line 23 of Algorithm 2), an antecedent of this previously examined element. Therefore when the stack becomes empty, the procedure returns a **synchronous run**. □

Lemma 7.8 (Correct Result Upon Termination)**.** *Whenever the algorithm terminates, it returns the correct result.*

Proof. There are only two lines in which the algorithm returns, namely, 32 and 35.

When line 32 is executed, by Lemma 7.3 we have that the last element of $run_{max}(SynchStack)$ is q', which by line 31 must be `SUCCESS`. Therefore,

$run_{max}(SynchStack)$ is actually a **feasible run** and by Lemma 7.7 it will be correctly built by `BuildRun()`. Hence, the returned values are correct.

Concerning line 35, clearly it can only be reached if no **feasible run** of length less than or equal to $depth_{max} + 1$ was found. Because of Proposition 7.2, this means that there is no such run, and therefore the result must be either `INCONCLUSIVE` or `FAILURE`.

If at some point it was not possible to synchronize $t = run_{max}(SynchStack)$ because $\mid t \mid = depth_{max} + 1$, then clearly there could be some **run** of length greater than $depth_{max} + 1$ that was not examined, and therefore the result in this case must be `INCONCLUSIVE`. Indeed, in such a case, line 33 would have set $verdic$ = `INCONCLUSIVE` permanently, and therefore the returned result would be correct. On the other hand, if this situation did not arise, then it is plain that there is no possibility of finding a **feasible run** of any length, and therefore the default setting $verdict$ = `FAILURE` is correct. □

Finally, we provide the proposition that guarantees the **observational soundness** Algorithm 1 for **weak feasibility**.

Proposition 7.5 (Observational Soundness of Algorithm 1 for Weak Feasibility)**.** *If Algorithm 1 terminates when checking* **weak feasibility***, then one of the cases is true:*

- *It returns both* `SUCCESS` *and a* **weak feasible run** $t \in runs^1_{obs}(\mathcal{SP} \otimes \mathcal{M})$ *such that* $\mid t \mid \leq depth_{max} + 1$. **Weak feasibility** *holds with respect to* $runs^1_{obs}(\mathcal{SP} \otimes \mathcal{M})$.
- *It returns* `FAILURE` *and there is no* **weak feasible run** *in* $runs^1_{obs}(\mathcal{SP} \otimes \mathcal{M})$. **Weak feasibility** *does not hold with respect to* $runs^1_{obs}(\mathcal{SP} \otimes \mathcal{M})$*, and no extension of the* **runs** *therein would change this.*
- *It returns* `INCONCLUSIVE` *and there is no* **weak feasible run** *in* $runs^1_{obs}(\mathcal{SP} \otimes \mathcal{M})$. **Weak feasibility** *does not hold with respect to* $runs^1_{obs}(\mathcal{SP} \otimes \mathcal{M})$*, but an extension of the* **runs** *therein could change this.*

Proof. The algorithm is divided in an initialization part (lines 1 – 7), an outer-most loop (lines 8 – 34) and a last return statement (line 35). It suffices then to prove: (i) that Main Invariant holds during the entire execution of the algorithm, and in particular during the outer-most loop; (ii) that this invariant is established in the initialization part, before the outer-most loop; and (iii) that if the algorithms terminates, its output complies with what is required by the proposition. Let us begin by (i) and (ii):

Initialization. By Lemma, 7.6, the initialization part establishes Main Invariant.

Invariant Maintenance. By Lemma 7.5, the outer-most loop maintains Main Invariant. Moreover, the last return statement (line 35) clearly does not violate it.

Finally, by Lemma 7.8 we have that upon termination it returns the correct result. This establishes (iii).

□

Observational Soundness of Algorithm 1 (for Strong Feasibility) The **observational soundness** with respect to **strong feasibility** follows from the fact that it suffices to eliminate elements that have transitions that lead to *Failure* from the search stack in order to verify this stronger variant. This merely strengthens the Main Invariant used to prove Proposition 7.5, by ensuring that the **synchronous runs** in the search stack are all **strong feasible runs**, and thus a similar result holds.

Proposition 7.6 (Observational Soundness of Algorithm 1 for Strong Feasibility)**.** *If Algorithm 1 terminates when checking* **strong feasibility***, then one of the cases is true:*

- *It returns both* `SUCCESS` *and a* **strong feasible run** $t \in runs^1_{obs}(\mathcal{SP} \otimes \mathcal{M})$ *such that* $\mid t \mid \leq depth_{max} + 1$. **Strong feasibility** *holds with respect to* $runs^1_{obs}(\mathcal{SP} \otimes \mathcal{M})$.
- *It returns* `FAILURE` *and there is no* **strong feasible run** *in* $runs^1_{obs}(\mathcal{SP} \otimes \mathcal{M})$. **Strong feasibility** *does not hold with respect to* $runs^1_{obs}(\mathcal{SP} \otimes \mathcal{M})$*, and no extension of the* **runs** *therein would change this.*
- *It returns* `INCONCLUSIVE` *and there is no* **strong feasible run** *in* $runs^1_{obs}(\mathcal{SP} \otimes \mathcal{M})$. **Strong feasibility** *does not hold with respect to* $runs^1_{obs}(\mathcal{SP} \otimes \mathcal{M})$*, but an extension of the* **runs** *therein could change this.*

Proof. It suffices to show in what the algorithm changes when checking **strong feasibility**, and how these changes affect the proof of of Proposition 7.5. When checking **strong feasibility**, the only difference with respect to **weak feasibility** is that Algorithm 1 will set $dist[Failure] = -1$ (line 2) and, potentially, execute lines 22 – 25.

The only effect of setting $dist[Failure] = -1$ is that `RemoveBest()` will always return first a $q \overset{f}{\rightsquigarrow} Failure$ if one exists, since by construction no other q' can

be such that $dist[q'] \leq dist[Failure]$. This implies that if during an iteration of the while loop of line 11 there exists a $q \overset{f}{\leadsto} Failure$ and a $s \overset{g}{\rightarrow} s'$ that synchronize, the first call to *CanSynch* will return true and lines 22 – 25 will be executed.

This execution will: pop *SynchStack*, thus eliminating (q, s) from the **synchronous run**; and set *progress* and *Succ* in such a way that the algorithm will proceed to the beginning of the outer while in line 8, without performing any further iteration of the other inner while loops, thus effectively ignoring (q, s) and its successors. So whatever **synchronous run** is stored in *SynchStack*, none of its elements can contain a transition to some $(Failure, s')$ **state**. This implies that if the algorithm finds a **weakly feasible run**, it will actually be a **strong feasible run**.

Finally, since the modifications induced by **strong feasibility** are very limited and do not compromise the proof of Proposition 7.5, it follows that similar guarantees hold for **strong feasibility** verification, but with the difference that they concern **strong feasible runs** instead of **weak feasible runs**. □

Special Case of `SUCCESS` Verdict in Algorithm 1 As remarked before, the soundness analysis of Algorithm 1 can be strengthened with respect to the `SUCCESS` verdict.

Proposition 7.7 (Soundness of Algorithm 1 with respect to `SUCCESS`)**.** *If Algorithm 1 ever outputs a `SUCCESS` verdict, $\mathcal{SP}$ is* **feasible** *with respect to $\mathcal{M}$ and runs($\mathcal{SP} \otimes \mathcal{M}$).*

Proof. Algorithm 1 outputs a `SUCCESS` if, and only if, it finds a **feasible run** r. Since no further observations can change the fact that r exists in $runs(\mathcal{SP} \otimes \mathcal{M})$, it follows that $\mathcal{SP}$ is indeed **feasible** with respect to $\mathcal{M}$, since otherwise Definition 7.14 would be violated. □

7.6.2.3 Justification of Soundness of Algorithm 2

Observational Soundness of Algorithm 2 Algorithm 2 is merely a slightly modified version of Algorithm 1, which, instead of searching for a **feasible run**, searches for a **run** which is not **feasible**. Hence, the properties and proofs concerning both algorithms are very similar. Below we analyse Algorithm 2 in the light of the proofs already given to Algorithm 1. In this way we avoid repeating most of what has already been said, and instead focus on the important differences that must be stressed.

Proposition 7.8 (Observational Soundness of Algorithm 2)**.** *Algorithm 2 terminates and one of the following cases is true:*

- *It returns* `SUCCESS`*, and all* **runs** *of maximal length in* $runs^2_{obs}(\mathcal{SP} \otimes \mathcal{M})$ *terminate in some* **state** $(Success, s)$*.* **Certainty** *holds with respect to* $runs^2_{obs}(\mathcal{SP} \otimes \mathcal{M})$*, and no extension of the* **observed runs** *therein would change this.*

- *It returns both* `FAILURE` *and a* **run** *of maximal length in* $runs^2_{obs}(\mathcal{SP} \otimes \mathcal{M})$ *such that it terminates either in a* **state** *without a verdict or in some* **state** $(Failure, s)$*.* **Certainty** *does not hold with respect to* $runs^2_{obs}(\mathcal{SP} \otimes \mathcal{M})$*.*

- *It returns* `INCONCLUSIVE`*.* **Certainty** *does not hold with respect to* $runs^2_{obs}(\mathcal{SP} \otimes \mathcal{M})$*, but an extension of the* **observed runs** *therein could change this.*

Proof. Both Algorithm 2 and Algorithm 1 are searching for a kind of **run**. The proof follows by observing the few points in which they differ, from which we can put the former in terms of the latter and then use Proposition 7.5 to show that the former is sound as well.

By construction, the differences are the following:

- In line 26, Algorithm 2 finds a **run** that terminates in some **state** $(Failure, s)$, whereas in the equivalent place of Algorithm 1 a **state** $(Success, s)$ is found. Since Algorithm 1 indeed finds such a **state** if it exists, if follows that Algorithm 2 will also find the **state** it is searching for if it exists. This establishes that it correctly returns `FAILURE` and an associated **run**.

- In line 28, Algorithm 2 reaches some **state** (q, s) such that $q \neq Success$ and $q \neq Failure$, which is indicated by $progress =$ **false**. In Algorithm 1, such a **state** merely indicates that it is time to try another path in the **synchronous product**, and thus a check for it is located in line 11. However, in Algorithm 2 this **state** does indicate a result, that is to say, the fact that there is a **run** which terminates in a **state** that has no verdict, an undesirable condition. Like in the point above, then, in this case the algorithm also correctly returns `FAILURE` and a related **run**.

- In lines 30 – 31, Algorithm 2 pops from *SynchStack* only after examining all the contents of the topmost *Unexplored* set. This is similar to what Algorithm 1 does, which pops only if $progress =$ **false** after trying all elements in *Unexplored*. The difference is that in Algorithm 2 all elements of *Unexplored* are expected to synchronize, and thus it suffices to check whether *Unexplored* is empty after the analysis to pop it, whereas in Algorithm 1 it may be the case that *Unexplored* is not empty and at the same time synchronization is no longer possible.

- If the search does not go beyond the depth $depth_{max}$ and none of the other conditions for `FAILURE` are met, it means that all **runs** of the **synchronous product** terminate in some **state** $(Success, s)$. Algorithm 2 correctly reports this by setting $verdict =$ `SUCCESS` in line 6, which under these conditions remain unmodified until line 32, when it is returned.

□

Special Case of `FAILURE` Verdict in Algorithm 2 As remarked before, the soundness analysis of Algorithm 2 can be strengthened with respect to the `FAILURE` verdict.

Proposition 7.9 (Soundness of Algorithm 2 with respect to `FAILURE`)**.** *If Algorithm 2 ever outputs a `FAILURE` verdict, $\mathcal{SP}$ is not* **certain** *with respect to $\mathcal{M}$ and runs$(\mathcal{SP} \otimes \mathcal{M})$.*

Proof. Algorithm 2 outputs a `FAILURE` if, and only if, it finds an **unfeasible run** r before reaching the maximum search depth. Since no further observations can change the fact that r exists in $runs(\mathcal{SP} \otimes \mathcal{M})$, it follows that $\mathcal{SP}$ cannot be **certain** with respect to $\mathcal{M}$, since otherwise Definition 7.16 would be violated. □

7.6.3 Justification of Termination

Both algorithms terminate, as the following results show.

7.6.3.1 Justification of Termination of Algorithm 1

The following auxiliary lemma is necessary before showing that the algorithm terminates. It argues that, by the construction of the algorithm, similar tuples used in the search can only be pushed a finite amount of times on the search stack.

Lemma 7.9 (Finite Consideration)**.** *Let $(q, s, e, p, sim, Unexplored, depth)$ be an arbitrary tuple on SynchStack of Algorithm 1 at an arbitrary point of its execution. Then, only a finite amount of tuples with the same q, s, e, p, Unexplored and depth can be pushed on SynchStack (let us call this the property of being* finitely considered*).*

Proof. By induction on *depth*.

Base (depth = 0): Because of line 13, $depth'$ is always greater than 0 in the outer-most while loop. Thus, the only time in which such a tuple can be

pushed is in line 6, which is clearly executed only once, and therefore makes it finitely considered.

Induction Hypothesis: The tuples $(q, s, e, p, sim, Unexplored, depth - 1)$ are finitely considered.

Inductive Step (depth > 0): Take a tuple $tup = (q', s', f, q, sim', unexplored', depth)$ from *SynchStack*. It was either pushed on *SynchStack* after examining some tuple (in line 29) or resulted from a removal of an element from the set of unexplored transitions in a tuple already in *SynchStack* (in line 12).

Suppose that *tup* was pushed on *SynchStack*. Then there is a tuple tup_{pre} of depth $k = depth - 1$ that was peeked to generate *tup*. tup_{pre} can only be peeked once in line 9, because in line 12 one of its components is permanently modified (i.e., a transition is removed from its set of unexplored transitions). So once tup_{pre} is peeked, *tup* is calculated and pushed on *SynchStack* once. Since by hypothesis tuples marked with depth $k = depth - 1$ are finitely considered, this can only be repeated a finite amount of times. Moreover, by construction, if there are other $tup'_{pre} \neq tup_{pre}$ that can be used to generate *tup*, the only difference in tup'_{pre} with respect to tup_{pre} has to be the **state** s of $\mathcal{M}$. But because of the finite branching of $\mathcal{M}$, there are only finitely many such **states** up to $depth - 1$, so in all *tup* can only be pushed on *SynchStack* a finite amount of times, and is therefore finitely considered.

Suppose, on the other hand, that *tup* was obtained by reducing the set of unexplored transitions of another tuple already on *SynchStack*. In this case, clearly $unexplored' \neq$ `Successors(`$\mathcal{SP}$, q'`)`. But tuples of depth greater than zero can be pushed only in in line 29, in which by construction the equality $unexplored' =$ `Successors(`$\mathcal{SP}$, q'`)` must hold. So *tup* has never been pushed on *SynchStack*, and thus is finitely considered.

□

The following proposition can now be proved.

Proposition 7.10 (Termination of Algorithm 1)**.** *Algorithm 1 terminates.*

Proof. Termination follows from the fact that each **state** of $\mathcal{SP}$ is examined only a finite amount of times, after which either a **feasible run** is found and the algorithm terminates, or synchronizations are no longer possible and thus *SynchStack* becomes empty, thereby achieving termination as well. The former case arises if indeed there are **feasible runs**, because by Proposition 7.2 one of them will be found, and since this implies the execution of the return statement in line 32, the algorithm terminates. The latter case, however, require some more analysis.

Since the number of **states** in $\mathcal{SP}$ is finite, the possible search depths are finite (for $depth_{max}$ is finite), and the amount of **events** are also finite, there are only finitely many events g, sets of $unexplored'$ and natural numbers $depth'$ to be considered in line 29. Furthermore, although $\mathcal{M}$ may have an infinite number of **states**, for any finite depth there is a corresponding finite amount of reachable **states** s' (and thus of simulation states sim'), owing to the property of finite branching of **ATSs**. So there are only finitely many tuples to be considered in in line 29. Moreover, by Lemma 7.9 all pushed tuples are finitely considered, which implies that each of these finitely many tuples can be pushed on *SynchStack* at most a finite amount of times. Hence, the size of *SynchStack* is bounded by a finite quantity.

Every time a tuple $(q, s, e, p, sim, Unexplored, depth)$ is chosen in line 9, *Unexplored* can only decrease, since the only statement that modifies it is the call to `RemoveBest()` of line 12. So for any set *Unexplored*, it will decrease until it reaches size 0 and is popped. That this will indeed happen follows from these cases for the guard in line 11:

- If $Unexplored = \emptyset$, we have that $progress =$ **false**, and in line 34 the tuple will be popped.
- If $Unexplored \neq \emptyset$, but $depth \geq depth_{max}$, it will be again the case that $progress =$ **false** and in line 34 the tuple will be popped.
- Otherwise, the while loop begins and `RemoveBest()` of line 12 is executed, thereby reducing its size.

Now, since there can be only finitely many tuples on *SynchStack*, and that each is guaranteed to be popped eventually, it follows that the outermost while loop eventually terminates, thus establishing that the algorithm itself terminates. □

7.6.3.2 Justification of Termination of Algorithm 2

A similar result holds for Algorithm 2.

Proposition 7.11 (Termination of Algorithm 2)**.** *Algorithm 2 terminates.*

Proof. Similar to the proof of Proposition 7.10, but taking in account the fact that Algorithm 2 is searching for a **refutable run**, and not a **feasible run**. □

7.6.4 Justification of Correctness

Based on the results shown, we can now formalize their correctness with the following propositions.

Proposition 7.12 (Correctness of Algorithm 1)**.** *Algorithm 1:*

- *is* **observationally sound** *for both* **weak** *and* **strong feasibility***;*
- *is* **observationally complete up to** $depth_{max} + 1$ *for both* **weak** *and* **strong feasibility***;*
- *gives a sound verdict in the classical sense (i.e., with respect to runs*$(\mathcal{SP} \otimes \mathcal{M})$*) whenever it outputs the* `SUCCESS` *verdict;*
- *terminates.*

Proof. Follows directly from Propositions 7.5, 7.6, 7.7, 7.2 and 7.10. □

Proposition 7.13 (Correctness of Algorithm 2)**.** *Algorithm 2:*

- *is* **observationally sound***;*
- *is* **observationally complete up to** $depth_{max} + 1$*;*
- *gives a sound verdict in the classical sense (i.e., with respect to runs*$(\mathcal{SP} \otimes \mathcal{M})$*) whenever it outputs the* `FAILURE` *verdict;*
- *terminates.*

Proof. Follows directly from Propositions 7.8, 7.9, 7.3 and 7.11. □

7.6.5 Justification of Worst-Case Complexities

Let us now provide the exact worst-case complexities of the auxiliary procedures and of the verification algorithms themselves. Owing to their similarity, Algorithms 1 and 2 actually have the same such complexities, and this follows from the same proof.

Worst-Case Complexities of Auxiliary Procedures

Lemma 7.10 (Worst-Case Time Complexity of `Preprocess()`)**.** *Let* $\mathcal{SP} = \langle Q, E, P, \rightsquigarrow, L, q_0 \rangle$ *be a* **simulation purpose**. *Then* `Preprocess()` *runs in* $O(\mid Q \mid^2)$ *time.*

Proof. In `Preprocess()` itself, it is clear that each **state** in Q must be visited twice for initializations, and that the rest of the running time is given by two invocations of the `PreprocessAux()` subprocedure. Because to each such **state** initialization a constant amount of operations is performed, these initializations together take time proportional to $m_0 = 2 \cdot \mid Q \mid$. `PreprocessAux()`, in turn, will recursively visit each **state** in Q accessible from q_0, which in the worst case are all the **states** of Q. In each such visit, a **state** *source* is specified, all of its successors q' may be examined in order to determine whether they have or have not been visited yet, and a maximum constant amount of other operations are performed, without taking in account the recursive call. Since in the worst case there may be $\mid Q \mid$ successors, each such visit to a **state** takes time proportional to $m_1 = \mid Q \mid$,

In considering the successors q', two cases may arise. If q' has already been visited, then $visited[q'] =$ **true** (because this is the first thing `PreprocessAux()` sets when visiting a state), and no recursive call happens. On the other hand, if q' has not been visited, then $visited[q'] =$ **false** and a recursive call to `PreprocessAux()` takes place in line 11. Since in each visit the **state** is marked as visited, there can be at most $\mid Q \mid$ such recursive calls, hence at most $m_2 = \mid Q \mid + 1$ visited **states** (counting the initial visit of q_0).

Therefore, in all, `Preprocess()` has a time complexity of $O(m_0 + 2\dot{m}_1 \cdot m_2) = O(2 \cdot \mid Q \mid + 2 \cdot \mid Q \mid \cdot \mid Q \mid) = O(\mid Q \mid^2)$. □

Lemma 7.11 (Worst-Case Space Complexity of `Preprocess()`)**.** *Let* $\mathcal{SP} = \langle Q, E, P, \rightsquigarrow, L, q_0 \rangle$ *be a* **simulation purpose**. *Then* `Preprocess()` *consumes* $O(\mid Q \mid)$ *memory.*

Proof. This follows from two observations:

- The only data structures employed, $dist[]$ and $visited[]$, each consumes memory proportional to $\mid Q \mid$ by construction. This can be seen by noticing that in these data structures only elements of Q are used as keys, and only numeric or boolean elements are used as values;
- The recursive calls of line 11 of `PreprocessAux()` cannot induce a call stack larger than $\mid Q \mid$, since each **state** is visited at most once.

Therefore, in all, $O(2 \cdot \mid Q \mid) = O(\mid Q \mid)$ space is used. □

Worst-Case Complexities of Algorithms 1 and 2 The complexities of Algorithms 1 and 2 must be given considering the fact that they perform a depth-first search on a transition system with possibly infinitely many **states** that is built on-the-fly (i.e., $\mathcal{SP} \otimes \mathcal{M}$), without keeping track of the visited

states, and only up to a maximum depth (i.e., $depth_{max}$). This means that the complexities must be given mainly in terms of $depth_{max}$ and the maximum branching factor (i.e., the maximum number of possible successors of any state).

Moreover, since **states** in $\mathcal{M}$ are actually related to an **EMMAS environment**, the complexities of processing such environments must be taken in account as well. This is done by considering the size of environments, according to the following definition.

Definition 7.24 (Environment Size)**.** *Let* $Env = \langle AG, AT, EB \rangle$ *be an* **environment**. *Then its* ***environment size*** *is defined as the sum of:* ENVIRONMENT SIZE

- $\sum_{ag \in AG} \mid ag \mid$ *(i.e., the sum of the sizes of each* **agent profile** *in* AG*). The size* $\mid ag \mid$ *of each* **agent profile** $ag = \langle id, S, A \rangle$ *is defined as* $\mid S \mid + \mid A \mid$*;*
- $\mid AT \mid$*;*
- $\sum_{Op \in EB} \mid Op \mid$ *(i.e., the sum of the sizes of each* **operation** *in* EB*). The size* $\mid Op \mid$ *of each* **operation** Op *is defined recursively as follows:*

$$\mid Op \mid = \begin{cases} \sum_{i=1}^{k} \mid Op_i \mid & \text{if } Op \text{ can be decomposed in } k > 1 \\ & \textbf{operations}; \\ 1 & \text{otherwise.} \end{cases}$$

This definition establishes the **environment size** as the amount of primitive elements in it. In particular, **operations** are first decomposed in the simplest possible **sub-operations**, which are then counted. An advantage of this definition is that each of the counted elements has a translation to π-calculus with length bounded by a constant, so that the length of the final π-calculus expression representing the **environment** is proportional to the **environment size**.

The **simulation purpose** $\mathcal{SP}$ should also be taken in account in calculations, since its size is not fixed.

The maximum branching factor of $\mathcal{SP} \otimes \mathcal{M}$ is calculated taking into account both the **environment size** and $\mathcal{SP}$, as follows.

Proposition 7.14 (Branching Factor)**.** *Let Env be an* **environment**, *and* n *its* **environment size**. *Then the branching factor (i.e., the maximum number of possible successors of any state) of* $\mathcal{SP} \otimes \mathcal{M}$ *is* $O(\mid Q \mid \cdot \mid E_{sp} \mid \cdot n^2)$.

Proof. Take an arbitrary **state** (q, s) of $\mathcal{SP} \otimes \mathcal{M}$. Let us first consider the number of possible transitions $q \overset{f}{\rightsquigarrow} q'$ in $\mathcal{SP}$. Certainly this number is less than

or equal to $\mid Q \mid \cdot \mid E_{sp} \mid$, which corresponds to all possible such transitions from q.

Let us now consider the number of possible transitions $s \xrightarrow{g} s'$ in $\mathcal{M}$. Each such transition is calculated by applying the π-calculus operational semantics to the π-calculus process $[Env]_\pi$ of **state** s that represents the **environment**. By construction, there are no more than n components in parallel in $[Env]_\pi$. Thus, the COM rule of the π-calculus operational semantics will generate no more than n^2 successors to $[Env]_\pi$. Since no other rule can generate more successors than this, it follows that the number of successors is bounded by $c \cdot n^2$, for some constant c. Hence, the number of possible $s \xrightarrow{g} s'$ in $\mathcal{M}$ transitions is also bounded by $c \cdot n^2$.

Let us suppose the worst case and consider that at (q, s) each $q \overset{f}{\rightsquigarrow} q'$ in $\mathcal{SP}$ synchronizes with all $s \xrightarrow{g} s'$ in $\mathcal{M}$. Then the maximum branching factor is bounded by $\mid Q \mid \cdot \mid E_{sp} \mid \cdot c \cdot n^2$, and thus is $O(\mid Q \mid \cdot \mid E_{sp} \mid \cdot n^2)$. □

It is now possible to give the complexities in space and time.

Proposition 7.15 (Worst-Case Space Complexity of Algorithms 1 and 2)**.** *Let Env be an* **environment***, n its* **environment size***, and $\mathcal{M}$ be the corresponding* **ATS***. Then, in the worst case, Algorithms 1 and 2 consume $O(\mid Q \mid \cdot \mid E_{sp} \mid \cdot n^3 \cdot depth_{max})$ space.*

Proof. Since the algorithm performs a depth-first search in a graph that is built on-the-fly and only up to a maximum depth (i.e., $depth_{max}$), it follows that the space complexity is given by the maximum size that the search stack may attain. This size, in turn, is given by the amount and size of elements that are put on the stack at each depth. The amount of these elements is no more than the maximum branching factor b, since it is a depth-first search. And the size of each element is proportional to n, since all that these elements contain is proportional to the current state of the **environment**. Hence, the size of the search stack is less than or equal to $c_1 \cdot b \cdot n \cdot depth_{max}$, for some constant c_1.

By Proposition 7.14, $b \leq c_2 \cdot \mid Q \mid \cdot \mid E_{sp} \mid \cdot n^2$, for some constant c_2. Therefore, the size of the search stack is bounded by $c_1 \cdot c_2 \cdot \mid Q \mid \cdot \mid E_{sp} \mid \cdot n^2 \cdot n \cdot depth_{max}$, hence $O(\mid Q \mid \cdot \mid E_{sp} \mid \cdot n^3 \cdot depth_{max})$. □

Proposition 7.16 (Worst-Case Time Complexity of Algorithm 1)**.** *Let $\mathcal{SP} = \langle Q, E_{sp}, P_{sp}, \rightsquigarrow, L_{sp}, q_0 \rangle$ be a* **simulation purpose***, $\mathcal{M} = \langle S, E, P, \rightarrow, L, s_0 \rangle$ an* **ATS***, and $b = \mid Q \mid \cdot \mid E_{sp} \mid \cdot n^2$. Then, in the worst case, Algorithms 1 and 2 have $O(b^{depth_{max}})$ running time.*

Proof. At each new depth reached by the search, no more than b elements are put on the search stack, where b is the maximum branching factor. Since in

the worst case all elements reached up to a maximum depth (i.e., $depth_{max}$) will be examined, it follows that the total quantity of examined elements is bounded by $c \cdot b^{depth_{max}}$, for some constant c. This results in $O(b^{depth_{max}})$ running time. □

7.7 Conclusion

In this chapter we have seen how to make an **EMMAS** environment suitable for simulation and how to check whether the resulting **ATS** (describing a system of interest) satisfies a **simulation purpose** (describing a property of interest) in a number of different and precisely defined senses. In this way, the method operates on formal structures in order to verify something – it is thus a *formal* verification approach. Nevertheless, there is a crucial counterpoint to this formal aspect: the **ATS** under verification is, itself, representing the results of a simulation, whose internal details are not available for the verification procedure, and therefore, from an algorithmic perspective, can be seen as not formal. Verification and simulation interact by means of a simulator interface, which provides the required elements for the construction of transition systems, but do not reveal how they were generated.

This interplay between verification and simulation is a distinctive feature of our approach to the analysis of simulations. The required simulator interface is simple to implement: in essence, it merely requires that (i) simulations proceed from state to state by discrete events; and that (ii) simulation states may be saved and restored. The technique, therefore, has clear practical applications.

From a theoretical perspective, an interesting result concerns the analysis of soundness and completeness. Owing the autonomy of agents and the transition systems with possibly infinitely many **states** , it is not possible to guarantee that all relevant **runs** are simulated, and thus solutions to the verification of the proposed satisfiability relations are inherently incomplete, and sometimes unsound. This is a limitation present in the problem itself, and not a particular issue of the algorithms given.

To handle these difficulties, we found it necessary to introduce weaker notions of soundness and completeness, namely, **observational soundness** and **observational completeness**. The intuition behind them is that although it is not possible to have access to all the relevant information for the verification, it is possible to use all the observed information in a systematic and exhaustive way, thus issuing a verdict which is as accurate as possible with respect to what can be observed. Accordingly, the provided algorithms are designed to be at least **observationally sound and complete** (Propositions 7.12 and 7.13).

Methods based in simulation in general do not offer similar guarantees. Our contribution is in providing both a framework to reason about such matters and algorithms to perform simulations in automated manner so that certain guarantees may be given. All this is in line with the methodological and philosophical outlook presented in Section 1.1 of Chapter 1.

The proposed satisfiability relations are not the only possible ones: one could create new such relations, whose verification could be accomplished in a similar manner. The ones we provide were partly chosen because they help to answer questions that arise in the examples we shall see in Chapter 9. It is therefore possible that different examples could inspire different such relations.

At last, it is worth to emphasize that the proposed verification framework is not limited to **EMMAS**. It was developed to address the concerns of **EMMAS**, but **ATSs** are general formal structures to represent **states** and transitions, and agent autonomy is a general property of MASs. Therefore, anything that can be formalized in such terms can employ the technique. This is an unexpected but fortunate consequence of our efforts.

Part IV

Implementation

CHAPTER 8

Simulator Implementation

We have so far presented the theoretical aspects of the technique developed in this thesis. Let us now proceed to their actual implementation. In this chapter we present our tool, called Formally Guided Simulator (FGS), which we have developed as a proof-of-concept implementation of our method. FGS is a component-oriented simulator, and as such it is necessary to provide components in order to have a functioning system: simulations are built using software components, among which there are agent components. The **Behaviourist Agent Architecture** has been implemented as one such component.

To be able to simulate **EMMAS** specifications, we developed a π-calculus simulation library. FGS loads such a specification and converts it to π-calculus, which is readily implemented by instantiating elements from this library.

In this chapter our focus is on the general design principles that we applied, the architectural choices that we have made, as well as on certain specific optimizations important in an implementation like ours. Details of the source code itself are avoided as much as possible, since they have no exceptional scientific or engineering importance.

The text is organized as follows. In Section 8.1 we present the general software architecture[1] underlying FGS, through which agents and environments can be defined, and verifications can be requested and carried out. Then, in Section 8.2 we explain how a simulation is actually executed given this design. In Section 8.3 we provide an account of the **Behaviourist Agent Architecture**

[1] *Software architecture* is not to be confused with *agent architecture*. The latter is a kind of the former. Here we are referring to the architecture of the tool, not the agent architecture given in Chapter 4.

implementation and of how it is used as an FGS component. The π-calculus simulation library, in turn, is considered in Section 8.4. Finally, Section 8.5 concludes. Since our discussion here is concerned with tool design and not with how to use it, the actual input format and commands necessary to execute FGS are left for Appendix C.

8.1 Architecture of FGS

The technique of verification presented in this thesis depends on the possibility of separating agents from their environment. The former are given as black-boxes, the latter is defined as a formal specification. The technique then consists in formulating simulation experiments that can be automatically carried out by systematically exploring the formal environment specification. In our software architecture, we realize these elements through the following main entities:

- *Software components.* Agents are implemented as software components that obey certain conventions.
- *Scenarios definitions.* Agents and properties about them are initialized and composed in an environment by specifying scenarios.
- *Experiments definitions.* Verification and simulation procedures are requested by specifying experiments.

Given these artefacts, we have the following dependencies. First, components for a particular domain must be created. Then, particular scenarios for this domain can be formulated. Finally, experimentation and analysis can be carried out using these scenarios. Thus, we have a layered structure as shown in Figure 8.1.

This is a rather general architecture, and is not restricted to the particular kinds of agents and environments considered in this thesis. However, in FGS we have provided specific components and specialized the definitions of scenarios and experiments to account for our particular needs. Thus a component that implements the **Behaviourist Agent Architecture** (see Section 8.3) is provided, as well as the possibility of specifying **EMMAS**-based scenarios and experiments concerning verification using **simulation purposes**.

In what follows we examine the main features of components, scenarios and experiments as used in FGS.

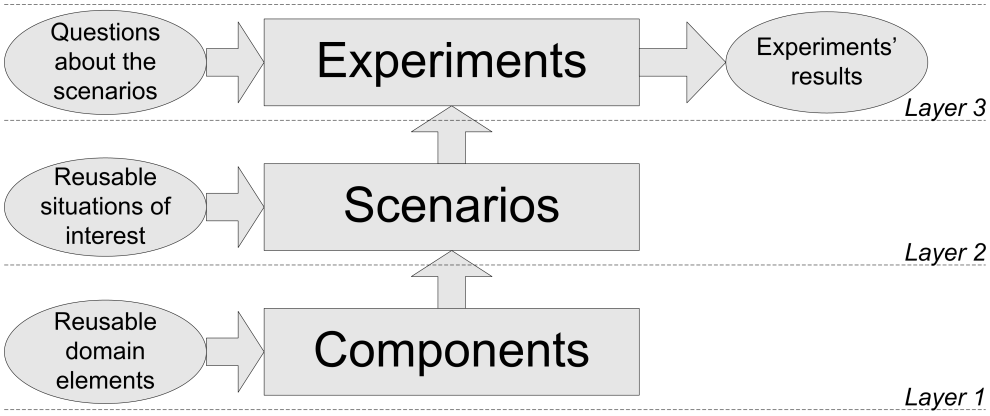


Figure 8.1: The dependencies between the several methodological concerns (in ellipses) and the artefacts that address them (in rectangles). Each layer has its particular worries and goals. **EMMAS** specifications are given within scenarios, whereas **simulation purposes** are provided by experiments. The **Behaviourist Agent Architecture** is implemented as a component.

8.1.1 Components

Our components represent very specific entities. We define two kinds of components, namely:

- *Agent components*, which account for the simulated agents. The **agent profiles** of **EMMAS** provide the interfaces of such components, so that they can be represented and used in **environments**;

- *Property components*, which define what can be locally (i.e., at each MAS global state) measured and calculated concerning the agents. These are used to compute the annotations (i.e., literals) present in each state in an **ATS** during simulations.

An agent component defines a particular kind of agent. That is, it defines how such an agent behaves and what parameters it has. In a simulation, one must *instantiate* agent components in order to have actual agents. All of the instances of a particular component will behave similarly, varying only according to the parameters specified during instantiation. Property components are analogous. The set of all components is called the *components repository*.

8.1.2 Scenarios

In order to be used, components must be instantiated in particular scenarios. Scenarios define which instances must be created (i.e., agents and properties about them), and how they are composed in an environment, thus specifying an MAS. They are also reusable assets, because they can be frequently used in distinct situations (e.g., different people might want to study a scenario in different manners) and may not be cheap to produce (e.g., because the information needed to create them is hard to obtain). Therefore, scenarios must be kept in separate files, allowing their storage and sharing.

In FGS, a scenario is specified as a XML file, in which in particular an **EMMAS** specification can be given in XML format as well. See Appendix C for an explanation of the input format, and Appendix B for actual examples of scenario files.

8.1.3 Experiments

It is not sufficient to have a description of an interesting scenario. It is also necessary to be able to do something useful with it. To this end, experiments provide strategies to explore them. Moreover, experiments should be performed in order to answer, automatically, particular questions about scenarios, instead of just "watching" the simulation unfolds. In this thesis, the relevant strategy is the one in which one may specify a **simulation purpose** and check whether it is satisfiable with respect to the specified MAS.

The same scenario can be used to perform different experiments. And the same experiment, or a slightly modified version, can be performed in different scenarios. Thus, experiment definitions should also be specified in separate files.

In FGS, an experiment is specified as a XML file, in which a **simulation purpose** can be described, along with the desired satisfiability relation. See Appendix C for an explanation of the input format, and Appendix B for actual examples of experiment files.

8.2 Simulation Execution and Analysis

To request the simulation and verification of an MAS, three kinds of XML files must be provided to FGS:

- One or more parametrizations to the agents present. These parametrizations are used to instantiate the implementation of the **Behaviourist**

Agent Architecture;

- One scenario description;
- One experiment description.

Once FGS is invoked, it first uses the scenario description to instantiate the **Behavioural Agent Architecture**, so that the relevant agents become available to the simulator. It then transforms the **EMMAS** specification provided in the scenario description in a π-calculus expression. This expression is implemented directly by instantiating classes from the π-calculus simulation library according to the implementation of the **translation function**. This corresponds to the initial **state** of the MAS to be simulated.

FGS then reads the experiment description in order to obtain the **simulation purpose** and the satisfiability relation to be checked. The **simulation purpose** is implemented trivially, since its structure is very simple. The satisfiability relation, in turn, determines which verification algorithm should be used.

After all this has been done, FGS merely executes the requested verification algorithm. To calculate the next **states** in the **ATS** implementation, it uses the π-calculus operational semantics rules which are implemented in the π-calculus library. A few optimizations are used in the π-calculus implementation, so that this computation is more efficient (e.g., expressions are balanced trees, and the results of applying the operational semantic rules are cached for later reuse whenever possible). The state of the agents are changed and inspected by manipulating the objects that instantiate them. This information added to **states** , and used to apply the relevant constraints. Finally, the property components specified in the scenario description are also simulated and the resulting values are used to annotate the **states** with literals.

As seen in Chapter 7, there is a special *commit* event which signals that agents should receive stimulation and provide behavioural responses. This is implemented by FGS as follows. **Events** are stored as they are found in a **run** of the **ATS**, but are not delivered to the agents. When a commit **event** is found, then the stored **events** are delivered to the relevant agents, and they may also change whether they are emitting or not emitting their actions.

If the appropriate options have been set, FGS will output information about every synchronization made in the **synchronous product**, which allows the visualization of the simulations. When the algorithm terminates, the verdict is shown, and a **run** is displayed as well if relevant (e.g., the **feasible run** that led to *Success*).

8.3 Behaviourist Agent Architecture Component

The **Behaviourist Agent Architecture** specified in Chapter 4 is implemented in Java. For most of the schemas in the specification, we provide a class that implements it. The objective of such a strategy is to allow an easy correspondence between the implementation and its specification. Besides reducing the chance of introducing errors in the implementation, this also allows modifications in the specifications to be implemented quickly. This is an important point because the architecture is designed to be extensible, and therefore it is convenient that such extensions can be easily integrated in the implementation. A disadvantage of this approach, however, is that the specification structure is not necessarily the most efficient implementation structure. For example, in the specification many operation schemas are defined by putting together several other operation schemas by means of the schema calculus. This division is maintained in the implementation, where each schema has its own class, although it would be more efficient if they were all implemented in only one class.

In order to instantiate an agent, it suffices to create a new `Organism` object and initialize it with a special kind of XML file, in which several agent parameters must be defined (e.g., the stimuli it recognizes, the actions it can perform, the reflexes). Appendix C provides a complete description of the required XML format.

This implementation is independent of FGS. However, to be used in FGS, it is wrapped in another class, the `OrganismComponent`, which implements the component interfaces required by FGS. In this manner, the agent implementation is integrated in the simulation and verification algorithms implemented by FGS.

8.4 π-Calculus Simulation Library

The π-calculus process algebra, as we have already pointed out, is a formalism for specification and verification of concurrent systems. Moreover, much like the λ-calculus for sequential computation, it is particularly suitable for theoretical analysis, since it is composed of few elements and notions. For example, since there are few basic operators, there are also few cases to prove whenever one wishes to establish new properties about the calculus. Nevertheless, it can also be used to implement concurrent systems. In this case, an important advantage also comes from the fact that there are few operators, for any implementation can be reduced to implementing them. Furthermore, since any such implementation inherits the theoretical properties of the calculus, it follows that programs thus built will by definition be able to express

a number of important concurrency notions, as well as be subject to certain forms of formal verification.

The environment model we described in Chapter 5 is largely given in terms of π-calculus, owing to its good theoretical properties. This motivated us to go one step further and actually implement a simulator for the π-calculus, so that the simulation of environments can be carried out directly from their formal definitions. This is provided as a library, which in turn is used by FGS.

This π-calculus simulation library is implemented in as a direct manner as possible, as follows:

- Each possible π-calculus processes is given a Java class, which holds references to classes implementing its sub-processes. A complete π-calculus processes, then, is implemented as a tree structure.
- Each rule in the π-calculus operational semantics is also implemented as a Java class. Processes hold references to the rules that apply to them. Once one asks for the successors of a given process, the rules associated with it are applied. This is done in a recursive manner, analogously to the definition of the π-calculus operational semantics.

There are a number of efficiency issues which are particular to π-calculus implementation, and which therefore have no treatment on a theoretical level. These issues require special optimizations to be made, and, if anything, these are the real contribution of such an implementation. Let us then turn to them.

8.4.1 Optimizations

Most of the optimizations below do not change the calculus in any way, but only make efficient choices regarding the many possible ways to implement theoretical features. Therefore, most of them do not significantly compromise the correctness of the implementation. The only exception to this is the *custom process* presented in Section 8.4.1.4. But for the sake of correctness, as well as other matters to be seen in that section, we have not implemented this particular idea, and thus it is left as a mere possibility which do not affect the π-calculus simulation library. Let us now present each optimization technique and the problems that they address.

8.4.1.1 Caching

In order to build an LTS for a π-calculus process, it is necessary to calculate its successors. As we have explained, this is achieved by a recursive application of the calculus' operational semantics rules. In principle, then, whenever

such successors are required, they can be computed in this fashion. However, once computed, they are guaranteed to do not change unless a substitution is applied to them. Therefore, it is possible to *cache* (i.e., record for latter use) the successors of any π-calculus process after their first computation, so that when they are needed again their computation is avoided.

Since a process is recursively composed by sub-processes, this caching is an excellent optimization, for without it the whole specification (which is one big process) would be constantly subject to successor calculation when in reality only a fraction of it actually needs re-computation.

Our implementation is simple and as follows. Each process keeps a cache of successors which is calculated in the first time that successors are requested. Later, this cache is returned. However, if the process is requested to perform a substitution *and* this substitution actually changes something, then the cache is cleared so that the successors can be recomputed in the next time that they are requested. This is necessary because some successors may change after the substitution.

8.4.1.2 Expression Simplification

In π-calculus, processes are composed of subproceses through operators. The only primitive process that exists is $\mathbf{0}$ (i.e., the *Nil* process). This special process signals a state in which nothing more can be done, and therefore it is incapable of interacting with any other process.

Now consider a system composed of several processes in parallel, such as this:

$$P_1 \mid P_2 \mid P_3 \mid P_4 \mid P_5 \mid P_6 \mid P_7$$

Since every P_i process contains at least one reference to the *Nil* process, it follows that it may be the case that many such references eventually appear in the composition after several applications of the operational semantics rules. For instance, at some point the parallel composition may be reduced to the following:

$$P_1 \mid \mathbf{0} \mid P_3 \mid P_4 \mid \mathbf{0} \mid \mathbf{0} \mid \mathbf{0}$$

Owing to the fact that the *Nil* process is inert, in principle one could leave the parallel composition as it is. However, in an implementation these references consume resources, which can become a hindrance if their quantity increases too much. To avoid this, we simplify the parallel composition, removing all such useless *Nil* processes. In the above example, this would result in the following shorter process:

$$P_1 \mid P_3 \mid P_4$$

This optimization is particularly important for the implementation of **EMMAS** because it employs the replication operator frequently in processes with few events. For example, the following process is always a parallel component in the generated π-calculus process of an **EMMAS** specification destined to simulation (see Definition 7.1):

$$!(\mathit{commit}.\mathbf{0})$$

By structural congruence, this is equivalent to the following:

$$\mathit{commit}.\mathbf{0} \mid !(\mathit{commit}.\mathbf{0})$$

Which produces the *commit* event and leaves us with:

$$\mathbf{0} \mid !(\mathit{commit}.\mathbf{0})$$

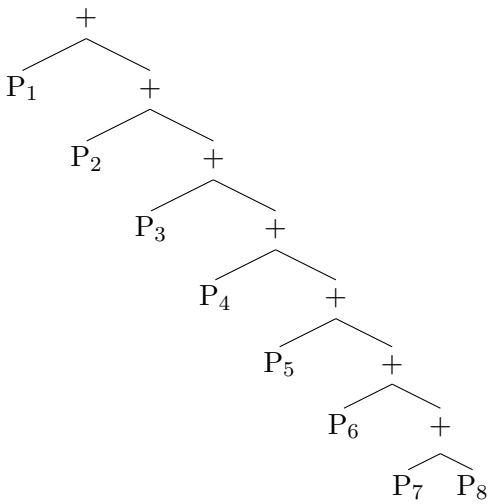
Which is the kind of expression the optimization discussed here aims at simplifying. Since during simulations many of these *commit* events will be generated, it is clear that there would be a very large accumulation of useless Nil processes if the optimization was not applied.

8.4.1.3 Balanced Expression Structure

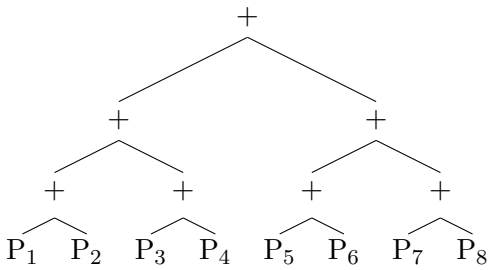
Most operators in π-calculus are binary, and therefore they induce a binary tree to represent expressions. A naive implementation could build an unbalanced binary tree, in which one of the branches is much deeper than the others. For instance, consider an unbalanced representation of the process

$$P_1 + P_2 + P_3 + P_4 + P_5 + P_6 + P_7 + P_8$$

such as the following:



This would imply that any modification in the lower nodes of the tree would affect almost all other preceding nodes, since the latter use the former to calculate their successors. This problem can be solved by simply balancing the tree, so that the tree's structure reflects the actual computational dependencies and avoids unnecessary recalculations. In the example just considered, the following balanced representation could be used:



Notice the difference with respect to the tree's height. In fact, we can prove that there is a logarithmic decrease in the height by transforming a perfectly unbalanced tree (i.e., one in which there is as much unbalance as possible) into a perfectly balanced one (i.e., one in which there is as much balance as possible).

Proposition 8.1. *Let T be a perfectly unbalanced binary tree with n nodes and height h. Let T' be the perfectly balanced version of T, with height h'. Then either $h' = lg(h+1)$ or $h' = lg(h+2)$.*

Proof. Clearly, $h = n - 1$, hence $n = h + 1$. In T', owing to the balance imposed, at each level of the tree there are two sub-trees with the same structure

(except for the lack of one node, if x is odd). So if n is even, it follows that:

$$\begin{aligned} n &= 2^{h'} \\ h+1 &= 2^{h'} \\ lg(h+1) &= h' \end{aligned}$$

On the other hand, if n is odd, we have that:

$$\begin{aligned} n &= 2^{h'} - 1 \\ h+1 &= 2^{h'} - 1 \\ h+2 &= 2^{h'} \\ lg(h+2) &= h' \end{aligned}$$

Therefore, either $h' = lg(h+1)$ or $h' = lg(h+2)$.

□

8.4.1.4 Custom Processes

It is possible to extend the framework with new kinds of primitive processes. This allows the possibility of implementing what was originally a complex π-calculus expressions as merely a primitive process, by providing both such an implementation and a corresponding operational semantic rule. In this manner more efficient calculations can be achieved. This possibility, however, has two related main disadvantages: it creates the problem of proving that the new implementation corresponds its original π-calculus definition; and as a consequence, it makes it difficult to modify the original π-calculus definition if necessary, for any such modification would require one to make sure that the modified implementation is still correct. It is also not a general solution, since any new complex expression, to take advantage of this, would have to be individually implemented. For these reasons we chose not to use such custom processes, though they offer a potential way of optimizing the technique.

8.5 Conclusion

In this chapter we have seen how the approach is actually implemented. There are three main implementation artefacts involved. Most importantly, the simulator itself, which we call FGS. But to work with the MASs developed in this thesis, FGS requires two other things: a component that implements the **Behaviourist Agent Architecture**; and a π-calculus simulation library to

implement **EMMAS** specifications. All of these were considered in the chapter.

In this thesis, our main formal effort has been devoted to the technique of formally guided simulations and the related verification algorithms, which are proved to be correct with respect to precisely defined notions of soundness and completeness (in Section 7.6). However, we have not proved that the *implementation* (in Java) of the various formal definitions, transformations and algorithms are correct as well. Nevertheless, since by construction these implementations follow their formal counterparts closely, they are likely to be correct.

An important consequence of decoupling the agent architecture implementation from the simulator, in the form of external components, is that one may try different architectures, or different implementations of the same architecture, without changing the simulator. This allows one to consider several possible agent implementations, which could be developed independently of each other. Such a feature is important in a tool designed for experimentation, since one way to experiment with an MAS is to consider variations of the agents under experimentation. Moreover, the **Behaviour Agent Architecture** itself is designed to be extended, so it is desirable that its implementation can be easily extended as well.

The idea of implementing the π-calculus directly in order to account for **EMMAS** has both positive and negative aspects. The positive aspect is that such an approach makes modifications to the **EMMAS** model quite easy to implement: it suffices to provide a π-calculus translation for the modification, which is trivially implemented by instantiating elements of the π-calculus simulation library. However, a naive implementation of π-calculus is bound to suffer from efficiency problems, for there are a number of algorithmic issues that are not addressed in the calculus' formal definition. This requires optimizations to be performed. And therein lies the negative aspect. The π-calculus is very expressive, but very fine-grained as well, so translations from one high-level concept of **EMMAS** usually require several π-calculus elements. Ultimately, one gets a π-calculus process that is much larger than the original **EMMAS** specification. A more direct implementation of **EMMAS** would be more in this respect, although this would possibly undermine the flexibility gained by using the π-calculus. In this thesis we have opted for this flexibility, but one may consider other implementation strategies.

CHAPTER 9

Case Studies

In this chapter we present some concrete case studies to illustrate the approach proposed in this thesis. Each one is given with:

- an informal introduction;
- parametrizations of the **Behaviourist Agent Architecture**;
- the **EMMAS** specification;
- the **simulation purpose** to be checked;
- the results of executing FGS.

The input files to FGS, as well as its output, are not provided in this chapter, but can be found in Appendix B. Here we focus our attention on the most important aspects of the models themselves, and not on input notation, which is rather verbose (see Appendix C for a description of the input format).

The case studies show the kinds of problems that the **Behaviourist Agent Architecture** and **EMMAS** are capable of modelling. Given the distinctive behaviourist point of view adopted, these problems differ considerably from those typically used in the MAS literature (e.g., auctions). Here we are concerned with how environmental stimulation can influence the behaviour of agents endowed with specific adaptation and learning mechanisms. These case studies are partly inspired by the literature on behaviourist psychology, but add innovations that arise from our particular computational approach. For instance, the fact that agents in **EMMAS** are related by a social network is the basis for specifying experiments in which multiple such networks can be simulated.

Moreover, each case study highlights a particular set of features of the approach. In this way, we show that our method has a certain generality in so far as it is able to express different circumstances in a uniform manner.

The experiments were executed in the Ubuntu 10.04 Desktop operating system (a GNU/Linux distribution), using a machine with an Intel Core 2 Duo processor (2.26 GHz) and 4 GB of RAM. The OpenJDK implementation of Java 6 was employed.

The case studies are divided in two groups. Section 9.1 presents those concerning isolated agents, while Section 9.2 provides multi-agent ones. We thus show how our approach is relevant in both circumstances. To conclude, Section 9.3 summarises and analyses the findings we obtained through all of the examples.

9.1 Single Agent Examples

Experiments with isolated agents are the means by which much of behavioural psychology is developed. Indeed, perhaps its most memorable instrument is the *operant conditioning chamber* (also known as the *Skinner box*), a cage in which an organism, such as a rat or a pigeon, can be put in order to undergo experimentation (Catania, 1998). The box allows the experimenter to provide stimulation (e.g., lights, food), and the subject animal to perform certain actions (e.g., push a lever), all of which can be recorded for later analysis. By this method, it is possible to study the animal's behaviour under several circumstances, as well as to teach certain behaviours to it.

In this section we consider a similar idea, but in a more abstract version. Instead of a box, we have any **EMMAS environment** such that only one agent is present. And instead of a human operator, we use **simulation purposes** to conduct the experiments.

The following examples model some classical phenomena that can be studied through behavioural means. In this way, they also illustrate the connection between our technology and the underlying psychological approach that inspired it.

9.1.1 Pavlovian Dog: Classical Conditioning

Classical conditioning consists in teaching an organism that a certain desired stimulus happens after another, initially neutral, stimulus. The neutral stimulus thus becomes a *conditioned stimulus*. It is a phenomenon also known as Pavlovian conditioning, since it was discovered by Ivan Pavlov, a Nobel prize

winner physiologist active from late XIXth to early XXth century. Pavlov performed experiments on dogs in order to discover this principle.

Let us then model such a dog using our agent architecture. We include the elements necessary for performing a classical conditioning experiment, but also elements that are not at all necessary. This is done for the sake of illustration, in order to show that an agent can be made rather complex, and that, nevertheless, we are able to experiment with the subset of its behaviours that we are interested in. The practical importance of this is that elements that in principle are not subject to experimentation can interfere with those that are, and when modelling an organism one may face such situations. Hence, our method must be able to deal with it.

9.1.1.1 Agent Parameters

First, we need an instance of an organism.

$$dog : Organism$$

Before the experiment, the dog must have the following elements defined:

- *Unconditioned stimulus.* A stimulus that is desired by the organism. For Pavlov's experiments, some kind of food. We also specify other stimuli to render the dog more complex.

$$food, injection, neutral, bark_sound : Stimulus$$

$$dog.primaryStimuli = \{food, injection, neutral, bark_sound\}$$

$$dog.primary_utility(food) = 0.9^{1}$$

$$dog.primary_utility(injection) = -0.6$$

$$dog.primary_utility(neutral) = neutral_utility$$

$$dog.primary_utility(bark_sound) = 0.1$$

$$dog.max_delay = 100$$

$$dog.c = 0.5$$

$$dog.pleasureHints = \{food\}$$

$$dog.painHints = \{veterinary\}$$

[1] We write rational numbers without fractions for readability, but formally we would have to specify a pair of integers (a, b).

- *Conditioned stimulus.* A stimulus that, initially, has no effect over the organism. For instance, the sound of a bell, a whistle and a veterinary.

$$bell, whistle, veterinary : Stimulus$$

- *Actions.* Pavlov would measure the salivation of the dog in order to asses whether the presented stimulus had any effect. We thus model this action, but also other actions that are available to the dog.

$$\begin{array}{|l}
salivate, bark, sit, push_lever : Action \\
\hline
dog.operantActions = \{bark, sit, push_lever\} \\
dog.reflexActions = \{salivate, bark\} \\
dog.baseLevel(salivate) = 0.0 \\
dog.baseLevel(bark) = 0.2 \\
dog.baseLevel(sit) = 0.1 \\
dog.baseLevel(push_lever) = 0.0
\end{array}$$

- *Salivation reflex.* The dog has a salivation reflex that has food as its antecedent.

$$\begin{array}{|l}
salivation : Reflex \\
\hline
dog.reflexes = \{salivation\} \\
salivation.antecedent = food \\
salivation.action = salivate
\end{array}$$

- *Operants.* Pavlov's experiments did not deal with operant behaviour, a notion that would only come to light later. However, we may here introduce some operants in order to make the example more interesting and show how classical conditioning interacts with them. So let us suppose that the dog has some previous operant learning. First, it knows that *push_lever* leads to *bell* (we will see that the environment complies with this). Second, let us make the dog aware that its *bark* results in a *bark_sound* it can hear.

$$\begin{array}{|l}
o_1, o_2 : Operant \\
\hline
dog.operants = \{o_1, o_2\}
\end{array}$$

$$\begin{array}{|l}
o_1.antecedents = \{\emptyset\} \\
o_1.action = push_lever \\
o_1.consequence = bell \\
o_1.consequenceContingency = \{(\emptyset, 0.9)\}
\end{array}$$

$$
\begin{aligned}
& o_2.antecedents = \{\emptyset\} \\
& o_2.action = bark \\
& o_2.consequence = bark_sound \\
& o_2.consequenceContingency = \{(\emptyset, 0.9)\}
\end{aligned}
$$

This gives us an agent suitable to experimentation.

9.1.1.2 EMMAS Specification

We first need to define an **agent profile** to the dog.

$$
\begin{aligned}
S &= \{food, bell, whistle, injection, veterinary, neutral, bark_sound\} \\
A &= \{salivate, bark, sit, push_lever\} \\
dog &= \langle 0, S, A \rangle
\end{aligned}
$$

Since there is only one dog in the example, the set of agents AG is a singleton.

$$AG = \{dog\}$$

For the same reason, there are no communication with other agents.

$$AT = \emptyset$$

It is in the environment behaviours that most of the environment's structure lies.

$$
\begin{aligned}
eb_1 &= ER(salivate, dog, NOP) \\
eb_2 &= ER(bark, dog, NOP) \\
eb_3 &= ER(push_lever, dog, Stimulate(bell, dog)) \\
eb_4 &= (Stimulate(bell, dog);\ Stimulate(food, dog)) + \\
&\quad (Stimulate(whistle, dog);\ Stimulate(food, dog)) \\
EB &= \{eb_1, eb_2, eb_3, eb_4\}
\end{aligned}
$$

Let us comment on the role of each of these:

- eb_1 and eb_2 define **environment behaviours** that react to the actions of *salivate* and *bark*, respectively. However, the reaction is merely a **do nothing operation**. The reason is that only the actions that are relevant for the **environment** are visible to the simulator. By specifying eb_1 and eb_2 in this way, we ensure that *salivate* and *bark* will be visible. These actions have no particular consequence in this **environment**, but

will be relevant later, when we specify the **simulation purpose** to be checked. This is also a way to reuse the same **agent profile** in different **environments**, since in each such **environment** one may select the relevant actions and stimuli for the agent.

- eb_3, on the other hand, is an **environment behaviour** that actually does something. Whenever the dog pushes a lever, it gets stimulated with the sound of a bell. This can be understood as an apparatus present available for the dog's manipulation. If eventually the utility of *bell* becomes positive and the dog learns that *push_lever* leads to *bell*, it will become more likely to perform *push_lever*. This is an example of how the structure of the environment can be closely related to an agent's learning mechanisms.

- eb_4 has a more experimental role. It defines two alternative ways in which the **environment** can manipulate the dog. In the first case, the *bell* stimulus is delivered, and later the *food* stimulus. In the second case, the *whistle* stimulus is delivered, and later the *food* stimulus. The objective of both is to try to condition either *bell* or *whistle* to *food*, a primary stimulus. The reason why these two alternatives are given here, and not merely one, is that the **environment** must express a range of possibilities. The actual experiments to be performed will be defined later by a **simulation purpose**, which will guide the choices among all the possibilities offered by the **environment**.

Given all that, we at last define the **environment**.

$$\langle AG, AT, EB \rangle$$

9.1.1.3 Simulation Purpose

Let us now define a **simulation purpose** that can perform some experiments in the dog in order to show its classical conditioning capabilities. We begin by defining everything except the transitions.

$$\begin{aligned}
Q = \; & \{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7, \\
& q_8, q_9, q_{10}, q_{11}, q_{12}, Success, Failure\} \\
E = \; & \{emit^0_{salivate}, stop^0_{salivate}, emit^0_{push_lever}, beginning^0_{food}, stable^0_{food}, \\
& beginning^0_{whistle}, stable^0_{whistle}, beginning^0_{bell}, stable^0_{bell}, ending^0_{bell}, \\
& absent^0_{bell}\} \\
P = \; & \{\} \\
L = \; & \{\}
\end{aligned}$$

The interesting thing now is to define, through the possible transitions, the experiments to perform. We will define two such experiments.

$$
\begin{aligned}
\rightsquigarrow = \quad & \{(q_0, beginning^0_{whistle}, q_1), (q_1, \circledast, q_1), (q_1, stable^0_{whistle}, q_3), \\
& (q_3, \circledast, q_3), (q_3, beginning^0_{food}, q_4), (q_4, \circledast, q_4), \\
& (q_4, stable^0_{food}, q_4), (q_4, emit^0_{salivate}, q_4), (q_4, beginning^0_{whistle}, Failure), \\
& (q_4, stop^0_{salivate}, q_5), (q_5, \circledast, q_5), (q_5, beginning^0_{whistle}, q_6), \\
& (q_6, \circledast, q_6), (q_6, emit^0_{salivate}, Success), \\
& \\
& (q_0, beginning^0_{bell}, q_{10}), (q_{10}, stable^0_{bell}, q_{10}), (q_{10}, ending^0_{bell}, q_{10}), \\
& (q_{10}, absent^0_{bell}, q_{10}), (q_{10}, \circledast, q_{10}), (q_{10}, beginning^0_{whistle}, Failure), \\
& (q_{10}, beginning^0_{food}, q_{12}), (q_{12}, \circledast, q_{12}), (q_{12}, emit^0_{push_lever}, Success)\}
\end{aligned}
$$

The first experiment consists in stimulating the dog with *whistle*, then giving it *food* later, and finally stimulating it again with *whistle* to check whether it emits a *salivate* action. If it does, it means that *whistle* was successfully conditioned to *food*.

In the second experiment we condition *bell*, instead of *whistle*, to *food*. It is assumed that the dog has an operant that states that *push_lever* leads to *bell*. The experiment, then, consists in checking whether *push_lever* is emitted by the dog, which shows how stimulus conditioning influences operant behaviour.

With all this we have the **simulation purpose**.

$$\langle Q, E, P, \rightsquigarrow, L, q_0 \rangle$$

9.1.1.4 Result

FGS verified in two seconds that the **simulation purpose** is indeed **weakly feasible**. It outputed the verdict and a **feasible run**. In an abbreviated form, this corresponds to the following:

```
Simulation Purpose Verification strategy
=======================================================
Result = SUCCESS
Running time = 2s

Run found:
    [depth = 0] State (in SP): initial
    [depth = 1] Events synch'd: <?beginning[agentId = 0]_[Stimulus type='bell'],
                                !beginning[agentId = 0]_[Stimulus type='bell']>;
                State annotations synch'd: <[],     []>  State (in SP): 10
    [depth = 2] Events synch'd: <(*)other,
                                !commit>;

    (...)

    [depth = 29] Events synch'd: <!emit[agentId = 0]_[Action type='push_lever'],
                                 ?emit[agentId = 0]_[Action type='push_lever']>;
```

```
                State annotations synch'd: <[],     []>  State (in SP): success


Result = SUCCESS
Running time = 2s


Finished.
```

The complete **feasible run** is given in Appendix B. In the next examples we do not show the actual output, since they are all similar to the above, but they can be found in Appendix B as well.

9.1.2 Worker: Operant Chaining

An operant is a learning unit that associates an action to a consequence. Often, though, the desired consequence might need a series of different actions. Because each action requires an operant, we call this *operant chaining*. This method consists in reinforcing actions in the appropriate order, so that the performance of one action sets the antecedents for the next one.

To see how this can be modelled, let us consider a worker. In many cases, the worker has no interest in his job. Rather, he works in order to be able to make money. And money itself has only value because it can be used to buy pleasant things, like food.

More than one operant is required to model how the worker acquires what he truly enjoys. In this example, we provide a worker with several operants that form a chain. The environment, in turn, is designed to make sure the chain is maintained. That is to say, every action is rewarded with the stimulus necessary for the next operant in the chain. The **simulation purpose** then merely checks that the chain is actually executed, by verifying that actions are emitted in the appropriate order.

9.1.2.1 Agent Parameters

Let us define a worker.

> $worker : Organism$

The relevant parametrizations for this example are the following:

- *Food stimulus.* Ultimately, the worker wants to have food.

$$\begin{array}{|l} food : Stimulus \\ \hline worker.primaryStimuli = \{food\} \end{array}$$

- *Money stimulus.* Money will be used to buy this food, but in itself has no utility.

$$\begin{array}{|l} money : Stimulus \end{array}$$

- *Work place stimulus.* The work place has several characteristic features (e.g., forms, colors, sounds). However, for our modelling, it suffices to abstract all of these features and define a single stimulus that correspond to the perception of being in the work place.

$$\begin{array}{|l} workPlace : Stimulus \end{array}$$

- *Home stimulus.* Similarly, the worker's home is synthesized in a single stimulus.

$$\begin{array}{|l} home : Stimulus \end{array}$$

- *Work action.* This is the action that the employer wishes the agent to perform.

$$\begin{array}{|l} work : Action \end{array}$$

- *Buy food action.* This is the action that will lead to actual pleasure.

$$\begin{array}{|l} buyFood : Action \end{array}$$

- *Wake up early action.* This will be required in order to work properly.

$$\begin{array}{|l} wakeUpEarly : Action \end{array}$$

- *Operants.* We need one operant for each action above.

$$\begin{array}{|l} worker.operants = \{wakeUpEarlyO, workO, buyFoodO\} \end{array}$$

- *Wake up early operant.* Notice that *home* is a required antecedent stimulus. Once the action *wakeUpEarly* is performed, our simplified environment defines that the worker reaches the *workPlace* stimulus.

$$\begin{array}{|l} wakeUpEarlyO : Operant \\ \hline wakeUpEarlyO.antecedents = \{\{home\}\} \\ wakeUpEarlyO.action = wakeUpEarly \\ wakeUpEarlyO.consequence = workPlace \\ wakeUpEarlyO.consequenceContingency = \{(\{home\}, 0.9)\} \end{array}$$

- *Work operant.* Notice that the *workPlace* is the consequent of the previous operant and a required antecedent for the *workO* operant below. That is to say, one operant has set the opportunity to do another, and that's why they are chained. The *workO* operant below, in turn, will cause the worker to acquire *money*.

$$
\begin{array}{|l}
workO : Operant \\
\hline
workO.antecedents = \{\{workPlace\}\} \\
workO.action = work \\
workO.consequence = money \\
workO.consequenceContingency = \{(\{workPlace\}, 0.9)\}
\end{array}
$$

- *Buy food operant.* Finally, because of the *money* acquired with the previous operant, it is possible to buy food.

$$
\begin{array}{|l}
buyFoodO : Operant \\
\hline
workO.antecedents = \{\{money\}\} \\
workO.action = buyFood \\
workO.consequence = food \\
workO.consequenceContingency = \{(\{money\}, 0.9)\}
\end{array}
$$

9.1.2.2 EMMAS Specification

This chain depends on the environment reinforcing the several operants. For example, if the worker no longer receives money after his work, he will eventually stop working and the chain will be broken. And by the same token, the chain could be extended by reinforcing other actions in a similar pattern before providing the *money* stimulus.

The following defines the **agent profile** of the worker.

$$
\begin{array}{ll}
S = & \{food, money, home, work_place\} \\
A = & \{work, buy_food, wake_up_early\} \\
worker = & \langle 0, S, A \rangle \\
AG = & \{worker\}
\end{array}
$$

Since the agent is alone, there are no **action transformers**.

$$AT = \emptyset$$

The **environment behaviours**, in turn, are designed to start and maintain the operant chain.

$$\begin{aligned}
eb_1 &= \mathit{Stimulate}(\mathit{home}, \mathit{worker}) \\
eb_2 &= \mathit{ER}(\mathit{wake_up_early}, \mathit{worker}, \mathit{Stimulate}(\mathit{work_place}, \mathit{worker})) \\
eb_3 &= \mathit{ER}(\mathit{work}, \mathit{worker}, \mathit{Stimulate}(\mathit{money}, \mathit{worker})) \\
eb_4 &= \mathit{ER}(\mathit{buy_food}, \mathit{worker}, \mathit{Stimulate}(\mathit{food}, \mathit{worker})) \\
EB &= \{eb_1, eb_2, eb_3, eb_4\}
\end{aligned}$$

Let us comment on these:

- eb_1 merely puts the agent in an initial state (i.e., his home) so that the initial condition for the chain may hold.
- eb_2 and eb_3 define **environment responses**, which rewards each operant in the chain with the stimulus required for the next operant.
- eb_4 defines an **environment response** that delivers the primary stimulus sought by the agent (i.e., food).

This gives us the **environment**.

$$\langle AG, AT, EB \rangle$$

9.1.2.3 Simulation Purpose

Let us build a **simulation purpose** to exercise the operant chain. We first need the **states** and **events** related to the operants to be performed.

$$\begin{aligned}
Q &= \{q_0, q_1, q_2, q_3, \mathit{Success}, \mathit{Failure}\} \\
E &= \{?\mathit{beginning}^0_{\mathit{home}}, !\mathit{emit}^0_{\mathit{wake_up_early}}, !\mathit{emit}^0_{\mathit{work}}, !\mathit{emit}^0_{\mathit{buy_food}}\} \\
P &= \{\} \\
L &= \{\}
\end{aligned}$$

The transition then require that the three actions (i.e., *wake_up_early*, *work*, *buy_food*) be performed in order.

$$\begin{aligned}
\rightsquigarrow = \ & \{(q_0, \circledast, q_0), (q_0, ?\mathit{beginning}^0_{\mathit{home}}, q_1), \\
& (q_1, \circledast, q_1), (q_1, !\mathit{emit}^0_{\mathit{wake_up_early}}, q_2), \\
& (q_2, \circledast, q_2), (q_2, !\mathit{emit}^0_{\mathit{work}}, q_3), \\
& (q_3, \circledast, q_3), (q_3, !\mathit{emit}^0_{\mathit{buy_food}}, \mathit{Success})\}
\end{aligned}$$

We thus have the **simulation purpose**.

$$\langle Q, E, P, \rightsquigarrow, L, q_0 \rangle$$

9.1.2.4 Result

FGS checked in one second that the **simulation purpose** is indeed **weakly feasible**.

9.2 Multi-Agent Examples

Though isolated agent experiments provide the basis for the analysis of behaviour, it is ultimately agents within a social context that one wishes to study, since in practice organism always interact. In this section, then, we consider examples of such social interaction that can be modelled using our approach.

9.2.1 Violent Child: Behaviour Elimination Through Reinforcement

It is often the case that one wishes to eliminate some operant behaviour. An obvious way to do it is to punish the agent when it performs such an operant. But punishment has its own undesirable consequences, such as the emotional influence that it brings.

Interestingly, there is an alternate way of eliminating operants, by using reinforcement instead. This approach consists in reinforcing some other behaviour which: (i) is something desirable; and (ii) is in conflict with the operant to be eliminated (i.e., the agent cannot perform both at the same time). By this method, one both avoids the problems with punishment and create a new and valuable behaviour.

As an example, let us consider a child who often misbehaves by beating her dog. She does so because she finds the resulting dog's scream amusing. This child, moreover, sometimes do caress the dog, albeit rarely.

Clearly, one cannot beat and caress the dog at the same time, since these actions depend on the same mechanism (i.e., the child's hand). So to eliminate the behaviour of beating the dog, we reinforce the action of caress. If the reinforcement is more pleasant to the child than the dog's scream, then the behaviour is successfully changed.

We model both the child and the dog. Their **environment** can provide some candy, which is reinforcing to the child. By means of a **simulation purpose**, we guide the simulations so that the child only gets the candy when performing the caress action. The **simulation purpose**, to be **feasible**, also requires the beating behaviour to disappear.

9.2.1.1 Agents Parameters

Let us begin by modelling the child as follows:

- *Stimuli.* We assume that this child enjoys receiving candy, as well as hearing the dog's scream. But for the technique to work, we will assume that the child actually prefers the candy.

$$
\begin{array}{|l}
child : Organism \\
candy, scream_sound : Stimulus \\
\hline
child.primaryStimuli = \{candy, scream_sound\} \\
child.primary_utility(candy) = 0.6 \\
child.primary_utility(scream_sound) = 0.2
\end{array}
$$

$$
\begin{array}{|l}
child.max_delay = 100 \\
child.c = 0.5 \\
child.pleasureHints = \{\} \\
child.painHints = \{scream_sound\}
\end{array}
$$

- *Actions.* The actions, in turn, are as follows. It is specially important to state that they are conflicting and that they may happen spontaneously.

$$
\begin{array}{|l}
caress, beat : Action \\
\hline
child.operantActions = \{caress, beat\} \\
child.reflexActions = \{\} \\
child.conflict(caress, beat) = conflicting \\
child.baseLevel(caress) = 0.2 \\
child.baseLevel(beat) = 0.0
\end{array}
$$

- *Operants.* Finally, let us assume the child already knows that when she beats the dog, it will scream.

$$
\begin{array}{|l}
o_1 : Operant \\
\hline
child.operants = \{o_1\} \\
o_1.antecedents = \{\emptyset\} \\
o_1.action = beat \\
o_1.consequence = scream_sound \\
o_1.consequenceContingency = \{(\emptyset, 0.9)\}
\end{array}
$$

- *Drive.* The child periodically gets tired of hearing the dog's scream. This is modelled as a drive.

$$
\begin{array}{|l}
d : \mathit{Drive} \\
\hline
\mathit{child.activeDrives} = \{d\} \\
d.\mathit{desires} = \{\mathit{scream_sound}\} \\
d.\mathit{importance} = 0.0 \\
d.\mathit{maxImportance} = 0.0 \\
d.\mathit{minImportance} = -1.0
\end{array}
$$

The dog, in turn, is modelled as follows:

- *Stimuli.* Both of the relevant stimuli for the dog are primary, but one is unpleasant, whereas the other is pleasant.

$$
\begin{array}{|l}
\mathit{dog} : \mathit{Organism} \\
\mathit{punch}, \mathit{caress} : \mathit{Stimulus} \\
\hline
\mathit{dog.primaryStimuli} = \{\mathit{punch}, \mathit{caress}\} \\
\mathit{dog.primary_utility}(\mathit{punch}) = -0.5 \\
\mathit{dog.primary_utility}(\mathit{caress}) = 0.5
\end{array}
$$

$$
\begin{array}{|l}
\mathit{dog.max_delay} = 100 \\
\mathit{dog.c} = 0.5 \\
\mathit{dog.pleasureHints} = \{\} \\
\mathit{dog.painHints} = \{\}
\end{array}
$$

- *Actions.* The dog can scream in pain or wag its tail when happy. Moreover, to make the example more complex, the dog also barks spontaneously.

$$
\begin{array}{|l}
\mathit{bark}, \mathit{scream}, \mathit{wag_tail} : \mathit{Action} \\
\hline
\mathit{dog.operantActions} = \{\mathit{bark}, \mathit{wag_tail}\} \\
\mathit{dog.reflexActions} = \{\mathit{scream}, \mathit{wag_tail}\} \\
\mathit{dog.baseLevel}(\mathit{bark}) = 0.01 \\
\mathit{dog.baseLevel}(\mathit{scream}) = 0.0 \\
\mathit{dog.baseLevel}(\mathit{wag_tail}) = 0.0
\end{array}
$$

- *Reflexes.* The dog has two reflexes, one to model the fact that it can complain when it receives a punch, and the other to model that it wags its tail when caressed.

$$
\begin{array}{|l}
complain, enjoy : Reflex \\
\hline
dog.reflexes = \{complain, enjoy\} \\
complain.antecedent = punch \\
complain.action = scream \\
enjoy.antecedent = caress \\
enjoy.action = wag_tail
\end{array}
$$

9.2.1.2 EMMAS Specification

Profiles are needed for the child and the dog.

$$
\begin{array}{ll}
S_c = & \{candy, scream_sound, neutral\} \\
A_c = & \{beat, caress\} \\
S_d = & \{punch, caress\} \\
A_d = & \{bark, scream, wag_tail\} \\
child = & \langle 0, S_c, A_c \rangle \\
dog = & \langle 1, S_d, A_d \rangle \\
AG = & \{child, dog\}
\end{array}
$$

We must specify how the actions of the agents affect each other. For this example, it suffices to allow the child to both punch and caress the dog, and allow the dog to scream to the child.

$$
\begin{array}{ll}
AT = & \{ \\
& \langle child, beat, punch, dog \rangle, \langle child, caress, caress, dog \rangle, \\
& \langle dog, scream, scream_sound, child \rangle \\
& \}
\end{array}
$$

The **environment** allows an experimenter to give as many candies to the child as desired. This does not mean that an unbounded amount of candy will be effectively given to the child, but simply that it is *possible* to do so. This shows how an **environment** can provide a kind of instrument (in this case, a candy dispenser) that is later put to use (i.e., by means of a **simulation purpose** and the related verification technique). Moreover, the environment also allows the provision of a neutral stimulus, which can be used as a discriminative stimulus when teaching new behaviours.

$$
\begin{aligned}
eb_1 = &\quad Forever(Stimulate(candy, child)) \\
eb_2 = &\quad Forever(Stimulate(neutral, child)) \\
EB = &\quad \{eb_1, eb_2\}
\end{aligned}
$$

We thus have the following **environment**.

$$\langle AG, AT, EB \rangle$$

9.2.1.3 Simulation Purpose

In order to guide properly the simulation, we use several **states** and **events**. We also employ a proposition to check whether the child really likes candy, for otherwise the procedure cannot possibly work.

$$
\begin{aligned}
Q = &\quad \{q_i \mid 0 \leq i \leq 47\} \cup \{Success, Failure\} \\
P = &\quad \{LikesCandy\} \\
L = &\quad \{(q_1, \{LikesCandy\})\} \\
E = &\quad \{emit^0_{beat}, stop^0_{beat}, emit^0_{caress}, stop^0_{caress}, \\
&\quad beginning^0_{candy}, stable^0_{candy}, ending^0_{candy}, absent^0_{candy}, \\
&\quad beginning^0_{neutral}, stable^0_{neutral}, ending^0_{neutral}, absent^0_{neutral}, \\
&\quad beginning^0_{scream_sound}, stable^0_{scream_sound}, ending^0_{scream_sound}, \\
&\quad absent^0_{scream_sound}, emit^1_{scream}, stop^1_{scream}, beginning^1_{caress}, \\
&\quad stable^1_{caress}, ending^1_{caress}, absent^1_{caress}, beginning^1_{punch}, \\
&\quad stable^1_{punch}, ending^1_{punch}, absent^1_{punch}\}
\end{aligned}
$$

The procedure to adopt is the following sequence:

1. let the child beat the dog. This involves a number of **events**, such as the beating action, the punch perceived by the dog, and the resulting scream. Because the desire to hear the scream is regulated by a drive, the child will get tired of beating the dog;
2. wait for a caress and reward it;
3. make sure that the child is no longer beating the dog, nor the dog is feeling the previous punch, and in the process define any new beating as a leading to *Failure*;
4. wait for the next caress action, which indicates that although the child had time to recover the wish to hear screams, it now prefers to caress the dog instead.

This corresponds to the following set of transitions.

$$
\begin{aligned}
\rightsquigarrow = \; & \{(q_0, beginning^0_{neutral}, q_1), (q_1, ?commit, q_1), (q_1, emit^0_{beat}, q_3), \\
& (q_3, ?commit, q_4), (q_4, beginning^1_{punch}, q_5), (q_5, ?commit, q_6), \\
& (q_6, stable^1_{punch}, q_7), (q_7, stable^0_{neutral}, q_8), (q_8, ?commit, q_9), \\
& (q_9, ?commit, q_{10}), (q_{10}, emit^1_{scream}, q_{11}), (q_{11}, beginning^0_{scream_sound}, q_{12}), \\
& (q_{12}, ?commit, q_{13}), (q_{13}, stable^0_{scream_sound}, q_{14}), (q_{14}, ?commit, q_{14}), \\
& (q_{14}, stop^1_{scream}, q_{15}), (q_{15}, ?commit, q_{16}), (q_{16}, ending^0_{scream_sound}, q_{17}), \\
& (q_{17}, ?commit, q_{18}), (q_{18}, absent^0_{scream_sound}, q_{19}), (q_{19}, ?commit, q_{20}), \\
& \\
& (q_{20}, emit^0_{caress}, q_{21}), (q_{21}, beginning^1_{caress}, q_{22}), (q_{22}, ?commit, q_{23}), \\
& (q_{23}, stable^1_{caress}, q_{24}), (q_{24}, ?commit, q_{24}), (q_{24}, stop^0_{caress}, q_{25}), \\
& (q_{25}, beginning^0_{candy}, q_{26}), (q_{26}, ending^1_{caress}, q_{27}), (q_{27}, ?commit, q_{28}), \\
& (q_{28}, absent^1_{caress}, q_{29}), (q_{29}, stable^0_{candy}, q_{30}), (q_{30}, ?commit, q_{31}), \\
& (q_{31}, \tau, q_{32}), \\
& \\
& (q_{32}, ending^0_{candy}, q_{33}), (q_{33}, ?commit, q_{34}), \\
& (q_{33}, emit^0_{beat}, Failure), \\
& (q_{34}, stop^0_{beat}, q_{35}), (q_{34}, emit^0_{beat}, Failure), \\
& (q_{35}, ending^1_{punch}, q_{36}), (q_{35}, emit^0_{beat}, Failure), \\
& (q_{36}, ?commit, q_{37}), \\
& (q_{36}, emit^0_{beat}, Failure), \\
& (q_{37}, absent^1_{punch}, q_{38}), (q_{37}, emit^0_{beat}, Failure), \\
& (q_{38}, ?commit, q_{39}), (q_{38}, emit^0_{beat}, Failure), (q_{39}, \circledast, q_{40}), \\
& (q_{39}, emit^0_{beat}, Failure), \\
& (q_{40}, \circledast, q_{41}), (q_{37}, emit^0_{beat}, Failure), \\
& (q_{41}, \circledast, q_{42}), (q_{41}, emit^0_{beat}, Failure), \\
& \\
& (q_{42}, emit^0_{caress}, Success) \\
& \}
\end{aligned}
$$

This gives us the **simulation purpose**.

$$\langle Q, E, P, \rightsquigarrow, L, q_0 \rangle$$

9.2.1.4 Result

FGS verified in four seconds that the **simulation purpose** is **strongly feasible**. The **strong** variant is used because we wish to assert that it is not even possible at a certain point for the event $emit^0_{beat}$ to take place (which leads to *Failure*).

9.2.2 Factory: Rearranging a Social Network

Many tasks require the cooperation of several individuals. The modern factory, with its assembly line, is an example of this. The main aspect in such a factory is, of course, the division of labour according to the technical requirements of the product being built. This we do not address here. We wish, here, to show a different dimension to this problem, namely, that of social interaction.

The factory owner would like workers to be replaceable commodities. However, this is not quite the case. Workers may, for instance, have a hard time working with certain others. They may be lazy, or uninterested. They might or might not have better things to think about. All this influences in their productive behaviour, and therefore is important.

In this example we assume the existence of a certain set of workers and managers. We wish to discover how to setup an assembly line so that work actually gets done.

9.2.2.1 Agents Parameters

In this example we consider the following agents.

$$m_1, m_2 : \mathit{Organism}$$

$$w_1, w_2, w_3 : \mathit{Organism}$$

Since the specification of each individual agent takes considerable space, in this example we show explicitly only one agent of each kind to be considered (i.e., managers and workers).

Let us begin by the managers:

- *Stimuli.* Managers only receive money.

$$\mathit{money} : \mathit{Stimulus}$$

$$m_1.\mathit{primaryStimuli} = \{\mathit{money}\}$$

$$m_1.\mathit{primary_utility}(\mathit{money}) = 0.9$$

$$m_1.\mathit{max_delay} = 10$$

$$m_1.c = 0.5$$

$$m_1.\mathit{pleasureHints} = \{\mathit{money}\}$$

$$m_1.\mathit{painHints} = \{\}$$

- *Actions.* The job of a manager is to give orders.

$$order_work : Action$$

$$m_1.operantActions = \{order_work\}$$
$$m_1.reflexActions = \{\}$$
$$m_1.baseLevel(order_work) = 0.0$$

- *Operants.* Managers have already learned that their work is rewarded.

$$o_1, o_2 : Operant$$
$$m_1.operants = \{o_1\}$$

$$o_1.antecedents = \{\emptyset\}$$
$$o_1.action = order_work$$
$$o_1.consequence = money$$
$$o_1.consequenceContingency = \{(\emptyset, 1.0)\}$$

$$o_2.antecedents = \{\emptyset\}$$
$$o_2.action = order_work$$
$$o_2.consequence = money$$
$$o_2.consequenceContingency = \{(\emptyset, 1.0)\}$$

Workers, in turn, are specified as follows.

- Stimuli. In an assembly line, each worker's output is the input for the next one. In this example we abstract the nature of the object being manufactured, and define different "work products" to represent the intermediary and final stages of the object. Besides working, the workers may hear others' speeches, and may also have some food.

$$food : Stimulus$$
$$work_product_0, work_product_1, work_product_2, work_product_3 : Stimulus$$
$$speech_1, speech_2, speech_3 : Stimulus$$

$$w_1.primaryStimuli = \{money, food\}$$
$$w_1.primary_utility(money) = 0.9$$
$$w_1.primary_utility(food) = 0.7$$

$$w_1.max_delay = 10$$
$$w_1.c = 0.5$$
$$w_1.pleasureHints = \{money\}$$
$$w_1.painHints = \{\}$$

- *Actions.* Workers can work in several parts of the assembly line, and to each part a different kind of work is required. They may also chat.

$$work_1, work_2, work_3 : Action$$
$$chat_1, chat_2, chat_3 : Action$$

$$w_1.operantActions = \{work_1, work_2, work_3, chat_1, chat_2, chat_3\}$$
$$w_1.reflexActions = \{\}$$
$$w_1.baseLevel(work_1) = 0.0$$
$$w_1.baseLevel(work_2) = 0.0$$
$$w_1.baseLevel(work_3) = 0.0$$
$$w_1.baseLevel(chat_1) = 0.3$$
$$w_1.baseLevel(chat_2) = 0.3$$
$$w_1.baseLevel(chat_3) = 0.3$$

$$w_1.conflict(work_1, chat_1) = conflicting$$
$$w_1.conflict(work_1, chat_2) = conflicting$$
$$w_1.conflict(work_1, chat_3) = conflicting$$
$$w_1.conflict(work_2, chat_1) = conflicting$$
$$w_1.conflict(work_2, chat_2) = conflicting$$
$$w_1.conflict(work_2, chat_3) = conflicting$$
$$w_1.conflict(work_3, chat_1) = conflicting$$
$$w_1.conflict(work_3, chat_2) = conflicting$$
$$w_1.conflict(work_3, chat_3) = conflicting$$

- *Operants.* Workers know that given a certain input, if they perform the right kind of work, they are going to get paid.

$$\frac{o_1^1, o_2^1, o_3^1 : Operant}{w_1.operants = \{o_1^1, o_2^1, o_3^1\}}$$

$$\begin{aligned}
&o_1^1.antecedents = \{\{work_product_0\}\}\\
&o_1^1.action = work_1\\
&o_1^1.consequence = money\\
&o_1^1.consequenceContingency = \{(\{work_product_0\}, 1.0)\}
\end{aligned}$$

$$\begin{aligned}
&o_1^2.antecedents = \{\{work_product_1\}\}\\
&o_1^2.action = work_2\\
&o_1^2.consequence = money\\
&o_1^2.consequenceContingency = \{(\{work_product_1\}, 1.0)\}
\end{aligned}$$

$$\begin{aligned}
&o_1^1.antecedents = \{\{work_product_2\}\}\\
&o_1^1.action = work_3\\
&o_1^1.consequence = money\\
&o_1^1.consequenceContingency = \{(\{work_product_2\}, 1.0)\}
\end{aligned}$$

9.2.2.2 EMMAS Specification

As we have seen, there are two types of agents: managers and workers. While each agent has a different behaviour, the **agent profiles** of each type share the same actions and stimuli. Below, m_1 and m_2 are possible managers, and w_1, w_2 and w_3 are workers.

$$\begin{aligned}
S_m &= \{money\}\\
A_m &= \{order_work\}\\
S_w &= \{money, work_product_0, work_product_1, work_product_2,\\
&\quad work_product_3, conversation_1, conversation_2\}\\
A_w &= \{work_1, work_2, work_3, chat_1, chat_2\}\\
m_1 &= \langle 0, S_m, A_m\rangle\\
m_2 &= \langle 1, S_m, A_m\rangle\\
w_1 &= \langle 11, S_w, A_w\rangle\\
w_2 &= \langle 12, S_w, A_w\rangle\\
w_3 &= \langle 13, S_w, A_w\rangle
\end{aligned}$$

We thus have the following agents.

$$AG = \{m_1, m_2, w_1, w_2, w_3\}$$

The communication among the agents is determined during the simulations, so initially there are no **action transformers**.

$$AT = \emptyset$$

To specify the environment's behaviours, we first define a set comprising of only workers.

$$W = \{w_1, w_2, w_3\}$$

The behaviours are then as follows.

$$\begin{aligned}
eb_1 = \;& ER(order_work, m_1, Stimulate(money, m_1)) \\
eb_2 = \;& ER(order_work, m_2, Stimulate(money, m_2)) \\
eb_3 = \;& ER(work_3, w_1, Stimulate(money, m_1) \parallel Stimulate(money, m_2) \parallel \\
& Stimulate(money, w_1) \parallel Stimulate(money, w_2) \parallel Stimulate(money, w_3)) \\
eb_4 = \;& ER(work_3, w_2, Stimulate(money, m_1) \parallel Stimulate(money, m_2) \parallel \\
& Stimulate(money, w_1) \parallel Stimulate(money, w_2) \parallel Stimulate(money, w_3)) \\
eb_5 = \;& ER(work_3, w_3, Stimulate(money, m_1) \parallel Stimulate(money, m_2) \parallel \\
& Stimulate(money, w_1) \parallel Stimulate(money, w_2) \parallel Stimulate(money, w_3)) \\
eb_6 = \;& \forall_+ x : W \bullet \forall_+ y : W \mid x \neq y \bullet \forall_+ z : W \mid z \neq x \wedge z \neq y \bullet \\
& (Create(m_1, order_work, work_product_0, x) + \\
& Create(m_2, order_work, work_product_0, x)); \\
& Create(x, work_1, work_product_1, y); \\
& Create(y, work_2, work_product_2, z); \\
& Create(x, chat_1, conversation_1, y); \\
& Create(y, chat_1, conversation_1, x); \\
& Create(y, chat_1, conversation_1, z); \\
& Create(z, chat_1, conversation_1, y) \\
EB = \;& \{eb_1, eb_2, eb_3, eb_4, eb_5, eb_6\}
\end{aligned}$$

Let us comment on them:

- eb_1 and eb_2 define that the managers are paid for performing their work. This ensures that they keep doing it, for otherwise the associated operant would be extinct (i.e., unlearned).
- eb_3, eb_4 and eb_5 define that when the action $work_3$ is performed by one of the agents, everyone gets paid. That is to say, the final step in the production results in a reward for everyone that participated, and not just the last agent in the assembly line.

- eb_6 is a succinct way of specifying many possible configurations of the communications between the agents. It defines that only one manager will be chosen, and that later any of the possible sequences of three workers constitutes the assembly line. Besides passing work products to each other, the workers can also chat if they are adjacent. By expanding this definition, we get several choices, among which the three first could look like this:

$$
\begin{aligned}
&((Create(m_1, order_work, work_product_0, w_1)+\\
&Create(m_2, order_work, work_product_0, w_1));\\
&Create(w_1, work_1, work_product_1, w_2);\\
&Create(w_2, work_2, work_product_2, w_3);\\
&Create(w_1, chat_1, conversation_1, w_2);\\
&Create(w_2, chat_1, conversation_1, w_1);\\
&Create(w_2, chat_1, conversation_1, w_3);\\
&Create(w_3, chat_1, conversation_1, w_2))\\
&+\\
&((Create(m_1, order_work, work_product_0, w_2)+\\
&Create(m_2, order_work, work_product_0, w_2));\\
&Create(w_2, work_1, work_product_1, w_1);\\
&Create(w_2, work_2, work_product_2, w_3);\\
&Create(w_2, chat_1, conversation_1, w_1);\\
&Create(w_1, chat_1, conversation_1, w_2);\\
&Create(w_1, chat_1, conversation_1, w_3);\\
&Create(w_3, chat_1, conversation_1, w_1))\\
&+\\
&((Create(m_1, order_work, work_product_0, w_3)+\\
&Create(m_2, order_work, work_product_0, w_3));\\
&Create(w_3, work_1, work_product_1, w_1);\\
&Create(w_1, work_2, work_product_2, w_2);\\
&Create(w_3, chat_1, conversation_1, w_1);\\
&Create(w_1, chat_1, conversation_1, w_3);\\
&Create(w_1, chat_1, conversation_1, w_2);\\
&Create(w_2, chat_1, conversation_1, w_1))\\
&+\ldots
\end{aligned}
$$

This highlights one of the important features of our modelling approach: the definition of several situations of interest in a short form.[2]

[2]Our implementation, FGS, does not currently support the $\forall_+$ quantifier, because we have implemented only composition and core operations of **EMMAS**, the basis on which the others are defined. Therefore, in the XML input file for this example, we have manually expanded the quantifier. Nonetheless, it is of course possible to add the quantifier directly to the XML notation of FGS (and perform the corresponding expansion automatically) in future versions.

Given all that, we at last define the **environment**.

$$\langle AG, AT, EB \rangle$$

9.2.2.3 Simulation Purpose

We have the following sets of **states** and **events**, and no labelling in the **states**.

$$\begin{aligned}
Q = \; & \{q_i \mid 19 \leq i \leq 40\} \cup \{q_0, Success, Failure\} \\
E = \; & \{beginning^{11}_{work_product_0}, beginning^{11}_{work_product_1}, beginning^{11}_{work_product_2}, \\
& beginning^{12}_{work_product_0}, beginning^{12}_{work_product_1}, beginning^{12}_{work_product_2}, \\
& beginning^{13}_{work_product_0}, beginning^{13}_{work_product_1}, beginning^{13}_{work_product_2}, \\
& emit^{0}_{order_work}, emit^{1}_{order_work}, emit^{11}_{work_3}, emit^{12}_{work_3}, \\
& emit^{13}_{work_3}\} \\
P = \; & \{\} \\
L = \; & \{\}
\end{aligned}$$

Let us now specify transitions defining that the following (abstract) sequence takes place:

1. establish a social network configuration. The creation of each new **action transformer** generates a τ (internal) **event**, so we look for as many of them as necessary;
2. a manager must give the initial work order;
3. the order arrives to some agent;
4. some worker is defined as the first in the assembly line. We abstract which one by merely specifying the $\circledast$ (other) **event**. In a **feasible run**, this will synchronize with some $emit^{i}_{work_1}$ of some agent i. A number of !*commit* **events** are required in order to ensure that the action is properly accounted for;
5. the partial product gets to some next worker;
6. some worker is defined as the second in the assembly line, similarly to the one defined as the first;
7. the partial product gets to some next worker;
8. some worker is defined as the third (and last) in the assembly line, similarly to the ones defined as first and second. However, the result of this agent's action is *Success*.

This corresponds to the following set of transitions.

$$
\begin{aligned}
\rightsquigarrow = \quad & \{(q_0, \tau, q_{19}), (q_{19}, \tau, q_{19}), \\
& (q_{19}, emit^0_{order_work}, q_{20}), (q_{19}, emit^1_{order_work}, q_{20}), \\
& (q_{20}, beginning^{11}_{work_product_0}, q_{21}), (q_{20}, beginning^{12}_{work_product_0}, q_{21}), \\
& (q_{20}, beginning^{13}_{work_product_0}, q_{21}), \\
& (q_{21}, ?commit, q_{22}), (q_{22}, \circledast, q_{26}), \\
& (q_{22}, ?commit, q_{23}), (q_{23}, \circledast, q_{26}), (q_{23}, ?commit, q_{24}), \\
& (q_{24}, \circledast, q_{26}), (q_{24}, ?commit, q_{25}), (q_{25}, \circledast, q_{26}), \\
& (q_{26}, beginning^{12}_{work_product_1}, q_{27}), (q_{26}, beginning^{11}_{work_product_1}, q_{27}), \\
& (q_{26}, beginning^{13}_{work_product_1}, q_{27}), \\
& (q_{27}, ?commit, q_{28}), (q_{28}, \circledast, q_{32}), (q_{28}, ?commit, q_{29}), \\
& (q_{29}, \circledast, q_{32}), (q_{29}, ?commit, q_{30}), (q_{30}, \circledast, q_{32}), \\
& (q_{30}, ?commit, q_{31}), (q_{31}, \circledast, q_{32}), \\
& (q_{32}, beginning^{13}_{work_product_2}, q_{33}), \\
& (q_{32}, beginning^{11}_{work_product_2}, q_{33}), (q_{32}, beginning^{12}_{work_product_2}, q_{33}), \\
& (q_{33}, ?commit, q_{34}), (q_{34}, emit^{13}_{work_3}, Success), \\
& (q_{34}, emit^{11}_{work_3}, Success), (q_{34}, emit^{12}_{work_3}, Success), \\
& (q_{34}, ?commit, q_{35}), (q_{35}, emit^{13}_{work_3}, Success), \\
& (q_{34}, emit^{11}_{work_3}, Success), (q_{34}, emit^{12}_{work_3}, Success), \\
& (q_{35}, ?commit, q_{36}), (q_{36}, emit^{13}_{work_3}, Success), \\
& (q_{36}, emit^{11}_{work_3}, Success), (q_{36}, emit^{12}_{work_3}, Success), \\
& (q_{36}, ?commit, q_{37}), (q_{37}, emit^{13}_{work_3}, Success), \\
& (q_{37}, emit^{11}_{work_3}, Success), (q_{37}, emit^{12}_{work_3}, Success) \\
& \}
\end{aligned}
$$

In this sequence of transitions many things are abstracted. In particular, the **simulation purpose** does not define the order of the assembly line. It merely stipulates that some of the agents must occupy each position. It is during the verification that several configurations will be simulated, and one that is capable of really leading to *Success* is chosen. For instance, if two adjacent workers prefer to chat than to work, no work gets done in the assembly line, and therefore such a configuration would not lead to a **feasible run**.

We thus have the **simulation purpose**.

$$\langle Q, E, P, \rightsquigarrow, L, q_0 \rangle$$

9.2.2.4 Result

FGS took 47 seconds to verify **weak feasibility**. Part of this time is spent in backtracking from configurations that do not lead to *Success*, and reconfiguring the network to try again. This happens because we have setup two agents that prefer to chat than to work, so when they are placed in adjacent positions no work gets done.

Moreover, FGS took one second to check that the same **simulation purpose** is not **certain**. The reason is that it allows a network configuration in which only one **action transformer** is created, which of course cannot lead to *Success*, and therefore violates a requirement for **certainty** to hold.

9.2.3 School Children: From Chaos to Order

Social behaviour is nothing but the composition of individual behaviour. In this example we explore this idea by imagining a class of misbehaving school children put under the coordination of a teacher. The task can be complicated by the fact that children can respond differently to the same stimuli, may live in different environments, and may socialize with each other in a non-homogeneous manner. For instance, a child that has a television available at home may well do less of his or her homework than another child that has no such distraction available. On the other hand, an elevated interest in learning, which we assume to be a rather personal trait, may be sufficient to compensate for this.

The central idea to put order in this chaotic classroom is to allow the teacher to reward and punish students according to their behaviour (e.g., whether they have done their homework). This strategy, however, may not be sufficient. For example, even an interested child may take more pleasure in playing with certain others, provided that such friends are reachable. The simulation, then, helps in uncovering these less direct interferences. Indeed, since one can define an arbitrary topology to the social network, specify different traits for the kids, and provide each with different environmental possibilities, it does seem difficult to imagine *a priori* what the result of the teacher's action will be.

There are many behaviours one may wish the children to have. In this example we limit ourselves to checking whether they do their assigned homework, despite distractions and competing interests, provided that they are aware of the rewards involved.

9.2.3.1 Agents Parameters

In this example we consider a teacher and some children.

$t : Organism$

$c_1, c_2, c_3 : Organism$

We give the parameters for the teacher and one of the children, since the other are very similar.

The teacher is parametrized as follows:

- *Stimuli.* The teacher can perceive whether each student has individually done his or her homework, as well as whether he or she is annoying other students. Of course, the teacher is paid too.

  $money : Stimulus$

  $homework_1, homework_2, homework_3 : Stimulus$

  $see_annoying_1, see_annoying_2, see_annoying_3 : Stimulus$

  $t.primaryStimuli = \{money\}$

  $t.primary_utility(money) = 0.9$

  $t.max_delay = 10$

  $t.c = 0.5$

- *Actions.* The teacher can assign homework to the class, as well as reward and punish students individually.

  $assign_homework : Action$

  $reward_1, reward_2, reward_3 : Action$

  $punish_1, punish_2, punish_3 : Action$

  $t.operantActions = \{assign_homework, reward_1, reward_2, reward_3, punish_1, punish_2, punish_3\}$

  $t.reflexActions = \{\}$

  $t.baseLevel(assign_homework) = 0.6$

  $t.baseLevel(reward_1) = 0.0$

  $t.baseLevel(reward_2) = 0.0$

  $t.baseLevel(reward_3) = 0.0$

  $t.baseLevel(punish_1) = 0.0$

  $t.baseLevel(punish_2) = 0.0$

  $t.baseLevel(punish_3) = 0.0$

- *Operants.* The teacher knows that in doing her job she gets compensated. This includes not only assigning homework to students, but also rewarding and punishing them according to what they do.

$$\frac{o_1^t, o_2^t, o_3^t, o_4^t, o_5^t, o_6^t, o_7^t : \mathit{Operant}}{t.\mathit{operants} = \{o_1^t, o_2^t, o_3^t, o_4^t, o_5^t, o_6^t, o_7^t\}}$$

$$\begin{aligned}
&o_1^t.\mathit{antecedents} = \{\emptyset\}\\
&o_1^t.\mathit{action} = \mathit{assign_homework}\\
&o_1^t.\mathit{consequence} = \mathit{money}\\
&o_1^t.\mathit{consequenceContingency} = \{(\emptyset, 1.0)\}
\end{aligned}$$

$$\begin{aligned}
&o_2^t.\mathit{antecedents} = \{\{\mathit{homework}_1\}\}\\
&o_2^t.\mathit{action} = \mathit{reward}_1\\
&o_2^t.\mathit{consequence} = \mathit{money}\\
&o_2^t.\mathit{consequenceContingency} = \{(\{\mathit{homework}_1\}, 1.0)\}
\end{aligned}$$

$$\begin{aligned}
&o_3^t.\mathit{antecedents} = \{\{\mathit{homework}_2\}\}\\
&o_3^t.\mathit{action} = \mathit{reward}_2\\
&o_3^t.\mathit{consequence} = \mathit{money}\\
&o_3^t.\mathit{consequenceContingency} = \{(\{\mathit{homework}_2\}, 1.0)\}
\end{aligned}$$

$$\begin{aligned}
&o_4^t.\mathit{antecedents} = \{\{\mathit{homework}_3\}\}\\
&o_4^t.\mathit{action} = \mathit{reward}_3\\
&o_4^t.\mathit{consequence} = \mathit{money}\\
&o_4^t.\mathit{consequenceContingency} = \{(\{\mathit{homework}_3\}, 1.0)\}
\end{aligned}$$

$$\begin{aligned}
&o_5^t.\mathit{antecedents} = \{\{\mathit{see_annoying}_1\}\}\\
&o_5^t.\mathit{action} = \mathit{punish}_1\\
&o_5^t.\mathit{consequence} = \mathit{money}\\
&o_5^t.\mathit{consequenceContingency} = \{(\{\mathit{see_annoying}_1\}, 1.0)\}
\end{aligned}$$

$$o_6^t.antecedents = \{\{see_annoying_2\}\}$$
$$o_6^t.action = punish_2$$
$$o_6^t.consequence = money$$
$$o_6^t.consequenceContingency = \{(\{see_annoying_2\}, 1.0)\}$$

$$o_7^t.antecedents = \{\{see_annoying_3\}\}$$
$$o_7^t.action = punish_3$$
$$o_7^t.consequence = money$$
$$o_7^t.consequenceContingency = \{(\{see_annoying_3\}, 1.0)\}$$

A typical child, in turn, is described as follows:

- *Stimuli.* A number of stimuli are available to children. Some of them are related to their studies (e.g., prize, homework), and others to their everyday life (e.g., provocation, tv).

$$prize, disapproval, homework, provocation : Stimulus$$
$$information, tv, cry_sound, neutral : Stimulus$$

$$c_1.primaryStimuli = \{prize, disapproval, provocation, information, tv, cry_sound, neutral\}$$
$$c_1.primary_utility(prize) = 0.5$$
$$c_1.primary_utility(disapproval) = -0.2$$
$$c_1.primary_utility(provocation) = -0.4$$
$$c_1.primary_utility(information) = 0.8$$
$$c_1.primary_utility(tv) = 0.6$$
$$c_1.primary_utility(cry_sound) = 0.3$$
$$c_1.primary_utility(neutral) = 0.0$$

$$c_1.max_delay = 10$$
$$c_1.c = 0.5$$

- *Actions.* A child can do their homework, study, annoy others, watch tv, cry, or just let the time pass.

$$do_homework, study, annoy, watch_tv, cry, idle : Action$$

$$c_1.operantActions = \{do_homework, study, annoy, watch_tv, cry, idle\}$$
$$c_1.reflexActions = \{annoy\}$$
$$c_1.baseLevel(do_homework) = 0.0$$
$$c_1.baseLevel(study) = 0.3$$
$$c_1.baseLevel(annoy) = 0.0$$
$$c_1.baseLevel(watch_tv) = 0.1$$
$$c_1.baseLevel(cry) = 0.0$$
$$c_1.baseLevel(idle) = 0.0$$
$$c_1.conflict(study, watch_tv) = conflicting$$
$$c_1.conflict(do_homework, watch_tv) = conflicting$$
$$c_1.conflict(study, annoy) = conflicting$$
$$c_1.conflict(do_homework, annoy) = conflicting$$

- *Operants.* The child know a few things. First, doing nothing is boring. Second, doing homework is rewarded. Third, annoying other children makes them cry.

$$o_1^{c1}, o_2^{c1}, o_3^{c1} : Operant$$
$$c_1.operants = \{o_1^{c1}, o_2^{c1}, o_3^{c1}\}$$

$$o_1^{c1}.antecedents = \{\emptyset\}$$
$$o_1^{c1}.action = idle$$
$$o_1^{c1}.consequence = neutral$$
$$o_1^{c1}.consequenceContingency = \{(\emptyset, 1.0)\}$$

$$o_2^{c1}.antecedents = \{\{homework\}\}$$
$$o_2^{c1}.action = do_homework$$
$$o_2^{c1}.consequence = prize$$
$$o_2^{c1}.consequenceContingency = \{(\{homework\}, 1.0)\}$$

$$o_3^{c1}.antecedents = \{\emptyset\}$$
$$o_3^{c1}.action = annoy$$
$$o_3^{c1}.consequence = cry_sound$$
$$o_3^{c1}.consequenceContingency = \{(\emptyset, 1.0)\}$$

- *Reflexes.* Finally, the child has a reflex that elicits crying whenever the child suffers provocation.

$$revolt : Reflex$$
$$c_1.reflexes = \{revolt\}$$
$$revolt.antecedent = provocation$$
$$revolt.action = cry$$

9.2.3.2 EMMAS Specification

In this example we have again two types of agents, namely, a teacher (denoted by t) and the children (denoted by c_1, c_2, c_3). The teacher has an **agent profile** that allows him to provide homework, prizes, and rewards to students, as well as to receive money as payment from the school. The children, in turn, can perceive the relevant stimuli and perform the necessary actions to interact with such a teacher, but are also capable of interacting both among themselves and with other elements (e.g., television).

$$\begin{aligned}
S_t &= \{money, homework_1, homework_2, homework_3, see_annoying_1, \\
&\quad see_annoying_2, see_annoying_3\} \\
A_t &= \{assign_homework, reward_1, reward_2, reward_3, punish_1, \\
&\quad punish_2, punish_3\} \\
S_c &= \{prize, disapproval, homework, provocation, information, \\
&\quad tv, cry_sound, neutral\} \\
A_c &= \{do_homework, study, annoy, watch_tv, cry, idle\} \\
t &= \langle 0, S_t, A_t \rangle \\
c_1 &= \langle 1, S_c, A_c \rangle \\
c_2 &= \langle 2, S_c, A_c \rangle \\
c_3 &= \langle 3, S_c, A_c \rangle
\end{aligned}$$

We thus have the following agents.

$$AG = \{t, c_1, c_2, c_3\}$$

Owing to the authority of the teacher, and to the configuration of the class, there is a fixed social network that allows the agents to interact in several ways.

$$
\begin{aligned}
AT = \; & \{\langle t, reward_1, prize, c_1\rangle, \langle t, reward_2, prize, c_2\rangle, \langle t, reward_3, prize, c_3\rangle, \\
& \langle t, punish_1, disapproval, c_1\rangle, \langle t, punish_2, disapproval, c_2\rangle, \\
& \langle t, punish_3, disapproval, c_3\rangle, \\
& \langle t, assign_homework, homework, c_1\rangle, \langle t, assign_homework, homework, c_2\rangle, \\
& \langle t, assign_homework, homework, c_3\rangle, \\
& \langle c_1, do_homework, homework, t\rangle, \langle c_2, do_homework, homework, t\rangle, \\
& \langle c_3, do_homework, homework, t\rangle, \\
& \langle c_1, annoy, provocation, c_2\rangle, \langle c_2, annoy, provocation, c_1\rangle, \\
& \langle c_1, annoy, provocation, c_3\rangle, \langle c_3, annoy, provocation, c_1\rangle, \\
& \langle c_2, annoy, provocation, c_3\rangle, \langle c_3, annoy, provocation, c_2\rangle, \\
& \langle c_1, annoy, see_annoying_1, t\rangle, \langle c_2, annoy, see_annoying_2, t\rangle, \\
& \langle c_3, annoy, see_annoying_3, t\rangle, \\
& \langle c_1, cry, cry_sound, t\rangle, \langle c_1, cry, cry_sound, c_2\rangle, \\
& \langle c_1, cry, cry_sound, c_3\rangle, \\
& \langle c_2, cry, cry_sound, t\rangle, \langle c_2, cry, cry_sound, c_1\rangle, \\
& \langle c_2, cry, cry_sound, c_3\rangle, \\
& \langle c_3, cry, cry_sound, t\rangle, \langle c_3, cry, cry_sound, c_1\rangle, \\
& \langle c_3, cry, cry_sound, c_2\rangle\}
\end{aligned}
$$

The intuitive meaning of these several **action transformers** can be grasped from their definition. Nonetheless, let us comment on some of them:

- $\langle t, reward_1, prize, c_1\rangle$ specifies that the teacher has the power to reward student c_1 with a prize. Other similar **action transformers** specify other powers with respect to the children, such as the power to punish and assign homework. These are formalizations of the social norm that dictates that the teacher has a certain authority over the students.
- $\langle c_1, annoy, provocation, c_2\rangle$ specifies that the child c_1 can annoy c_2, an act that is perceived as a provocation. This does not mean that c_1 will annoy c_2, but only that it has the possibility of doing so. This can be the case, for example, if c_1 and c_2 sit close to one another in the class. However, **EMMAS** abstracts any such physical location properties, and preserves only the logical relation between the agents.
- $\langle c_1, annoy, see_annoying_1, t\rangle$ specifies that whenever c_1 annoys another child, the teacher will observe it. This is an example of how the same action can have more than one kind of consequence (i.e., annoy another child and allow the teacher to see this).

This social network allows teachers and children interact, which is already a source of behaviours for the MAS. Nevertheless, we can enrich the MAS further by specifying other environment behaviours to interact with the agents.

$$\begin{aligned}
eb_1 &= ER(study, c_1, Stimulate(information, c_1)) \\
eb_2 &= ER(study, c_2, Stimulate(information, c_2)) \\
eb_3 &= ER(study, c_3, Stimulate(information, c_3)) \\
eb_4 &= ER(watch_tv, c_1, Stimulate(tv, c_1)) \\
eb_5 &= ER(watch_tv, c_2, Stimulate(tv, c_2)) \\
eb_6 &= ER(watch_tv, c_3, Stimulate(tv, c_3)) \\
EB &= \{eb_1, eb_2, eb_3, eb_4, eb_5, eb_6\}
\end{aligned}$$

These behaviours allow students to either study or waste time in watching television. This gives an opportunity for the children to procrastinate their homework. During simulation, this might interfere in the observed behaviours, and thus either allow or prevent the satisfaction of a **simulation purpose**.

We thus have the **environment**.

$$\langle AG, AT, EB \rangle$$

9.2.3.3 Simulation Purpose

Let us build a **simulation purpose** to check whether homework is being done. To this end, the most important is to define the relevant **events**.

$$\begin{aligned}
Q &= \{q_0, q_1, q_2, q_3, Success, Failure\} \\
E &= \{!emit^0_{assign_homework}, !emit^1_{do_homework} \\
&\quad !emit^2_{do_homework}, !emit^3_{do_homework}\} \\
P &= \{\} \\
L &= \{\}
\end{aligned}$$

The transitions, in turn, are as follows. First we require the teacher to assign some homework. Then anything can happen, but in the end we require that at least one of the students actually do the homework.

$$\begin{aligned}
\rightsquigarrow = \ & \{(q_0, \circledast, q_0), \\
& (q_0, !emit^0_{assign_homework}, q_1), (q_1, \circledast, q_1), \\
& \\
& (q_1, !emit^1_{do_homework}, Success), (q_1, !emit^2_{do_homework}, Success), \\
& (q_1, !emit^3_{do_homework}, Success)\}
\end{aligned}$$

Thus we have the **simulation purpose**.

$$\langle Q, E, P, \rightsquigarrow, L, q_0 \rangle$$

9.2.3.4 Result

FGS took six seconds to check **weak feasibility**. In the **feasible run** found, it determined that at least agent identified by 1 did its assigned homework. In the process of doing so other **events** took place, such as children annoying each other and crying. These, however, did not prevent that agent from doing the homework.

9.2.4 Online Social Networks: Spreading a Message

We have so far investigated MASs in which the relationships among agents are an abstraction of what happens in the physical world. It is interesting then to consider now an example in which these abstract relations are already present in the object we wish to model. Such is the case in communications through the Internet, where interactions are necessarily mediated by network connections. Let us then consider a fictitious but truthful application, an online social network, which is nothing but a website where users may register themselves and interact with other registered users.[3] From a modelling perspective, the advantage of such an application is in its complete description of the underlying social network. So we can easily imagine that the application's owner may well use this information to model and simulate the network in order to learn more about it, or test certain interventions before applying them to the real network.

In this example, for simplicity, we consider an online social network for philosophers. In this manner we can model some typical philosophers and the things they enjoy doing, such as talking about philosophy (obviously), complaining about sophists or buying books. Once such a network is in place, we can then investigate its properties, such as whether the actions of an agent can affect the behaviour of agents which are not directly related to it (i.e., how actions propagate in the network). More specifically, we consider how advertisements displayed on the website affect agents which are not directly exposed to them.

9.2.4.1 Agents Parameters

The agents in this example are all similar to one another, so let us model explicitly only one of them here.

$$p_1 : \mathit{Organism}$$

[3] Actual examples of such networks currently include popular websites such as `www.facebook.com`, `www.twitter.com`, `plus.google.com`, `www.orkut.com` and `www.myspace.com`.

The relevant parameters are as follows:

- *Stimuli.* Philosophers can understand philosophy and sophisms, the first being delightful and the later hateful. They also enjoy books and jokes. Money is valued, but not much. Advertisement are valued because they normally relate to books in the website. Finally, virtual points distributed by the website only have value because they can be exchanged for real money.

$$philosophy, sophism, book, joke, money, ad_1 : Stimulus$$

$$p_1.primaryStimuli = \{philosophy, sophism, book, joke, money\}$$
$$p_1.primary_utility(philosophy) = 1.0$$
$$p_1.primary_utility(sophism) = -1.0$$
$$p_1.primary_utility(book) = 0.7$$
$$p_1.primary_utility(joke) = 0.6$$
$$p_1.primary_utility(money) = 0.3$$
$$ad_1 \ \underline{p_1.sCauses} \ book$$
$$points \ \underline{p_1.sCauses} \ money$$

$$p_1.max_delay = 100$$
$$p_1.c = 0.5$$

- *Actions.* There are two kinds of actions in this model. First, actions having to do with being a philosopher: writing philosophy, telling jokes and complaining. Second, actions related to operating the website, which are reduced merely to forwarding an advertisement to friends and buying what is advertised.

$$tell_joke, write_philosophy, complain, forward_ad_1, buy_ad_1 : Action$$

$$p_1.operantActions = \{tell_joke, write_philosophy, complain, forward_ad_1, buy_ad_1\}$$

$$p_1.reflexActions = \{complain\}$$

$$p_1.baseLevel(tell_joke) = 0.01$$

$$p_1.baseLevel(write_philosophy) = 0.3$$

$$p_1.baseLevel(complain) = 0.0$$

$$p_1.baseLevel(forward_ad_1) = 0.0$$

$$p_1.baseLevel(buy_ad_1) = 0.0$$

$$p_1.conflict(tell_joke, buy_ad_1) = conflicting$$

- *Operants.* The philosopher has learned previously that when forwarding and advertisement, he is rewarded with virtual points, which can be exchanged for real money later. This information is captured in an operant. The stimulus ad_1 must be present in order to be forwarded, so it is defined as a required antecedent. The action is defined as $forward_ad_1$, and the consequence is *points*.

$$o_1^{p1} : Operant$$

$$p_1.operants = \{o_1^{p1}\}$$

$$o_1^{p1}.antecedents = \{\{ad_1\}\}$$

$$o_1^{p1}.action = forward_ad_1$$

$$o_1^{p1}.consequence = points$$

$$o_1^{p1}.consequenceContingency = \{(\{ad_1\}, 1.0)\}$$

- *Reflexes.* Philosophers instinctively complain when reading sophisms.

$$revolt : Reflex$$

$$p_1.reflexes = \{revolt\}$$

$$revolt.antecedent = sophism$$

$$revolt.action = complain$$

9.2.4.2 EMMAS Specification

Let us begin by defining a few philosophers to populate the website.

$$
\begin{aligned}
S &= \{philosophy, sophism, book, joke, money, ad_1, points\} \\
A &= \{tell_joke, write_philosophy, complain, forward_ad_1, buy_ad_1\} \\
p_0 &= \langle 0, S, A\rangle \\
p_1 &= \langle 1, S, A\rangle \\
p_2 &= \langle 2, S, A\rangle \\
p_3 &= \langle 3, S, A\rangle \\
p_4 &= \langle 4, S, A\rangle \\
AG &= \{p_0, p_1, p_2, p_3, p_4\}
\end{aligned}
$$

The relations between agents are established by the actions they are capable of performing to each other. In a website this can be made much more concrete, since the user interface can define explicitly the available actions. The simple interface here allow the philosophers to forward ads and buy the related product, tell jokes and write philosophy to other agents.

$$
\begin{aligned}
AT = \; & \{ \\
& \langle p_0, forward_ad_1, ad_1, p_1\rangle, \langle p_0, forward_ad_1, ad_1, p_2\rangle, \\
& \langle p_2, forward_ad_1, ad_1, p_3\rangle, \\
& \\
& \langle p_0, tell_joke, joke, p_0\rangle, \langle p_1, tell_joke, joke, p_1\rangle, \\
& \langle p_2, tell_joke, joke, p_2\rangle, \langle p_3, tell_joke, joke, p_3\rangle, \\
& \\
& \langle p_0, tell_joke, joke, p_4\rangle, \langle p_1, tell_joke, joke, p_4\rangle, \\
& \langle p_2, tell_joke, joke, p_4\rangle, \langle p_3, tell_joke, joke, p_4\rangle, \\
& \\
& \langle p_0, write_philosophy, philosophy, p_1\rangle, \langle p_1, write_philosophy, philosophy, p_0\rangle, \\
& \langle p_0, write_philosophy, philosophy, p_2\rangle, \langle p_2, write_philosophy, philosophy, p_0\rangle, \\
& \langle p_2, write_philosophy, philosophy, p_3\rangle, \langle p_3, write_philosophy, philosophy, p_2\rangle, \\
& \\
& \langle p_0, buy_ad_1, book, p_0\rangle, \langle p_1, buy_ad_1, book, p_1\rangle, \\
& \langle p_2, buy_ad_1, book, p_2\rangle, \langle p_3, buy_ad_1, book, p_3\rangle \\
& \langle p_4, buy_ad_1, book, p_4\rangle \\
& \\
& \}
\end{aligned}
$$

Let us comment on some representative **action transformers**:

- $\langle p_0, forward_ad_1, ad_1, p_1\rangle$: defines that to forward an advertisement to p_1 means to stimulate p_1 with the advertisement.
- $\langle p_0, tell_joke, joke, p_4\rangle$ and $\langle p_0, write_philosophy, philosophy, p_1\rangle$: these represent the capability that p_1 has of sending messages to other agents (p_4 and p_1).

- p_4 only receives *joke*, but does not send *tell_joke*. This models the fact that p_4 subscribes to the jokes of others, but does not publish any himself.
- $\langle p_1, buy_ad_1, book, p_1 \rangle$: this models the fact that the action of buying results in a book to the buyer. This exemplifies the possibility of causing something to oneself.

The environment behaviours in this example are concerned with the advertisement infrastructure.

$$
\begin{aligned}
eb_1 = \; & BeginStimulation(ad_1, p_0) + \\
& BeginStimulation(ad_1, p_1) + \\
& BeginStimulation(ad_1, p_2) + \\
& BeginStimulation(ad_1, p_3) + \\
& BeginStimulation(ad_1, p_4) \\
eb_2 = \; & ER(forward_ad_1, p_0, Stimulate(points, p_0)) \\
eb_3 = \; & ER(forward_ad_1, p_1, Stimulate(points, p_1)) \\
eb_4 = \; & ER(forward_ad_1, p_2, Stimulate(points, p_2)) \\
eb_5 = \; & ER(forward_ad_1, p_3, Stimulate(points, p_3)) \\
eb_6 = \; & ER(forward_ad_1, p_4, Stimulate(points, p_4)) \\
EB = \; & \{eb_1, eb_2, eb_3, eb_4, eb_5, eb_6\}
\end{aligned}
$$

Let us examine these behaviours:

- eb_1 defines that the advertisement can be delivered to one of the agents. The objective in doing so is to see how an ad delivered to a particular agent reaches others.
- eb_2, eb_3, eb_4, eb_5 and eb_6 define, for each concerned agent, that a $forward_ad_1$ action is rewarded with *points*, a virtual currency that has value because it can be exchanged for real money. That is to say, agents have an incentive to forward advertisements.

We thus have the **environment**.

$$\langle AG, AT, EB \rangle$$

9.2.4.3 Simulation Purpose

This **simulation purpose** is concerned with checking whether an advertisement delivered to one agent can somehow cause *another* agent to buy the advertised product. This shows how **events** can propagate in the social network.

To this end, it is first necessary to define the relevant **events** for advertising and buying.

$$
\begin{aligned}
Q = \; & \{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7, \\
& q_8, q_9, q_{10}, q_{11}, Success, Failure\} \\
E = \; & \{beginning^0_{ad_1}, beginning^1_{ad_1}, beginning^2_{ad_1}, beginning^3_{ad_1}, \\
& emit^0_{forward_ad_1}, emit^2_{forward_ad_1}, emit^3_{buy_ad_1}\} \\
P = \; & \{\} \\
L = \; & \{\}
\end{aligned}
$$

With that we define the transitions. First the advertisement is delivered to an agent, which then forward it to others. The simulation is guided through some main points until the agent identified by 3 buys the advertised product.

$$
\begin{aligned}
\rightsquigarrow = \; & \{(q_0, beginning^0_{ad_1}, q_1), (q_1, \circledast, q_1), (q_1, emit^0_{forward_ad_1}, q_2), \\
& (q_2, \circledast, q_2), (q_2, beginning^2_{ad_1}, q_3), (q_2, \circledast, q_2), \\
& (q_3, emit^2_{forward_ad_1}, q_4), (q_3, \circledast, q_3), (q_4, beginning^3_{ad_1}, q_5), \\
& (q_4, \circledast, q_4), (q_5, emit^3_{buy_ad_1}, Success), (q_5, \circledast, q_5) \\
& \}
\end{aligned}
$$

This gives us the **simulation purpose**.

$$\langle Q, E, P, \rightsquigarrow, L, q_0 \rangle$$

9.2.4.4 Result

FGS verified in 30 seconds that the **simulation purpose** is **weakly feasible**. In the corresponding **feasible run**, we can see that an advertisement originally delivered to agent p_0 was eventually passed to agent p_3. In the course of verification, many irrelevant **events** were found (but promptly ignored), which explains the long time that it took to finish.

9.3 Conclusion

In this chapter we have seen concrete application examples of the approach proposed in this thesis. For each one, we have given an intuitive explanation of its importance, the corresponding formal specification and the result of running FGS.

Each example addresses different requirements, such as whether one or more agents are concerned and whether the communications between them are fixed or not. The objective of this variety is to show that our approach is sufficiently

general to express different situations in a uniform way. The importance of uniformity is that it allows the reuse of all underlying theory and technology, and therefore frees the user from having to come up with *ad hoc* solutions for individual cases. For instance, in the examples in which the communications between agents are fixed, one could devise an *ad hoc* solution to represent such an environment instead of using **EMMAS**. However, such a solution would imply in losing access to all the theory and tools that assume an **EMMAS environment**. Indeed, by modelling the situation of interest using the agents and **environments** described in this thesis, as well as asking questions through **simulation purposes**, one can then use the associated simulation and verification technology, which would be unavailable otherwise.

There are, of course, legitimate reasons why one would avoid using our modelling elements. First, they may not provide some required feature (e.g., the explicit numeric representation of time in the environment). Second, it may be more efficient to implement specific solutions instead of reusing our theory, since one can then optimize the implementation with no regard for other possible uses. Our method has a certain domain of application, and within it strives to provide a general approach, but this imposes such trade-offs.

The verification technique employs the specified **simulation purposes** to guide the simulation only through relevant simulation states. This means that the more details are present in a **simulation purpose**, the quicker the verification will be. The case studies showed that the actual level of detail required for a successful verification can vary. In some cases, such as the school from Section 9.2.3, an apparently complex situation may required only a simple **simulation purpose**. In other cases, such as the behaviour elimination from Section 9.2.1, a seemingly simple situation may demand a lengthy **simulation purpose**. The reason for such discrepancies is simply that each MAS has different subtleties concerning the interaction of its several elements, which can be independent of the quantity of the elements involved. The more such subtleties must be mastered, the more restrictive (and long) a **simulation purpose** gets. In practice, we have found that if a verification takes too long, one may make the **simulation purpose** employed more restrictive and try again. Furthermore, an **unfeasible simulation purpose** may become **feasible** if only minor adjustments are made (e.g., leaving enough !*commit* **events** to allow an action to be properly accounted for by the **environment** and other agents). Hence, the construction of a **simulation purpose** can also be done incrementally, by experimenting with successive variants.

The examples also showed that the approach can be used to model the MASs themselves at different levels of detail. For instance, one can assign fine grained actions to an agent, such as *beating a dog*, but also coarse and abstract actions such as *to work*. Thus, despite being based on the study of detailed actions of organisms (e.g., the pecking of birds), the behaviourist concepts we employ

are actually suitable to a broader class of modelling problems. It remains to study, however, how to properly relate several different abstraction levels of behaviour that could exist in the same MAS. In this respect, we have merely seen that individual operants may be chained in order to compose a more complex behaviour (in Section 9.1.2). We have not, however, provided any means by which the *same* operant could somehow be decomposed in finer behaviours. This is an interesting topic for future research, as it could allow the modelling of *shaping* (i.e., the gradual modification of a behaviour towards some desired form).

Finally, property components (which assign propositions to the simulation states) were less necessary in modelling than we had anticipated. The reason is that in a behaviourist framework most phenomena of interest can be determined by observing the interaction of agents with their environment, and therefore can be accounted for by **events**. Nonetheless, by providing such propositions, it is possible to determine such information in a more direct manner. In the example of Section 9.2.1, for instance, we define the *LikesCandy* proposition, which determines immediately whether the agent actually likes candy, instead of setting up an experiment to test the reinforcing power of candy. In practice, this might be used to model preferences declared by agents (e.g., in the example just mentioned, the child may have verbally revealed her preference for candy).

Part V

Conclusion

CHAPTER 10

Conclusion

In this thesis we have considered the broad problem of studying human and animal societies through computation. We have proposed a particular solution, in the form of a novel manner to model and analyse multi-agent systems. With respect to modelling, we presented a new agent architecture and an environment model. These are intended to provide a behaviourist account of MASs, which contrasts with more cognitive-oriented approaches. Our environments, in particular, are designed to facilitate experimentation with agents, in the spirit of the behaviourist tradition in psychology. Concerning analysis, we developed a new method, which combines ideas of both simulation and verification into a new technique. Together with the experimentation features of our environments, this technique provides a way to systematically explore possible simulations and check whether desired properties, in the form of **simulation purposes**, hold or not. To illustrate the capabilities and limitations of all this, we have developed a number of examples too. A tool, called FGS, has been implemented in order to give support to the approach.

In previous chapters we have already considered the immediate consequences and merits of the several elements that form our approach. Let us now take a wider view of our developments, and venture also in how they can be carried out further.

The **Behaviourist Agent Architecture** described in Chapter 4, while suitable for use as it is, can be subject to a number of improvements. As we have pointed out throughout the text, it offers a basis on which researchers may create their own mechanisms. This might be useful, for instance, if one wishes to specialize the architecture for a particular kind of agent, whose properties are better understood. It would also be interesting to augment the architecture with other psychological phenomena that can be given in

behavioural terms. For instance, to make the notion of schedules of reinforcement (Ferster and Skinner, 1957) explicit would be an important progress. It would also be important to incorporate the capability of *shaping* behaviours, by which the *same* behaviour is progressively modified until it reaches a final desired form.[1]

In its purely formal aspects, a significant improvement would consists in considering first-order actions and stimuli. This would allow the establishment of relations between predicates, independently of their variables. For example, the organism could learn that, for every x, $Push(x)$ implies in $SeeMoving(x)$. This would allow learning to be generalized, and not confined to particular values of x, as it is now. From a psychological point of view, though, this raises a number of questions concerning the inductive capabilities of organisms, and therefore it is not clear how to best use this formal device. For instance, how many particular observations of different x are required to establish the general rule?

Architecturally, yet another change may be useful. Our agents follow a behaviourist theory in order to learn about and interact with their environments. Behaviourism, however, says nothing about their internal physiology. As far as psychology is concerned, this allows the separation of concerns between behavioural problems and physiological ones. But behaviour may have physiological consequences. For example, if an organism acts in such a way that it contracts a virus, then it may very well get sick and transmit the virus to other organisms. Sickness may imply in reduced behaviour (much like depression), and transmission may be carried out by specific behaviours. Such an interaction between behaviour and other aspects of an organism could be achieved by adding layers in the agent architecture, so that a behavioural layer could somehow communicate with a physiological layer, perhaps among others. An improvement like this would allow the consideration of problems involving both aspects, such as epidemiology questions, which depend partly on the behaviour of agents, and partly on their internal susceptibility to disease. The difficulty would be in establishing the relations between these several layers.

EMMAS, the environment model introduced in Chapter 5, has been designed from the start to work with the **Behaviourist Agent Architecture**. However, it may be possible to adapt it to other agent architectures. For instance, the division of stimulation in four stages (i.e., absent, beginning, stable, ending) is very particular to the **Behaviourist Agent Architecture**, so perhaps it could suffice to change this (e.g., by reducing the phases merely to two: absent and present) in order to make it compatible with others, specially reactive architectures. Another example would consist in making the stim-

[1] This is distinct from operant chaining, a process by which a sequence of *different* behaviours is learned as the consequences of one operant establishes the required antecedents of the next (thus "chaining" them together).

uli non-primitive entities. Such a modification could be useful to cognitive architectures, since they usually require the exchange of structured messages between agents, which could impose further constraints on the possible evolutions of the MAS. However that may be, a successful adaptation to other agent architectures would bring with itself the possibility of applying the verification technique presented in this thesis.

Although the proposed approach is in principle capable of modelling and analysing large MASs, we have not investigated deeply the related scalability issues that may arise. Notably, as remarked in Chapter 8, each **EMMAS environment** is translated into a quite large π-calculus expression, which in practice brings a number of efficiency problems. We implemented a few optimizations to partly address this matter, it is likely that more can be done, but it is unclear to what extent such optimizations can scale. One may also wonder how easy or hard it is in general to specify a large MAS using the modelling elements given in the thesis (e.g., **agent profiles**, **action transformers**). However, if it turns out that the provided elements are insufficient to conveniently write certain large specifications, the problem could probably be addressed by providing a new set of higher-level specification elements, with an automatic mapping to the ones provided in the thesis. For instance, if one wishes to describe a random social network with respect to certain agents, stimuli and actions, an algorithm could be provided to translate such an abstract requirement into a set of **action transformers** of **EMMAS**. With respect to the scalability of the verification technique itself, it may be possible to parallelize the algorithms given. In principle, one could explore different parts of the **synchronous product** (e.g., disjoint subtrees) at the same time.

Finally, it is interesting to note that the verification method proposed may not be limited to simulations. The simulator interface provided in Section 7.5.1 could be adapted so that it interacts with the *actual world*, not a simulator. All the ideas concerning automated experiments could then be used to perform actual *empirical* experiments. So instead of, say, experimenting with a simulation of an online community, one could perform experiments in a real online community, provided that the user interface was properly adapted. The main problem to do such things is that, contrary to simulations, states of the actual world cannot be stored at one point and restored later. So the algorithms that we provided would have to be either changed to avoid this, or the states of the actual world would have to be partitioned into equivalence classes, so that one can at least return to an equivalent state.

APPENDIX A

Full Agent Specification

This appendix provides the full formal specification of the **Behaviour Agent Architecture** presented in Chapter 4. To facilitate consultation, the sections below follow the same structure found on Section 4.2 of that chapter.

A.1 Formal Specification of Agent Behaviour

Organism

StimulationSubsystem

RespondingSubsystem

DriveSubsystem

EmotionSubsystem

A.1.1 Preliminary Definitions

A.1.1.1 Magnitude

$$
\begin{aligned}
Q == \{ & q : \mathbb{Z} \times \mathbb{Z} \mid \mathbf{let}\ a == \mathit{first}\ q;\ b == \mathit{second}\ q \bullet \\
& b \neq 0 \land \\
& a \operatorname{div} b \leq 1 \land \\
& a \operatorname{div} b \geq -1 \}
\end{aligned}
$$

$$
\begin{aligned}
PositiveQ == \{ & q : \mathbb{Z} \times \mathbb{Z} \mid \mathbf{let}\ a == first\ q;\ b == second\ q \bullet \\
& b \neq 0 \wedge \\
& a \operatorname{div} b \leq 1 \wedge \\
& a \operatorname{div} b \geq 0 \}
\end{aligned}
$$

$$
\begin{array}{|l}
max_q : Q \\
neutral_q : Q \\
min_q : Q \\
\hline
max_q = (1, 1) \\
neutral_q = (0, 1) \\
min_q = (-1, 1)
\end{array}
$$

A.1.1.2 Intensity, Correlation and Probability

$$Intensity == PositiveQ$$

$$Correlation == PositiveQ$$

$$Probability == PositiveQ$$

$$max_intensity == max_q$$

$$min_intensity == neutral_q$$

$$max_correlation == max_q$$

$$min_correlation == neutral_q$$

$$max_probability == max_q$$

$$min_probability == neutral_q$$

A.1.1.3 Equality of Magnitudes

$$
\begin{array}{|l}
_ =_1 _ : \mathbb{P}(Q \times Q) \\
\hline
\forall\, p, q : Q \bullet \\
\quad \mathbf{let}\ a == first\ p;\ b == second\ p;\ c == first\ q;\ d == second\ q \bullet \\
\qquad p =_1 q \Leftrightarrow a * d = b * c
\end{array}
$$

A.1.1.4 Order Relations of Magnitudes

$$
\begin{array}{l}
_ \leq_1 _ : \mathbb{P}(Q \times Q) \\
\hline
\forall\, p, q : Q \bullet \\
\quad \mathbf{let}\ a == \mathit{first}\ p;\ b == \mathit{second}\ p;\ c == \mathit{first}\ q;\ d == \mathit{second}\ q \bullet \\
\qquad p \leq_1 q \Leftrightarrow \\
\qquad\quad (b * d > 0 \wedge a * d \leq b * c) \vee \\
\qquad\quad (b * d < 0 \wedge a * d \geq b * c)
\end{array}
$$

$$
\begin{array}{l}
_ <_1 _ : \mathbb{P}(Q \times Q) \\
\hline
\forall\, p, q : Q \bullet \\
\quad p <_1 q \Leftrightarrow (p \leq_1 q \wedge (\neg\, (p =_1 q)))
\end{array}
$$

$$
\begin{array}{l}
_ \geq_1 _ : \mathbb{P}(Q \times Q) \\
\hline
\forall\, p, q : Q \bullet \\
\quad p \geq_1 q \Leftrightarrow \neg\, (p <_1 q)
\end{array}
$$

$$
\begin{array}{l}
_ >_1 _ : \mathbb{P}(Q \times Q) \\
\hline
\forall\, p, q : Q \bullet \\
\quad p >_1 q \Leftrightarrow (p \geq_1 q \wedge (\neg\, (p =_1 q)))
\end{array}
$$

A.1.1.5 Magnitude Operators

$$
\begin{array}{l}
_ +_1 _ : (\mathit{PositiveQ} \times \mathit{PositiveQ}) \rightarrow \mathit{PositiveQ} \\
_ +_2 _ : (Q \times Q) \rightarrow Q \\
\hline
\forall\, p, q : \mathit{PositiveQ} \bullet \\
\quad \mathbf{let}\ a == \mathit{first}\ p;\ b == \mathit{second}\ p;\ c == \mathit{first}\ q;\ d == \mathit{second}\ q \bullet \\
\qquad \mathbf{let}\ \mathit{sum} == (a * d + c * b, b * d) \bullet \\
\qquad\quad \mathit{sum} \leq_1 \mathit{max_q} \wedge \mathit{sum} \geq_1 \mathit{neutral_q} \Rightarrow p +_1 q = \mathit{sum} \wedge \\
\qquad\quad \mathit{sum} <_1 \mathit{neutral_q} \Rightarrow p +_1 q = \mathit{neutral_q} \wedge \\
\qquad\quad \mathit{sum} >_1 \mathit{max_q} \Rightarrow p +_1 q = \mathit{max_q} \\
\forall\, p, q : Q \bullet \\
\quad \mathbf{let}\ a == \mathit{first}\ p;\ b == \mathit{second}\ p;\ c == \mathit{first}\ q;\ d == \mathit{second}\ q \bullet \\
\qquad \mathbf{let}\ \mathit{sum} == (a * d + c * b, b * d) \bullet \\
\qquad\quad \mathit{sum} \leq_1 \mathit{max_q} \wedge \mathit{sum} \geq_1 \mathit{min_q} \Rightarrow p +_2 q = \mathit{sum} \wedge \\
\qquad\quad \mathit{sum} <_1 \mathit{min_q} \Rightarrow p +_2 q = \mathit{min_q} \wedge \\
\qquad\quad \mathit{sum} >_1 \mathit{max_q} \Rightarrow p +_2 q = \mathit{max_q}
\end{array}
$$

$$
\begin{array}{l}
_ -_1 _ : (PositiveQ \times PositiveQ) \rightarrow PositiveQ \\
_ -_2 _ : (Q \times Q) \rightarrow Q \\
\hline
\forall\, p, q : PositiveQ \bullet \\
\quad \mathbf{let}\ a == first\ p;\ b == second\ p;\ c == first\ q;\ d == second\ q \bullet \\
\quad\quad \mathbf{let}\ sub == (a * d - c * b, b * d) \bullet \\
\quad\quad\quad sub <_1 neutral_q \Rightarrow p -_1 q = neutral_q \wedge \\
\quad\quad\quad sub \geq_1 neutral_q \wedge sub \leq_1 max_q \Rightarrow p -_1 q = sub \wedge \\
\quad\quad\quad sub >_1 max_q \Rightarrow p -_1 q = max_q \\
\forall\, p, q : Q \bullet \\
\quad \mathbf{let}\ a == first\ p;\ b == second\ p;\ c == first\ q;\ d == second\ q \bullet \\
\quad\quad \mathbf{let}\ sub == (a * d - c * b, b * d) \bullet \\
\quad\quad\quad sub <_1 min_q \Rightarrow p -_2 q = min_q \wedge \\
\quad\quad\quad sub \geq_1 min_q \wedge sub \leq_1 max_q \Rightarrow p -_2 q = sub \wedge \\
\quad\quad\quad sub >_1 max_q \Rightarrow p -_2 q = max_q
\end{array}
$$

$$
\begin{array}{l}
_ *_1 _ : (PositiveQ \times PositiveQ) \rightarrow PositiveQ \\
_ *_2 _ : (Q \times Q) \rightarrow Q \\
\hline
\forall\, p, q : PositiveQ \bullet \\
\quad \mathbf{let}\ a == first\ p;\ b == second\ p;\ c == first\ q;\ d == second\ q \bullet \\
\quad\quad \mathbf{let}\ mult == (a * c, b * d) \bullet \\
\quad\quad\quad mult <_1 neutral_q \Rightarrow p *_1 q = neutral_q \wedge \\
\quad\quad\quad mult >_1 max_q \Rightarrow p *_1 q = max_q \wedge \\
\quad\quad\quad mult \geq_1 neutral_q \wedge mult \leq_1 max_q \Rightarrow p *_1 q = mult \\
\forall\, p, q : Q \bullet \\
\quad \mathbf{let}\ a == first\ p;\ b == second\ p;\ c == first\ q;\ d == second\ q \bullet \\
\quad\quad \mathbf{let}\ mult == (a * c, b * d) \bullet \\
\quad\quad\quad mult <_1 min_q \Rightarrow p *_2 q = min_q \wedge \\
\quad\quad\quad mult >_1 max_q \Rightarrow p *_2 q = max_q \wedge \\
\quad\quad\quad mult \geq_1 min_q \wedge mult \leq_1 max_q \Rightarrow p *_2 q = mult
\end{array}
$$

$div_1 : (PositiveQ \times PositiveQ) \nrightarrow PositiveQ$

$div_2 : (Q \times Q) \nrightarrow Q$

$\forall\, p, q : PositiveQ \bullet$
$\quad \mathbf{let}\ a == first\ p;\ b == second\ p;\ c == first\ q;\ d == second\ q \bullet$
$\quad\quad c \neq 0 \wedge (\mathbf{let}\ divi == (a * d, b * c) \bullet$
$\quad\quad\quad divi <_1 neutral_q \Rightarrow div_1(p, q) = neutral_q \wedge$
$\quad\quad\quad divi >_1 max_q \Rightarrow div_1(p, q) = max_q \wedge$
$\quad\quad\quad divi \geq_1 neutral_q \wedge divi \leq_1 max_q \Rightarrow div_1(p, q) = divi)$

$\forall\, p, q : Q \bullet$
$\quad \mathbf{let}\ a == first\ p;\ b == second\ p;\ c == first\ q;\ d == second\ q \bullet$
$\quad\quad c \neq 0 \wedge (\mathbf{let}\ divi == (a * d, b * c) \bullet$
$\quad\quad\quad divi <_1 min_q \Rightarrow div_2(p, q) = min_q \wedge$
$\quad\quad\quad divi >_1 max_q \Rightarrow div_2(p, q) = max_q \wedge$
$\quad\quad\quad divi \geq_1 min_q \wedge divi \leq_1 max_q \Rightarrow div_2(p, q) = divi)$

A.1.1.6 Random Numbers

$random : Instant \rightarrow Q$

A.1.1.7 Time

$Instant == \mathbb{N}$

$Duration == \mathbb{N}$

A.1.2 Stimulation

A.1.2.1 Basic Entities

$[Stimulus]$

StimulationSubsystem

$StimulationParameters$

$StimulusImplication$

$StimulusEquivalence$

$currentStimuli : \mathbb{P}\, Stimulus$

$pastStimuli : Instant \nrightarrow \mathbb{P}\, Stimulus$

$stimulus_status : Stimulus \rightarrow StimulusStatus$

$stimulusBeginning : Stimulus \nrightarrow Instant$

$StimulationParameters$

$StimulationHints$

$Conditioning_Ref1_Parameters$

$stimuli : \mathbb{P}\, Stimulus$

$primaryStimuli : \mathbb{P}\, Stimulus$

$primary_utility : Stimulus \nrightarrow Utility$

$max_delay : Duration$

$\mathrm{dom}\, primary_utility = primaryStimuli$

$StimulusStatus ::= Beginning \mid Ending \mid Stable \mid Absent$

$StimulationHints$

$pleasureHints : \mathbb{P}\, Stimulus$

$painHints : \mathbb{P}\, Stimulus$

$pleasureHints \cap painHints = \emptyset$

$Stimulation$

$stimulus : Stimulus$

$intensity : Intensity$

$status : StimulusStatus$

$status = Beginning \lor status = Ending$

A.1.2.2 Relations Among Stimuli

$StimulusImplication$

$sCauses : \mathbb{P}(Stimulus \times Stimulus)$

$sCorrelation : Stimulus \times Stimulus \nrightarrow Correlation$

$\forall\, s_1, s_2, s_3 : Stimulus \bullet$

$\quad (s_1\ \underline{sCauses}\ s_1) \land$

$\quad (((s_1\ \underline{sCauses}\ s_2) \land (s_2\ \underline{sCauses}\ s_3)) \Rightarrow (s_1\ \underline{sCauses}\ s_3))$

$\forall\, s_1, s_2 : Stimulus \mid s_1\ \underline{sCauses}\ s_2 \bullet$

$\quad \exists\, c : Correlation \bullet ((s_1, s_2) \mapsto c) \in sCorrelation$

StimulusEquivalence

$StimulusImplication$

$equals : \mathbb{P}(Stimulus \times Stimulus)$

$\forall\, s_1, s_2 : Stimulus \bullet$

$\quad (s_1 \ \underline{equals}\ s_2) \Leftrightarrow (s_1 \ \underline{sCauses}\ s_2) \wedge (s_2 \ \underline{sCauses}\ s_1)$

$\forall\, s_1, s_2 : Stimulus \mid s_1 \ \underline{equals}\ s_2 \bullet$

$\quad sCorrelation(s_1, s_2) = sCorrelation(s_2, s_1)$

A.1.2.3 Stimulus Utility

$Utility : \mathbb{P}\, Q$

$max_utility : Utility$

$neutral_utility : Utility$

$min_utility : Utility$

$max_utility = max_q$

$neutral_utility = neutral_q$

$min_utility = min_q$

$\forall\, u : Utility \bullet (u \geq_1 min_utility) \wedge (u \leq_1 max_utility)$

$(neutral_utility \geq_1 min_utility) \wedge (neutral_utility \leq_1 max_utility)$

StimulusUtility

$StimulationSubsystem$

$EmotionSubsystem$

$DriveSubsystem$

$sUtility : Stimulus \rightarrow Utility$

StimulusUtility_Ref1

$StimulusUtilityBase$

$StimulusEmotionalRegulator$

$StimulusDriveRegulator$

$\forall\, s : Stimulus \bullet$

$\quad sUtility(s) = driveRegulator(s, emotionalRegulator(s, base(s)))$

$StimulusUtilityBase$

$StimulusUtility$

$StimulusImplication$

$base : Stimulus \rightarrow Utility$

$\forall\, s : Stimulus \bullet$
$\quad (\exists\, p : primaryStimuli \bullet$
$\quad\quad base(s) = primary_utility(p) \wedge$
$\quad\quad (\forall\, q : primaryStimuli \mid s\ \underline{sCauses}\ q \bullet$
$\quad\quad\quad primary_utility(p) \geq_1 primary_utility(q) \wedge$
$\quad\quad (s\ \underline{sCauses}\ p))) \vee$
$\quad (\forall\, p : primaryStimuli \bullet$
$\quad\quad \neg\, (s\ \underline{sCauses}\ p) \wedge$
$\quad\quad sUtility(s) = neutral_utility)$

A.1.2.4 Stimulus Conditioning

$ConditioningOp_1$

$\Delta StimulusImplication$

$StimulationParameters$

$s_1? : Stimulus$

$s_2? : Stimulus$

$delay? : Duration$

$delay? \leq max_delay$

$sCauses' = sCauses \oplus \{s_1? \mapsto s_2?\}$

$(s_1?\ \underline{sCauses}\ s_2?) \wedge sCorrelation(s_1?, s_2?) <_1 max_correlation \Rightarrow$
$\quad sCorrelation'(s_1?, s_2?) >_1 sCorrelation(s_1?, s_2?)$

$(s_1?\ \underline{sCauses}\ s_2?) \wedge sCorrelation(s_1?, s_2?) = max_correlation \Rightarrow$
$\quad sCorrelation'(s_1?, s_2?) = sCorrelation(s_1?, s_2?)$

$\neg\, (s_1?\ \underline{sCauses}\ s_2?) \Rightarrow$
$\quad sCorrelation'(s_1?, s_2?) \geq_1 min_correlation$

ConditioningOp_2

$\Xi StimulusImplication$

$StimulationParameters$

$delay? : Duration$

$delay? > max_delay$

$T_ConditioningOp \mathrel{\widehat{=}} ConditioningOp_1 \lor ConditioningOp_2$

UnconditioningOp_1

$\Delta StimulusImplication$

$StimulationSubsystem$

$s_1? : Stimulus$

$s_2? : Stimulus$

$currentInstant? : Instant$

$s_1? \in pastStimuli(currentInstant? - max_delay) \land$
$\quad \neg (\exists\, t : Instant \mid currentInstant? - max_delay < t \leq currentInstant? \bullet$
$\quad\quad s_2? \in pastStimuli(t))$

$s_1?\ \underline{sCauses}\ s_2?$

$sCorrelation'(s_1?, s_2?) \leq_1 sCorrelation(s_1?, s_2?)$

$sCorrelation'(s_1?, s_2?) \leq_1 min_correlation \Rightarrow$
$\quad sCauses' = sCauses \setminus \{s_1? \mapsto s_2?\}$

UnconditioningOp_2

$\Xi StimulusImplication$

$StimulationSubsystem$

$s_1? : Stimulus$

$s_2? : Stimulus$

$currentInstant? : Instant$

$\neg (s_1? \in pastStimuli(currentInstant? - max_delay) \land$
$\quad \neg (\exists\, t : Instant \mid currentInstant? - max_delay < t \leq currentInstant? \bullet$
$\quad\quad s_2? \in pastStimuli(t))) \lor$
$\neg (s_1?\ \underline{sCauses}\ s_2?)$

$$T_UnconditioningOp \mathrel{\widehat{=}} UnconditioningOp_1 \lor UnconditioningOp_2$$

$Conditioning_Ref1_Parameters$

$c : Correlation$

$increment : Correlation$

$delay? : Duration$

$\mathbf{let}\ q == (delay?, 1) \bullet increment = div_1(c, q)$

$ConditioningOp_Ref1$

$\Delta StimulusImplication$

$StimulationParameters$

$s_1? : Stimulus$

$s_2? : Stimulus$

$delay? : Duration$

$Conditioning_Ref1_Parameters$

$delay? \leq max_delay$

$sCauses' = sCauses \oplus \{s_1? \mapsto s_2?\}$

$sCorrelation'(s_1?, s_2?) = sCorrelation(s_1?, s_2?) +_1 increment$

Proposition A.1. *ConditioningOp is refined by ConditioningOp_Ref*1.

Proof. From Definition D.1, there are two predicates that must be satisfied to prove the operation's refinement.

First, we must show that

$$\forall\, StimulusImplication;\ s_1?, s_2? : Stimulus;\ delay? : Duration \bullet \\ \mathrm{pre}(ConditioningOp) \Rightarrow \mathrm{pre}(ConditioningOp_Ref1)$$

This follows immediately from the fact that both have same pre-condition, namely, $delay? \leq max_delay$.

Second, we must show that

$$\forall\, StimulusImplication;\ StimulusImplication';\ s_1?, s_2? : Stimulus; \\ delay? : Duration \bullet \\ \mathrm{pre}(ConditioningOp) \land ConditioningOp_Ref1 \Rightarrow ConditioningOp$$

Intuitively, this holds because the result of the $+_1$ operator is bounded by *max_correlation*, so incrementing the correlation using it actually results in what is defined by *ConditioningOp* (e.g., the correlation's value may remain the same). Mathematically, we have the following proof.[1]

1	$\mathrm{pre}(ConditioningOp) \land$ $ConditioningOp_Ref1$	[Assumption]
1.1	$(s_1? \ \underline{sCauses} \ s_2?) \land$ $sCorrelation(s_1?, s_2?) <_1 max_correlation$	[Assumption]
1.1.1	$sCorrelation(s_1?, s_2?) <_1 max_correlation$	[$\land$ elimination using step 1.1]
1.1.2	$sCorrelation(s_1?, s_2?) <_1 max_q$	[Definition of *max_correlation*]
1.1.3	$sum =$ $(first\ sCorrelation(s_1?, s_2?) * second\ increment +$ $first\ increment * second\ sCorrelation(s_1?, s_2?),$ $second\ sCorrelation(s_1?, s_2?) * second\ increment)$	[Definition of $+_1$]
1.1.4	$sum \geq_1 sCorrelation(s_1?, s_2?)$	[*increment* is positive]
1.1.5	$sum \geq_1 neutral_q$	[Definition of *Correlation*]
1.1.6	$(sum \leq_1 max_q) \lor (sum >_1 max_q)$	[Excluded middle]
1.1.7	$sum \leq_1 max_q$	[Assumption]
1.1.7.1	$sum \leq_1 max_q \land sum \geq_1 neutral_q$	[$\land$ introduction using steps 1.1.7 and 1.1.5]
1.1.7.2	$sCorrelation'(s_1?, s_2?) = sum$	[Definition of $+_1$ and *ConditioningOp_Ref*1, and step 1.1.3]
1.1.7.3	$sCorrelation'(s_1?, s_2?) >_1 sCorrelation(s_1?, s_2?)$	[Steps 1.1.7.2 and 1.1.4]
1.1.8	$sum >_1 max_q$	[Assumption]
1.1.8.1	$sCorrelation'(s_1?, s_2?) = max_q$	[Definition of $+_1$ and *ConditioningOp_Ref*1]
1.1.8.2	$sCorrelation'(s_1?, s_2?) >_1 sCorrelation(s_1?, s_2?)$	[Steps 1.1.8.1 and 1.1.2]
1.1.9	$sCorrelation'(s_1?, s_2?) >_1 sCorrelation(s_1?, s_2?)$	[$\lor$ elimination using steps 1.1.6, 1.1.7.3 and 1.1.8.2]
1.2	$(s_1? \ \underline{sCauses} \ s_2?) \land$ $sCorrelation(s_1?, s_2?) <_1 max_correlation \Rightarrow$ $sCorrelation'(s_1?, s_2?) >_1 sCorrelation(s_1?, s_2?)$	[$\Rightarrow$ introduction using steps 1.1 and 1.1.9]
1.3	$(s_1? \ \underline{sCauses} \ s_2?) \land$ $sCorrelation(s_1?, s_2?) = max_correlation$	[Assumption]
1.3.1	$sCorrelation(s_1?, s_2?) = max_correlation$	[$\land$ elimination]
1.3.2	$sCorrelation(s_1?, s_2?) = max_q$	[Definition of

[1]For greater readability, this proof is presented in a manner similar to Fitch-style proofs: each line is a numbered step, followed by a proposition and a justification for that proposition; indentation and nested numbering are used to show that lines are under an assumption.

		$max_correlation$]
1.3.3	$sum =$ $(first\ sCorrelation(s_1?, s_2?) * second\ increment +$ $first\ increment * second\ sCorrelation(s_1?, s_2?),$ $second\ sCorrelation(s_1?, s_2?) * second\ increment)$	[Definition of $+_1$]
1.3.4	$sum \geq_1 sCorrelation(s_1?, s_2?)$	[$increment$ is positive]
1.3.5	$sum \geq_1 max_q$	[Steps 1.3.2 and 1.3.4]
1.3.6	$(sum \leq_1 max_q) \vee (sum >_1 max_q)$	[Excluded middle]
1.3.7	$sum \leq_1 max_q$	[Assumption]
1.3.7.1	$sum \leq_1 max_q \wedge sum \geq_1 neutral_q$	[$\wedge$ introduction using steps 1.3.7 and 1.3.5]
1.3.7.2	$sCorrelation'(s_1?, s_2?) = sum$	[Definition of $+_1$ and $ConditioningOp_Ref1$, and step 1.3.3]
1.3.7.3	$sum \leq_1 max_q$	[$\wedge$ elimination in step 1.3.7.1]
1.3.7.4	$sum \geq_1 max_q \wedge sum \leq_1 max_q$	[Steps 1.3.7 and 1.3.7.3]
1.3.7.5	$sum = max_q$	[Step 1.3.7.4]
1.3.7.6	$sCorrelation'(s_1?, s_2?) = sCorrelation(s_1?, s_2?)$	[Steps 1.3.7.2 and 1.3.4]
1.3.8	$sum >_1 max_q$	[Assumption]
1.3.8.1	$sCorrelation'(s_1?, s_2?) = max_q$	[Definition of $+_1$ and $ConditioningOp_Ref1$]
1.3.8.2	$sCorrelation'(s_1?, s_2?) = sCorrelation(s_1?, s_2?)$	[Steps 1.3.8.1 and 1.3.2]
1.3.9	$sCorrelation'(s_1?, s_2?) = sCorrelation(s_1?, s_2?)$	[$\vee$ elimination using steps 1.3.6, 1.3.7.3 and 1.3.8.2]
1.4	$(s_1?\ \underline{sCauses}\ s_2?) \wedge$ $sCorrelation(s_1?, s_2?) = max_correlation \Rightarrow$ $sCorrelation'(s_1?, s_2?) = sCorrelation(s_1?, s_2?)$	[$\Rightarrow$ introduction using steps 1.3 and 1.3.9]
1.5	$\neg\ (s_1?\ \underline{sCauses}\ s_2?)$	[Assumption]
1.5.1	$sCorrelation(s_1?, s_2?) \geq_1 min_correlation$	[Definition of $Correlation$]
1.5.2	$sum =$ $(first\ sCorrelation(s_1?, s_2?) * second\ increment +$ $first\ increment * second\ sCorrelation(s_1?, s_2?),$ $second\ sCorrelation(s_1?, s_2?) * second\ increment)$	[Definition of $+_1$]
1.5.3	$sum \geq_1 sCorrelation(s_1?, s_2?)$	[$increment$ is positive]
1.5.4	$sum \geq_1 min_q$	[Definition of $Correlation$]
1.5.5	$sCorrelation'(s_1?, s_2?) = sum \vee$ $sCorrelation'(s_1?, s_2?) = netural_q \vee$ $sCorrelation'(s_1?, s_2?) = max_q$	[Definition of $+_1$]
1.5.6	$sCorrelation'(s_1?, s_2?) \geq_1 min_q \vee$ $sCorrelation'(s_1?, s_2?) = netural_q \vee$ $sCorrelation'(s_1?, s_2?) = max_q$	[Steps 1.5.4

		and 1.5.5]
1.5.7	$sCorrelation'(s_1?, s_2?) \geq_1 min_q$	[Definitions of min_q, $neutral_q$ and max_q and step 1.5.6]
1.5.8	$sCorrelation'(s_1?, s_2?) \geq_1 min_correlation$	[Definition of $min_correlation$]
1.6	$\neg\ (s_1?\ \underline{sCauses}\ s_2?) \Rightarrow$ $sCorrelation'(s_1?, s_2?) \geq_1 min_correlation$	[$\Rightarrow$ introduction using steps 1.5 and 1.5.8]
1.7	$((s_1?\ \underline{sCauses}\ s_2?) \wedge$ $sCorrelation(s_1?, s_2?) <_1 max_correlation \Rightarrow$ $sCorrelation'(s_1?, s_2?) >_1 sCorrelation(s_1?, s_2?)) \wedge$ $((s_1?\ \underline{sCauses}\ s_2?) \wedge$ $sCorrelation(s_1?, s_2?) = max_correlation \Rightarrow$ $sCorrelation'(s_1?, s_2?) = sCorrelation(s_1?, s_2?)) \wedge$ $(\neg\ (s_1?\ \underline{sCauses}\ s_2?) \Rightarrow$ $sCorrelation'(s_1?, s_2?) \geq_1 min_correlation)$	[$\wedge$ introduction using steps 1.2, 1.4 and 1.6]
1.8	$delay? \leq max_delay \wedge$ $sCauses' = sCauses \oplus \{s_1? \mapsto s_2?\}$	[Definition of $ConditioningOp_Ref1$ assumed in step 1]
1.9	$ConditioningOp$	[Steps 1.7 and 1.8]
2	$\mathrm{pre}(ConditioningOp) \wedge ConditioningOp_Ref1 \Rightarrow$ $ConditioningOp$	[$\Rightarrow$ introduction using steps 1 and 1.9]

□

$$T_ConditioningOp_Ref1 \mathrel{\widehat{=}} ConditioningOp_Ref1 \vee ConditioningOp_2$$

$UnconditioningOp_Ref1$

$UnconditioningOp_1$

$Conditioning_Ref1_Parameters$

$sCorrelation'(s_1?, s_2?) = sCorrelation(s_1?, s_2?) -_1 increment$

$$T_UnconditioningOp_Ref1 \mathrel{\widehat{=}} UnconditioningOp_Ref1 \vee UnconditioningOp_2$$

A.1.2.5 Stimulation

$StimulationUpdateOp_1$

$\Delta StimulationSubsystem$

$currentInstant? : Instant$

$stimulation? : Stimulation$

$stimulation?.status = Beginning$

$currentStimuli' = currentStimuli \cup \{stimulation?.stimulus\}$

$stimulus_status' = stimulus_status \oplus$
$\quad \{stimulation?.stimulus \mapsto Beginning\}$

$stimulusBeginning' = stimulusBeginning \oplus$
$\quad \{stimulation?.stimulus \mapsto currentInstant?\}$

$StimulationUpdateOp_2$

$\Delta StimulationSubsystem$

$stimulation? : Stimulation$

$stimulation?.status = Ending$

$currentStimuli' = currentStimuli$

$stimulus_status' = stimulus_status \oplus$
$\quad \{stimulation?.stimulus \mapsto Ending\}$

$stimulusBeginning' = stimulusBeginning$

$StimulationUpdateOp_3$

$\Delta StimulationSubsystem$

$stimulation? : Stimulation$

$stimulation?.status = Stable \lor stimulation?.status = Absent$

$currentStimuli' = currentStimuli$

$stimulus_status' = stimulus_status \oplus$
$\quad \{stimulation?.stimulus \mapsto stimulation?.status\}$

$stimulusBeginning' = stimulusBeginning$

$$T_StimulationUpdateOp \mathrel{\widehat{=}} StimulationUpdateOp_1 \lor StimulationUpdateOp_2 \lor StimulationUpdateOp_3$$

$CurrentStimuliUpdateOp_1$

$\Delta StimulationSubsystem$

$s? : Stimulus$

$stimulus_status(s?) = Beginning$

$stimulus_status'(s?) = Stable$

$currentStimuli' = currentStimuli$

$CurrentStimuliUpdateOp_2$

$\Delta StimulationSubsystem$

$s? : Stimulus$

$stimulus_status(s?) = Stable$

$stimulus_status'(s?) = Stable$

$currentStimuli' = currentStimuli$

$CurrentStimuliUpdateOp_3$

$\Delta StimulationSubsystem$

$s? : Stimulus$

$stimulus_status(s?) = Ending$

$stimulus_status'(s?) = Absent$

$currentStimuli' = currentStimuli \setminus \{s?\}$

$CurrentStimuliUpdateOp_4$

$\Delta StimulationSubsystem$

$s? : Stimulus$

$stimulus_status(s?) = Absent$

$stimulus_status'(s?) = Absent$

$currentStimuli' = currentStimuli$

$$
\begin{aligned}
T_CurrentStimuliUpdateOp \mathrel{\widehat{=}} \; & CurrentStimuliUpdateOp_1 \lor \\
& CurrentStimuliUpdateOp_2 \lor \\
& CurrentStimuliUpdateOp_3 \lor \\
& CurrentStimuliUpdateOp_4
\end{aligned}
$$

$T_PastStimuliUpdateOp$

$\Delta StimulationSubsystem$

$currentInstant? : Instant$

$pastStimuli' = pastStimuli \oplus \{currentInstant? \mapsto currentStimuli\}$

A.1.2.6 Integration

$Organism_StimulusProcessing$

$\Delta Organism$

$stimulations? : \mathbb{P}\, Stimulation$

$currentInstant? : Instant$

$\forall\, st : stimulations? \mid st.status = Beginning \bullet$
$\quad \forall\, s : Stimulus \mid s \in \operatorname{dom} stimulusBeginning \wedge$
$\quad\quad stimulusBeginning(s) \geq currentInstant? - max_delay \bullet$
$\quad\quad \exists\, T_ConditioningOp_Ref1 \bullet$
$\quad\quad\quad s_1? = s \wedge$
$\quad\quad\quad s_2? = st.stimulus \wedge$
$\quad\quad\quad delay? = currentInstant? - stimulusBeginning(s)$

$\forall\, cause : sCauses \bullet$
$\quad \mathbf{let}\ s_1 == first\ cause;\ s_2 == second\ cause \bullet$
$\quad\quad \exists\, T_UnconditioningOp_Ref1 \bullet$
$\quad\quad\quad s_1? = s_1 \wedge$
$\quad\quad\quad s_2? = s_2$

$\forall\, st : stimulations? \bullet$
$\quad \exists\, T_StimulationUpdateOp \bullet$
$\quad\quad stimulation? = st$

$\forall\, s : currentStimuli \bullet$
$\quad \exists\, T_CurrentStimuliUpdateOp \bullet$
$\quad\quad s? = s$

$T_PastStimuliUpdateOp$

A.1.3 General Responding

A.1.3.1 Basic Entities

RespondingSubsystem

$CurrentBehaviors$

$CurrentResponses$

$Actions$

$ActionHistory$

$ActionConflict$

$ActionBaselevel$

$operants : \mathbb{P}\ Operant$

$reflexes : \mathbb{P}\ Reflex$

CurrentBehaviors

$elicited : \mathbb{P}\ Reflex$

$emitted : \mathbb{P}\ Operant$

$spontaneous : \mathbb{P}\ Action$

CurrentResponses

$responses : \mathbb{P}\ Response$

$activeResponses : \mathbb{P}\ Response$

$inactiveResponses : \mathbb{P}\ Response$

$reflexResponse : Reflex \nrightarrow Response$

$operantResponse : Operant \nrightarrow Response$

$spontaneousResponse : Action \nrightarrow Response$

$reflexElicitationTime : Reflex \nrightarrow Instant$

$responses = activeResponses \cup inactiveResponses$

$activeResponses \cap inactiveResponses = \emptyset$

$\operatorname{ran} reflexResponse = responses$

$\operatorname{ran} operantResponse = responses$

$\operatorname{ran} spontaneousResponse = responses$

A.1.3.2 Actions

$[Action]$

$Conflict ::= conflicting \mid nonconflicting$

ActionConflict

$conflict : Action \times Action \rightarrow Conflict$

Actions

$operantActions : \mathbb{P}\, Action$

$reflexActions : \mathbb{P}\, Action$

ActionBaselevel

$Actions$

$baseLevel : Action \rightarrow Probability$

$\forall\, a : Action \mid a \in operantActions \bullet baseLevel(a) >_1 min_probability$

$\forall\, a : Action \mid a \in reflexActions \wedge a \notin operantActions \bullet$
$\quad baseLevel(a) =_1 min_probability$

ActionHistory

$actionsHistory : Instant \nrightarrow \mathbb{P}\, Action$

A.1.3.3 Behavioural Responses

Response

$action : Action$

$latency : Duration$

$duration : Duration$

$magnitude : Intensity$

A.1.3.4 Response Scheduling Operations

$$
\begin{array}{l}
\textit{ReflexSchedulingOp} \\
\hline
\Delta \mathit{CurrentBehaviors} \\
\mathit{RespondingSubsystem} \\
\mathit{StimulationSubsystem} \\
s? : \mathit{Stimulus} \\
i? : \mathit{Intensity} \\
\hline
\forall\, r : \mathit{reflexes} \bullet \\
\quad (\exists\, \mathit{ReflexElicitationCond} \bullet s = s? \wedge i = i?) \Rightarrow r \in \mathit{elicited}' \\
\mathit{elicited} \subseteq \mathit{elicited}' \\
\hline
\end{array}
$$

$$
\begin{array}{l}
\textit{OperantSchedulingOp} \\
\hline
\Delta \mathit{CurrentBehaviors} \\
\mathit{RespondingSubsystem} \\
\mathit{StimulationSubsystem} \\
\mathit{OperantUtility} \\
\hline
\forall\, o : \mathit{operants} \bullet \\
\quad \mathit{OperantEmissionCond} \Rightarrow o \in \mathit{emitted}' \\
\mathit{emitted} \subseteq \mathit{emitted}' \\
\hline
\end{array}
$$

$$
\begin{array}{l}
\textit{BaseLevelSchedulingOp} \\
\hline
\Delta \mathit{CurrentBehaviors} \\
\mathit{RespondingSubsystem} \\
\mathit{ResponseEmotionalRegulator} \\
t? : \mathit{Instant} \\
\hline
\forall\, a : \mathit{operantActions} \bullet \\
\quad \mathit{responseRegulator}(a, \mathit{baseLevel}(a)) \geq_1 \mathit{random}(t?) \\
\qquad \Rightarrow a \in \mathit{spontaneous}' \\
\mathit{spontaneous} \subseteq \mathit{spontaneous}' \\
\hline
\end{array}
$$

A.1.3.5 Conflict Resolution Operations

$OperantConflictCond$

$ActionConflict$

$o_1, o_2 : Operant$

$conflict(o_1.action, o_2.action) = conflicting$

$OperantConflictResolutionOp$

$\Delta CurrentBehaviors$

$StimulationSubsystem$

$OperantUtility$

$ActionConflict$

$removeO : \mathbb{P}\, Operant$

$\forall\, o_1, o_2 : emitted \mid OperantConflictCond \bullet$
$\quad (oUtility(o_1, currentStimuli) >_1 oUtility(o_2, currentStimuli) \Rightarrow$
$\quad\quad o_2 \in removeO) \wedge$
$\quad (oUtility(o_1, currentStimuli) =_1 oUtility(o_2, currentStimuli) \Rightarrow$
$\quad\quad (o_1 \in removeO) \vee (o_2 \in removeO))$

$ReflexConflictCond$

$ActionConflict$

$r_1, r_2 : Reflex$

$conflict(r_1.action, r_2.action) = conflicting$

$ReflexConflictResolutionOp$

$\Delta CurrentBehaviors$

$removeR : \mathbb{P}\, Reflex$

$\forall\, r_1, r_2 : elicited \mid ReflexConflictCond \bullet$
$\quad (r_1 \in removeR) \vee (r_2 \in removeR)$

$ReflexConflictResolutionOp_Ref1$

$ReflexConflictResolutionOp$

$\forall\, r_1, r_2 : elicited \mid ReflexConflictCond \bullet$
$\quad (r_1.magnitude \geq_1 r_2.magnitude \Rightarrow r_2 \in removeR) \wedge$
$\quad (r_1.magnitude <_1 r_2.magnitude \Rightarrow r_1 \in removeR)$

$ReflexConflictResolutionOp_Ref2$

$ReflexConflictResolutionOp$

$$\forall\, r_1, r_2 : elicited \mid ReflexConflictCond \bullet$$
$$(r_1.latency \geq r_2.latency \Rightarrow r_1 \in removeR) \wedge$$
$$(r_1.latency < r_2.latency \Rightarrow r_2 \in removeR)$$

$OperantReflexConflicCond$

$ActionConflict$

$o : Operant$

$r : Reflex$

$$conflict(o.action, r.action) = conflicting$$

$OperantReflexConflictResolutionOp$

$\Delta\, CurrentBehaviors$

$ActionConflict$

$removeO : \mathbb{P}\, Operant$

$removeR : \mathbb{P}\, Reflex$

$$\forall\, o : emitted;\ r : elicited \mid OperantReflexConflicCond \bullet$$
$$(r \in removeR) \vee (o \in removeO)$$

$BaseLevelConflictResolutionOp$

$\Delta CurrentBehaviors$

$ActionConflict$

$removeO : \mathbb{P}\, Operant$

$removeR : \mathbb{P}\, Reflex$

$removeA : \mathbb{P}\, Action$

$\forall\, o : emitted;\ a : spontaneous \mid o.action = a \bullet$
$\quad a \in removeA$

$\forall\, o : emitted;\ a : spontaneous \mid conflict(o.action, a) = conflicting \bullet$
$\quad (a \in removeA) \vee (o \in removeO)$

$\forall\, r : elicited;\ a : spontaneous \mid$
$\quad r.action = a \vee conflict(r.action, a) = conflicting \bullet$
$\qquad a \in removeA$

$\forall\, a_1, a_2 : spontaneous \mid conflict(a_1, a_2) = conflicting \bullet$
$\quad (a_1 \in removeA) \vee (a_2 \in removeA)$

$AuxConflictResolutionOp$

$\Delta CurrentBehaviors$

$ActionConflict$

$removeO : \mathbb{P}\, Operant$

$removeR : \mathbb{P}\, Reflex$

$removeA : \mathbb{P}\, Action$

$removeO \subseteq emitted$

$removeR \subseteq elicited$

$removeA \subseteq spontaneous$

$\forall\, o_1 : emitted \bullet$
$\quad ((\neg\, (\exists\, o_2 : emitted \bullet OperantConflictCond)) \wedge$
$\quad (\neg\, (\exists\, r : elicited \bullet (\mathbf{let}\ o == o_1 \bullet OperantReflexConflictCond))) \wedge$
$\quad (\neg\, (\exists\, a : spontaneous \bullet conflict(o_1.action, a) = conflicting))) \Rightarrow$
$\quad (o_1 \notin removeO)$

$\forall\, r_1 : elicited \bullet$
$\quad ((\neg\, (\exists\, r_2 : elicited \bullet ReflexConflictCond)) \wedge$
$\quad (\neg\, (\exists\, o : emitted \bullet (\mathbf{let}\ r = r_1 \bullet OperantReflexConflictCond))) \wedge$
$\quad (\neg\, (\exists\, a : spontaneous \bullet r_1.action = a \vee conflict(r_1.action, a) = conflicting))) \Rightarrow$
$\quad (r_1 \notin removeR)$

$\forall\, a_1 : spontaneous \bullet$
$\quad ((\neg\, (\exists\, a_2 : spontaneous \bullet conflict(a_1, a_2) = conflicting)) \wedge$
$\quad (\neg\, (\exists\, o : emitted \bullet o.action = a_1 \vee conflict(o.action, a_1) = conflicting)) \wedge$
$\quad (\neg\, (\exists\, r : elicited \bullet r.action = a \vee conflict(r.action, a) = conflicting))) \Rightarrow$
$\quad (a_1 \notin removeA)$

$emitted' = emitted \setminus removeO$

$elicited' = elicited \setminus removeR$

$spontaneous' = spontaneous \setminus removeA$

$ConflictResolutionOp \mathrel{\widehat{=}}$
$\quad OperantConflictResolutionOp \wedge$
$\quad ReflexConflictResolutionOp \wedge$
$\quad OperantReflexConflictResolutionOp \wedge$
$\quad BaseLevelConflictResolutionOp \wedge$
$\quad AuxConflictResolutionOp$

$$
\begin{aligned}
&ConflictResolutionOp_Ref1 \mathrel{\widehat{=}} \\
&\qquad OperantConflictResolutionOp \land \\
&\qquad ReflexConflictResolutionOp_Ref1 \land \\
&\qquad OperantReflexConflictResolutionOp \land \\
&\qquad BaseLevelConflictResolutionOp \land \\
&\qquad AuxConflictResolutionOp
\end{aligned}
$$

$$
\begin{aligned}
&ConflictResolutionOp_Ref2 \mathrel{\widehat{=}} \\
&\qquad OperantConflictResolutionOp \land \\
&\qquad ReflexConflictResolutionOp_Ref2 \land \\
&\qquad OperantReflexConflictResolutionOp \land \\
&\qquad BaseLevelConflictResolutionOp \land \\
&\qquad AuxConflictResolutionOp
\end{aligned}
$$

A.1.3.6 Response Emission, Update and Termination Operations

$$
\begin{array}{|l}
\hline
OperantEmissionOp \\
\Delta ActionHistory \\
\Delta CurrentResponses \\
currentInstant? : Instant \\
o? : Operant \\
\hline
\neg (\exists\, rp : Response \bullet operantResponse(o?) = rp) \\
\exists\, rp : Response \bullet \\
\qquad rp.action = o?.action \land \\
\qquad inactiveResponses' = inactiveResponses \cup \{rp\} \land \\
\qquad operantResponse' = operantResponse \oplus \{o? \mapsto rp\} \\
actionsHistory'(currentInstant?) = \\
\qquad actionsHistory(currentInstant?) \cup \{o?.action\} \\
\hline
\end{array}
$$

ReflexElicitationOp

$\Delta ActionHistory$

$\Delta CurrentResponses$

$currentInstant? : Instant$

$r? : Reflex$

$\neg (\exists\, rp : Response \bullet reflexResponse(r?) = rp)$

$\exists\, rp : Response \bullet$
$\quad rp.action = r?.action \wedge$
$\quad rp.latency = r?.latency \wedge$
$\quad rp.duration = r?.duration \wedge$
$\quad rp.magnitude = r?.magnitude \wedge$
$\quad inactiveResponses' = inactiveResponses \cup \{rp\} \wedge$
$\quad reflexResponse' = reflexResponse \oplus \{r? \mapsto rp\}$

$\exists\, ReflexAdjustmentOp \bullet$
$\quad \theta Reflex = r? \wedge$
$\quad t_1? = reflexElicitationTime(r?) \wedge t_2? = currentInstant?$

$reflexElicitationTime' = reflexElicitationTime \oplus \{r? \mapsto currentInstant?\}$

$actionsHistory'(currentInstant?) =$
$\quad actionsHistory(currentInstant?) \cup \{r?.action\}$

BaseLevelEmissionOp

$\Delta ActionHistory$

$\Delta CurrentResponses$

$currentInstant? : Instant$

$a? : Action$

$\neg (\exists\, rp : Response \bullet spontaneousResponse(a?) = rp)$

$\exists\, rp : Response \bullet$
$\quad inactiveResponses' = inactiveResponses \cup \{rp\} \wedge$
$\quad rp.action = a?$

$actionsHistory'(currentInstant?) =$
$\quad actionsHistory(currentInstant?) \cup \{a?\}$

$InactiveResponseUpdateOp_1$

$\Delta CurrentResponses$

$\Delta Response$

$\theta Response \in inactiveResponses$

$\theta Response \notin activeResponses$

$latency > 0$

$latency' = latency - 1$

$activeResponses' = activeResponses$

$inactiveResponses' = (inactiveResponses \setminus \{\theta Response\}) \cup \{\theta Response'\}$

$InactiveResponseUpdateOp_2$

$\Delta CurrentResponses$

$\Delta Response$

$\theta Response \in inactiveResponses$

$\theta Response \notin activeResponses$

$latency \leq 0$

$activeResponses' = activeResponses \cup \{\theta Response'\}$

$inactiveResponses' = inactiveResponses \setminus \{\theta Response\}$

$ActiveResponseUpdateOp$

$\Delta CurrentResponses$

$\Delta Response$

$\theta Response \notin inactiveResponses$

$\theta Response \in activeResponses$

$duration' = duration - 1$

$activeResponses' = (activeResponses \setminus \{\theta Response\}) \cup \{\theta Response'\}$

$inactiveResponses' = inactiveResponses$

$NeutralResponseUpdateOp$

$\Xi CurrentResponses$

$\Xi Response$

$(\theta Response \notin inactiveResponses \wedge \theta Response \notin activeResponses) \vee$
$(\theta Response \in inactiveResponses \wedge \theta Response \in activeResponses)$

$T_ResponseUpdateOp \mathrel{\widehat{=}}$
$\quad InactiveResponseUpdateOp_1 \vee$
$\quad InactiveResponseUpdateOp_2 \vee$
$\quad ActiveResponseUpdateOp \vee$
$\quad NeutralResponseUpdateOp$

$ResponseTerminationOp_1$

$\Delta CurrentResponses$

$\Delta CurrentBehaviors$

$rp? : Response$

$rp? \in activeResponses$

$rp?.duration \leq 0$

$activeResponses' = activeResponses \setminus \{rp?\}$

$inactiveResponses' = inactiveResponses \setminus \{rp?\}$

$\forall\, o : emitted \bullet operantResponse(o) = rp? \Rightarrow o \notin emitted'$

$\forall\, r : elicited \bullet reflexResponse(r) = rp? \Rightarrow r \notin elicited'$

$\forall\, a : spontaneous \bullet spontaneousResponse(a) = rp? \Rightarrow a \notin spontaneous'$

$$
\begin{array}{|l}
\hline
\textit{ResponseTerminationOp_2} \\
\Delta \mathit{CurrentResponses} \\
\Xi \mathit{CurrentBehaviors} \\
rp? : \mathit{Response} \\
\hline
rp? \in \mathit{activeResponses} \\
\neg\, (\exists\, r : \mathit{elicited} \bullet \mathit{reflexResponse}(r) = rp?) \\
\neg\, (\exists\, o : \mathit{emitted} \bullet \mathit{operantResponse}(o) = rp?) \\
\neg\, (\exists\, a : \mathit{spontaneous} \bullet \mathit{spontaneousResponse}(a) = rp?) \\
\mathit{activeResponses}' = \mathit{activeResponses} \setminus \{rp?\} \\
\mathit{inactiveResponses}' = \mathit{inactiveResponses} \setminus \{rp?\} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{ResponseTerminationOp_3} \\
\Xi \mathit{CurrentResponses} \\
\Xi \mathit{CurrentBehaviors} \\
rp? : \mathit{Response} \\
\hline
rp?.\mathit{duration} > 0 \\
(\exists\, r : \mathit{elicited} \bullet \mathit{reflexResponse}(r) = rp?) \lor \\
\qquad (\exists\, o : \mathit{emitted} \bullet \mathit{operantResponse}(o) = rp?) \lor \\
\qquad (\exists\, a : \mathit{spontaneous} \bullet \mathit{spontaneousResponse}(a) = rp?) \\
\hline
\end{array}
$$

$$
\begin{array}{l}
T_\mathit{ResponseTerminationOp} \mathrel{\widehat{=}} \\
\qquad \mathit{ResponseTerminationOp_1} \lor \\
\qquad \mathit{ResponseTerminationOp_2} \lor \\
\qquad \mathit{ResponseTerminationOp_3}
\end{array}
$$

A.1.3.7 Integration

$Organism_BehaviorSelection$

$\Delta Organism$

$OperantSchedulingOp$

$stimulations? : \mathbb{P}\, Stimulation$

$currentInstant? : Instant$

$\forall\, st : stimulations? \mid st.status = Beginning \bullet$
$\quad \exists\, ReflexSchedulingOp \bullet$
$\quad\quad s? = st.stimulus \wedge i? = st.intensity$

$\exists\, BaseLevelSchedulingOp \bullet t? = currentInstant?$

$Organism_ConflicResolution$

$\Delta Organism$

$ConflictResolutionOp_Ref1$

$Organism_ResponseEmission$

$\Delta Organism$

$currentInstant? : Instant$

$\forall\, o : emitted \bullet$
$\quad \exists\, OperantEmissionOp \bullet o? = o$

$\forall\, r : elicited \bullet$
$\quad \exists\, ReflexElicitationOp \bullet r? = r$

$\forall\, a : spontaneous \bullet$
$\quad \exists\, BaseLevelEmissionOp \bullet a? = a$

$Organism_ResponseMaintenance$

$\Delta Organism$

$\forall\, rp : responses \bullet$
$\quad \exists\, T_ResponseUpdateOp \bullet \theta Response = rp$

$\forall\, rp : responses \bullet$
$\quad \exists\, T_ResponseTerminationOp \bullet rp? = rp$

A.1.4 Operant Behaviour

A.1.4.1 Basic Entities

Operant

$StimulusUtility$

$antecedents : \mathbb{P}(\mathbb{P}\, Stimulus)$

$action : Action$

$consequence : Stimulus$

$consequenceContingency : (\mathbb{P}\, Stimulus) \nrightarrow Correlation$

$sUtility(consequence) \neq neutral_utility$

$\mathrm{dom}\, consequenceContingency = antecedents$

A.1.4.2 Operant Implication

OperantImplication

$StimulusImplication$

$Discrimination$

$oCauses : \mathbb{P}(Operant \times Stimulus)$

$oCorrelation : Operant \times Stimulus \nrightarrow Correlation$

$\forall\, o : Operant \bullet o\ \underline{oCauses}\ o.consequence$

$\forall\, o_1, o_2 : Operant;\ S : \mathbb{P}\, Stimulus \mid S\ \underline{discriminatesNonEmpty}\ o_2 \bullet$

$\quad (\forall\, s : S \bullet o_1.consequence\ \underline{sCauses}\ s) \Rightarrow$

$\quad\quad o_1\ \underline{oCauses}\ o_2.consequence$

$\mathrm{dom}\, oCorrelation = oCauses$

Discrimination

$discriminates : \mathbb{P}(\mathbb{P}\, Stimulus \times Operant)$

$discriminatesNonEmpty : \mathbb{P}(\mathbb{P}\, Stimulus \times Operant)$

$\forall\, S : \mathbb{P}\, Stimulus;\ o : Operant \bullet$

$\quad S\ \underline{discriminates}\ o \Leftrightarrow (\exists\, A : o.antecedents \bullet A \subseteq S)$

$\forall\, S : \mathbb{P}\, Stimulus;\ o : Operant \bullet$

$\quad S\ \underline{discriminatesNonEmpty}\ o \Leftrightarrow$

$\quad\quad (\exists\, A : o.antecedents \mid A \neq \emptyset \bullet A \subseteq S)$

A.1.4.3 Operant Utility

$$
\begin{array}{l}
\textit{OperantUtility} \\
\hline
\textit{StimulusUtility} \\
\\
\textit{OperantImplication} \\
\\
\textit{oUtility} : (\textit{Operant} \times \mathbb{P}\,\textit{Stimulus}) \rightarrow \textit{Utility} \\
\hline
\end{array}
$$

$$
\begin{array}{l}
\textit{OperantUtility_Ref1} \\
\hline
\textit{OperantUtility} \\
\\
\textit{Discrimination} \\
\hline
\forall\, o : \textit{Operant};\ S : \mathbb{P}\,\textit{Stimulus} \bullet \\
\quad (\exists\, s : \textit{Stimulus} \mid \\
\qquad S\ \underline{\textit{discriminates}}\ o \wedge o\ \underline{\textit{oCauses}}\ s \bullet \\
\qquad\quad \textit{oUtility}(o, S) = \textit{sUtility}(s) \wedge \\
\qquad\quad (((\forall\, s' : \textit{Stimulus} \mid \\
\qquad\qquad (S\ \underline{\textit{discriminates}}\ o \wedge o\ \underline{\textit{oCauses}}\ s') \bullet \\
\qquad\qquad\quad \textit{sUtility}(s') \geq_1 \textit{neutral_utility} \wedge \\
\qquad\qquad\quad \textit{sUtility}(s) \geq_1 \textit{sUtility}(s')) \wedge \\
\qquad\quad \textit{sUtility}(s) \geq_1 \textit{neutral_utility}) \vee \\
\qquad\quad ((\forall\, s' : \textit{Stimulus} \mid \\
\qquad\qquad (S\ \underline{\textit{discriminates}}\ o \wedge o\ \underline{\textit{oCauses}}\ s') \bullet \\
\qquad\qquad\quad \textit{sUtility}(s) \leq_1 \textit{sUtility}(s')) \wedge \\
\qquad\quad \textit{sUtility}(s) <_1 \textit{neutral_utility})) \vee \\
\quad ((\neg\, (\exists\, s : \textit{Stimulus} \bullet \\
\qquad S\ \underline{\textit{discriminates}}\ o \wedge o\ \underline{\textit{oCauses}}\ s)) \wedge \\
\quad (\textit{oUtility}(o, S) = \textit{neutral_utility}))
\end{array}
$$

A.1.4.4 Fundamental Operations

$OperantOp$

$\Delta Operant$

$\Delta EmotionSubsystem$

$OperantUtility$

$discriminativeStimuli? : \mathbb{P}\, Stimulus$

$consequence? : Stimulus$

$action? : Action$

$delay? : Duration$

$action? = action$

$delay? \leq max_delay$

$OperantFormationOp$

$StimulationParameters$

$\Delta EmotionSubsystem$

$action? : Action$

$consequence? : Stimulus$

$discriminativeStimuli? : \mathbb{P}\, Stimulus$

$delay? : Duration$

$new! : Operant$

$delay? \leq max_delay$

$discriminativeStimuli? \neq \emptyset$

$consequence? \notin discriminativeStimuli?$

$new!.antecedents = \{discriminativeStimuli?\}$

$new!.consequence = consequence?$

$new!.action = action?$

$\mathrm{dom}\, new!.consequenceContingency = \{discriminativeStimuli?\}$

OperantEliminationOp_1

$\Delta RespondingSubsystem$

$operant? : Operant$

$\neg\ (\exists\, c : Correlation \mid c \in \operatorname{ran} operant?.consequenceContingency \bullet$
$\quad c \geq_1 min_correlation)$

$operants' = operants \setminus \{operant?\}$

OperantEliminationOp_2

$\Xi RespondingSubsystem$

$operant? : Operant$

$\exists\, c : Correlation \mid c \in \operatorname{ran} operant?.consequenceContingency \bullet$
$\quad c \geq_1 min_correlation$

$$T_OperantEliminationOp \mathrel{\widehat{=}} OperantEliminationOp_1 \lor OperantEliminationOp_2$$

DiscriminationOp

$OperantOp$

$discriminativeStimuli? \notin \operatorname{dom} consequenceContingency$

$consequence?\ \underline{sCauses}\ consequence$

$consequence? \notin discriminativeStimuli?$

$\operatorname{dom} consequenceContingency' =$
$\quad \operatorname{dom} consequenceContingency \cup \{discriminativeStimuli?\}$

$consequenceContingency'(discriminativeStimuli?) >_1 min_correlation$

OperantConditioningOp

$OperantOp$

$discriminativeStimuli? \in \operatorname{dom} consequenceContingency$

$consequence?\ \underline{sCauses}\ consequence$

$consequenceContingency'(discriminativeStimuli?)$
$\quad \geq_1 consequenceContingency(discriminativeStimuli?)$

$$
\begin{array}{|l}
\hline
\textit{ExtinctionOp} \\
\textit{OperantOp} \\
\textit{StartFrustrationOp} \\
\hline
\textit{discriminativeStimuli?} \in \mathrm{dom}\ \textit{consequenceContingency} \\
\neg\ (\textit{consequence?}\ \underline{\textit{sCauses}}\ \textit{consequence}) \\
\textit{consequenceContingency}'(\textit{discriminativeStimuli?}) \\
\qquad \leq_1 \textit{consequenceContingency}(\textit{discriminativeStimuli?}) \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{NeutralOp} \\
\textit{OperantOp} \\
\hline
\textit{discriminativeStimuli?} \notin \mathrm{dom}\ \textit{consequenceContingency} \\
\neg\ (\textit{consequence?}\ \underline{\textit{sCauses}}\ \textit{consequence}) \\
\textit{consequenceContingency}'(\textit{discriminativeStimuli?}) \\
\qquad = \textit{consequenceContingency}(\textit{discriminativeStimuli?}) \\
\hline
\end{array}
$$

$$
\begin{aligned}
&\textit{FundamentalOperantOp} \mathrel{\widehat{=}} \\
&\qquad \textit{DiscriminationOp} \lor \\
&\qquad \textit{OperantConditioningOp} \lor \\
&\qquad \textit{ExtinctionOp} \lor \\
&\qquad \textit{NeutralOp}
\end{aligned}
$$

A.1.4.5 Reinforcement and Punishment Operations

$$
\begin{array}{|l}
\hline
\textit{PositiveReinforcement} \\
\textit{StimulusUtility} \\
\textit{consequence?} : \textit{Stimulus} \\
\hline
\textit{sUtility}(\textit{consequence?}) >_1 \textit{neutral_utility} \\
\textit{stimulus_status}(\textit{consequence?}) = \textit{Beginning} \\
\hline
\end{array}
$$

$$
\begin{aligned}
\textit{PositiveReinforcementOp_1} \mathrel{\widehat{=}}\ & \textit{FundamentalOperantOp} \land \\
& \textit{PositiveReinforcement}
\end{aligned}
$$

$$
\begin{aligned}
\textit{PositiveReinforcementOp_2} \mathrel{\widehat{=}}\ & \textit{OperantFormationOp} \land \\
& \textit{PositiveReinforcement}
\end{aligned}
$$

$$
\begin{array}{|l}
\hline
\textit{NegativeReinforcement} \\
\textit{StimulusUtility} \\
\textit{consequence?} : \textit{Stimulus} \\
\hline
\textit{sUtility}(\textit{consequence?}) <_1 \textit{neutral_utility} \\
\textit{stimulus_status}(\textit{consequence?}) = \textit{Ending} \\
\hline
\end{array}
$$

$$
\begin{aligned}
\textit{NegativeReinforcementOp_1} \mathrel{\widehat{=}} & \\
& \textit{FundamentalOperantOp} \wedge \\
& \textit{NegativeReinforcement}
\end{aligned}
$$

$$
\begin{aligned}
\textit{NegativeReinforcementOp_2} \mathrel{\widehat{=}} & \\
& \textit{OperantFormationOp} \wedge \\
& \textit{NegativeReinforcement}
\end{aligned}
$$

$$
\begin{array}{|l}
\hline
\textit{PositivePunishment} \\
\textit{StimulusUtility} \\
\textit{consequence?} : \textit{Stimulus} \\
\hline
\textit{sUtility}(\textit{consequence?}) <_1 \textit{neutral_utility} \\
\textit{stimulus_status}(\textit{consequence?}) = \textit{Beginning} \\
\hline
\end{array}
$$

$$
\begin{aligned}
\textit{PositivePunishmentOp_1} \mathrel{\widehat{=}} & \\
& \textit{FundamentalOperantOp} \wedge \\
& \textit{PositivePunishment} \wedge \\
& \textit{StartAngerOp}
\end{aligned}
$$

$$
\begin{aligned}
\textit{PositivePunishmentOp_2} \mathrel{\widehat{=}} & \\
& \textit{OperantFormationOp} \wedge \\
& \textit{PositivePunishment} \wedge \\
& \textit{StartAngerOp}
\end{aligned}
$$

$$
\begin{array}{|l}
\hline
\textit{NegativePunishment} \\
\textit{StimulusUtility} \\
\textit{consequence?} : \textit{Stimulus} \\
\hline
\textit{sUtility}(\textit{consequence?}) >_1 \textit{neutral_utility} \\
\textit{stimulus_status}(\textit{consequence?}) = \textit{Ending} \\
\hline
\end{array}
$$

$$NegativePunishmentOp_1 \mathrel{\widehat{=}} FundamentalOperantOp \land NegativePunishment \land StartDepressionOp$$

$$NegativePunishmentOp_2 \mathrel{\widehat{=}} OperantFormationOp \land NegativePunishment \land StartDepressionOp$$

$NeutralReinforcementOp_1$

$\Xi Operant$

$\Xi EmotionSubsystem$

$StimulationParameters$

$StimulusUtility$

$consequence? : Stimulus$

$delay? : Duration$

$action? : Action$

$sUtility(consequence?) = neutral_utility \lor$
$\quad delay? > max_delay \lor$
$\quad action? \neq action$

$NeutralReinforcementOp_2$

$\Xi EmotionSubsystem$

$StimulationParameters$

$StimulusUtility$

$consequence? : Stimulus$

$discriminativeStimuli? : \mathbb{P}\, Stimulus$

$delay? : Duration$

$sUtility(consequence?) = neutral_utility \lor$
$\quad delay? > max_delay \lor$
$\quad discriminativeStimuli? = \emptyset$

$$T_OperantOp \mathrel{\widehat{=}} PositiveReinforcementOp_1 \lor NegativeReinforcementOp_1 \lor PositivePunishmentOp_1 \lor NegativePunishmentOp_1 \lor NeutralReinforcementOp_1$$

$$T_OperantFormationOp \mathrel{\widehat{=}} PositiveReinforcementOp_2 \lor NegativeReinforcementOp_2 \lor PositivePunishmentOp_2 \lor NegativePunishmentOp_2 \lor NeutralReinforcementOp_2$$

A.1.4.6 Emission Condition

$$\begin{array}{|l}
\hline
\textit{OperantEmissionCond} \\
\hline
RespondingSubsystem \\
StimulationSubsystem \\
OperantUtility \\
Discrimination \\
o : Operant \\
\hline
currentStimuli \ \underline{discriminates} \ o \\
oUtility(o, currentStimuli) >_1 neutral_utility \\
\forall\, x : operants \mid \\
\quad x \neq o \land currentStimuli \ \underline{discriminates} \ x \land x.action = o.action \bullet \\
\qquad oUtility(x, currentStimuli) \geq_1 neutral_utility \\
\hline
\end{array}$$

A.1.4.7 Integration

$$
\begin{array}{l}
\textit{Organism_OperantOp} \\
\hline
\Delta \mathit{Organism} \\
\mathit{stimulations?} : \mathbb{P}\, \mathit{Stimulation} \\
\mathit{currentInstant?} : \mathit{Instant} \\
\hline
\forall\, t : \mathit{Instant} \mid \mathit{currentInstant?} > t \geq \mathit{currentInstant?} - \mathit{max_delay} \wedge t \geq 0 \bullet \\
\quad \forall\, a : \mathit{Action} \mid a \in \mathit{actionsHistory}(t) \bullet \\
\quad\quad \forall\, o : \mathit{operants} \mid o.\mathit{action} = a \bullet \\
\quad\quad\quad \forall\, \mathit{st} : \mathit{stimulations?} \mid \mathit{st.status} \neq \mathit{Absent} \bullet \\
\quad\quad\quad\quad \exists\, \mathit{T_OperantOp} \bullet \\
\quad\quad\quad\quad\quad \mathit{discriminativeStimuli?} = \mathit{pastStimuli}(t) \wedge \\
\quad\quad\quad\quad\quad \mathit{consequence?} = \mathit{st.stimulus} \wedge \\
\quad\quad\quad\quad\quad \mathit{action?} = a \wedge \\
\quad\quad\quad\quad\quad \mathit{delay?} = \mathit{currentInstant?} - t
\end{array}
$$

$$
\begin{array}{l}
\textit{Organism_OperantFormationOp} \\
\hline
\Delta \mathit{Organism} \\
\mathit{stimulations?} : \mathbb{P}\, \mathit{Stimulation} \\
\mathit{currentInstant?} : \mathit{Instant} \\
\hline
\forall\, t : \mathit{Instant} \mid \mathit{currentInstant?} > t \geq \mathit{currentInstant?} - \mathit{max_delay} \wedge t \geq 0 \bullet \\
\quad \forall\, a : \mathit{Action} \mid a \in \mathit{actionsHistory}(t) \bullet \\
\quad\quad \forall\, \mathit{st} : \mathit{stimulations?} \mid \mathit{st.status} \neq \mathit{Absent} \bullet \\
\quad\quad\quad \exists\, \mathit{T_OperantFormationOp} \bullet \\
\quad\quad\quad\quad \mathit{discriminativeStimuli?} = \mathit{pastStimuli}(t) \wedge \\
\quad\quad\quad\quad \mathit{consequence?} = \mathit{st.stimulus} \wedge \\
\quad\quad\quad\quad \mathit{action?} = a \wedge \\
\quad\quad\quad\quad \mathit{delay?} = \mathit{currentInstant?} - t \wedge \\
\quad\quad\quad\quad \mathit{new!} \in \mathit{operants}'
\end{array}
$$

$$
\begin{array}{l}
\textit{Organism_OperantEliminationOp} \\
\hline
\Delta \mathit{Organism} \\
\hline
\forall\, o : \mathit{operants} \bullet \\
\quad \exists\, \mathit{T_OperantEliminationOp} \bullet \mathit{operant?} = o
\end{array}
$$

A.1.5 Respondent Behaviour

A.1.5.1 Basic Entities

Reflex

$Actions$

$ReflexParameters$

$antecedent : Stimulus$

$action : Action$

$threshold : Intensity$

$elicitation : Probability$

$magnitude : Intensity$

$duration : Duration$

$latency : Duration$

$action \in reflexActions$

$min_elicitation \leq_1 elicitation \leq_1 max_elicitation$

$min_magnitude \leq_1 magnitude \leq_1 max_magnitude$

$min_duration \leq duration \leq max_duration$

$min_latency \leq latency \leq max_latency$

$min_threshold \leq_1 threshold \leq_1 max_threshold$

ReflexParameters

$delta_elicitation : Probability \times Instant \times Instant \rightarrow Probability$

$delta_magnitude : Intensity \times Instant \times Instant \rightarrow Intensity$

$delta_duration : Duration \times Instant \times Instant \rightarrow Duration$

$delta_latency : Duration \times Instant \times Instant \rightarrow Duration$

$delta_threshold : Intensity \times Instant \times Instant \rightarrow Intensity$

$max_elicitation : Probability$

$min_elicitation : Probability$

$max_magnitude : Intensity$

$min_magnitude : Intensity$

$max_duration : Duration$

$min_duration : Duration$

$max_latency : Duration$

$min_latency : Duration$

$max_threshold : Intensity$

$min_threshold : Intensity$

$$\begin{aligned}
&\forall\, t_1, t_2 : Instant;\ p : Probability;\ i_1, i_2 : Intensity;\ d : Duration \bullet \\
&\quad min_elicitation \leq_1 delta_elicitation(p, t_1, t_2) \leq_1 max_elicitation \wedge \\
&\quad min_magnitude \leq_1 delta_magnitude(i_1, t_1, t_2) \leq_1 max_magnitude \wedge \\
&\quad min_duration \leq delta_duration(d, t_1, t_2) \leq max_duration \wedge \\
&\quad min_latency \leq delta_latency(d, t_1, t_2) \leq max_latency \wedge \\
&\quad min_threshold \leq_1 delta_threshold(i_2, t_1, t_2) \leq_1 max_threshold
\end{aligned}$$

A.1.5.2 Operations

ReflexAdjustmentOp

$\Delta Reflex$

$t_1? : Instant$

$t_2? : Instant$

$elicitation' = delta_elicitation(elicitation, t_1?, t_2?)$

$magnitude' = delta_magnitude(magnitude, t_1?, t_2?)$

$duration' = delta_duration(duration, t_1?, t_2?)$

$latency' = delta_latency(latency, t_1?, t_2?)$

$\theta ReflexParameters' = \theta ReflexParameters$

$antecedent' = antecedent$

$action' = action$

A.1.5.3 Elicitation Condition

ReflexElicitationCond

$StimulusImplication$

$r : Reflex$

$s : Stimulus$

$i : Intensity$

$s \ \underline{sCauses} \ (r.antecedent)$

$(r.threshold) \leq_1 i$

A.1.6 Drives

A.1.6.1 Basic Entities

DriveSubsystem

$activeDrives : \mathbb{P}\, Drive$

Drive

$importance : Utility$

$desires : \mathbb{P}\, Stimulus$

$deprivation : Utility \rightarrow Utility$

$satiation : Utility \rightarrow Utility$

$maxImportance, minImportance : Utility$

$importance \geq_1 minImportance$

$importance \leq_1 maxImportance$

$\forall\, u : Utility \bullet deprivation(u) \geq_1 u$

$\forall\, u : Utility \bullet satiation(u) \leq_1 u$

A.1.6.2 Stimulus Regulation

StimulusDriveRegulator

$DriveSubsystem$

$driveRegulator : (Stimulus \times Utility) \rightarrow Utility$

$\forall\, s : Stimulus;\ u : Utility \bullet$
$\quad driveRegulator(s, u) = u +_2 f(s, neutral_utility, activeDrives)$

$f : (Stimulus \times Utility \times (\mathbb{P}\, Drive)) \rightarrow Utility$

$\forall\, s : Stimulus;\ u : Utility;\ ds : \mathbb{P}\, Drive \mid ds = \emptyset \bullet$
$\quad f(s, u, ds) = neutral_utility$

$\forall\, s : Stimulus;\ u : Utility;\ ds : \mathbb{P}\, Drive \mid ds \neq \emptyset \bullet$
$\quad \exists\, d : ds \bullet$
$\qquad (s \in d.desires \Rightarrow f(s, u, ds) =$
$\qquad\quad u +_2 d.importance +_2 f(s, u, ds \setminus \{d\})) \wedge$
$\qquad (s \notin d.desires \Rightarrow f(s, u, ds) =$
$\qquad\quad f(s, u, ds \setminus \{d\}))$

A.1.6.3 Operations

$$
\begin{array}{|l}
\hline
\textit{DriveOp} \\
\Delta Drive \\
stimulations? : \mathbb{P}\, Stimulation \\
present : \mathbb{P}\, Stimulus \\
\hline
present = \{\, s : Stimulus \mid \\
\quad (\exists\, st : stimulations? \bullet \\
\qquad st.stimulus = s \wedge st.status = Stable)\,\} \\
desires' = desires \\
deprivation' = deprivation \\
satiation' = satiation \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{SatiationOp} \\
DriveOp \\
\hline
desires \subseteq present \\
importance' = satiation(importance) \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{DeprivationOp} \\
DriveOp \\
\hline
\neg\, (desires \subseteq present) \\
importance' = deprivation(importance) \\
\hline
\end{array}
$$

$$T_DriveOp \mathrel{\widehat{=}} SatiationOp \vee DeprivationOp$$

A.1.6.4 Integration

$$
\begin{array}{|l}
\hline
\textit{Organism_DrivesUpdate} \\
\Delta Organism \\
stimulations? : \mathbb{P}\, Stimulation \\
\hline
\forall\, d : activeDrives \bullet \\
\quad \exists\, T_DriveOp \bullet \theta Drive = d \\
\hline
\end{array}
$$

A.1.7 Emotions

A.1.7.1 Basic Entities

EmotionSubsystem

$anger : Anger$

$depression : Depression$

$frustration : Frustration$

$EmotionStatus ::= Active \mid Inactive$

Emotion

$status : EmotionStatus$

$intensity : Intensity$

$duration : Duration$

UtilityRegulatorEmotion

$Emotion$

$utilityChange : Intensity \rightarrow Utility$

ProbabilityRegulatorEmotion

$Emotion$

$probabilityChange : Intensity \rightarrow Probability$

Anger

$UtilityRegulatorEmotion$

Depression

$UtilityRegulatorEmotion$

Frustration

$ProbabilityRegulatorEmotion$

A.1.7.2 Stimulus Regulation

$StimulusEmotionalRegulator$

$DepressionRegulator$

$AngerRegulator$

$emotionalRegulator : (Stimulus \times Utility) \rightarrow Utility$

$\forall\, s : Stimulus;\ u : Utility \bullet$
$\quad emotionalRegulator(s, u) =$
$\quad\quad angerRegulator(s, depressionRegulator(s, u))$

$DepressionRegulator_1$

$EmotionSubsystem$

$depressionRegulator : (Stimulus \times Utility) \rightarrow Utility$

$depression.status = Active$

$\forall\, s : Stimulus;\ u : Utility \bullet$
$\quad depressionRegulator(s, u) =$
$\quad\quad u +_2 depression.utilityChange(depression.intensity)$

$DepressionRegulator_2$

$EmotionSubsystem$

$depressionRegulator : (Stimulus \times Utility) \rightarrow Utility$

$depression.status = Inactive$

$\forall\, s : Stimulus;\ u : Utility \bullet$
$\quad depressionRegulator(s, u) = u$

$$DepressionRegulator \mathrel{\widehat{=}} DepressionRegulator_1 \lor DepressionRegulator_2$$

$AngerRegulator_1$

$EmotionSubsystem$

$StimulationHints$

$angerRegulator : (Stimulus \times Utility) \rightarrow Utility$

$anger.status = Active$

$\forall\, s : Stimulus;\ u : Utility \mid s \in painHints \bullet$
$\quad angerRegulator(s, u) = u +_2 anger.utilityChange(anger.intensity)$

$\forall\, s : Stimulus;\ u : Utility \mid s \notin painHints \bullet$
$\quad angerRegulator(s, u) = u$

$AngerRegulator_2$

$EmotionSubsystem$

$angerRegulator : (Stimulus \times Utility) \rightarrow Utility$

$anger.status = Inactive$

$\forall\, s : Stimulus;\ u : Utility \bullet$
$\quad angerRegulator(s, u) = u$

$AngerRegulator \mathrel{\widehat{=}} AngerRegulator_1 \vee AngerRegulator_2$

A.1.7.3 Response Regulation

$ResponseEmotionalRegulator$

$FrustrationRegulator$

$responseRegulator : (Action \times Probability) \rightarrow Probability$

$\forall\, a : Action;\ p : Probability \bullet$
$\quad responseRegulator(a, p) =$
$\qquad frustrationRegulator(a, p)$

$FrustrationRegulator_1$

$EmotionSubsystem$

$frustrationRegulator : (Action \times Probability) \rightarrow Probability$

$frustration.status = Active$

$\forall\, a : Action;\ p : Probability \bullet$
$\quad frustrationRegulator(a, p) =$
$\qquad p +_1 frustration.probabilityChange(frustration.intensity)$

$FrustrationRegulator_2$

$EmotionSubsystem$

$frustrationRegulator : (Action \times Probability) \rightarrow Probability$

$frustration.status = Inactive$

$\forall\, a : Action;\ p : Probability \bullet$
$\quad frustrationRegulator(a, p) = p$

$FrustrationRegulator \mathrel{\widehat{=}} FrustrationRegulator_1 \vee FrustrationRegulator_2$

A.1.7.4 Operations

$StartDepressionOp$

$\Delta EmotionSubsystem$

$intensity? : Intensity$

$duration? : Duration$

$depression'.status = Active$

$depression'.intensity = intensity?$

$depression'.duration = duration?$

$anger' = anger$

$frustration' = frustration$

$StartAngerOp$

$\Delta EmotionSubsystem$

$intensity? : Intensity$

$duration? : Duration$

$anger'.status = Active$

$anger'.intensity = intensity?$

$anger'.duration = duration?$

$depression' = depression$

$frustration' = frustration$

$StartFrustrationOp$

$\Delta EmotionSubsystem$

$intensity? : Intensity$

$duration? : Duration$

$frustration'.status = Active$

$frustration'.intensity = intensity?$

$frustration'.duration = duration?$

$anger' = anger$

$depression' = depression$

UpdateDepressionOp

$\Delta EmotionSubsystem$

$depression.duration > 0$

$depression'.duration = depression.duration - 1$

$anger' = anger$

$frustration' = frustration$

UpdateAngerOp

$\Delta EmotionSubsystem$

$anger.duration > 0$

$anger'.duration = anger.duration - 1$

$depression' = depression$

$frustration' = frustration$

UpdateFrustrationOp

$\Delta EmotionSubsystem$

$frustration.duration > 0$

$frustration'.duration = frustration.duration - 1$

$anger' = anger$

$depression' = depression$

EndDepressionOp

$\Delta EmotionSubsystem$

$depression.duration \leq 0$

$depression'.status = Inactive$

$anger' = anger$

$frustration' = frustration$

$EndAngerOp$

$\Delta EmotionSubsystem$

$anger.duration \leq 0$

$anger'.status = Inactive$

$depression' = depression$

$frustration' = frustration$

$EndFrustrationOp$

$\Delta EmotionSubsystem$

$frustration.duration \leq 0$

$frustration'.status = Inactive$

$anger' = anger$

$depression' = depression$

A.1.7.5 Integration

$Organism_EmotionUpdate$

$\Delta Organism$

$UpdateDepressionOp \lor EndDepressionOp$

$UpdateAngerOp \lor EndAngerOp$

$UpdateFrustrationOp \lor EndFrustrationOp$

A.1.8 Subsystems Integration

A.1.8.1 Simulator

$Simulator$

$Organism$

$currentInstant : Instant$

A.1.8.2 Initial State

Init

$Simulator$

$currentInstant = 0$

A.1.8.3 Simulator Iteration Operation

SimulatorIterationOp

$\Delta Simulator$

$Organism_ConflicResolution$

$Organism_ResponseMaintenance$

$Organism_OperantEliminationOp$

$Organism_DrivesUpdate$

$Organism_EmotionUpdate$

$stimulations? : \mathbb{P}\, Stimulation$

$responses! : \mathbb{P}\, Response$

$currentInstant' = currentInstant + 1$

$\exists\, Organism_StimulusProcessing \bullet currentInstant? = currentInstant$

$\exists\, Organism_BehaviorSelection \bullet$
$\quad currentInstant? = currentInstant \land responses! = activeResponses$

$\exists\, Organism_ResponseEmission \bullet currentInstant? = currentInstant$

$\exists\, Organism_OperantOp \bullet currentInstant? = currentInstant$

$\exists\, Organism_OperantFormationOp \bullet currentInstant? = currentInstant$

APPENDIX B

Input Files and Tool Output for the Case Studies

This appendix contains actual XML input files to FGS, which correspond to the examples presented in Chapter 9, as well as the related FGS output. For each example, the following files are provided:

- One ore more parametrizations to the agents present in the example. These parametrizations are used to instantiate the implementation of the **Behaviourist Agent Architecture**;
- One scenario description, in which the agents are declared to exist, and the **EMMAS** specification is provided;
- One experiment description, in which a **simulation purpose** and the satisfiability relation to be checked are specified;
- The output of FGS, which includes the verdict, the time it took to apply the algorithm and a **synchronous run**.

Each section of this appendix correspond to an example of Chapter 9. Nonetheless, the files shown here may be somewhat more up-to-date than the formal specifications given there.

B.1 Pavlovian Dog

B.1.1 Agent

```
<?xml version="1.0" encoding="UTF-8"?>

<organism>

  <stimulation-subsystem>

    <stimulation-parameters>
      <stimulation-hints>
        <pleasure-hints>
          <stimulus id="0" />
        </pleasure-hints>
        <pain-hints>
          <stimulus id="4" />
        </pain-hints>
      </stimulation-hints>

      <stimuli>
        <stimulus id="0" name="food" primary="true"
           utility="0.9" />
        <stimulus id="1" name="bell" />
        <stimulus id="2" name="whistle" />
        <stimulus id="3" name="injection" primary="true"
           utility="-0.6" />
        <stimulus id="4" name="veterinary" primary="false"
            />
        <stimulus id="5" name="neutral" primary="true"
           utility="0.0" />
        <stimulus id="6" name="bark_sound" primary="true"
           utility="0.1" />
      </stimuli>

      <max-delay value="100" />

    </stimulation-parameters>

    <conditioning-parameters>
        <c value="0.5"/>
    </conditioning-parameters>

    <stimulus-implication>

    </stimulus-implication>
  </stimulation-subsystem>


  <responding-subsystem>

    <actions>
```

```
44        <action id="0" name="salivate" base-level="0.0"
             operant="false" reflex="true" />
45        <action id="1" name="bark" base-level="0.2" operant=
             "true" reflex="true" />
46        <action id="2" name="sit" base-level="0.1" operant="
             true" reflex="true" />
47        <action id="3" name="push_lever" base-level="0.0"
             operant="true" reflex="true" />
48      </actions>
49
50      <action-conflict />
51
52      <operants>
53
54        <operant>
55          <antecedents>
56            <antecedent contingency="0.9"/>
57          </antecedents>
58          <action id="3"/> <!-- push_lever -->
59          <consequence id="1"/> <!-- bell -->
60        </operant>
61
62        <operant>
63          <antecedents>
64            <antecedent contingency="0.9"/>
65          </antecedents>
66          <action id="1"/> <!-- bark -->
67          <consequence id="6"/> <!-- bark_sound -->
68        </operant>
69
70      </operants>
71
72
73      <reflexes>
74        <reflex>
75          <reflex-parameters>
76            <max-elicitation value="1.0" />
77            <min-elicitation value="0.9" />
78            <max-strength value="1.0" />
79            <min-strength value="0.5" />
80            <max-duration value="10" />
81            <min-duration value="2" />
82            <max-latency value="10" />
83            <min-latency value="1" />
84            <max-threshold value="0.1" />
85            <min-threshold value="0.3" />
86          </reflex-parameters>
87          <antecedent-stimulus id="0" />
88          <action id="0" />
```

```
89          <threshold value="0.3" />
90          <elicitation value="1.0" />
91          <strength value="1.0" />
92          <duration value="2" />
93          <latency value="1" />
94        </reflex>
95      </reflexes>
96
97    </responding-subsystem>
98
99    <drive-subsystem>
100     <drives/>
101   </drive-subsystem>
102
103   <emotion-subsystem>
104     <anger status="INACTIVE" intensity="0.0" duration="0"
          />
105     <depression status="INACTIVE" intensity="0.0" duration
          ="0" />
106     <frustration status="INACTIVE" intensity="0.0"
          duration="0" />
107   </emotion-subsystem>
108
109 </organism>
```

B.1.2 Scenario

```
1  <?xml version="1.0"?>
2
3  <scenario name="EMMAS dog training"
4                description="A dog trainment environment.">
5
6    <agent component-id="organism.OrganismComponent" id="0"
       name="Dog">
7      <initializer file="dog.agent.xml"/>
8    </agent>
9
10   <emmas>
11
12     <action-transformers>
13       <!-- The dog is alone. -->
14     </action-transformers>
15
16     <behaviors>
17
18       <behavior>
19         <environment-response agent-id="0" action="
             salivate">
20           <nop/>
```

```
21          </environment-response>
22        </behavior>
23
24
25        <behavior>
26          <environment-response agent-id="0" action="bark">
27            <nop/>
28          </environment-response>
29        </behavior>
30
31
32        <!-- Pushing a lever generates a bell sound -->
33        <behavior>
34          <environment-response agent-id="0" action="
              push_lever">
35            <stimulate stimulus="bell" agent-id="0" />
36          </environment-response>
37        </behavior>
38
39
40        <!-- Experiments that can be performed -->
41        <behavior>
42          <choice>
43            <sequential-composition><stimulate stimulus="
                bell" agent-id="0" /><stimulate stimulus="
                food" agent-id="0" /></sequential-composition
                >
44            <sequential-composition><stimulate stimulus="
                whistle" agent-id="0" /><stimulate stimulus="
                food" agent-id="0" /><stimulate stimulus="
                whistle" agent-id="0" /></sequential-
                composition>
45          </choice>
46        </behavior>
47
48      </behaviors>
49    </emmas>
50
51  </scenario>
```

B.1.3 Experiment

```
1  <?xml version="1.0"?>
2
3  <experiment name="Dog training program" description="A
     training program for dogs." >
4
5    <simulation-purpose-verification relation="feasibility">
6
```

```
7       <states>
8        <state id="initial"/>
9        <state id="1"/>
10       <state id="2"/>
11       <state id="3"/>
12       <state id="4"/>
13       <state id="5"/>
14       <state id="6"/>
15       <state id="7"/>
16       <state id="8"/>
17       <state id="9"/>
18       <state id="10"/>
19       <state id="11"/>
20       <state id="12"/>
21      </states>
22
23      <events>
24        <emmas-event id="!emit_salivate" type="output" name=
             "emit" action="salivate" agent-id="0"/>
25        <emmas-event id="!stop_salivate" type="output" name=
             "stop" action="salivate" agent-id="0"/>
26        <emmas-event id="!emit_push_lever" type="output"
             name="emit" action="push_lever" agent-id="0"/>
27        <emmas-event id="?beg_food" type="input" name="
             beginning" stimulus="food" agent-id="0"/>
28        <emmas-event id="?sta_food" type="input" name="
             stable" stimulus="food" agent-id="0"/>
29
30        <emmas-event id="?beg_whistle" type="input" name="
             beginning" stimulus="whistle" agent-id="0"/>
31        <emmas-event id="?sta_whistle" type="input" name="
             stable" stimulus="whistle" agent-id="0"/>
32
33        <emmas-event id="?beg_bell" type="input" name="
             beginning" stimulus="bell" agent-id="0"/>
34        <emmas-event id="?sta_bell" type="input" name="
             stable" stimulus="bell" agent-id="0"/>
35        <emmas-event id="?end_bell" type="input" name="
             ending" stimulus="bell" agent-id="0"/>
36        <emmas-event id="?abs_bell" type="input" name="
             absent" stimulus="bell" agent-id="0"/>
37      </events>
38
39
40      <!-- Two ways of training the dog, one of them must
           work. -->
41      <transitions>
42
```

```
43        <!-- Try to condition food upon whistle and check
             whether it worked. -->
44        <transition state-id1="initial" event-id="?
             beg_whistle" state-id2="1"/>
45        <transition state-id1="1" event-id="other" state-id2
             ="1"/>
46        <transition state-id1="1" event-id="?sta_whistle"
             state-id2="3"/>
47        <transition state-id1="3" event-id="other" state-id2
             ="3"/>
48        <transition state-id1="3" event-id="?beg_food" state
             -id2="4"/>
49        <transition state-id1="4" event-id="other" state-id2
             ="4"/>
50        <transition state-id1="4" event-id="?sta_food" state
             -id2="4"/>
51        <transition state-id1="4" event-id="!emit_salivate"
             state-id2="4"/>
52        <transition state-id1="4" event-id="?beg_whistle"
             state-id2="failure"/>
53        <transition state-id1="4" event-id="!stop_salivate"
             state-id2="5"/> <!-- make sure it is done -->
54        <transition state-id1="5" event-id="other" state-id2
             ="5"/>
55        <transition state-id1="5" event-id="?beg_whistle"
             state-id2="6"/>
56        <transition state-id1="6" event-id="other" state-id2
             ="6"/>
57        <transition state-id1="6" event-id="!emit_salivate"
             state-id2="success"/> <!-- the whistle alone
             managed to trigger salivation -->
58
59
60        <!-- Condition food upon bell and check whether the
             dog knows how to sound the bell. -->
61        <transition state-id1="initial" event-id="?beg_bell"
              state-id2="10"/>
62        <transition state-id1="10" event-id="?sta_bell"
             state-id2="10"/>
63        <transition state-id1="10" event-id="?end_bell"
             state-id2="10"/>
64        <transition state-id1="10" event-id="?abs_bell"
             state-id2="10"/>
65        <transition state-id1="10" event-id="other" state-
             id2="10"/>
66        <transition state-id1="10" event-id="?beg_whistle"
             state-id2="failure"/>
67        <transition state-id1="10" event-id="?beg_food"
             state-id2="12"/>
```

```
68        <transition state-id1="12" event-id="other" state-
             id2="12"/>
69        <transition state-id1="12" event-id="!
             emit_push_lever" state-id2="success"/>
70
71
72      </transitions>
73
74    </simulation-purpose-verification>
75
76  </experiment>
```

B.1.4 Result

```
Simulation Purpose Verification strategy
=====================================================
Result = SUCCESS
Running time = 2s

Run found:
    [depth = 0] State (in SP): initial
    [depth = 1] Events synch'd: <?beginning[agentId = 0]_[Stimulus type='bell'],
                                !beginning[agentId = 0]_[Stimulus type='bell']>;
                State annotations synch'd: <[],     []>  State (in SP): 10
    [depth = 2] Events synch'd: <(*)other,
                                !commit>;
                State annotations synch'd: <[],     []>  State (in SP): 10
    [depth = 3] Events synch'd: <?stable[agentId = 0]_[Stimulus type='bell'],
                                !stable[agentId = 0]_[Stimulus type='bell']>;
                State annotations synch'd: <[],     []>  State (in SP): 10
    [depth = 4] Events synch'd: <(*)other,
                                !commit>;
                State annotations synch'd: <[],     []>  State (in SP): 10
    [depth = 5] Events synch'd: <(*)other,
                                ?emit[agentId = 0]_[Action type='bark']>;
                State annotations synch'd: <[],     []>  State (in SP): 10
    [depth = 6] Events synch'd: <(*)other,
                                !commit>;
                State annotations synch'd: <[],     []>  State (in SP): 10
    [depth = 7] Events synch'd: <(*)other,
                                TAU>;
                State annotations synch'd: <[],     []>  State (in SP): 10
    [depth = 8] Events synch'd: <(*)other,
                                !commit>;
                State annotations synch'd: <[],     []>  State (in SP): 10
    [depth = 9] Events synch'd: <(*)other,
                                TAU>;
                State annotations synch'd: <[],     []>  State (in SP): 10
    [depth = 10] Events synch'd: <(*)other,
                                 !commit>;
                 State annotations synch'd: <[],     []>  State (in SP): 10
    [depth = 11] Events synch'd: <(*)other,
```

```
                              ?stop[agentId = 0]_[Action type='bark']>;
             State annotations synch'd: <[],     []>  State (in SP): 10
[depth = 12] Events synch'd: <(*)other,
                              TAU>;
             State annotations synch'd: <[],     []>  State (in SP): 10
[depth = 13] Events synch'd: <(*)other,
                              TAU>;
             State annotations synch'd: <[],     []>  State (in SP): 10
[depth = 14] Events synch'd: <(*)other,
                              !commit>;
             State annotations synch'd: <[],     []>  State (in SP): 10
[depth = 15] Events synch'd: <?ending[agentId = 0]_[Stimulus type='bell'],
                              !ending[agentId = 0]_[Stimulus type='bell']>;
             State annotations synch'd: <[],     []>  State (in SP): 10
[depth = 16] Events synch'd: <(*)other,
                              ?emit[agentId = 0]_[Action type='bark']>;
             State annotations synch'd: <[],     []>  State (in SP): 10
[depth = 17] Events synch'd: <(*)other,
                              !commit>;
             State annotations synch'd: <[],     []>  State (in SP): 10
[depth = 18] Events synch'd: <(*)other,
                              TAU>;
             State annotations synch'd: <[],     []>  State (in SP): 10
[depth = 19] Events synch'd: <(*)other,
                              !commit>;
             State annotations synch'd: <[],     []>  State (in SP): 10
[depth = 20] Events synch'd: <(*)other,
                              !commit>;
             State annotations synch'd: <[],     []>  State (in SP): 10
[depth = 21] Events synch'd: <?absent[agentId = 0]_[Stimulus type='bell'],
                              !absent[agentId = 0]_[Stimulus type='bell']>;
             State annotations synch'd: <[],     []>  State (in SP): 10
[depth = 22] Events synch'd: <(*)other,
                              !commit>;
             State annotations synch'd: <[],     []>  State (in SP): 10
[depth = 23] Events synch'd: <(*)other,
                              !commit>;
             State annotations synch'd: <[],     []>  State (in SP): 10
[depth = 24] Events synch'd: <(*)other,
                              TAU>;
             State annotations synch'd: <[],     []>  State (in SP): 10
[depth = 25] Events synch'd: <(*)other,
                              TAU>;
             State annotations synch'd: <[],     []>  State (in SP): 10
[depth = 26] Events synch'd: <?beginning[agentId = 0]_[Stimulus type='food'],
                              !beginning[agentId = 0]_[Stimulus type='food']>;
             State annotations synch'd: <[],     []>  State (in SP): 12
[depth = 27] Events synch'd: <(*)other,
                              ?emit[agentId = 0]_[Action type='salivate']>;
             State annotations synch'd: <[],     []>  State (in SP): 12
[depth = 28] Events synch'd: <(*)other,
                              !commit>;
             State annotations synch'd: <[],     []>  State (in SP): 12
[depth = 29] Events synch'd: <!emit[agentId = 0]_[Action type='push_lever'],
```

```
                                ?emit[agentId = 0]_[Action type='push_lever']>;
                State annotations synch'd: <[],     []>  State (in SP): success


Result = SUCCESS
Running time = 2s


Finished.
```

B.2 Worker

B.2.1 Agent

```xml
<?xml version="1.0" encoding="UTF-8"?>

<organism>

  <stimulation-subsystem>

    <stimulation-parameters>
      <stimulation-hints>
        <pleasure-hints>
          <stimulus id="0" />
        </pleasure-hints>
        <pain-hints>
        </pain-hints>
      </stimulation-hints>

      <stimuli>
        <stimulus id="0" name="food" primary="true"
           utility="0.9" />
        <stimulus id="1" name="money" />
        <stimulus id="2" name="work_place" />
        <stimulus id="3" name="home" />
      </stimuli>

      <max-delay value="10" />

    </stimulation-parameters>

    <conditioning-parameters>
        <c value="0.5"/>
    </conditioning-parameters>

    <stimulus-implication>

```

```
33      </stimulus-implication>
34    </stimulation-subsystem>
35
36
37    <responding-subsystem>
38
39      <actions>
40        <action id="0" name="work" base-level="0.0" operant=
            "true" reflex="false" />
41        <action id="1" name="buy_food" base-level="0.0"
            operant="true" reflex="false" />
42        <action id="2" name="wakeup_early" base-level="0.0"
            operant="true" reflex="true" />
43      </actions>
44
45      <action-conflict />
46
47      <operants>
48
49        <!-- Wake up and go to work -->
50        <operant>
51          <antecedents>
52            <antecedent contingency="0.9">
53              <stimulus id="3"/>
54            </antecedent>
55          </antecedents>
56          <action id="2"/>
57          <consequence id="2"/>
58        </operant>
59
60        <!-- Work -->
61        <operant>
62          <antecedents>
63            <antecedent contingency="1.0">
64              <stimulus id="2"/>
65            </antecedent>
66          </antecedents>
67          <action id="0"/>
68          <consequence id="1"/>
69        </operant>
70
71        <!-- Buy food -->
72        <operant>
73          <antecedents>
74            <antecedent contingency="1.0">
75              <stimulus id="1"/>
76            </antecedent>
77          </antecedents>
78          <action id="1"/>
```

```
79          <consequence id="0"/>
80        </operant>
81
82      </operants>
83
84      <reflexes/>
85
86    </responding-subsystem>
87
88    <drive-subsystem>
89      <drives>
90
91        <drive>
92          <importance value="0.0"/>
93          <max-importance value="1.0"/>
94          <min-importance value="-1.0"/>
95          <desires>
96            <stimulus id="0" />
97          </desires>
98        </drive>
99
100     </drives>
101   </drive-subsystem>
102
103   <emotion-subsystem>
104     <anger status="INACTIVE" intensity="0.0" duration="0"
          />
105     <depression status="INACTIVE" intensity="0.0" duration
          ="0" />
106     <frustration status="INACTIVE" intensity="0.0"
          duration="0" />
107   </emotion-subsystem>
108
109 </organism>
```

B.2.2 Scenario

```
1  <?xml version="1.0"?>
2  <scenario name="EMMAS worker"
3                description="En example of operant chaining.
                   ">
4
5
6    <agent component-id="organism.OrganismComponent" id="0"
       name="Worker">
7      <initializer file="worker.agent.xml"/>
8    </agent>
9
10   <emmas>
```

```

    <action-transformers>
      <!-- The worker is alone. -->
    </action-transformers>

    <behaviors>

      <!-- The agent begin in his home -->
      <behavior>
        <stimulate stimulus="home" agent-id="0" />
      </behavior>

      <!-- Reinforce the first operant in the chain -->
      <behavior>
        <environment-response agent-id="0" action="
            wakeup_early">
          <stimulate stimulus="work_place" agent-id="0" />
        </environment-response>
      </behavior>


      <!-- Reinforce the second operant in the chain -->
      <behavior>
        <environment-response agent-id="0" action="work">
          <stimulate stimulus="money" agent-id="0" />
        </environment-response>
      </behavior>


      <!-- Reinforce the third operant in the chain -->
      <behavior>
        <environment-response agent-id="0" action="
            buy_food">
          <stimulate stimulus="food" agent-id="0" />
        </environment-response>
      </behavior>


    </behaviors>

  </emmas>

</scenario>
```

B.2.3 Experiment

```
<?xml version="1.0"?>

```

```
3  <experiment name="Operant chaining mainenance" description
       ="An operant chaining mantenaince program for workers."
       >
4
5    <simulation-purpose-verification relation="feasibility">
6
7      <states>
8        <state id="initial"/>
9        <state id="1"/>
10       <state id="2"/>
11       <state id="3"/>
12     </states>
13
14     <events>
15       <emmas-event id="?beg_home" type="input" name="
             beginning" stimulus="home" agent-id="0"/>
16
17       <emmas-event id="!emit_wakeup_early" type="output"
             name="emit" action="wakeup_early" agent-id="0"/>
18       <emmas-event id="!emit_work" type="output" name="
             emit" action="work" agent-id="0"/>
19       <emmas-event id="!emit_buy_food" type="output" name=
             "emit" action="buy_food" agent-id="0"/>
20     </events>
21
22     <transitions>
23        <!-- Walk the agent through the chain -->
24
25        <transition state-id1="initial" event-id="other"
             state-id2="initial"/>
26        <transition state-id1="initial" event-id="?beg_home
             " state-id2="1"/>
27
28        <transition state-id1="1" event-id="other" state-
             id2="1"/>
29        <transition state-id1="1" event-id="!
             emit_wakeup_early" state-id2="2"/>
30
31        <transition state-id1="2" event-id="other" state-
             id2="2"/>
32        <transition state-id1="2" event-id="!emit_work"
             state-id2="3"/>
33
34        <transition state-id1="3" event-id="other" state-
             id2="3"/>
35        <transition state-id1="3" event-id="!emit_buy_food"
              state-id2="success"/>
36     </transitions>
37
```

```
38   </simulation-purpose-verification>
39
40 </experiment>
```

B.2.4 Result

```
Simulation Purpose Verification strategy
=====================================================
Result = SUCCESS
Running time = 1s

Run found:
    [depth = 0] State (in SP): initial
    [depth = 1] Events synch'd: <?beginning[agentId = 0]_[Stimulus type='home'],
                                !beginning[agentId = 0]_[Stimulus type='home']>;
                State annotations synch'd: <[],     []>  State (in SP): 1
    [depth = 2] Events synch'd: <(*)other,
                                !commit>;
                State annotations synch'd: <[],     []>  State (in SP): 1
    [depth = 3] Events synch'd: <!emit[agentId = 0]_[Action type='wakeup_early'],
                                ?emit[agentId = 0]_[Action type='wakeup_early']>;
                State annotations synch'd: <[],     []>  State (in SP): 2
    [depth = 4] Events synch'd: <(*)other,
                                TAU>;
                State annotations synch'd: <[],     []>  State (in SP): 2
    [depth = 5] Events synch'd: <(*)other,
                                !beginning[agentId = 0]_[Stimulus type='work_place']>;
                State annotations synch'd: <[],     []>  State (in SP): 2
    [depth = 6] Events synch'd: <(*)other,
                                !stable[agentId = 0]_[Stimulus type='home']>;
                State annotations synch'd: <[],     []>  State (in SP): 2
    [depth = 7] Events synch'd: <(*)other,
                                !commit>;
                State annotations synch'd: <[],     []>  State (in SP): 2
    [depth = 8] Events synch'd: <!emit[agentId = 0]_[Action type='work'],
                                ?emit[agentId = 0]_[Action type='work']>;
                State annotations synch'd: <[],     []>  State (in SP): 3
    [depth = 9] Events synch'd: <(*)other,
                                TAU>;
                State annotations synch'd: <[],     []>  State (in SP): 3
    [depth = 10] Events synch'd: <(*)other,
                                 TAU>;
                 State annotations synch'd: <[],     []>  State (in SP): 3
    [depth = 11] Events synch'd: <(*)other,
                                 !commit>;
                 State annotations synch'd: <[],     []>  State (in SP): 3
    [depth = 12] Events synch'd: <(*)other,
                                 !beginning[agentId = 0]_[Stimulus type='money']>;
                 State annotations synch'd: <[],     []>  State (in SP): 3
    [depth = 13] Events synch'd: <(*)other,
                                 !stable[agentId = 0]_[Stimulus type='work_place']>;
                 State annotations synch'd: <[],     []>  State (in SP): 3
    [depth = 14] Events synch'd: <(*)other,
```

```
                                !commit>;
                State annotations synch'd: <[],     []>  State (in SP): 3
    [depth = 15] Events synch'd: <!emit[agentId = 0]_[Action type='buy_food'],
                                ?emit[agentId = 0]_[Action type='buy_food']>;
                State annotations synch'd: <[],     []>  State (in SP): success



Result = SUCCESS
Running time = 1s


Finished.
```

B.3 Violent Child

B.3.1 Agents

Child The child whose behaviour must be modified.

```
<?xml version="1.0" encoding="UTF-8"?>

<organism>

  <stimulation-subsystem>

    <stimulation-parameters>
      <stimulation-hints>
        <pleasure-hints>
          <stimulus id="5" />
        </pleasure-hints>
        <pain-hints>
          <stimulus id="3" />
          <stimulus id="4" />
        </pain-hints>
      </stimulation-hints>

      <stimuli>
        <stimulus id="0" name="praise" primary="true"
           utility="0.5" />
        <stimulus id="1" name="candy"  primary="true"
           utility="0.9" />
        <stimulus id="2" name="punch" primary="true"
           utility="-0.6" />
        <stimulus id="3" name="scream_sound" primary="true
           " utility="0.2"/>
        <stimulus id="4" name="cry_sound" primary="true"
           utility="0.2" />
```

```
24          <stimulus id="5" name="neutral"/>
25        </stimuli>
26
27        <max-delay value="10" />
28
29      </stimulation-parameters>
30
31      <conditioning-parameters>
32          <c value="0.5"/>
33      </conditioning-parameters>
34
35      <stimulus-implication>
36
37      </stimulus-implication>
38    </stimulation-subsystem>
39
40
41    <responding-subsystem>
42
43      <actions>
44        <action id="0" name="beat" base-level="0.0" operant=
            "true" reflex="true" />
45        <action id="1" name="caress" base-level="0.2"
            operant="true" reflex="true" />
46      </actions>
47
48
49      <action-conflict>
50        <conflict id1="0" id2="1"/>
51        <conflict id1="0" id2="2"/>
52      </action-conflict>
53
54      <operants>
55        <operant>
56          <antecedents>
57            <antecedent contingency="0.9"/>
58          </antecedents>
59          <action id="0"/> <!-- beat -->
60          <consequence id="3"/> <!-- scream_sound -->
61        </operant>
62
63      </operants>
64
65
66      <reflexes>
67
68      </reflexes>
69
70    </responding-subsystem>
```

```
71
72    <drive-subsystem>
73      <drives>
74
75        <drive>
76          <importance value="0.0"/>
77          <max-importance value="0.0"/>
78          <min-importance value="-1.0"/>
79          <desires>
80            <stimulus id="3" />
81          </desires>
82        </drive>
83
84
85      </drives>
86    </drive-subsystem>
87
88    <emotion-subsystem>
89      <anger status="INACTIVE" intensity="0.0" duration="0"
          />
90      <depression status="INACTIVE" intensity="0.0" duration
          ="0" />
91      <frustration status="INACTIVE" intensity="0.0"
          duration="0" />
92    </emotion-subsystem>
93
94  </organism>
```

Dog The dog that suffers the child's misbehaviour.

```
1   <?xml version="1.0" encoding="UTF-8"?>
2
3   <organism>
4
5     <stimulation-subsystem>
6
7       <stimulation-parameters>
8         <stimulation-hints>
9           <pleasure-hints>
10            <stimulus id="0" />
11          </pleasure-hints>
12          <pain-hints>
13            <stimulus id="4" />
14          </pain-hints>
15        </stimulation-hints>
16
17        <stimuli>
18          <stimulus id="0" name="food" primary="true"
              utility="0.9" />
```

```
19          <stimulus id="1" name="bell" />
20          <stimulus id="2" name="whistle" />
21          <stimulus id="3" name="injection" primary="true"
               utility="-0.6" />
22          <stimulus id="4" name="veterinary" primary="false"
                />
23          <stimulus id="5" name="neutral" primary="true"
               utility="0.0" />
24          <stimulus id="6" name="bark_sound" primary="true"
               utility="0.1" />
25          <stimulus id="7" name="punch" primary="true"
               utility="-0.5" />
26          <stimulus id="8" name="caress" primary="true"
               utility="0.6" />
27        </stimuli>
28
29        <max-delay value="100" />
30
31      </stimulation-parameters>
32
33      <conditioning-parameters>
34          <c value="0.5"/>
35      </conditioning-parameters>
36
37      <stimulus-implication>
38
39      </stimulus-implication>
40    </stimulation-subsystem>
41
42
43    <responding-subsystem>
44
45      <actions>
46        <action id="0" name="salivate" base-level="0.0"
             operant="false" reflex="true" />
47        <action id="1" name="bark" base-level="0.2" operant=
             "true" reflex="true" />
48        <action id="2" name="sit" base-level="0.1" operant="
             true" reflex="true" />
49        <action id="3" name="push_lever" base-level="0.0"
             operant="true" reflex="true" />
50        <action id="4" name="scream" base-level="0.0"
             operant="true" reflex="true" />
51      </actions>
52
53      <action-conflict />
54
55      <operants>
56
```

```
57        <operant>
58          <antecedents>
59            <antecedent contingency="0.9"/>
60          </antecedents>
61          <action id="3"/> <!-- push_lever -->
62          <consequence id="1"/> <!-- bell -->
63        </operant>
64
65        <operant>
66          <antecedents>
67            <antecedent contingency="0.9"/>
68          </antecedents>
69          <action id="1"/> <!-- bark -->
70          <consequence id="6"/> <!-- bark_sound -->
71        </operant>
72
73      </operants>
74
75
76      <reflexes>
77
78        When fed, the dog salivates.
79        <reflex>
80          <reflex-parameters>
81            <max-elicitation value="1.0" />
82            <min-elicitation value="0.9" />
83            <max-strength value="1.0" />
84            <min-strength value="0.5" />
85            <max-duration value="10" />
86            <min-duration value="2" />
87            <max-latency value="10" />
88            <min-latency value="1" />
89            <max-threshold value="0.1" />
90            <min-threshold value="0.3" />
91          </reflex-parameters>
92          <antecedent-stimulus id="0" />
93          <action id="0" />
94          <threshold value="0.3" />
95          <elicitation value="1.0" />
96          <strength value="1.0" />
97          <duration value="10" />
98          <latency value="2" />
99        </reflex>
100
101
102       When punched, the dog screams.
103       <reflex>
104         <reflex-parameters>
105           <max-elicitation value="1.0" />
```

```
106            <min-elicitation value="0.9" />
107            <max-strength value="1.0" />
108            <min-strength value="0.5" />
109            <max-duration value="10" />
110            <min-duration value="2" />
111            <max-latency value="10" />
112            <min-latency value="1" />
113            <max-threshold value="0.1" />
114            <min-threshold value="0.3" />
115          </reflex-parameters>
116          <antecedent-stimulus id="7" />
117          <action id="4" />
118          <threshold value="0.3" />
119          <elicitation value="1.0" />
120          <strength value="1.0" />
121          <duration value="10" />
122          <latency value="2" />
123        </reflex>
124
125
126      </reflexes>
127
128    </responding-subsystem>
129
130    <drive-subsystem>
131      <drives/>
132    </drive-subsystem>
133
134    <emotion-subsystem>
135      <anger status="INACTIVE" intensity="0.0" duration="0"
         />
136      <depression status="INACTIVE" intensity="0.0" duration
         ="0" />
137      <frustration status="INACTIVE" intensity="0.0"
         duration="0" />
138    </emotion-subsystem>
139
140  </organism>
```

B.3.2 Scenario

```
1  <?xml version="1.0"?>
2
3  <scenario name="EMMAS violent child"
4                description="An example of how to change the
                     behaviour of a problem child.">
5
6
```

```
7   <!-- Organisms -->
8   <agent component-id="organism.OrganismComponent" id="0"
       name="Child">
9     <initializer file="child.agent.xml"/>
10  </agent>
11
12  <agent component-id="organism.OrganismComponent" id="1"
       name="Dog">
13    <initializer file="dog.agent.xml"/>
14  </agent>
15
16  <!-- A property that inspects whether the child likes
       candy -->
17  <property component-id = "organism.
       StimulusUtilityProperty" id="0" name="LikesCandy">
18    <primitive-parameter name="TargetStimulus" value="
         candy" />
19    <primitive-parameter name="TargetValue" value="0.1" />
20
21    <agent-target id="0" />
22  </property>
23
24
25
26  <emmas>
27
28    <action-transformers>
29      <action-transformer agent-id1="0" action="beat"
           stimulus="punch" agent-id2="1"/>
30      <action-transformer agent-id1="0" action="caress"
           stimulus="caress" agent-id2="1"/>
31      <action-transformer agent-id1="1" action="scream"
           stimulus="scream_sound" agent-id2="0"/>
32    </action-transformers>
33
34    <behaviors>
35
36      <!-- Candy dispenser (we can give as much candy as
           we like to the child). -->
37      <behavior>
38        <unbounded-sequence><stimulate stimulus="candy"
             agent-id="0" /></unbounded-sequence>
39      </behavior>
40
41      <!-- A neutral background can always be provided.
           -->
42      <behavior>
43        <unbounded-sequence><stimulate stimulus="neutral"
             agent-id="0" /></unbounded-sequence>
```

```
44        </behavior>
45
46      </behaviors>
47
48    </emmas>
49
50  </scenario>
```

B.3.3 Experiment

```
1  <?xml version="1.0"?>
2
3  <experiment name="Behaviour elimination"
4              description="Eliminates undesirable behaviours
                   by reinforcement instead of punishment." >
5
6    <simulation-purpose-verification relation="strong
       feasibility">
7
8      <states>
9       <state id="initial"/>
10
11      <!-- Requires that this state must be annotated with
          the specified literal -->
12      <state id="1">
13        <literal proposition="LikesCandy" type="positive"/>
14      </state>
15
16      <state id="2"/>   <state id="3"/>   <state id="4"/>
          <state id="5"/>   <state id="6"/>   <state id="7
          "/>
17      <state id="8"/>   <state id="9"/>   <state id="10"/>
          <state id="11"/>   <state id="12"/>   <state id=
          "13"/>
18      <state id="14"/>   <state id="15"/>   <state id="16"/
          >   <state id="17"/>   <state id="18"/>   <state
          id="19"/>
19      <state id="20"/>   <state id="21"/>   <state id="22"/
          >   <state id="23"/>   <state id="24"/>   <state
          id="25"/>
20      <state id="26"/>   <state id="27"/>   <state id="28"/
          >   <state id="29"/>   <state id="30"/>   <state
          id="31"/>
21      <state id="32"/>   <state id="33"/>   <state id="34"/
          >   <state id="35"/>   <state id="36"/>   <state
          id="37"/>
22      <state id="38"/>   <state id="39"/>   <state id="40"/
          >   <state id="41"/>   <state id="42"/>   <state
          id="43"/>
```

```
23      <state id="44"/>   <state id="45"/>   <state id="46"/
           >   <state id="47"/>   <state id="50"/>
24    </states>
25
26    <events>
27      <!-- Child events -->
28      <emmas-event id="!emit_beat_0" type="output" name="
           emit" action="beat" agent-id="0"/>
29      <emmas-event id="!stop_beat_0" type="output" name="
           stop" action="beat" agent-id="0"/>
30      <emmas-event id="!emit_caress_0" type="output" name=
           "emit" action="caress" agent-id="0"/>
31      <emmas-event id="!stop_caress_0" type="output" name=
           "stop" action="caress" agent-id="0"/>
32      <emmas-event id="?beg_candy_0" type="input" name="
           beginning" stimulus="candy" agent-id="0"/>
33      <emmas-event id="?sta_candy_0" type="input" name="
           stable" stimulus="candy" agent-id="0"/>
34      <emmas-event id="?end_candy_0" type="input" name="
           ending" stimulus="candy" agent-id="0"/>
35      <emmas-event id="?abs_candy_0" type="input" name="
           absent" stimulus="candy" agent-id="0"/>
36      <emmas-event id="?beg_neutral_0" type="input" name="
           beginning" stimulus="neutral" agent-id="0"/>
37      <emmas-event id="?sta_neutral_0" type="input" name="
           stable" stimulus="neutral" agent-id="0"/>
38      <emmas-event id="?end_neutral_0" type="input" name="
           ending" stimulus="neutral" agent-id="0"/>
39      <emmas-event id="?abs_neutral_0" type="input" name="
           absent" stimulus="neutral" agent-id="0"/>
40      <emmas-event id="?beg_scream_sound_0" type="input"
           name="beginning" stimulus="scream_sound" agent-id
           ="0"/>
41      <emmas-event id="?sta_scream_sound_0" type="input"
           name="stable" stimulus="scream_sound" agent-id="0
           "/>
42      <emmas-event id="?end_scream_sound_0" type="input"
           name="ending" stimulus="scream_sound" agent-id="0
           "/>
43      <emmas-event id="?abs_scream_sound_0" type="input"
           name="absent" stimulus="scream_sound" agent-id="0
           "/>
44
45      <!-- Dog events -->
46      <emmas-event id="!emit_scream_1" type="output" name=
           "emit" action="scream" agent-id="1"/>
47      <emmas-event id="!stop_scream_1" type="output" name=
           "stop" action="scream" agent-id="1"/>
```

```
48        <emmas-event id="?beg_caress_1" type="input" name="
             beginning" stimulus="caress" agent-id="1"/>
49        <emmas-event id="?sta_caress_1" type="input" name="
             stable" stimulus="caress" agent-id="1"/>
50        <emmas-event id="?end_caress_1" type="input" name="
             ending" stimulus="caress" agent-id="1"/>
51        <emmas-event id="?abs_caress_1" type="input" name="
             absent" stimulus="caress" agent-id="1"/>
52        <emmas-event id="?beg_punch_1" type="input" name="
             beginning" stimulus="punch" agent-id="1"/>
53        <emmas-event id="?sta_punch_1" type="input" name="
             stable" stimulus="punch" agent-id="1"/>
54        <emmas-event id="?end_punch_1" type="input" name="
             ending" stimulus="punch" agent-id="1"/>
55        <emmas-event id="?abs_punch_1" type="input" name="
             absent" stimulus="punch" agent-id="1"/>
56
57      </events>
58
59
60      <transitions>
61
62       <!-- Let  the child beat the dog and (temporarily)
             get tired of it -->
63       <transition state-id1="initial" event-id="?
             beg_neutral_0" state-id2="1"/> <!-- Establish a
             discriminative stimulus -->
64       <transition state-id1="1" event-id="?commit" state-
             id2="1"/>              <!-- Loop here -->
65       <transition state-id1="1" event-id="!emit_beat_0"
             state-id2="3"/>
66       <transition state-id1="3" event-id="?commit" state-
             id2="4"/>
67       <transition state-id1="4" event-id="?beg_punch_1"
             state-id2="5"/>
68       <transition state-id1="5" event-id="?commit" state-
             id2="6"/>
69       <transition state-id1="6" event-id="?sta_punch_1"
             state-id2="7"/>
70       <transition state-id1="7" event-id="?sta_neutral_0"
             state-id2="8"/>
71       <transition state-id1="8" event-id="?commit" state-
             id2="9"/>
72       <transition state-id1="9" event-id="?commit" state-
             id2="10"/>
73       <transition state-id1="10" event-id="!emit_scream_1"
             state-id2="11"/>
74       <transition state-id1="11" event-id="?
             beg_scream_sound_0" state-id2="12"/>
```

```
75      <transition state-id1="12" event-id="?commit" state-
           id2="13"/>
76      <transition state-id1="13" event-id="?
           sta_scream_sound_0" state-id2="14"/>
77
78
79      <transition state-id1="14" event-id="?commit" state-
           id2="14"/>     <!-- Loop here -->
80      <transition state-id1="14" event-id="!stop_scream_1"
           state-id2="15"/>
81
82      <transition state-id1="15" event-id="?commit" state-
           id2="16"/>
83      <transition state-id1="16" event-id="?
           end_scream_sound_0" state-id2="17"/>
84      <transition state-id1="17" event-id="?commit" state-
           id2="18"/>
85      <transition state-id1="18" event-id="?
           abs_scream_sound_0" state-id2="19"/>
86      <transition state-id1="19" event-id="?commit" state-
           id2="20"/>
87
88      <!-- Reward an eventual caress with candy -->
89      <transition state-id1="20" event-id="!emit_caress_0"
           state-id2="21"/>
90
91      <transition state-id1="21" event-id="?beg_caress_1"
           state-id2="22"/>
92
93      <transition state-id1="22" event-id="?commit" state-
           id2="23"/>
94      <transition state-id1="23" event-id="?sta_caress_1"
           state-id2="24"/>
95      <transition state-id1="24" event-id="?commit" state-
           id2="24"/>
96
97      <transition state-id1="24" event-id="!stop_caress_0"
           state-id2="25"/>
98      <transition state-id1="25" event-id="?beg_candy_0"
           state-id2="26"/>
99
100
101     <transition state-id1="26" event-id="?end_caress_1"
           state-id2="27"/>
102     <transition state-id1="27" event-id="?commit" state-
           id2="28"/>
103     <transition state-id1="28" event-id="?abs_caress_1"
           state-id2="29"/>
104
```

```
105     <transition state-id1="29" event-id="?sta_candy_0"
            state-id2="30"/>
106     <transition state-id1="30" event-id="?commit" state-
            id2="31"/>
107     <transition state-id1="31" event-id="internal" state-
            id2="32"/>
108
109     <!-- Make sure that the child is no longer beating
            the dog. To do this, we
110           assume the use of the strong feasibility
                relation. -->
111     <transition state-id1="32" event-id="?end_candy_0"
            state-id2="33"/>
112     <transition state-id1="33" event-id="?commit" state-
            id2="34"/>
113     <transition state-id1="33" event-id="!emit_beat_0"
            state-id2="failure"/>
114     <transition state-id1="34" event-id="!stop_beat_0"
            state-id2="35"/>
115     <transition state-id1="34" event-id="!emit_beat_0"
            state-id2="failure"/>
116     <transition state-id1="35" event-id="?end_punch_1"
            state-id2="36"/>
117     <transition state-id1="35" event-id="!emit_beat_0"
            state-id2="failure"/>
118     <transition state-id1="36" event-id="?commit" state-
            id2="37"/>
119     <transition state-id1="36" event-id="!emit_beat_0"
            state-id2="failure"/>
120     <transition state-id1="37" event-id="?abs_punch_1"
            state-id2="38"/>
121     <transition state-id1="37" event-id="!emit_beat_0"
            state-id2="failure"/>
122     <transition state-id1="38" event-id="?commit" state-
            id2="39"/>
123     <transition state-id1="38" event-id="!emit_beat_0"
            state-id2="failure"/>
124     <transition state-id1="39" event-id="other" state-id2
            ="40"/>
125     <transition state-id1="39" event-id="!emit_beat_0"
            state-id2="failure"/>
126     <transition state-id1="40" event-id="other" state-id2
            ="41"/>
127     <transition state-id1="37" event-id="!emit_beat_0"
            state-id2="failure"/>
128     <transition state-id1="41" event-id="other" state-id2
            ="42"/>
129     <transition state-id1="41" event-id="!emit_beat_0"
            state-id2="failure"/>
```

```
130
131       <!-- If after all this trials no beating took place,
             and later a caress happens,
132            it means that the beating behaviour has been
                successfully eliminated -->
133       <transition state-id1="42" event-id="!emit_caress_0"
             state-id2="success"/>
134      </transitions>
135
136
137    </simulation-purpose-verification>
138
139  </experiment>
```

B.3.4 Result

```
Simulation Purpose Verification strategy (group = GROUP_0)
=====================================================
Result = SUCCESS
Running time = 4s

Run found:
    [depth = 0] State (in SP): initial
    [depth = 1] Events synch'd: <?beginning[agentId = 0]_[Stimulus type='neutral'],
                                !beginning[agentId = 0]_[Stimulus type='neutral']>;
                State annotations synch'd: <[LikesCandy],     [LikesCandy]>  State (in SP): 1
    [depth = 2] Events synch'd: <!emit[agentId = 0]_[Action type='beat'],
                                ?emit[agentId = 0]_[Action type='beat']>;
                State annotations synch'd: <[],     [LikesCandy]>  State (in SP): 3
    [depth = 3] Events synch'd: <?commit,
                                !commit>;
                State annotations synch'd: <[],     [LikesCandy]>  State (in SP): 4
    [depth = 4] Events synch'd: <?beginning[agentId = 1]_[Stimulus type='punch'],
                                !beginning[agentId = 1]_[Stimulus type='punch']>;
                State annotations synch'd: <[],     [LikesCandy]>  State (in SP): 5
    [depth = 5] Events synch'd: <?commit,
                                !commit>;
                State annotations synch'd: <[],     [LikesCandy]>  State (in SP): 6
    [depth = 6] Events synch'd: <?stable[agentId = 1]_[Stimulus type='punch'],
                                !stable[agentId = 1]_[Stimulus type='punch']>;
                State annotations synch'd: <[],     [LikesCandy]>  State (in SP): 7
    [depth = 7] Events synch'd: <?stable[agentId = 0]_[Stimulus type='neutral'],
                                !stable[agentId = 0]_[Stimulus type='neutral']>;
                State annotations synch'd: <[],     [LikesCandy]>  State (in SP): 8
    [depth = 8] Events synch'd: <?commit,
                                !commit>;
                State annotations synch'd: <[],     [LikesCandy]>  State (in SP): 9
    [depth = 9] Events synch'd: <?commit,
                                !commit>;
                State annotations synch'd: <[],     [LikesCandy]>  State (in SP): 10
    [depth = 10] Events synch'd: <!emit[agentId = 1]_[Action type='scream'],
                                 ?emit[agentId = 1]_[Action type='scream']>;
```

```
             State annotations synch'd: <[],     [LikesCandy]>  State (in SP): 11
[depth = 11] Events synch'd: <?beginning[agentId = 0]_[Stimulus type='scream_sound'],
                              !beginning[agentId = 0]_[Stimulus type='scream_sound']>;
             State annotations synch'd: <[],     [LikesCandy]>  State (in SP): 12
[depth = 12] Events synch'd: <?commit,
                              !commit>;
             State annotations synch'd: <[],     [LikesCandy]>  State (in SP): 13
[depth = 13] Events synch'd: <?stable[agentId = 0]_[Stimulus type='scream_sound'],
                              !stable[agentId = 0]_[Stimulus type='scream_sound']>;
             State annotations synch'd: <[],     [LikesCandy]>  State (in SP): 14
[depth = 14] Events synch'd: <?commit,
                              !commit>;
             State annotations synch'd: <[],     [LikesCandy]>  State (in SP): 14
[depth = 15] Events synch'd: <?commit,
                              !commit>;
             State annotations synch'd: <[],     [LikesCandy]>  State (in SP): 14
[depth = 16] Events synch'd: <?commit,
                              !commit>;
             State annotations synch'd: <[],     [LikesCandy]>  State (in SP): 14
[depth = 17] Events synch'd: <?commit,
                              !commit>;
             State annotations synch'd: <[],     [LikesCandy]>  State (in SP): 14
[depth = 18] Events synch'd: <?commit,
                              !commit>;
             State annotations synch'd: <[],     [LikesCandy]>  State (in SP): 14
[depth = 19] Events synch'd: <?commit,
                              !commit>;
             State annotations synch'd: <[],     [LikesCandy]>  State (in SP): 14
[depth = 20] Events synch'd: <?commit,
                              !commit>;
             State annotations synch'd: <[],     [LikesCandy]>  State (in SP): 14
[depth = 21] Events synch'd: <?commit,
                              !commit>;
             State annotations synch'd: <[],     [LikesCandy]>  State (in SP): 14
[depth = 22] Events synch'd: <?commit,
                              !commit>;
             State annotations synch'd: <[],     [LikesCandy]>  State (in SP): 14
[depth = 23] Events synch'd: <?commit,
                              !commit>;
             State annotations synch'd: <[],     [LikesCandy]>  State (in SP): 14
[depth = 24] Events synch'd: <?commit,
                              !commit>;
             State annotations synch'd: <[],     [LikesCandy]>  State (in SP): 14
[depth = 25] Events synch'd: <!stop[agentId = 1]_[Action type='scream'],
                              ?stop[agentId = 1]_[Action type='scream']>;
             State annotations synch'd: <[],     [LikesCandy]>  State (in SP): 15
[depth = 26] Events synch'd: <?commit,
                              !commit>;
             State annotations synch'd: <[],     [LikesCandy]>  State (in SP): 16
[depth = 27] Events synch'd: <?ending[agentId = 0]_[Stimulus type='scream_sound'],
                              !ending[agentId = 0]_[Stimulus type='scream_sound']>;
             State annotations synch'd: <[],     [LikesCandy]>  State (in SP): 17
[depth = 28] Events synch'd: <?commit,
                              !commit>;
```

```
             State annotations synch'd: <[],     [LikesCandy]>  State (in SP): 18
[depth = 29] Events synch'd: <?absent[agentId = 0]_[Stimulus type='scream_sound'],
                              !absent[agentId = 0]_[Stimulus type='scream_sound']>;
             State annotations synch'd: <[],     [LikesCandy]>  State (in SP): 19
[depth = 30] Events synch'd: <?commit,
                              !commit>;
             State annotations synch'd: <[],     [LikesCandy]>  State (in SP): 20
[depth = 31] Events synch'd: <!emit[agentId = 0]_[Action type='caress'],
                              ?emit[agentId = 0]_[Action type='caress']>;
             State annotations synch'd: <[],     [LikesCandy]>  State (in SP): 21
[depth = 32] Events synch'd: <?beginning[agentId = 1]_[Stimulus type='caress'],
                              !beginning[agentId = 1]_[Stimulus type='caress']>;
             State annotations synch'd: <[],     [LikesCandy]>  State (in SP): 22
[depth = 33] Events synch'd: <?commit,
                              !commit>;
             State annotations synch'd: <[],     [LikesCandy]>  State (in SP): 23
[depth = 34] Events synch'd: <?stable[agentId = 1]_[Stimulus type='caress'],
                              !stable[agentId = 1]_[Stimulus type='caress']>;
             State annotations synch'd: <[],     [LikesCandy]>  State (in SP): 24
[depth = 35] Events synch'd: <?commit,
                              !commit>;
             State annotations synch'd: <[],     [LikesCandy]>  State (in SP): 24
[depth = 36] Events synch'd: <?commit,
                              !commit>;
             State annotations synch'd: <[],     [LikesCandy]>  State (in SP): 24
[depth = 37] Events synch'd: <?commit,
                              !commit>;
             State annotations synch'd: <[],     [LikesCandy]>  State (in SP): 24
[depth = 38] Events synch'd: <!stop[agentId = 0]_[Action type='caress'],
                              ?stop[agentId = 0]_[Action type='caress']>;
             State annotations synch'd: <[],     [LikesCandy]>  State (in SP): 25
[depth = 39] Events synch'd: <?beginning[agentId = 0]_[Stimulus type='candy'],
                              !beginning[agentId = 0]_[Stimulus type='candy']>;
             State annotations synch'd: <[],     [LikesCandy]>  State (in SP): 26
[depth = 40] Events synch'd: <?ending[agentId = 1]_[Stimulus type='caress'],
                              !ending[agentId = 1]_[Stimulus type='caress']>;
             State annotations synch'd: <[],     [LikesCandy]>  State (in SP): 27
[depth = 41] Events synch'd: <?commit,
                              !commit>;
             State annotations synch'd: <[],     [LikesCandy]>  State (in SP): 28
[depth = 42] Events synch'd: <?absent[agentId = 1]_[Stimulus type='caress'],
                              !absent[agentId = 1]_[Stimulus type='caress']>;
             State annotations synch'd: <[],     [LikesCandy]>  State (in SP): 29
[depth = 43] Events synch'd: <?stable[agentId = 0]_[Stimulus type='candy'],
                              !stable[agentId = 0]_[Stimulus type='candy']>;
             State annotations synch'd: <[],     [LikesCandy]>  State (in SP): 30
[depth = 44] Events synch'd: <?commit,
                              !commit>;
             State annotations synch'd: <[],     [LikesCandy]>  State (in SP): 31
[depth = 45] Events synch'd: <internal,
                              TAU>;
             State annotations synch'd: <[],     [LikesCandy]>  State (in SP): 32
[depth = 46] Events synch'd: <?ending[agentId = 0]_[Stimulus type='candy'],
                              !ending[agentId = 0]_[Stimulus type='candy']>;
```

```
                State annotations synch'd: <[],     [LikesCandy]>  State (in SP): 33
    [depth = 47] Events synch'd: <?commit,
                                  !commit>;
                State annotations synch'd: <[],     [LikesCandy]>  State (in SP): 34
    [depth = 48] Events synch'd: <!stop[agentId = 0]_[Action type='beat'],
                                  ?stop[agentId = 0]_[Action type='beat']>;
                State annotations synch'd: <[],     [LikesCandy]>  State (in SP): 35
    [depth = 49] Events synch'd: <?ending[agentId = 1]_[Stimulus type='punch'],
                                  !ending[agentId = 1]_[Stimulus type='punch']>;
                State annotations synch'd: <[],     [LikesCandy]>  State (in SP): 36
    [depth = 50] Events synch'd: <?commit,
                                  !commit>;
                State annotations synch'd: <[],     [LikesCandy]>  State (in SP): 37
    [depth = 51] Events synch'd: <?absent[agentId = 1]_[Stimulus type='punch'],
                                  !absent[agentId = 1]_[Stimulus type='punch']>;
                State annotations synch'd: <[],     [LikesCandy]>  State (in SP): 38
    [depth = 52] Events synch'd: <?commit,
                                  !commit>;
                State annotations synch'd: <[],     [LikesCandy]>  State (in SP): 39
    [depth = 53] Events synch'd: <(*)other,
                                  TAU>;
                State annotations synch'd: <[],     [LikesCandy]>  State (in SP): 40
    [depth = 54] Events synch'd: <(*)other,
                                  !absent[agentId = 0]_[Stimulus type='candy']>;
                State annotations synch'd: <[],     [LikesCandy]>  State (in SP): 41
    [depth = 55] Events synch'd: <(*)other,
                                  !ending[agentId = 0]_[Stimulus type='neutral']>;
                State annotations synch'd: <[],     [LikesCandy]>  State (in SP): 42
    [depth = 56] Events synch'd: <!emit[agentId = 0]_[Action type='caress'],
                                  ?emit[agentId = 0]_[Action type='caress']>;
                State annotations synch'd: <[],     [LikesCandy]>  State (in SP): success



Result = SUCCESS
Running time = 4s


Finished.
```

B.4 Factory

B.4.1 Agents

Manager Type 1 One of the possible managers.

```
<?xml version="1.0" encoding="UTF-8"?>

<organism>

  <stimulation-subsystem>
```

```
6
7       <stimulation-parameters>
8         <stimulation-hints>
9           <pleasure-hints>
10          </pleasure-hints>
11          <pain-hints>
12          </pain-hints>
13        </stimulation-hints>
14
15        <stimuli>
16          <stimulus id="0" name="money" primary="true"
              utility="0.7" />
17        </stimuli>
18
19        <max-delay value="10" />
20
21      </stimulation-parameters>
22
23      <conditioning-parameters>
24          <c value="0.5"/>
25      </conditioning-parameters>
26
27      <stimulus-implication>
28
29          <!--
30          <cause id1="2" id2="3" correlation="1.0" />
31          -->
32
33      </stimulus-implication>
34    </stimulation-subsystem>
35
36
37    <responding-subsystem>
38
39      <actions>
40        <action id="0" name="order_work" base-level="0.0"
            operant="true" reflex="true" />
41      </actions>
42
43      <action-conflict/>
44
45      <operants>
46
47        <operant>
48          <antecedents>
49            <antecedent contingency="1.0"/>
50          </antecedents>
51          <action id="0"/>
52          <consequence id="0"/>
```

```
53        </operant>
54
55      </operants>
56
57
58
59      <reflexes/>
60
61    </responding-subsystem>
62
63    <drive-subsystem>
64      <drives>
65
66        <!-- Money is periodically needed -->
67        <drive>
68          <importance value="0.0"/>
69          <max-importance value="1.0"/>
70          <min-importance value="-1.0"/>
71          <desires>
72            <stimulus id="0" />
73          </desires>
74        </drive>
75
76      </drives>
77    </drive-subsystem>
78
79    <emotion-subsystem>
80      <anger status="INACTIVE" intensity="0.0" duration="0"
          />
81      <depression status="INACTIVE" intensity="0.0" duration
          ="0" />
82      <frustration status="INACTIVE" intensity="0.0"
          duration="0" />
83    </emotion-subsystem>
84
85  </organism>
```

Manager Type 2 The other manager, who is depressed.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <organism>
4
5    <stimulation-subsystem>
6
7      <stimulation-parameters>
8        <stimulation-hints>
9          <pleasure-hints>
10         </pleasure-hints>
```

```
11          <pain-hints>
12          </pain-hints>
13        </stimulation-hints>
14
15        <stimuli>
16          <stimulus id="0" name="money" primary="true"
               utility="0.5" />
17        </stimuli>
18
19        <max-delay value="10" />
20
21      </stimulation-parameters>
22
23      <conditioning-parameters>
24          <c value="0.5"/>
25      </conditioning-parameters>
26
27      <stimulus-implication/>
28
29    </stimulation-subsystem>
30
31    <responding-subsystem>
32
33      <actions>
34        <action id="0" name="order_work" base-level="0.0"
             operant="true" reflex="true" />
35      </actions>
36
37
38      <action-conflict>
39
40      </action-conflict>
41
42      <operants>
43
44        <operant>
45          <antecedents>
46            <antecedent contingency="1.0"/>
47          </antecedents>
48          <action id="0"/>
49          <consequence id="0"/>
50        </operant>
51
52      </operants>
53
54      <reflexes/>
55
56    </responding-subsystem>
57
```

```
58    <drive-subsystem>
59      <drives>
60
61        <!-- Money is periodically needed -->
62        <drive>
63          <importance value="0.0"/>
64          <max-importance value="1.0"/>
65          <min-importance value="-1.0"/>
66          <desires>
67            <stimulus id="0" />
68          </desires>
69        </drive>
70
71      </drives>
72    </drive-subsystem>
73
74
75    <emotion-subsystem>
76      <anger status="INACTIVE" intensity="0.0" duration="0"
          />
77      <depression status="ACTIVE" intensity="1.0" duration="
          20" /> <!-- Depressed -->
78      <frustration status="INACTIVE" intensity="0.0"
          duration="0" />
79    </emotion-subsystem>
80
81  </organism>
```

Worker Type 1 A worker that does not enjoy chatting.

```
1   <?xml version="1.0" encoding="UTF-8"?>
2
3
4   <organism>
5
6     <stimulation-subsystem>
7
8       <stimulation-parameters>
9         <stimulation-hints>
10          <pleasure-hints>
11          </pleasure-hints>
12          <pain-hints>
13          </pain-hints>
14        </stimulation-hints>
15
16        <stimuli>
17
18          <stimulus id="0" name="money" primary="true"
              utility="0.7" />
```

```
19
20          <stimulus id="1" name="work_product_0" primary="
               false" />
21          <stimulus id="2" name="work_product_1" primary="
               false" />
22          <stimulus id="3" name="work_product_2" primary="
               false" />
23          <stimulus id="4" name="work_product_3" primary="
               false" />
24
25          <stimulus id="5" name="conversation_1" primary="
               false"/>
26          <stimulus id="6" name="conversation_2" primary="
               false" />
27
28
29          <stimulus id="20" name="food" primary="true"
               utility="0.6"/>
30
31        </stimuli>
32
33
34        <max-delay value="10" />
35
36
37      </stimulation-parameters>
38
39      <conditioning-parameters>
40          <c value="0.5"/>
41      </conditioning-parameters>
42
43      <stimulus-implication>
44
45
46      </stimulus-implication>
47    </stimulation-subsystem>
48
49
50
51    <responding-subsystem>
52
53      <actions>
54
55        <action id="0" name="work_1" base-level="0.0"
             operant="true" reflex="true" />
56        <action id="1" name="work_2" base-level="0.0"
             operant="true" reflex="true" />
57        <action id="2" name="work_3" base-level="0.0"
             operant="true" reflex="true" />
```

```
58
59        <action id="3" name="chat_1" base-level="0.3"
             operant="true" reflex="true" />
60        <action id="4" name="chat_2" base-level="0.3"
             operant="true" reflex="true" />
61        <action id="5" name="chat_3" base-level="0.3"
             operant="true" reflex="true" />
62
63
64      </actions>
65
66
67      <action-conflict>
68
69        <!-- Either one works or one chats -->
70        <conflict id1="0" id2="3"/>
71        <conflict id1="0" id2="4"/>
72        <conflict id1="0" id2="5"/>
73
74        <conflict id1="1" id2="3"/>
75        <conflict id1="1" id2="4"/>
76        <conflict id1="1" id2="5"/>
77
78        <conflict id1="2" id2="3"/>
79        <conflict id1="2" id2="4"/>
80        <conflict id1="2" id2="5"/>
81
82      </action-conflict>
83
84      <operants>
85
86
87        <operant>
88          <antecedents>
89            <antecedent contingency="1.0">
90              <stimulus id="1"/>
91            </antecedent>
92          </antecedents>
93          <action id="0"/>
94          <consequence id="0"/>
95        </operant>
96
97        <operant>
98          <antecedents>
99            <antecedent contingency="1.0">
100             <stimulus id="2"/>
101           </antecedent>
102         </antecedents>
103         <action id="1"/>
```

```
104         <consequence id="0"/>
105       </operant>
106
107
108       <operant>
109         <antecedents>
110           <antecedent contingency="1.0">
111             <stimulus id="3"/>
112           </antecedent>
113         </antecedents>
114         <action id="2"/>
115         <consequence id="0"/>
116       </operant>
117
118     </operants>
119
120
121
122     <reflexes/>
123
124   </responding-subsystem>
125
126   <drive-subsystem>
127     <drives>
128
129       <!-- Money is periodically needed -->
130       <drive>
131         <importance value="0.0"/>
132         <max-importance value="1.0"/>
133         <min-importance value="-1.0"/>
134         <desires>
135           <stimulus id="0" />
136         </desires>
137       </drive>
138
139     </drives>
140   </drive-subsystem>
141
142
143   <emotion-subsystem>
144     <anger status="INACTIVE" intensity="0.0" duration="0"
          />
145     <depression status="INACTIVE" intensity="0.0" duration
          ="0" />
146     <frustration status="INACTIVE" intensity="0.0"
          duration="0" />
147   </emotion-subsystem>
148
149 </organism>
```

Worker Type 2 A worker that enjoys chatting very much.

```
<?xml version="1.0" encoding="UTF-8"?>

<organism>

  <stimulation-subsystem>

    <stimulation-parameters>
      <stimulation-hints>
        <pleasure-hints>
        </pleasure-hints>
        <pain-hints>
        </pain-hints>
      </stimulation-hints>

      <stimuli>

        <stimulus id="0" name="money" primary="true"
           utility="0.5" />

        <stimulus id="1" name="work_product_0" primary="
           false" />
        <stimulus id="2" name="work_product_1" primary="
           false" />
        <stimulus id="3" name="work_product_2" primary="
           false" />
        <stimulus id="4" name="work_product_3" primary="
           false" />

        <stimulus id="5" name="conversation_1" primary="
           true" utility="0.8" />
        <stimulus id="6" name="conversation_2" primary="
           true" utility="0.8" />

        <stimulus id="20" name="food" primary="true"
           utility="0.5"/>

      </stimuli>

      <max-delay value="10" />

    </stimulation-parameters>

    <conditioning-parameters>
        <c value="0.5"/>
    </conditioning-parameters>

    <stimulus-implication>

```

```
41
42      </stimulus-implication>
43    </stimulation-subsystem>
44
45
46
47    <responding-subsystem>
48
49      <actions>
50
51        <action id="0" name="work_1" base-level="0.0"
             operant="true" reflex="true" />
52        <action id="1" name="work_2" base-level="0.0"
             operant="true" reflex="true" />
53        <action id="2" name="work_3" base-level="0.0"
             operant="true" reflex="true" />
54
55        <action id="3" name="chat_1" base-level="0.3"
             operant="true" reflex="true" />
56        <action id="4" name="chat_2" base-level="0.3"
             operant="true" reflex="true" />
57        <action id="5" name="chat_3" base-level="0.3"
             operant="true" reflex="true" />
58
59      </actions>
60
61
62      <action-conflict>
63
64        <!-- Either one works or one chats -->
65        <conflict id1="0" id2="3"/>
66        <conflict id1="0" id2="4"/>
67        <conflict id1="0" id2="5"/>
68
69        <conflict id1="1" id2="3"/>
70        <conflict id1="1" id2="4"/>
71        <conflict id1="1" id2="5"/>
72
73        <conflict id1="2" id2="3"/>
74        <conflict id1="2" id2="4"/>
75        <conflict id1="2" id2="5"/>
76
77      </action-conflict>
78
79      <operants>
80
81        <!--
82          Each work product requires a different action,
               which is rewarded with money
```

```
83        -->
84
85        <operant>
86          <antecedents>
87            <antecedent contingency="1.0">
88              <stimulus id="1"/>
89            </antecedent>
90          </antecedents>
91          <action id="0"/>
92          <consequence id="0"/>
93        </operant>
94
95        <operant>
96          <antecedents>
97            <antecedent contingency="1.0">
98              <stimulus id="2"/>
99            </antecedent>
100         </antecedents>
101         <action id="1"/>
102         <consequence id="0"/>
103       </operant>
104
105
106       <operant>
107         <antecedents>
108           <antecedent contingency="1.0">
109             <stimulus id="3"/>
110           </antecedent>
111         </antecedents>
112         <action id="2"/>
113         <consequence id="0"/>
114       </operant>
115
116
117
118
119      <!-- When dealing with a particular kind of work,
            this agent knows that
120          a good conversation is possible. -->
121       <operant>
122         <antecedents>
123           <antecedent contingency="1.0">
124             <stimulus id="2"/>
125           </antecedent>
126         </antecedents>
127         <action id="3"/>      <!-- chat_1 -->
128         <consequence id="5"/> <!-- conversation_1 -->
129       </operant>
130
```

```
131     </operants>
132
133
134
135     <reflexes/>
136
137   </responding-subsystem>
138
139
140   <drive-subsystem>
141     <drives>
142
143       <!-- Money is periodically needed -->
144       <drive>
145         <importance value="0.0"/>
146         <max-importance value="1.0"/>
147         <min-importance value="-1.0"/>
148         <desires>
149           <stimulus id="0" />
150         </desires>
151       </drive>
152
153     </drives>
154   </drive-subsystem>
155
156
157   <emotion-subsystem>
158     <anger status="INACTIVE" intensity="0.0" duration="0"
          />
159     <depression status="INACTIVE" intensity="0.0" duration
          ="0" />
160     <frustration status="INACTIVE" intensity="0.0"
          duration="0" />
161   </emotion-subsystem>
162
163 </organism>
```

B.4.2 Scenario

```
1 <?xml version="1.0"?>
2 <scenario name="EMMAS Factory Example"
3               description="A factory where workers should
                  maximize production.">
4
5
6   <!-- Managers -->
7   <agent component-id="organism.OrganismComponent" id="0"
      name="Manager 1">
```

```
8       <initializer file="manager1.agent.xml"/>
9     </agent>
10
11    <agent component-id="organism.OrganismComponent" id="1"
        name="Manager 2">
12      <initializer file="manager2.agent.xml"/>
13    </agent>
14
15    <!-- Workers -->
16    <agent component-id="organism.OrganismComponent" id="11"
        name="Worker 1">
17      <initializer file="worker2.agent.xml"/>
18    </agent>
19
20    <agent component-id="organism.OrganismComponent" id="12"
        name="Worker 2">
21      <initializer file="worker2.agent.xml"/>
22    </agent>
23
24      <agent component-id="organism.OrganismComponent" id="
          13" name="Worker 3">
25      <initializer file="worker1.agent.xml"/>
26    </agent>
27
28
29    <emmas>
30
31      <action-transformers>
32        <!-- Initially, no connections. (They'll be built
            later, see the behaviors below.) -->
33      </action-transformers>
34
35      <behaviors>
36
37
38        <!-- When the product is finished by some agent (
            through the 'work3' action),
39              everyone is rewarded.  -->
40
41        <behavior>
42          <environment-response agent-id="11" action="work_3
              ">
43            <parallel>
44              <stimulate stimulus="money" agent-id="11" />
45              <stimulate stimulus="money" agent-id="12" />
46              <stimulate stimulus="money" agent-id="13" />
47              <stimulate stimulus="money" agent-id="0" />
48              <stimulate stimulus="money" agent-id="1" />
49            </parallel>
```

```
50          </environment-response>
51        </behavior>
52
53        <behavior>
54          <environment-response agent-id="12" action="work_3
              ">
55            <parallel>
56              <stimulate stimulus="money" agent-id="11" />
57              <stimulate stimulus="money" agent-id="12" />
58              <stimulate stimulus="money" agent-id="13" />
59              <stimulate stimulus="money" agent-id="0" />
60              <stimulate stimulus="money" agent-id="1" />
61            </parallel>
62          </environment-response>
63        </behavior>
64
65        <behavior>
66          <environment-response agent-id="13" action="work_3
              ">
67            <parallel>
68              <stimulate stimulus="money" agent-id="11" />
69              <stimulate stimulus="money" agent-id="12" />
70              <stimulate stimulus="money" agent-id="13" />
71              <stimulate stimulus="money" agent-id="0" />
72              <stimulate stimulus="money" agent-id="1" />
73            </parallel>
74          </environment-response>
75        </behavior>
76
77
78        <behavior>
79
80          <!-- There are several ways to setup the company.
              -->
81
82          <choice>
83
84            <sequential-composition>
85              <choice>
86                <create agent-id1="0" action="order_work"
                    stimulus="work_product_0" agent-id2 = "13
                    " />
87                <create agent-id1="1" action="order_work"
                    stimulus="work_product_0" agent-id2 = "13
                    " />
88              </choice>
89
90              <create agent-id1="13" action="work_1"
                  stimulus="work_product_1" agent-id2 = "12"
```

```
                />
91            <create agent-id1="12" action="work_2"
                stimulus="work_product_2" agent-id2 = "11"
                />
92
93            <create agent-id1="13" action="chat_1"
                stimulus="conversation_1" agent-id2 = "12"
                />
94            <create agent-id1="12" action="chat_1"
                stimulus="conversation_1" agent-id2 = "13"
                />
95
96            <create agent-id1="12" action="chat_1"
                stimulus="conversation_1" agent-id2 = "11"
                />
97            <create agent-id1="11" action="chat_1"
                stimulus="conversation_1" agent-id2 = "12"
                />
98
99          </sequential-composition>
100
101
102         <sequential-composition>
103           <choice>
104             <create agent-id1="0" action="order_work"
                  stimulus="work_product_0" agent-id2 = "11
                  " />
105             <create agent-id1="1" action="order_work"
                  stimulus="work_product_0" agent-id2 = "11
                  " />
106           </choice>
107
108           <create agent-id1="11" action="work_1"
                stimulus="work_product_1" agent-id2 = "12"
                />
109           <create agent-id1="12" action="work_2"
                stimulus="work_product_2" agent-id2 = "13"
                />
110
111           <create agent-id1="11" action="chat_1"
                stimulus="conversation_1" agent-id2 = "12"
                />
112           <create agent-id1="12" action="chat_1"
                stimulus="conversation_1" agent-id2 = "11"
                />
113
114           <create agent-id1="12" action="chat_1"
                stimulus="conversation_1" agent-id2 = "13"
                />
```

```
115             <create agent-id1="13" action="chat_1"
                   stimulus="conversation_1" agent-id2 = "12"
                   />
116
117           </sequential-composition>
118
119
120           <sequential-composition>
121
122             <choice>
123               <create agent-id1="0" action="order_work"
                     stimulus="work_product_0" agent-id2 = "11
                     " />
124               <create agent-id1="1" action="order_work"
                     stimulus="work_product_0" agent-id2 = "11
                     " />
125             </choice>
126
127             <create agent-id1="11" action="work_1"
                   stimulus="work_product_1" agent-id2 = "13"
                   />
128             <create agent-id1="13" action="work_2"
                   stimulus="work_product_2" agent-id2 = "12"
                   />
129
130             <create agent-id1="11" action="chat_1"
                   stimulus="conversation_1" agent-id2 = "13"
                   />
131             <create agent-id1="13" action="chat_1"
                   stimulus="conversation_1" agent-id2 = "11"
                   />
132
133             <create agent-id1="13" action="chat_1"
                   stimulus="conversation_1" agent-id2 = "12"
                   />
134             <create agent-id1="12" action="chat_1"
                   stimulus="conversation_1" agent-id2 = "13"
                   />
135
136           </sequential-composition>
137
138
139
140           <sequential-composition>
141             <choice>
142               <create agent-id1="0" action="order_work"
                     stimulus="work_product_0" agent-id2 = "12
                     " />
```

```
143                <create agent-id1="1" action="order_work"
                      stimulus="work_product_0" agent-id2 = "12
                      " />
144              </choice>
145
146              <create agent-id1="12" action="work_1"
                    stimulus="work_product_1" agent-id2 = "11"
                    />
147              <create agent-id1="11" action="work_2"
                    stimulus="work_product_2" agent-id2 = "13"
                    />
148
149              <create agent-id1="12" action="chat_1"
                    stimulus="conversation_1" agent-id2 = "11"
                    />
150              <create agent-id1="11" action="chat_1"
                    stimulus="conversation_1" agent-id2 = "12"
                    />
151
152              <create agent-id1="11" action="chat_1"
                    stimulus="conversation_1" agent-id2 = "13"
                    />
153              <create agent-id1="13" action="chat_1"
                    stimulus="conversation_1" agent-id2 = "11"
                    />
154            </sequential-composition>
155
156
157            <sequential-composition>
158              <choice>
159                <create agent-id1="0" action="order_work"
                      stimulus="work_product_0" agent-id2 = "12
                      " />
160                <create agent-id1="1" action="order_work"
                      stimulus="work_product_0" agent-id2 = "12
                      " />
161              </choice>
162
163              <create agent-id1="12" action="work_1"
                    stimulus="work_product_1" agent-id2 = "13"
                    />
164              <create agent-id1="13" action="work_2"
                    stimulus="work_product_2" agent-id2 = "11"
                    />
165
166              <create agent-id1="12" action="chat_1"
                    stimulus="conversation_1" agent-id2 = "13"
                    />
```

```
167             <create agent-id1="13" action="chat_1"
                   stimulus="conversation_1" agent-id2 = "12"
                   />
168
169             <create agent-id1="13" action="chat_1"
                   stimulus="conversation_1" agent-id2 = "11"
                   />
170             <create agent-id1="11" action="chat_1"
                   stimulus="conversation_1" agent-id2 = "13"
                   />
171
172           </sequential-composition>
173
174
175           <sequential-composition>
176             <choice>
177               <create agent-id1="0" action="order_work"
                     stimulus="work_product_0" agent-id2 = "13
                     " />
178               <create agent-id1="1" action="order_work"
                     stimulus="work_product_0" agent-id2 = "13
                     " />
179             </choice>
180
181             <create agent-id1="13" action="work_1"
                   stimulus="work_product_1" agent-id2 = "11"
                   />
182             <create agent-id1="11" action="work_2"
                   stimulus="work_product_2" agent-id2 = "12"
                   />
183
184             <create agent-id1="13" action="chat_1"
                   stimulus="conversation_1" agent-id2 = "11"
                   />
185             <create agent-id1="11" action="chat_1"
                   stimulus="conversation_1" agent-id2 = "13"
                   />
186
187             <create agent-id1="11" action="chat_1"
                   stimulus="conversation_1" agent-id2 = "12"
                   />
188             <create agent-id1="12" action="chat_1"
                   stimulus="conversation_1" agent-id2 = "11"
                   />
189
190           </sequential-composition>
191
192         </choice>
193       </behavior>
```

```
194
195     </behaviors>
196
197   </emmas>
198 </scenario>
```

B.4.3 Experiment

```
1  <?xml version="1.0"?>
2
3  <experiment name="Factory assembly" description="Seeks the
       best factory configuration." >
4
5    <!-- A simulation purpose verification  -->
6    <simulation-purpose-verification relation="feasibility">
7
8      <states>
9       <state id="initial"/>  <state id="1"/>  <state id="2"
           />  <state id="3"/>  <state id="4"/>  <state id="5
           "/>
10      <state id="6"/>   <state id="7"/>   <state id="8"/>
            <state id="9"/>   <state id="10"/>  <state id="
           11"/>
11      <state id="12"/>  <state id="13"/>  <state id="14"/>
            <state id="15"/>  <state id="16"/>  <state id="17
           "/>
12      <state id="18"/>  <state id="19"/>  <state id="20"/>
            <state id="21"/>  <state id="22"/>  <state id="23
           "/>
13      <state id="24"/>  <state id="25"/>  <state id="26"/>
            <state id="27"/>  <state id="28"/>  <state id="29
           "/>
14      <state id="30"/>  <state id="31"/>  <state id="32"/>
            <state id="33"/>  <state id="34"/>  <state id="35
           "/>
15      <state id="36"/>  <state id="37"/>  <state id="38"/>
            <state id="39"/>  <state id="40"/>
16     </states>
17
18     <events>
19       <emmas-event id="?beg_workproduct0_ag11" type="input
            " name="beginning" stimulus="work_product_0"
            agent-id="11"/>
20       <emmas-event id="?beg_workproduct1_ag11" type="input
            " name="beginning" stimulus="work_product_1"
            agent-id="11"/>
21       <emmas-event id="?beg_workproduct2_ag11" type="input
            " name="beginning" stimulus="work_product_2"
            agent-id="11"/>
```

```
22        <emmas-event id="?beg_workproduct0_ag12" type="input
             " name="beginning" stimulus="work_product_0"
             agent-id="12"/>
23        <emmas-event id="?beg_workproduct1_ag12" type="input
             " name="beginning" stimulus="work_product_1"
             agent-id="12"/>
24        <emmas-event id="?beg_workproduct2_ag12" type="input
             " name="beginning" stimulus="work_product_2"
             agent-id="12"/>
25        <emmas-event id="?beg_workproduct0_ag13" type="input
             " name="beginning" stimulus="work_product_0"
             agent-id="13"/>
26        <emmas-event id="?beg_workproduct1_ag13" type="input
             " name="beginning" stimulus="work_product_1"
             agent-id="13"/>
27        <emmas-event id="?beg_workproduct2_ag13" type="input
             " name="beginning" stimulus="work_product_2"
             agent-id="13"/>
28
29        <emmas-event id="!emit_orderwork0_ag0" type="output"
              name="emit" action="order_work" agent-id="0"/>
30        <emmas-event id="!emit_orderwork0_ag1" type="output"
              name="emit" action="order_work" agent-id="1"/>
31
32        <emmas-event id="!emit_work3_ag11" type="output"
             name="emit" action="work_3" agent-id="11"/>
33        <emmas-event id="!emit_work3_ag12" type="output"
             name="emit" action="work_3" agent-id="12"/>
34        <emmas-event id="!emit_work3_ag13" type="output"
             name="emit" action="work_3" agent-id="13"/>
35      </events>
36
37
38
39      <transitions>
40
41      <!-- There are several ways of building the company's
           assembly line.
42           The following transitions specify that the
                assembly line must
43           be built and that it must go through a number of
                phases.
44           The exact events that accomplish this will be
                found during verification.
45
46           For the sake of illustration, some things are
                left concrete ("do this event"), and others
47           abstract ("do any event such that X"). The more
                concrete, the easier
```

```
48          it is to perform the verification (but the harder
                it is to write the
49          simulation purpose).
50      -->
51
52
53
54      <!-- Establish a network configuration. Each new
            connection generates an 'internal' event,
55            so we look for as many of them as necessary. -->
56      <transition state-id1="initial" event-id="internal"
            state-id2="19"/>
57      <transition state-id1="19" event-id="internal" state-
            id2="19"/>
58
59
60      <!-- A manager must give the initial work order -->
61      <transition state-id1="19" event-id="!
            emit_orderwork0_ag0" state-id2="20"/>
62      <transition state-id1="19" event-id="!
            emit_orderwork0_ag1" state-id2="20"/>
63
64
65      <!-- The order arrives to some agent. -->
66      <transition state-id1="20" event-id="?
            beg_workproduct0_ag11" state-id2="21"/>
67      <transition state-id1="20" event-id="?
            beg_workproduct0_ag12" state-id2="21"/>
68      <transition state-id1="20" event-id="?
            beg_workproduct0_ag13" state-id2="21"/>
69
70      <!-- Some worker will be the first in the assembly
            line. Here we abstract which one by
71            merely specifying the 'other' event. In a
                feasible trace, this
72            will synchronize with some 'emit' of some agent
                (e.g., '!emit_work1_ag11'). -->
73      <transition state-id1="21" event-id="?commit" state-
            id2="22"/>
74      <transition state-id1="22" event-id="other" state-id2
            ="26"/>
75      <transition state-id1="22" event-id="?commit" state-
            id2="23"/>
76      <transition state-id1="23" event-id="other" state-id2
            ="26"/>
77      <transition state-id1="23" event-id="?commit" state-
            id2="24"/>
78      <transition state-id1="24" event-id="other" state-id2
            ="26"/>
```

```
79      <transition state-id1="24" event-id="?commit" state-
           id2="25"/>
80      <transition state-id1="25" event-id="other" state-id2
           ="26"/>
81
82      <!-- The partial product gets to the next worker. -->
83      <transition state-id1="26" event-id="?
           beg_workproduct1_ag12" state-id2="27"/>
84      <transition state-id1="26" event-id="?
           beg_workproduct1_ag11" state-id2="27"/>
85      <transition state-id1="26" event-id="?
           beg_workproduct1_ag13" state-id2="27"/>
86
87      <!-- Some worker will be the second in the assembly
           line. Here we abstract which one by
88          merely specifying the 'other' event. In a
               feasible trace, this
89          will synchronize with some 'emit' of some agent.
               -->
90      <transition state-id1="27" event-id="?commit" state-
           id2="28"/>
91      <transition state-id1="28" event-id="other" state-id2
           ="32"/>
92      <transition state-id1="28" event-id="?commit" state-
           id2="29"/>
93      <transition state-id1="29" event-id="other" state-id2
           ="32"/>
94      <transition state-id1="29" event-id="?commit" state-
           id2="30"/>
95      <transition state-id1="30" event-id="other" state-id2
           ="32"/>
96      <transition state-id1="30" event-id="?commit" state-
           id2="31"/>
97      <transition state-id1="31" event-id="other" state-id2
           ="32"/>
98
99      <!-- The partial product gets to the next worker. -->
100     <transition state-id1="32" event-id="?
           beg_workproduct2_ag13" state-id2="33"/>
101     <transition state-id1="32" event-id="?
           beg_workproduct2_ag11" state-id2="33"/>
102     <transition state-id1="32" event-id="?
           beg_workproduct2_ag12" state-id2="33"/>
103
104     <!-- Some worker will be the third (and last) in the
           assembly line. Here we
105         so not abstract which event corresponds to this.
               Notice the difference
106         w.r.t. the previous assembly line positions. -->
```

```
107        <transition state-id1="33" event-id="?commit" state-
               id2="34"/>
108        <transition state-id1="34" event-id="!emit_work3_ag13
               " state-id2="success"/>
109        <transition state-id1="34" event-id="!emit_work3_ag11
               " state-id2="success"/>
110        <transition state-id1="34" event-id="!emit_work3_ag12
               " state-id2="success"/>
111        <transition state-id1="34" event-id="?commit" state-
               id2="35"/>
112        <transition state-id1="35" event-id="!emit_work3_ag13
               " state-id2="success"/>
113        <transition state-id1="34" event-id="!emit_work3_ag11
               " state-id2="success"/>
114        <transition state-id1="34" event-id="!emit_work3_ag12
               " state-id2="success"/>
115        <transition state-id1="35" event-id="?commit" state-
               id2="36"/>
116        <transition state-id1="36" event-id="!emit_work3_ag13
               " state-id2="success"/>
117        <transition state-id1="36" event-id="!emit_work3_ag11
               " state-id2="success"/>
118        <transition state-id1="36" event-id="!emit_work3_ag12
               " state-id2="success"/>
119        <transition state-id1="36" event-id="?commit" state-
               id2="37"/>
120        <transition state-id1="37" event-id="!emit_work3_ag13
               " state-id2="success"/>
121        <transition state-id1="37" event-id="!emit_work3_ag11
               " state-id2="success"/>
122        <transition state-id1="37" event-id="!emit_work3_ag12
               " state-id2="success"/>
123
124      </transitions>
125
126    </simulation-purpose-verification>
127
128  </experiment>
```

B.4.4 Result

Weak feasibility Result of **weak feasibility** verification.

```
Simulation Purpose Verification strategy
=====================================================
Result = SUCCESS
Running time = 47s

Run found:
```

```
[depth = 0] State (in SP): initial
[depth = 1] Events synch'd: <internal,
                             TAU>;
            State annotations synch'd: <[],     []>  State (in SP): 19
[depth = 2] Events synch'd: <internal,
                             TAU>;
            State annotations synch'd: <[],     []>  State (in SP): 19
[depth = 3] Events synch'd: <internal,
                             TAU>;
            State annotations synch'd: <[],     []>  State (in SP): 19
[depth = 4] Events synch'd: <internal,
                             TAU>;
            State annotations synch'd: <[],     []>  State (in SP): 19
[depth = 5] Events synch'd: <internal,
                             TAU>;
            State annotations synch'd: <[],     []>  State (in SP): 19
[depth = 6] Events synch'd: <internal,
                             TAU>;
            State annotations synch'd: <[],     []>  State (in SP): 19
[depth = 7] Events synch'd: <!emit[agentId = 0]_[Action type='order_work'],
                             ?emit[agentId = 0]_[Action type='order_work']>;
            State annotations synch'd: <[],     []>  State (in SP): 20
[depth = 8] Events synch'd: <?beginning[agentId = 11]_[Stimulus type='work_product_0'],
                             !beginning[agentId = 11]_[Stimulus type='work_product_0']>;
            State annotations synch'd: <[],     []>  State (in SP): 21
[depth = 9] Events synch'd: <?commit,
                             !commit>;
            State annotations synch'd: <[],     []>  State (in SP): 22
[depth = 10] Events synch'd: <?commit,
                              !commit>;
             State annotations synch'd: <[],     []>  State (in SP): 23
[depth = 11] Events synch'd: <?commit,
                              !commit>;
             State annotations synch'd: <[],     []>  State (in SP): 24
[depth = 12] Events synch'd: <(*)other,
                              ?emit[agentId = 11]_[Action type='work_1']>;
             State annotations synch'd: <[],     []>  State (in SP): 26
[depth = 13] Events synch'd: <?beginning[agentId = 13]_[Stimulus type='work_product_1'],
                              !beginning[agentId = 13]_[Stimulus type='work_product_1']>;
             State annotations synch'd: <[],     []>  State (in SP): 27
[depth = 14] Events synch'd: <?commit,
                              !commit>;
             State annotations synch'd: <[],     []>  State (in SP): 28
[depth = 15] Events synch'd: <?commit,
                              !commit>;
             State annotations synch'd: <[],     []>  State (in SP): 29
[depth = 16] Events synch'd: <?commit,
                              !commit>;
             State annotations synch'd: <[],     []>  State (in SP): 30
[depth = 17] Events synch'd: <(*)other,
                              ?emit[agentId = 13]_[Action type='work_2']>;
             State annotations synch'd: <[],     []>  State (in SP): 32
[depth = 18] Events synch'd: <?beginning[agentId = 12]_[Stimulus type='work_product_2'],
                              !beginning[agentId = 12]_[Stimulus type='work_product_2']>;
```

```
                State annotations synch'd: <[],     []>  State (in SP): 33
    [depth = 19] Events synch'd: <?commit,
                                  !commit>;
                State annotations synch'd: <[],     []>  State (in SP): 34
    [depth = 20] Events synch'd: <?commit,
                                  !commit>;
                State annotations synch'd: <[],     []>  State (in SP): 35
    [depth = 21] Events synch'd: <?commit,
                                  !commit>;
                State annotations synch'd: <[],     []>  State (in SP): 36
    [depth = 22] Events synch'd: <!emit[agentId = 12]_[Action type='work_3'],
                                  ?emit[agentId = 12]_[Action type='work_3']>;
                State annotations synch'd: <[],     []>  State (in SP): success



Result = SUCCESS
Running time = 47s


Finished.
```

Certainty Result of **certainty** verification.

```
Simulation Purpose Verification strategy
=======================================================
Result = FAILURE
Running time = 1s

Run found:
    [depth = 0] State (in SP): initial
    [depth = 1] Events synch'd: <internal,
                                 TAU>;
                State annotations synch'd: <[],     []>  State (in SP): 19



Result = FAILURE
Running time = 1s


Finished.
```

B.5 School Children

B.5.1 Agents

Teacher The school teacher.

```
<?xml version="1.0" encoding="UTF-8"?>

<organism>

  <stimulation-subsystem>

    <stimulation-parameters>
      <stimulation-hints>
        <pleasure-hints>
        </pleasure-hints>
        <pain-hints>
        </pain-hints>
      </stimulation-hints>

      <stimuli>

        <stimulus id="0" name="money" primary="true"
           utility="0.7" />

        <stimulus id="1" name="homework_1" primary="false"
            />
        <stimulus id="2" name="homework_2" primary="false"
            />
        <stimulus id="3" name="homework_3" primary="false"
            />

        <stimulus id="4" name="see_annoying_1" primary="
           false" />
        <stimulus id="5" name="see_annoying_2" primary="
           false" />
        <stimulus id="6" name="see_annoying_3" primary="
           false" />

      </stimuli>

      <max-delay value="10" />

    </stimulation-parameters>

    <conditioning-parameters>
        <c value="0.5"/>
    </conditioning-parameters>

    <stimulus-implication/>

  </stimulation-subsystem>

  <responding-subsystem>

```

```
43      <actions>
44        <action id="0" name="assign_homework" base-level="
             0.6" operant="true" reflex="false" />
45
46        <action id="1" name="reward_1" base-level="0.0"
             operant="true" reflex="true" />
47        <action id="2" name="reward_2" base-level="0.0"
             operant="true" reflex="true" />
48        <action id="3" name="reward_3" base-level="0.0"
             operant="true" reflex="true" />
49
50        <action id="4" name="punish_1" base-level="0.0"
             operant="true" reflex="true" />
51        <action id="5" name="punish_2" base-level="0.0"
             operant="true" reflex="true" />
52        <action id="6" name="punish_3" base-level="0.0"
             operant="true" reflex="true" />
53      </actions>
54
55      <action-conflict/>
56
57      <operants>
58
59        <!-- The teacher has learned she will be paid when
             she performs her job properly. -->
60
61        <operant>
62          <antecedents>
63            <antecedent contingency="1.0"/>
64          </antecedents>
65          <action id="0"/>
66          <consequence id="0"/>
67        </operant>
68
69        <operant>
70          <antecedents>
71            <antecedent contingency="1.0">
72              <stimulus id="1"/>
73            </antecedent>
74          </antecedents>
75          <action id="1"/>
76          <consequence id="0"/>
77        </operant>
78
79
80        <operant>
81          <antecedents>
82            <antecedent contingency="1.0">
83              <stimulus id="2"/>
```

```
84            </antecedent>
85          </antecedents>
86          <action id="2"/>
87          <consequence id="0"/>
88        </operant>
89
90        <operant>
91          <antecedents>
92            <antecedent contingency="1.0">
93              <stimulus id="3"/>
94            </antecedent>
95          </antecedents>
96          <action id="3"/>
97          <consequence id="0"/>
98        </operant>
99
100
101       <operant>
102         <antecedents>
103           <antecedent contingency="1.0">
104             <stimulus id="4"/>
105           </antecedent>
106         </antecedents>
107         <action id="4"/>
108         <consequence id="0"/>
109       </operant>
110
111
112       <operant>
113         <antecedents>
114           <antecedent contingency="1.0">
115             <stimulus id="5"/>
116           </antecedent>
117         </antecedents>
118         <action id="5"/>
119         <consequence id="0"/>
120       </operant>
121
122
123       <operant>
124         <antecedents>
125           <antecedent contingency="1.0">
126             <stimulus id="6"/>
127           </antecedent>
128         </antecedents>
129         <action id="6"/>
130         <consequence id="0"/>
131       </operant>
132     </operants>
```

```
133
134     <reflexes/>
135   </responding-subsystem>
136
137   <drive-subsystem>
138     <drives>
139
140       <!-- Money is periodically needed -->
141       <drive>
142         <importance value="0.0"/>
143         <max-importance value="1.0"/>
144         <min-importance value="-1.0"/>
145         <desires>
146           <stimulus id="0" />
147         </desires>
148       </drive>
149
150     </drives>
151   </drive-subsystem>
152
153   <emotion-subsystem>
154     <anger status="INACTIVE" intensity="0.0" duration="0"
          />
155     <depression status="INACTIVE" intensity="0.0" duration
          ="0" />
156     <frustration status="INACTIVE" intensity="0.0"
          duration="0" />
157   </emotion-subsystem>
158
159 </organism>
```

Student Parameters for the students.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <organism>
4
5    <stimulation-subsystem>
6
7      <stimulation-parameters>
8        <stimulation-hints>
9          <pleasure-hints>
10         </pleasure-hints>
11         <pain-hints>
12         </pain-hints>
13       </stimulation-hints>
14
15       <stimuli>
```

```
16          <stimulus id="0" name="prize" primary="true"
               utility="0.5" />
17          <stimulus id="1" name="disapproval" primary="true"
                utility="-0.2" />
18          <stimulus id="2" name="homework" primary="false" /
               >
19          <stimulus id="3" name="provocation" primary="true"
                utility="-0.4" />
20          <stimulus id="4" name="information" primary="true"
                utility="0.8" />
21          <stimulus id="5" name="tv" primary="true" utility=
               "0.6" />
22          <stimulus id="6" name="cry_sound" primary="true"
               utility="0.3" />
23          <stimulus id="7" name="neutral" primary="true"
               utility="0.0" />
24        </stimuli>
25
26        <max-delay value="10" />
27
28      </stimulation-parameters>
29
30      <conditioning-parameters>
31          <c value="0.5"/>
32      </conditioning-parameters>
33
34      <stimulus-implication/>
35
36    </stimulation-subsystem>
37
38    <responding-subsystem>
39
40      <actions>
41        <action id="0" name="do_homework" base-level="0.0"
             operant="true" reflex="false" />
42        <action id="1" name="study" base-level="0.3" operant
             ="true" reflex="false" />
43        <action id="2" name="annoy" base-level="0.0" operant
             ="true" reflex="true" />
44        <action id="3" name="watch_tv" base-level="0.1"
             operant="true" reflex="false" />
45        <action id="4" name="cry" base-level="0.0" operant="
             true" reflex="false" />
46        <action id="5" name="idle" base-level="0.0" operant=
             "true" reflex="false" />
47      </actions>
48
49      <action-conflict>
50        <conflict id1="1" id2="3"/>
```

```
51        <conflict id1="0" id2="3"/>
52        <conflict id1="1" id2="2"/>
53        <conflict id1="0" id2="2"/>
54      </action-conflict>
55
56      <operants>
57
58        <!-- To idle is quite boring. -->
59        <operant>
60          <antecedents>
61            <antecedent contingency="1.0"/>
62          </antecedents>
63          <action id="5"/>
64          <consequence id="7"/>
65        </operant>
66
67
68        <!-- Doing the homework is rewarded. -->
69        <operant>
70          <antecedents>
71            <antecedent contingency="1.0">
72              <stimulus id="2"/>
73            </antecedent>
74          </antecedents>
75          <action id="0"/>
76          <consequence id="0"/>
77        </operant>
78
79
80        <!-- Annoying other children makes them cry. -->
81        <operant>
82          <antecedents>
83            <antecedent contingency="1.0"/>
84          </antecedents>
85          <action id="2"/>
86          <consequence id="6"/>
87        </operant>
88
89      </operants>
90
91      <reflexes>
92
93        <!-- A provocation elicits crying. -->
94        <reflex>
95
96          <reflex-parameters>
97            <max-elicitation value="1.0" />
98            <min-elicitation value="0.9" />
99            <max-strength value="1.0" />
```

```
100            <min-strength value="0.5" />
101            <max-duration value="10" />
102            <min-duration value="2" />
103            <max-latency value="10" />
104            <min-latency value="1" />
105            <max-threshold value="0.3" />
106            <min-threshold value="0.1" />
107          </reflex-parameters>
108
109          <antecedent-stimulus id="3" />
110          <action id="4" />
111          <threshold value="0.3" />
112          <elicitation value="1.0" />
113          <strength value="1.0" />
114          <duration value="30" />
115          <latency value="1" />
116        </reflex>
117      </reflexes>
118
119    </responding-subsystem>
120
121    <drive-subsystem>
122      <drives/>
123    </drive-subsystem>
124
125    <emotion-subsystem>
126      <anger status="INACTIVE" intensity="0.0" duration="0"
         />
127      <depression status="INACTIVE" intensity="0.0" duration
         ="0" />
128      <frustration status="INACTIVE" intensity="0.0"
         duration="0" />
129    </emotion-subsystem>
130
131  </organism>
```

B.5.2 Scenario

```
1  <?xml version="1.0"?>
2  <scenario name="EMMAS School children example"
3                 description="A class of misbehaving children
                     .">
4
5
6    <agent component-id="organism.OrganismComponent" id="0"
       name="Organism 0">
7      <initializer file="teacher.agent.xml"/>
8    </agent>
```

```
9
10  <agent component-id="organism.OrganismComponent" id="1"
       name="Organism 1">
11    <initializer file="child.agent.xml"/>
12  </agent>
13
14  <agent component-id="organism.OrganismComponent" id="2"
       name="Organism 2">
15    <initializer file="child.agent.xml"/>
16  </agent>
17
18  <agent component-id="organism.OrganismComponent" id="3"
       name="Organism 3">
19    <initializer file="child.agent.xml"/>
20  </agent>
21
22
23  <emmas>
24
25
26    <action-transformers>
27
28      <!-- Teacher can punish any student.  -->
29      <action-transformer agent-id1="0" action="punish_1"
           stimulus="disapproval" agent-id2="1"/>
30      <action-transformer agent-id1="0" action="punish_2"
           stimulus="disapproval" agent-id2="2"/>
31      <action-transformer agent-id1="0" action="punish_3"
           stimulus="disapproval" agent-id2="3"/>
32
33
34      <!-- Teacher can reward any student (e.g., for doing
            their homework).  -->
35      <action-transformer agent-id1="0" action="reward_1"
           stimulus="prize" agent-id2="1"/>
36      <action-transformer agent-id1="0" action="reward_2"
           stimulus="prize" agent-id2="2"/>
37      <action-transformer agent-id1="0" action="reward_3"
           stimulus="prize" agent-id2="3"/>
38
39
40      <!-- Teachers can assign homework. -->
41      <action-transformer agent-id1="0" action="
           assign_homework" stimulus="homework" agent-id2="1
           "/>
42      <action-transformer agent-id1="0" action="
           assign_homework" stimulus="homework" agent-id2="2
           "/>
```

```
43        <action-transformer agent-id1="0" action="
             assign_homework" stimulus="homework" agent-id2="3
             "/>
44
45
46        <!-- Students can do and deliver their homework. -->
47        <action-transformer agent-id1="1" action="
             do_homework" stimulus="homework_1" agent-id2="0"/
             >
48        <action-transformer agent-id1="2" action="
             dd_homework" stimulus="homework_2" agent-id2="0"/
             >
49        <action-transformer agent-id1="3" action="
             do_homework" stimulus="homework_3" agent-id2="0"/
             >
50
51
52        <!-- Students can annoy each other, according to who
              they have access to. -->
53        <action-transformer agent-id1="1" action="annoy"
             stimulus="provocation" agent-id2="2"/>
54        <action-transformer agent-id1="2" action="annoy"
             stimulus="provocation" agent-id2="1"/>
55        <action-transformer agent-id1="1" action="annoy"
             stimulus="provocation" agent-id2="3"/>
56        <action-transformer agent-id1="3" action="annoy"
             stimulus="provocation" agent-id2="1"/>
57        <action-transformer agent-id1="2" action="annoy"
             stimulus="provocation" agent-id2="3"/>
58        <action-transformer agent-id1="3" action="annoy"
             stimulus="provocation" agent-id2="2"/>
59
60
61        <!-- Teacher can see students annoying each other.
             -->
62        <action-transformer agent-id1="1" action="annoy"
             stimulus="see_annoying_1" agent-id2="0"/>
63        <action-transformer agent-id1="2" action="annoy"
             stimulus="see_annoying_2" agent-id2="0"/>
64        <action-transformer agent-id1="3" action="annoy"
             stimulus="see_annoying_3" agent-id2="0"/>
65
66
67        <!-- When an agent cries, everybody listens. -->
68        <action-transformer agent-id1="1" action="cry"
             stimulus="cry_sound" agent-id2="0"/>
69        <action-transformer agent-id1="1" action="cry"
             stimulus="cry_sound" agent-id2="2"/>
```

```
70        <action-transformer agent-id1="1" action="cry"
             stimulus="cry_sound" agent-id2="3"/>
71
72        <action-transformer agent-id1="2" action="cry"
             stimulus="cry_sound" agent-id2="0"/>
73        <action-transformer agent-id1="2" action="cry"
             stimulus="cry_sound" agent-id2="1"/>
74        <action-transformer agent-id1="2" action="cry"
             stimulus="cry_sound" agent-id2="3"/>
75
76        <action-transformer agent-id1="3" action="cry"
             stimulus="cry_sound" agent-id2="0"/>
77        <action-transformer agent-id1="3" action="cry"
             stimulus="cry_sound" agent-id2="1"/>
78        <action-transformer agent-id1="3" action="cry"
             stimulus="cry_sound" agent-id2="2"/>
79
80      </action-transformers>
81
82
83      <behaviors>
84
85        <!-- Students can study. -->
86        <behavior>
87          <environment-response agent-id="1" action="study">
88            <stimulate stimulus="information" agent-id="1" /
                >
89          </environment-response>
90        </behavior>
91
92        <behavior>
93          <environment-response agent-id="2" action="study">
94            <stimulate stimulus="information" agent-id="2" /
                >
95          </environment-response>
96        </behavior>
97
98
99        <behavior>
100         <environment-response agent-id="3" action="study">
101           <stimulate stimulus="information" agent-id="3" /
                >
102         </environment-response>
103       </behavior>
104
105
106       <!-- Distractions are always be available to bother
             the students. -->
107       <behavior>
```

```
108         <environment-response agent-id="1" action="
               watch_tv">
109           <stimulate stimulus="tv" agent-id="1" />
110         </environment-response>
111       </behavior>
112
113       <behavior>
114         <environment-response agent-id="2" action="
               watch_tv">
115           <stimulate stimulus="tv" agent-id="2" />
116         </environment-response>
117       </behavior>
118
119       <behavior>
120         <environment-response agent-id="3" action="
               watch_tv">
121           <stimulate stimulus="tv" agent-id="3" />
122         </environment-response>
123       </behavior>
124
125     </behaviors>
126   </emmas>
127
128 </scenario>
```

B.5.3 Experiment

```
 1 <?xml version="1.0"?>
 2
 3 <experiment name="School children verification"
 4             description="Checks whether children do their
                homework." >
 5
 6   <simulation-purpose-verification relation="feasibility">
 7
 8     <states>
 9      <state id="initial"/>
10      <state id="1"/>
11     </states>
12
13     <events>
14       <emmas-event id="!emit_assign_homework" type="output
            " name="emit" action="assign_homework" agent-id="
            0"/>
15
16       <emmas-event id="!emit_annoy_ag3" type="output" name
            ="emit" action="annoy" agent-id="3"/>
17
```

```
18        <emmas-event id="!emit_do_homework_ag1" type="output
             " name="emit" action="do_homework" agent-id="1"/>
19        <emmas-event id="!emit_do_homework_ag2" type="output
             " name="emit" action="do_homework" agent-id="2"/>
20        <emmas-event id="!emit_do_homework_ag3" type="output
             " name="emit" action="do_homework" agent-id="3"/>
21      </events>
22
23
24      <transitions>
25
26        <!-- Anything can happen for some time. -->
27        <transition state-id1="initial" event-id="other"
             state-id2="initial"/>
28        <transition state-id1="initial" event-id="!
             emit_assign_homework" state-id2="1"/>
29
30        <transition state-id1="1" event-id="other" state-id2
             ="1"/>
31
32        <!-- We wish that at least some of the children will
              eventually do homework. -->
33        <transition state-id1="1" event-id="!
             emit_do_homework_ag1" state-id2="success"/>
34        <transition state-id1="1" event-id="!
             emit_do_homework_ag2" state-id2="success"/>
35        <transition state-id1="1" event-id="!
             emit_do_homework_ag3" state-id2="success"/>
36
37
38      </transitions>
39
40    </simulation-purpose-verification>
41
42  </experiment>
```

B.5.4 Result

```
Simulation Purpose Verification strategy
=======================================================
Result = SUCCESS
Running time = 6s

Run found:
    [depth = 0] State (in SP): initial
    [depth = 1] Events synch'd: <(*)other,
                                ?emit[agentId = 2]_[Action type='annoy']>;
                State annotations synch'd: <[],     []>  State (in SP): initial
    [depth = 2] Events synch'd: <(*)other,
                                !beginning[agentId = 3]_[Stimulus type='provocation']>;
```

```
                State annotations synch'd: <[],     []>  State (in SP): initial
    [depth = 3] Events synch'd: <(*)other,
                                ?emit[agentId = 3]_[Action type='cry']>;
                State annotations synch'd: <[],     []>  State (in SP): initial
    [depth = 4] Events synch'd: <(*)other,
                                !commit>;
                State annotations synch'd: <[],     []>  State (in SP): initial
    [depth = 5] Events synch'd: <!emit[agentId = 0]_[Action type='assign_homework'],
                                ?emit[agentId = 0]_[Action type='assign_homework']>;
                State annotations synch'd: <[],     []>  State (in SP): 1
    [depth = 6] Events synch'd: <(*)other,
                                ?emit[agentId = 1]_[Action type='annoy']>;
                State annotations synch'd: <[],     []>  State (in SP): 1
    [depth = 7] Events synch'd: <(*)other,
                                ?emit[agentId = 3]_[Action type='annoy']>;
                State annotations synch'd: <[],     []>  State (in SP): 1
    [depth = 8] Events synch'd: <(*)other,
                                !beginning[agentId = 1]_[Stimulus type='homework']>;
                State annotations synch'd: <[],     []>  State (in SP): 1
    [depth = 9] Events synch'd: <(*)other,
                                !commit>;
                State annotations synch'd: <[],     []>  State (in SP): 1
    [depth = 10] Events synch'd: <!emit[agentId = 1]_[Action type='do_homework'],
                                 ?emit[agentId = 1]_[Action type='do_homework']>;
                 State annotations synch'd: <[],     []>  State (in SP): success



Result = SUCCESS
Running time = 6s


Finished.
```

B.6 Online Social Network

B.6.1 Agents

Philosopher Type 1 One type of philosopher, who has already read too much and is tired of such an activity.

```
<?xml version="1.0" encoding="UTF-8"?>


<organism>

  <stimulation-subsystem>

    <stimulation-parameters>
      <stimulation-hints>
```

```
10          <pleasure-hints>
11            <stimulus id="0" />
12          </pleasure-hints>
13          <pain-hints>
14          </pain-hints>
15        </stimulation-hints>
16
17        <stimuli>
18          <!-- Philosophers enjoy wine with moderation -->
19          <stimulus id="0" name="wine" primary="true"
               utility="0.4" />
20
21          <!-- Philosophers love Philosophy, obviously -->
22          <stimulus id="1" name="s_verbal_philosophy"
               primary="true" utility="1.0"/>
23
24          <stimulus id="2" name="s_verbal_sophisms" />
25
26          <!-- Philosophers detest ignorance -->
27          <stimulus id="3" name="s_verbal_ignorance" primary
               ="true" utility="-1.0"/>
28
29          <!-- In principle, Philosophers do not fear death
               -->
30          <stimulus id="4" name="s_verbal_death" />
31
32          <!-- Philosophers clearly like books -->
33          <stimulus id="5" name="books" primary="true"
               utility="0.7"/>
34
35          <stimulus id="6" name="s_verbal_joke" primary="
               true" utility="0.6"/>
36
37          <stimulus id="7" name="s_money" primary="true"
               utility="0.3"/>
38
39          <stimulus id="8" name="s_ad_1" />
40
41          <!-- Virtual points that can be exchanged for
               money -->
42          <stimulus id="9" name="points" />
43        </stimuli>
44
45
46        <!-- Philosophers have a long memory for causal
             relations -->
47        <max-delay value="100" />
48
49
```

```
50      </stimulation-parameters>
51
52      <conditioning-parameters>
53          <c value="0.05 "/>
54      </conditioning-parameters>
55
56
57      <stimulus-implication>
58
59          <!-- Philosophers consider sophisms as ignorance
               -->
60          <cause id1="2" id2="3" correlation="1.0" />
61
62          <!-- Death often reminds Philosophers of their
               Philosophy -->
63          <cause id1="4" id2="1" correlation="0.5" />
64
65          <!-- Advertisements are normally related to books.
                -->
66          <cause id1="8" id2="5" correlation="0.5" />
67
68
69          <!-- Virtual points can be redeemed as real money
               -->
70          <cause id1="9" id2="7" correlation="1.0" />
71
72      </stimulus-implication>
73    </stimulation-subsystem>
74
75    <responding-subsystem>
76
77      <actions>
78
79        <!-- Contrary to popular belief, true Philosophers
             like jokes  -->
80        <action id="0" name="a_verbal_joke" base-level="0.01
             " operant="true" reflex="true" />
81
82        <!-- Philosophers often talk about Philosophy for no
              good reason! -->
83        <action id="1" name="a_verbal_philosophy" base-level
             ="0.3" operant="true" reflex="true" />
84
85        <!-- Philosophers do not complain on their own very
             much -->
86        <action id="2" name="a_verbal_complaint" base-level=
             "0.01" operant="true" reflex="true" />
87
```

```
88        <action id="3" name="a_forward_ad_1" base-level="0.0
             " operant="true" reflex="true" />
89
90        <action id="4" name="a_buy_ad_1" base-level="0.0"
             operant="true" reflex="true" />
91
92      </actions>
93
94      <action-conflict>
95        <conflict id1="0" id2="4"/>
96      </action-conflict>
97
98      <operants>
99
100       <!-- Philosophers amuse themselves with their own
             jokes -->
101       <operant>
102         <antecedents>
103           <antecedent contingency="1.0"/>
104         </antecedents>
105         <action id="0"/>
106         <consequence id="6"/>
107       </operant>
108
109       <!-- "To fight ignorance, nature gave us reason."
             -->
110       <operant>
111         <antecedents>
112           <antecedent contingency="0.9">
113             <stimulus id="3"/>
114           </antecedent>
115         </antecedents>
116         <action id="1"/>
117         <consequence id="1"/>
118       </operant>
119
120
121        <!-- Ad 1 forwarding operant -->
122        <operant>
123         <antecedents>
124           <antecedent contingency="1.0">
125             <stimulus id="8"/>
126           </antecedent>
127         </antecedents>
128         <action id="3"/>
129         <consequence id="9"/>
130       </operant>
131
```

```
132         <!-- Ad 1 shopping operant. We are assuming that Ad
                1 is about books. -->
133         <operant>
134          <antecedents>
135           <antecedent contingency="1.0">
136              <stimulus id="8"/>
137            </antecedent>
138          </antecedents>
139          <action id="4"/>
140          <consequence id="5"/>
141        </operant>
142
143      </operants>
144
145
146      <reflexes>
147
148        <!-- Wine make Philosophers prone to telling jokes
               -->
149        <reflex>
150
151          <reflex-parameters>
152            <max-elicitation value="1.0" />
153            <min-elicitation value="0.9" />
154            <max-strength value="1.0" />
155            <min-strength value="0.5" />
156            <max-duration value="10" />
157            <min-duration value="2" />
158            <max-latency value="10" />
159            <min-latency value="1" />
160            <max-threshold value="0.3" />
161            <min-threshold value="0.1" />
162          </reflex-parameters>
163
164          <antecedent-stimulus id="0" />
165          <action id="0" />
166          <threshold value="0.3" />
167          <elicitation value="1.0" />
168          <strength value="1.0" />
169          <duration value="30" />
170          <latency value="1" />
171        </reflex>
172
173        <!-- Ignorance makes Philosophers complain -->
174        <reflex>
175
176          <reflex-parameters>
177            <max-elicitation value="1.0" />
178            <min-elicitation value="0.9" />
```

```
179           <max-strength value="1.0" />
180           <min-strength value="0.5" />
181           <max-duration value="10" />
182           <min-duration value="2" />
183           <max-latency value="10" />
184           <min-latency value="1" />
185           <max-threshold value="0.3" />
186           <min-threshold value="0.1" />
187         </reflex-parameters>
188
189         <antecedent-stimulus id="3" />
190         <action id="2" />
191         <threshold value="0.3" />
192         <elicitation value="1.0" />
193         <strength value="1.0" />
194         <duration value="10" />
195         <latency value="1" />
196       </reflex>
197
198     </reflexes>
199
200   </responding-subsystem>
201
202
203   <drive-subsystem>
204     <drives>
205
206
207       <!-- Philosophers must drink some wine from time to
             time. -->
208       <drive>
209         <importance value="0.0"/>
210         <max-importance value="1.0"/>
211         <min-importance value="-1.0"/>
212         <desires>
213           <stimulus id="0" />
214         </desires>
215       </drive>
216
217
218       <!-- Philosophers sometimes get tired of reading
             books, because they need
219             time to absorb knowledge. -->
220       <drive>
221         <importance value="-0.5"/>
222         <max-importance value="1.0"/>
223         <min-importance value="-0.5"/>
224         <desires>
225           <stimulus id="5" />
```

```
226         </desires>
227       </drive>
228
229     </drives>
230   </drive-subsystem>
231
232
233
234   <emotion-subsystem>
235     <anger status="INACTIVE" intensity="0.0" duration="0"
          />
236     <depression status="INACTIVE" intensity="0.0" duration
          ="0" />
237     <frustration status="INACTIVE" intensity="0.0"
          duration="0" />
238   </emotion-subsystem>
239
240 </organism>
```

Philosopher Type 2 Another type of philosopher, who is not tired of reading.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3
4  <organism>
5
6    <stimulation-subsystem>
7
8      <stimulation-parameters>
9        <stimulation-hints>
10         <pleasure-hints>
11           <stimulus id="0" />
12         </pleasure-hints>
13         <pain-hints>
14         </pain-hints>
15       </stimulation-hints>
16
17       <stimuli>
18         <!-- Philosophers enjoy wine with moderation -->
19         <stimulus id="0" name="wine" primary="true"
             utility="0.4" />
20
21         <!-- Philosophers love Philosophy, obviously -->
22         <stimulus id="1" name="s_verbal_philosophy"
             primary="true" utility="1.0"/>
23
24         <stimulus id="2" name="s_verbal_sophisms" />
```

```
25
26          <!-- Philosophers detest ignorance -->
27          <stimulus id="3" name="s_verbal_ignorance" primary
               ="true" utility="-1.0"/>
28
29          <!-- In principle, Philosophers do not fear death
               -->
30          <stimulus id="4" name="s_verbal_death" />
31
32          <!-- Philosophers clearly like books -->
33          <stimulus id="5" name="books" primary="true"
               utility="0.7"/>
34
35          <stimulus id="6" name="s_verbal_joke" primary="
               true" utility="0.6"/>
36
37          <stimulus id="7" name="s_money" primary="true"
               utility="0.3"/>
38
39          <stimulus id="8" name="s_ad_1" />
40
41          <!-- Virtual points that can be exchanged for
               money -->
42          <stimulus id="9" name="points" />
43        </stimuli>
44
45
46        <!-- Philosophers have a long memory for causal
             relations -->
47        <max-delay value="100" />
48
49
50      </stimulation-parameters>
51
52      <conditioning-parameters>
53          <c value="0.05 "/>
54      </conditioning-parameters>
55
56
57      <stimulus-implication>
58
59          <!-- Philosophers consider sophisms as ignorance
               -->
60          <cause id1="2" id2="3" correlation="1.0" />
61
62          <!-- Death often reminds Philosophers of their
               Philosophy -->
63          <cause id1="4" id2="1" correlation="0.5" />
64
```

```
65          <!-- Advertisements are normally related to books.
                -->
66          <cause id1="8" id2="5" correlation="0.5" />
67
68
69          <!-- Virtual points can be redeemed as real money
                -->
70          <cause id1="9" id2="7" correlation="1.0" />
71
72      </stimulus-implication>
73    </stimulation-subsystem>
74
75
76
77    <responding-subsystem>
78
79      <actions>
80
81        <!-- Contrary to popular belief, true Philosophers
              like jokes  -->
82        <action id="0" name="a_verbal_joke" base-level="0.01
              " operant="true" reflex="true" />
83
84        <!-- Philosophers often talk about Philosophy for no
               good reason! -->
85        <action id="1" name="a_verbal_philosophy" base-level
              ="0.3" operant="true" reflex="true" />
86
87        <!-- Philosophers do not complain on their own very
              much -->
88        <action id="2" name="a_verbal_complaint" base-level=
              "0.01" operant="true" reflex="true" />
89
90        <action id="3" name="a_forward_ad_1" base-level="0.0
              " operant="true" reflex="true" />
91
92        <action id="4" name="a_buy_ad_1" base-level="0.0"
              operant="true" reflex="true" />
93
94      </actions>
95
96      <action-conflict>
97        <conflict id1="0" id2="4"/>
98      </action-conflict>
99
100     <operants>
101
102       <!-- Philosophers amuse themselves with their own
              jokes -->
```

```
103        <operant>
104          <antecedents>
105            <antecedent contingency="1.0"/>
106          </antecedents>
107          <action id="0"/>
108          <consequence id="6"/>
109        </operant>
110
111        <!-- "To fight ignorance, nature gave us reason."
               -->
112        <operant>
113          <antecedents>
114            <antecedent contingency="0.9">
115              <stimulus id="3"/>
116            </antecedent>
117          </antecedents>
118          <action id="1"/>
119          <consequence id="1"/>
120        </operant>
121
122
123         <!-- Ad 1 forwarding operant -->
124         <operant>
125          <antecedents>
126            <antecedent contingency="1.0">
127              <stimulus id="8"/>
128            </antecedent>
129          </antecedents>
130          <action id="3"/>
131          <consequence id="9"/>
132        </operant>
133
134         <!-- Ad 1 shopping operant. We are assuming that Ad
                1 is about books. -->
135         <operant>
136          <antecedents>
137           <antecedent contingency="1.0">
138              <stimulus id="8"/>
139            </antecedent>
140          </antecedents>
141          <action id="4"/>
142          <consequence id="5"/>
143        </operant>
144
145      </operants>
146
147
148      <reflexes>
149
```

```
150        <!-- Wine make Philosophers prone to telling jokes
              -->
151        <reflex>
152
153          <reflex-parameters>
154            <max-elicitation value="1.0" />
155            <min-elicitation value="0.9" />
156            <max-strength value="1.0" />
157            <min-strength value="0.5" />
158            <max-duration value="10" />
159            <min-duration value="2" />
160            <max-latency value="10" />
161            <min-latency value="1" />
162            <max-threshold value="0.3" />
163            <min-threshold value="0.1" />
164          </reflex-parameters>
165
166          <antecedent-stimulus id="0" />
167          <action id="0" />
168          <threshold value="0.3" />
169          <elicitation value="1.0" />
170          <strength value="1.0" />
171          <duration value="30" />
172          <latency value="1" />
173        </reflex>
174
175        <!-- Ignorance makes Philosophers complain -->
176        <reflex>
177
178          <reflex-parameters>
179            <max-elicitation value="1.0" />
180            <min-elicitation value="0.9" />
181            <max-strength value="1.0" />
182            <min-strength value="0.5" />
183            <max-duration value="10" />
184            <min-duration value="2" />
185            <max-latency value="10" />
186            <min-latency value="1" />
187            <max-threshold value="0.3" />
188            <min-threshold value="0.1" />
189          </reflex-parameters>
190
191          <antecedent-stimulus id="3" />
192          <action id="2" />
193          <threshold value="0.3" />
194          <elicitation value="1.0" />
195          <strength value="1.0" />
196          <duration value="10" />
197          <latency value="1" />
```

```
198       </reflex>
199
200     </reflexes>
201
202   </responding-subsystem>
203
204
205
206
207
208   <drive-subsystem>
209     <drives>
210
211
212       <!-- Philosophers must drink some wine from time to
             time. -->
213       <drive>
214         <importance value="0.0"/>
215         <max-importance value="1.0"/>
216         <min-importance value="-1.0"/>
217         <desires>
218           <stimulus id="0" />
219         </desires>
220       </drive>
221
222
223       <!-- Philosophers sometimes get tired of reading
             books, because they need
224             time to absorb knowledge. -->
225       <drive>
226         <importance value="0.0"/>
227         <max-importance value="1.0"/>
228         <min-importance value="-0.5"/>
229         <desires>
230           <stimulus id="5" />
231         </desires>
232       </drive>
233
234     </drives>
235   </drive-subsystem>
236
237
238
239   <emotion-subsystem>
240     <anger status="INACTIVE" intensity="0.0" duration="0"
          />
241     <depression status="INACTIVE" intensity="0.0" duration
          ="0" />
```

```
242     <frustration status="INACTIVE" intensity="0.0"
            duration="0" />
243   </emotion-subsystem>
244
245 </organism>
```

B.6.2 Scenario

```
1  <?xml version="1.0"?>
2  <scenario name="EMMAS Online Social Network Example"
3                description="An online social network.">
4
5
6    <!-- Organisms -->
7    <agent component-id="organism.OrganismComponent" id="0"
         name="Philosopher 0">
8      <initializer file="philosopher1.agent.xml"/>
9    </agent>
10
11   <agent component-id="organism.OrganismComponent" id="1"
         name="Philosopher 1">
12     <initializer file="philosopher1.agent.xml"/>
13   </agent>
14
15   <agent component-id="organism.OrganismComponent" id="2"
         name="Philosopher 2">
16     <initializer file="philosopher1.agent.xml"/>
17   </agent>
18
19   <agent component-id="organism.OrganismComponent" id="3"
         name="Philosopher 3">
20     <initializer file="philosopher2.agent.xml"/>
21   </agent>
22
23   <agent component-id="organism.OrganismComponent" id="4"
         name="Philosopher 4">
24     <initializer file="philosopher1.agent.xml"/>
25   </agent>
26
27
28   <emmas>
29
30     <action-transformers>
31
32       <!-- Message forwarding -->
33       <action-transformer agent-id1="0" action="
             a_forward_ad_1" stimulus="s_ad_1" agent-id2="1"/>
```

```
34        <action-transformer agent-id1="0" action="
             a_forward_ad_1" stimulus="s_ad_1" agent-id2="2"/>
35        <action-transformer agent-id1="2" action="
             a_forward_ad_1" stimulus="s_ad_1" agent-id2="3"/>
36
37        <!-- Agents listen to their own jokes -->
38        <action-transformer agent-id1="0" action="
             a_verbal_joke" stimulus="s_verbal_joke" agent-id2
             ="0"/>
39        <action-transformer agent-id1="1" action="
             a_verbal_joke" stimulus="s_verbal_joke" agent-id2
             ="1"/>
40        <action-transformer agent-id1="2" action="
             a_verbal_joke" stimulus="s_verbal_joke" agent-id2
             ="2"/>
41        <action-transformer agent-id1="3" action="
             a_verbal_joke" stimulus="s_verbal_joke" agent-id2
             ="3"/>
42
43        <!-- Agent 4 listen to all jokes -->
44        <action-transformer agent-id1="0" action="
             a_verbal_joke" stimulus="s_verbal_joke" agent-id2
             ="4"/>
45        <action-transformer agent-id1="1" action="
             a_verbal_joke" stimulus="s_verbal_joke" agent-id2
             ="4"/>
46        <action-transformer agent-id1="2" action="
             a_verbal_joke" stimulus="s_verbal_joke" agent-id2
             ="4"/>
47        <action-transformer agent-id1="3" action="
             a_verbal_joke" stimulus="s_verbal_joke" agent-id2
             ="4"/>
48
49
50        <!-- Some agents exchange philosophy -->
51        <action-transformer agent-id1="0" action="
             a_verbal_philosophy" stimulus="
             s_verbal_philosophy" agent-id2="1"/>
52        <action-transformer agent-id1="1" action="
             a_verbal_philosophy" stimulus="
             s_verbal_philosophy" agent-id2="0"/>
53
54        <action-transformer agent-id1="0" action="
             a_verbal_philosophy" stimulus="
             s_verbal_philosophy" agent-id2="2"/>
55        <action-transformer agent-id1="2" action="
             a_verbal_philosophy" stimulus="
             s_verbal_philosophy" agent-id2="0"/>
56
```

```
57        <action-transformer agent-id1="2" action="
             a_verbal_philosophy" stimulus="
             s_verbal_philosophy" agent-id2="3"/>
58        <action-transformer agent-id1="3" action="
             a_verbal_philosophy" stimulus="
             s_verbal_philosophy" agent-id2="2"/>
59
60        <!-- Agents get what they buy -->
61        <action-transformer agent-id1="0" action="a_buy_ad_1
             " stimulus="books" agent-id2="0"/>
62        <action-transformer agent-id1="1" action="a_buy_ad_1
             " stimulus="books" agent-id2="1"/>
63        <action-transformer agent-id1="2" action="a_buy_ad_1
             " stimulus="books" agent-id2="2"/>
64        <action-transformer agent-id1="3" action="a_buy_ad_1
             " stimulus="books" agent-id2="3"/>
65        <action-transformer agent-id1="4" action="a_buy_ad_1
             " stimulus="books" agent-id2="4"/>
66
67      </action-transformers>
68
69
70
71      <behaviors>
72
73
74        <!-- The advertisement engine can choose among
             agents -->
75        <behavior>
76          <choice>
77            <begin-stimulation stimulus="s_ad_1" agent-id="0
                " />
78            <begin-stimulation stimulus="s_ad_1" agent-id="1
                " />
79            <begin-stimulation stimulus="s_ad_1" agent-id="2
                " />
80            <begin-stimulation stimulus="s_ad_1" agent-id="3
                " />
81            <begin-stimulation stimulus="s_ad_1" agent-id="4
                " />
82          </choice>
83        </behavior>
84
85
86        <!--
87           Message forwarding is rewarded with virtual
                points.
88        -->
89
```

```
90        <behavior>
91          <environment-response agent-id="0" action="
              a_forward_ad_1">
92            <stimulate stimulus="points" agent-id="0" />
93          </environment-response>
94        </behavior>
95
96        <behavior>
97          <environment-response agent-id="1" action="
              a_forward_ad_1">
98            <stimulate stimulus="points" agent-id="1" />
99          </environment-response>
100       </behavior>
101
102       <behavior>
103         <environment-response agent-id="2" action="
              a_forward_ad_1">
104           <stimulate stimulus="points" agent-id="2" />
105         </environment-response>
106       </behavior>
107
108       <behavior>
109         <environment-response agent-id="3" action="
              a_forward_ad_1">
110           <stimulate stimulus="points" agent-id="3" />
111         </environment-response>
112       </behavior>
113
114       <behavior>
115         <environment-response agent-id="4" action="
              a_forward_ad_1">
116           <stimulate stimulus="points" agent-id="4" />
117         </environment-response>
118       </behavior>
119
120     </behaviors>
121
122   </emmas>
123
124
125 </scenario>
```

B.6.3 Experiment

```
1 <?xml version="1.0"?>
2
3 <experiment name="Simulation Purpose test"
4             description="A simple test for the simulation
                purpose verification strategy." >
```

```
5
6   <simulation-purpose-verification relation="feasibility">
7
8     <states>
9      <state id="initial"/>  <state id="1"/>  <state id="2"
          />  <state id="3"/>   <state id="4"/>  <state id="
          5"/>
10     <state id="6"/>  <state id="7"/>  <state id="8"/>  <
          state id="9"/>  <state id="10"/>  <state id="11"/>
11    </states>
12
13
14    <events>
15     <emmas-event id="?beg_ad1_0" type="input" name="
          beginning" stimulus="s_ad_1" agent-id="0"/>
16     <emmas-event id="?beg_ad1_1" type="input" name="
          beginning" stimulus="s_ad_1" agent-id="1"/>
17     <emmas-event id="?beg_ad1_2" type="input" name="
          beginning" stimulus="s_ad_1" agent-id="2"/>
18     <emmas-event id="?beg_ad1_3" type="input" name="
          beginning" stimulus="s_ad_1" agent-id="3"/>
19     <emmas-event id="!emit_forward1_0" type="output" name
          ="emit" action="a_forward_ad_1" agent-id="0"/>
20     <emmas-event id="!emit_forward1_2" type="output" name
          ="emit" action="a_forward_ad_1" agent-id="2"/>
21     <emmas-event id="!emit_buy1_3" type="output" name="
          emit" action="a_buy_ad_1" agent-id="3"/>
22    </events>
23
24
25    <transitions>
26
27     <!-- Advertise to agent 0 -->
28     <transition state-id1="initial" event-id="?beg_ad1_0"
           state-id2="1"/>
29     <transition state-id1="1" event-id="other" state-id2=
          "1"/>
30
31     <!-- As soon as possible, agent 0 forwards the ad to
          its friends -->
32     <transition state-id1="1" event-id="!emit_forward1_0"
           state-id2="2"/>
33     <transition state-id1="2" event-id="other" state-id2=
          "2"/>
34
35     <!-- We are interested in agent 2 receiving this
          forwarded ad -->
36     <transition state-id1="2" event-id="?beg_ad1_2" state
          -id2="3"/>
```

```
37      <transition state-id1="2" event-id="other" state-id2=
           "2"/>
38
39
40      <!-- Agent 2 should as well forward the ad -->
41      <transition state-id1="3" event-id="!emit_forward1_2"
            state-id2="4"/>
42      <transition state-id1="3" event-id="other" state-id2=
           "3"/>
43
44      <!-- Agent 3 receives the ad -->
45      <transition state-id1="4" event-id="?beg_ad1_3" state
           -id2="5"/>
46      <transition state-id1="4" event-id="other" state-id2=
           "4"/>
47
48      <!-- Agent 3 buys the product -->
49      <transition state-id1="5" event-id="!emit_buy1_3"
           state-id2="success"/>
50      <transition state-id1="5" event-id="other" state-id2=
           "5"/>
51
52     </transitions>
53
54   </simulation-purpose-verification>
55
56 </experiment>
```

B.6.4 Result

```
Simulation Purpose Verification strategy
=======================================================
Result = SUCCESS
Running time = 30s

Run found:
    [depth = 0] State (in SP): initial
    [depth = 1] Events synch'd: <?beginning[agentId = 0]_[Stimulus type='s_ad_1'],
                                !beginning[agentId = 0]_[Stimulus type='s_ad_1']>;
                State annotations synch'd: <[],     []>  State (in SP): 1
    [depth = 2] Events synch'd: <(*)other,
                                !commit>;
                State annotations synch'd: <[],     []>  State (in SP): 1
    [depth = 3] Events synch'd: <!emit[agentId = 0]_[Action type='a_forward_ad_1'],
                                ?emit[agentId = 0]_[Action type='a_forward_ad_1']>;
                State annotations synch'd: <[],     []>  State (in SP): 2
    [depth = 4] Events synch'd: <(*)other,
                                !commit>;
                State annotations synch'd: <[],     []>  State (in SP): 2
    [depth = 5] Events synch'd: <(*)other,
```

```
                              ?emit[agentId = 3]_[Action type='a_verbal_joke']>;
            State annotations synch'd: <[],     []>  State (in SP): 2
[depth = 6] Events synch'd: <(*)other,
                              !stable[agentId = 0]_[Stimulus type='s_ad_1']>;
            State annotations synch'd: <[],     []>  State (in SP): 2
[depth = 7] Events synch'd: <(*)other,
                              ?emit[agentId = 0]_[Action type='a_forward_ad_1']>;
            State annotations synch'd: <[],     []>  State (in SP): 2
[depth = 8] Events synch'd: <?beginning[agentId = 2]_[Stimulus type='s_ad_1'],
                              !beginning[agentId = 2]_[Stimulus type='s_ad_1']>;
            State annotations synch'd: <[],     []>  State (in SP): 3
[depth = 9] Events synch'd: <(*)other,
                              !beginning[agentId = 3]_[Stimulus type='s_verbal_joke']>;
            State annotations synch'd: <[],     []>  State (in SP): 3
[depth = 10] Events synch'd: <(*)other,
                               ?emit[agentId = 2]_[Action type='a_verbal_joke']>;
             State annotations synch'd: <[],     []>  State (in SP): 3
[depth = 11] Events synch'd: <(*)other,
                               ?emit[agentId = 1]_[Action type='a_verbal_joke']>;
             State annotations synch'd: <[],     []>  State (in SP): 3
[depth = 12] Events synch'd: <(*)other,
                               ?emit[agentId = 0]_[Action type='a_verbal_philosophy']>;
             State annotations synch'd: <[],     []>  State (in SP): 3
[depth = 13] Events synch'd: <(*)other,
                               ?emit[agentId = 3]_[Action type='a_verbal_philosophy']>;
             State annotations synch'd: <[],     []>  State (in SP): 3
[depth = 14] Events synch'd: <(*)other,
                               !beginning[agentId = 1]_[Stimulus type='s_ad_1']>;
             State annotations synch'd: <[],     []>  State (in SP): 3
[depth = 15] Events synch'd: <(*)other,
                               !beginning[agentId = 1]_[Stimulus type='s_verbal_philosophy']>;
             State annotations synch'd: <[],     []>  State (in SP): 3
[depth = 16] Events synch'd: <(*)other,
                               ?emit[agentId = 0]_[Action type='a_verbal_joke']>;
             State annotations synch'd: <[],     []>  State (in SP): 3
[depth = 17] Events synch'd: <(*)other,
                               !commit>;
             State annotations synch'd: <[],     []>  State (in SP): 3
[depth = 18] Events synch'd: <!emit[agentId = 2]_[Action type='a_forward_ad_1'],
                               ?emit[agentId = 2]_[Action type='a_forward_ad_1']>;
             State annotations synch'd: <[],     []>  State (in SP): 4
[depth = 19] Events synch'd: <(*)other,
                               ?emit[agentId = 1]_[Action type='a_forward_ad_1']>;
             State annotations synch'd: <[],     []>  State (in SP): 4
[depth = 20] Events synch'd: <(*)other,
                               !beginning[agentId = 4]_[Stimulus type='s_verbal_joke']>;
             State annotations synch'd: <[],     []>  State (in SP): 4
[depth = 21] Events synch'd: <(*)other,
                               !stable[agentId = 1]_[Stimulus type='s_ad_1']>;
             State annotations synch'd: <[],     []>  State (in SP): 4
[depth = 22] Events synch'd: <(*)other,
                               ?emit[agentId = 3]_[Action type='a_verbal_joke']>;
             State annotations synch'd: <[],     []>  State (in SP): 4
[depth = 23] Events synch'd: <(*)other,
```

```
                              !beginning[agentId = 0]_[Stimulus type='s_verbal_joke']>;
             State annotations synch'd: <[],     []>  State (in SP): 4
[depth = 24] Events synch'd: <(*)other,
                              ?emit[agentId = 1]_[Action type='a_verbal_joke']>;
             State annotations synch'd: <[],     []>  State (in SP): 4
[depth = 25] Events synch'd: <(*)other,
                              TAU>;
             State annotations synch'd: <[],     []>  State (in SP): 4
[depth = 26] Events synch'd: <(*)other,
                              ?emit[agentId = 0]_[Action type='a_forward_ad_1']>;
             State annotations synch'd: <[],     []>  State (in SP): 4
[depth = 27] Events synch'd: <(*)other,
                              !beginning[agentId = 1]_[Stimulus type='points']>;
             State annotations synch'd: <[],     []>  State (in SP): 4
[depth = 28] Events synch'd: <(*)other,
                              !stable[agentId = 3]_[Stimulus type='s_verbal_joke']>;
             State annotations synch'd: <[],     []>  State (in SP): 4
[depth = 29] Events synch'd: <(*)other,
                              ?emit[agentId = 2]_[Action type='a_verbal_joke']>;
             State annotations synch'd: <[],     []>  State (in SP): 4
[depth = 30] Events synch'd: <(*)other,
                              !commit>;
             State annotations synch'd: <[],     []>  State (in SP): 4
[depth = 31] Events synch'd: <(*)other,
                              TAU>;
             State annotations synch'd: <[],     []>  State (in SP): 4
[depth = 32] Events synch'd: <(*)other,
                              ?emit[agentId = 2]_[Action type='a_forward_ad_1']>;
             State annotations synch'd: <[],     []>  State (in SP): 4
[depth = 33] Events synch'd: <?beginning[agentId = 3]_[Stimulus type='s_ad_1'],
                              !beginning[agentId = 3]_[Stimulus type='s_ad_1']>;
             State annotations synch'd: <[],     []>  State (in SP): 5
[depth = 34] Events synch'd: <(*)other,
                              !beginning[agentId = 2]_[Stimulus type='s_verbal_philosophy']>;
             State annotations synch'd: <[],     []>  State (in SP): 5
[depth = 35] Events synch'd: <(*)other,
                              !beginning[agentId = 2]_[Stimulus type='s_verbal_joke']>;
             State annotations synch'd: <[],     []>  State (in SP): 5
[depth = 36] Events synch'd: <(*)other,
                              !stable[agentId = 1]_[Stimulus type='points']>;
             State annotations synch'd: <[],     []>  State (in SP): 5
[depth = 37] Events synch'd: <(*)other,
                              ?emit[agentId = 0]_[Action type='a_verbal_philosophy']>;
             State annotations synch'd: <[],     []>  State (in SP): 5
[depth = 38] Events synch'd: <(*)other,
                              ?emit[agentId = 2]_[Action type='a_verbal_philosophy']>;
             State annotations synch'd: <[],     []>  State (in SP): 5
[depth = 39] Events synch'd: <(*)other,
                              ?emit[agentId = 0]_[Action type='a_verbal_joke']>;
             State annotations synch'd: <[],     []>  State (in SP): 5
[depth = 40] Events synch'd: <(*)other,
                              !stable[agentId = 0]_[Stimulus type='s_verbal_joke']>;
             State annotations synch'd: <[],     []>  State (in SP): 5
[depth = 41] Events synch'd: <(*)other,
```

```
                                !beginning[agentId = 1]_[Stimulus type='s_verbal_joke']>;
                State annotations synch'd: <[],     []>  State (in SP): 5
    [depth = 42] Events synch'd: <(*)other,
                                !stable[agentId = 1]_[Stimulus type='s_verbal_philosophy']>;
                State annotations synch'd: <[],     []>  State (in SP): 5
    [depth = 43] Events synch'd: <(*)other,
                                !commit>;
                State annotations synch'd: <[],     []>  State (in SP): 5
    [depth = 44] Events synch'd: <!emit[agentId = 3]_[Action type='a_buy_ad_1'],
                                ?emit[agentId = 3]_[Action type='a_buy_ad_1']>;
                State annotations synch'd: <[],     []>  State (in SP): success



Result = SUCCESS
Running time = 30s


Finished.
```

APPENDIX C

Simulator Input Format and Parameters

FGS is a command-line tool. To execute it, it is necessary to specify certain input files, as well as to give certain parameters as arguments to the tool. In Section C.1 we present the relevant input format, and in Section C.2 we explain the tool's parameters.

C.1 Input Format

The inputs are all given as XML files. There are three kinds of such files, namely:

- *Instantiation of the Behaviourist Agent Architecture.* The implementation of **Behaviourist Agent Architecture** takes several parameters in order to create a concrete agent.
- *Scenarios definitions.* Declares which agents are present, and define the **EMMAS** model to use.
- *Experiments definitions.* Defines the simulation and verification strategy to execute, including the **simulation purposes** to use.

The examples provided in Chapter 9 should be enough to allow one to learn how to specify these inputs. Nevertheless, for the sake of completeness, in what follows we document the three types of XML format. We do so by listing the required XML elements and providing an example. For readability,

each format is broken in several parts, and each part is explained individually. A mark of `(...)` in the examples indicates that a part of the XML has been suppressed because it is shown in a later example. This documentation is sufficient to relate the theoretical concepts seen in the thesis to their actual implementation. Features that go beyond the scope of the thesis (e.g., for experimental purposes) are not presented.

C.1.1 Behaviourist Agent Architecture

The parametrization of the **Behaviourist Agent Architecture** is closely related to its formal Z specification, presented in Chapter 4. Each XML element corresponds to some Z specification element, which is made more concrete (i.e., particular to an agent).

<organism> Organisms are composed of four subsystems, which must all be parametrized.

```
<organism>
 <stimulation-subsystem> (...) </stimulation-subsystem>
 <responding-subsystem> (...) </responding-subsystem>
 <drive-subsystem>  (...) </drive-subsystem>
 <emotion-subsystem> (...)  </emotion-subsystem>
</organism>
```

<stimulation-subsystem> The stimulation subsystem definition consists of the following sub-parts:

- A list `<pleasure-hints>` of stimuli that signals pleasure in another agent.
- A list `<pain-hints>` of stimuli that signals pain in another agent.
- A list `<stimuli>` containing all the stimuli recognized by the agent.
- A constant `<max-delay/>` that determines the maximum delay between one stimulus and another in order to classical conditioning to take place.
- A constant `<c/>` used by stimulus conditioning.
- A list `<stimulus-implication>` of relations between stimuli that are initially in a stimulus implication relation. Each element `<cause/>` indicates that stimulus identified by `id1` implies stimulus identified by `id2` with a correlation of `correlation`.

```
<stimulation-subsystem>
 <stimulation-parameters>
  <stimulation-hints>

   <pleasure-hints>
     <stimulus id="0"/>
     <stimulus id="1"/>
   </pleasure-hints>

   <pain-hints>
     <stimulus id="2"/>
     <stimulus id="3"/>
   </pain-hints>

  </stimulation-hints>

  <stimuli>

   <stimulus id="0" name="some_stimulus"    primary="true"
       utility="0.5" />
   <stimulus id="1" name="another_stimulus" />
   <!-- Other <stimulus/>  -->

  </stimuli>

  <max-delay value="10" />

 </stimulation-parameters>

 <conditioning-parameters>
  <c value="0.5"/>
 </conditioning-parameters>

 <stimulus-implication>
  <cause id1="0" id2="1" correlation="1.0" />
  <cause id1="1" id2="2" correlation="0.5" />
 </stimulus-implication>

</stimulation-subsystem>
```

<responding-subsystem> The responding subsystem parametrization is composed of the following sub-parts:

- A list `<actions>` of actions. For each `<action>`, one must specify: an identifier unique among actions, its name, its base level probability of

occurrence, whether it can be used in an operant, and whether it can be used in a reflex.

- A list `<action-conflict>` of conflicts between actions. Each element `<conflict>` must specify the identifier of two actions.
- A list `<operants>` of operants initially known by the agents.
- A list `<reflexes>` of reflexes.

```
<responding-subsystem>

 <actions>
  <action id="0" name="action_0" base-level="0.0" operant=
     "true" reflex="false" />
  <action id="1" name="action_1" base-level="0.3" operant=
     "true" reflex="false" />
  <action id="2" name="action_2" base-level="0.3" operant=
     "true" reflex="false" />
 </actions>

 <action-conflict>
  <conflict id1="1" id2="0"/>
  <conflict id1="2" id2="1"/>
 </action-conflict>

 <operants> (...) </operants>
 <reflexes> (...) </reflexes>

</responding-subsystem>
```

`<operants>` Zero or more `<operant>` must be specified. Each `<operant>` must define:

- A list `<antecedents>` of lists`<antecedent>` of stimuli. Each such `<antecedent>` must specify a correlation `contingency`. Moreover, each such element may be either empty or contain references to stimuli identified by `id`.
- A reference `<action>` to an action identified by `id`.
- A reference `<consequence>` to a stimulus identified by `id`.

```
<operants>

 <operant>
```

```
  <antecedents>
   <antecedent contingency="1.0"/>
  </antecedents>
  <action id="5"/>
  <consequence id="7"/>
 </operant>


 <operant>
  <antecedents>
   <antecedent contingency="1.0">
    <stimulus id="0"/>
    <stimulus id="2"/>
   </antecedent>
  </antecedents>
  <action id="0"/>
  <consequence id="0"/>
 </operant>

</operants>
```

<reflexes> Zero or more `<reflex>` must be specified. Each `<reflex>` must define:

- A list `<reflex-parameters>` with several parameters.
- A reference `<antecedent-stimulus/>` to a stimulus identified by `id`.
- An reference `<action/>` to an action identified by `id`.
- An element `<threshold/>` specifying an initial `value` for the threshold parameter.
- An element `<elicitation/>` specifying an initial `value` for the elicitation parameter.
- An element `<strength/>` specifying an initial `value` for the strength parameter.
- An element `<duration/>` specifying an initial `value` for the duration parameter.
- An element `<latency/>` specifying an initial `value` for the latency parameter.

```
<reflexes>

 <reflex>
  <reflex-parameters>
   <max-elicitation value="1.0" />
   <min-elicitation value="0.9" />
   <max-strength value="1.0" />
   <min-strength value="0.5" />
   <max-duration value="10" />
   <min-duration value="2" />
   <max-latency value="10" />
   <min-latency value="1" />
   <max-threshold value="0.3" />
   <min-threshold value="0.1" />
  </reflex-parameters>

  <antecedent-stimulus id="3" />
  <action id="4" />
  <threshold value="0.3" />
  <elicitation value="1.0" />
  <strength value="1.0" />
  <duration value="30" />
  <latency value="1" />
 </reflex>

</reflexes>
```

<drive-subsystem> Zero or more `<drive>` must be specified. Each `<drive>` must define:

- An `<importance>` specifying an initial `value`.
- A `<max-importance>` specifying a maximum `value`.
- A `<min-importance>` specifying a minimum `value`.
- A list `<desires>` of references to stimuli, each identified by an `id`.

```
<drive-subsystem>
  <drives>
    <drive>
      <importance value="0.0"/>
      <max-importance value="0.5"/>
      <min-importance value="-1.0"/>
      <desires>
        <stimulus id="0" />
```

```
      </desires>
    </drive>
    <!-- Other <drive></drive>  -->
   </drives>
</drive-subsystem>
```

<emotion-subsystem> Exactly three emotions must be specified: `anger`, `depression` and `frustration`. Each one has the same kind of attributes:

- A `status`, which must be either `ACTIVE` or `INACTIVE`.
- An `intensity` magnitude.
- A `duration` natural number.

```
<emotion-subsystem>
 <anger status="INACTIVE" intensity="0.0" duration="0" />
 <depression status="INACTIVE" intensity="0.0" duration="0
    " />
 <frustration status="INACTIVE" intensity="0.0" duration="
    0" />
</emotion-subsystem>
```

C.1.2 Scenarios

Scenarios define the elements available to the simulation. This includes agents, properties to be measured regarding them, as well as the environment itself, given as an **EMMAS** specification.

<scenario> A scenario specify the agents that exist, the properties to be measured (from which propositions about the MAS can be derived), and the **EMMAS** specification.

```
<scenario name="Some name" description="More information
   about the scenario comes here.">

 <!-- Agents -->
 (...)

 <!-- Properties -->
 (...)

 <!-- Environment -->
```

```
10  <emmas> (...) </emmas>
11
12 </scenario>
```

<agent> Each agent is defined by an <agent> element, which must:

- Define the attribute `component-id`, which is the full name (including packages) of the class that implement the agent. In this thesis, we are using the `organism.OrganismComponent` class, which is the component that implements the **Behaviourist Agent Architecture**.
- Define the attribute `id`, which must be a unique natural number to identify the agent.
- Define the attribute `name`, with a user friendly name for the agent.
- Contain as a child a XML element that shall be used to initialize the agent component when instantiating it. It is also possible (and preferable) to include a XML file containing this element through the `<initializer>` element, which takes a `file` attribute. This attribute must contain the name of a XML file that is in the same folder as the scenario file.

```
1  <agent component-id="organism.OrganismComponent" id="0"
      name="Organism 0">
2   <initializer file="agent1.xml"/>
3  </agent>
4
5  <agent component-id="organism.OrganismComponent" id="1"
      name="Organism 1">
6   <initializer file="agent2.xml"/>
7  </agent>
```

<property> Properties are used to calculate propositions during simulations. They must be declared to exist to be taken in account. For each property, one must:

- Define the `component-id` attribute, which is the full name (including packages) of the class that implement the property.
- Define the attribute `id`, which must be a unique natural number to identify the property.

- Define the attribute `name`, with a user friendly name for the property.
- Specify as children either one `<environment-target />` element, or a series of `<agent-target/>` elements. In the former case, the property uses the environment itself to be calculated, whereas in the later it uses the agents to this end. Each reference `<agent-target/>` must specify the `id` of an agent.
- Specify as children zero or more `<primitive-parameter />` elements, with attributes `name` and `value`, to configure the property.

```
<property component-id = "organism.StimulusUtilityProperty
    " id="0" name="LikesCandy">
  <primitive-parameter name="TargetStimulus" value="candy"
      />
  <primitive-parameter name="TargetValue" value="0.1" />

  <agent-target id="0" />
</property>
```

<emmas> An **EMMAS** specification is composed of a list of **action transformers** and a list of environment behaviours. Each `<behavior>` must contain one child **environment operation** element.

```
<emmas>
 <action-transformers> (...) </action-transformers>

 <behaviors>
  <!-- Zero or more <behavior> -->
  <behavior> (...) </behavior>
  <behavior> (...) </behavior>
 </behaviors>
</emmas>
```

<action-transformers> Defines zero or more `<action-transformer>`. Each `<action-transformer>` must define the following attributes:

- `agent-id1`: the identifier of the agent that performs the action.
- `action`: the name of an action. The possible action names are given by the specification of the agents, and correspond to the names of actions defined therein.

- `stimulus`: the name of a stimulus. The possible stimulus names are given by the specification of the agents, and correspond to the names of stimuli defined therein.
- `agent-id2`, the identifier of the agent that receives the stimulus.

```
<action-transformers>
 <action-transformer agent-id1="0" action="punish_1"
    stimulus="disapproval" agent-id2="1"/>
 <action-transformer agent-id1="0" action="punish_2"
    stimulus="disapproval" agent-id2="2"/>
 <action-transformer agent-id1="0" action="punish_3"
    stimulus="disapproval" agent-id2="3"/>
</action-transformers>
```

Environment Operations All composition and core **operations** described in Chapter 5 are implemented and can be used. In the present version of FGS, quantifiers and complex **operations** must be manually expanded in order to be converted in these composition and core **operations**.

Let `stimulus` be an attribute that takes as its value the name of a stimulus, `action` be an attribute that takes as its value the name of an action, and `agent-id`, `agent-id1` and `agent-id2` be agent identifiers. Then the following are the core **operations** that are not composed of other **operations**:

- `<begin-stimulation stimulus="bell" agent-id="0" />`: Implements $BeginStimulation(s, ag)$.
- `<end-stimulation stimulus="bell" agent-id="0" />`: Implements $EndStimulation(s, ag)$.
- `<stimulate stimulus="bell" agent-id="0" />`: Implements $Stimulate(s, ag)$.
- `<create agent-id1="0" action="a" stimulus="s" agent-id2 = "13" />`: Implements $Create(ag_1, a, s, ag_2)$.
- `<destroy agent-id1="0" action="a" stimulus="s" agent-id2 = "13" />`: Implements $Destroy(n, a, s, m)$.
- `<nop/>`: Implements NOP.

Environment responses are given by specifying `<environment-response>` elements which must:

- Define the attribute `agent-id` that indicates the agent identifier whose actions should be considered.
- Define the attribute `action` that indicates the name of the action to be considered.
- Define one **operation** as a child element.

```
<environment-response agent-id="1" action="watch_tv">
   (...)
</environment-response>
```

Composition **operations**, in turn, are as follows:

- `<parallel> (...) </parallel>`: Implements **parallel composition**. Must contain two or more children **operations**.
- `<choice> (...) </choice>`: Implements **choice**. Must contain two or more children **operations**.
- `<sequential-composition> (...) </sequential-composition>`: Implements **sequential composition**. Must contain two or more children **operations**.
- `<sequence times = "2"> (...) </sequence>`: Implements **sequence**. Must define an attribute `times` with a natural number, and contain one child **operation**.
- `<unbounded-sequence> (...) </unbounded-sequence>`: Implements **unbounded sequence**. Must contain one child **operation**.

C.1.3 Experiments

An experiment specify how the simulation should be conducted. In this thesis, we are concerned with the verification of satisfiability relations pertaining to **simulation purposes**.

<experiment> The experiment should contain one or more `<simulation-purpose-verification>` elements, describing the verification to be performed and containing a **simulation purpose** $\langle Q, E, P, \rightsquigarrow, L, q_0 \rangle$. Each such element must:

- Specify a `relation` attribute containing the name of the satisfiability relation to use.

- Contain a child list `<states>` that defines the set Q of states.
- Contain a child list `<events>` that defines the set E of events.
- Contain a child list `<transitions>` that defines the relation $\rightsquigarrow$ of transitions.

```
<experiment name="Some name" description="Some more
   descriptive text." >

 <simulation-purpose-verification relation="feasibility">
  <states> (...) </states>
  <events> (...) </events>
  <transitions> (...) </transitions>
 </simulation-purpose-verification>

</experiment>
```

<states> Contains one or more `<state>` elements. Each such element must:

- Define an attribute `id` with a unique identifier. There is a reserved identifier, `initial`, which must be used to designate q_0 (i.e., the initial state).
- Zero or more child `<literal>` elements. Each such element must define a `type` attribute, which is either `positive` or `negative`, and a `proposition` attribute, which contains the name of a proposition.

```
<states>
 <state id="initial"/>
 <state id="1"/>
   <literal type="negative" proposition="P"/>
   <literal type="positive" proposition="Q"/>
 </state>
 <state id="second"/>
 <state id="3rd"/>
</states>
```

<events> Zero or more `<emmas-event/>` elements must be specified. Each one must specify the following attributes:

- `id`: a unique identifier.
- `type`: either `input` or `output`.

- `name`: one of the following: `emit`, `stop`, `beginning`, `stable`, `ending`, `absent`.
- `agent-id`: the identifier of the agent concerned with the event.

```
<events>
 <emmas-event id="!emit_beat_ag0" type="output" name="emit
    " action="beat" agent-id="0"/>
 <emmas-event id="!stop_beat_ag0" type="output" name="stop
    " action="beat" agent-id="0"/>
 <emmas-event id="?begin_candy_ag0" type="input" name="
    beginning" stimulus="candy" agent-id="0"/>
 <emmas-event id="?stable_candy_ag0" type="input" name="
    stable" stimulus="candy" agent-id="0"/>
 <emmas-event id="?ending_candy_ag0" type="input" name="
    ending" stimulus="candy" agent-id="0"/>
 <emmas-event id="?absent_candy_ag0" type="input" name="
    absent" stimulus="candy" agent-id="0"/>
</events>
```

<transitions> One or more `<transition/>` elements must be specified. Each one must specify the following attributes:

- `state-id1`: the identifier of a previously defined state.
- `event-id`: either the identifier of a previously defined event, or a one of the following pre-defined events: `other` (i.e., $\circledast$) or `tau` (i.e., τ).
- `state-id2`: the identifier of a previously defined state.

```
<transitions>
 <transition state-id1="initial" event-id="other" state-
    id2="initial"/>
 <transition state-id1="initial" event-id="!
    emit_caress_ag0" state-id2="1"/>
 <transition state-id1="1" event-id="?begin_candy_ag0"
    state-id2="2"/>
 <transition state-id1="2" event-id="other" state-id2="2"/
    >

 <transition state-id1="2" event-id="!emit_caress_ag0"
    state-id2="3"/>
 <transition state-id1="3" event-id="?begin_candy_ag0"
    state-id2="4"/>
 <transition state-id1="4" event-id="other" state-id2="4"/
    >
</transitions>
```

C.2 Parameters

There are mandatory and optional parameters. The mandatory are the following two:

- `-s FILE`: Specifies that FGS should use the specified `FILE` as the scenario definition.
- `-e FILE`: Specifies that FGS should use the specified `FILE` as the experiment definition.

The optional parameters are as follows:

- `-verbose L`: Specifies how much output is produced. In FGS, each output message has an associated importance. This option defines that only the messages with importance greater than or equal to `L` are to be shown. `L` is an integer between 0 (least important) and 4 (most important).
- `-max-depth D`: Defines the maximum depth `D` allowed during the depth-first construction of the **synchronous product** used in the verification algorithms. `D` is a positive integer.
- `-dont-randomize`: By default, FGS chooses uniformly at random between non-deterministic alternatives. If present, this option eliminates this randomization.
- `-max-synch-steps S`: Defines the maximum amount `S` of synchronizations to be tried during the verification algorithms. If this parameter is not specified, no such limit is imposed.
- `-debug`: Allows debug output to be shown.
- `-version`: Displays information about the version of FGS.
- `-help`: Displays a list of possible parameters.

APPENDIX D

Z Notation Overview

The Z Notation (ISO/IEC, 2002; Woodcock and Davies, 1996; Jacky, 1996) is a formal method based on set theory and first-order logic. This means that a number of operations over sets are assumed to exist, and first-order logic is used to write invariants. Z is designed for the specification of systems composed of states and transitions among states, although stateless definitions are also possible. It provides a calculus that allow the composition of complex specifications out of simple ones. Moreover, it has a characteristic associated visual convention that aims at facilitating the reading of specifications.

In this appendix we briefly present the main elements of the Z Notation, paying special attention to the ones we actually employ in our specification.

D.1 Types, Functions and Predicates

Z provides some primitive elements with which to define variables and invariants. Table D.1 presents some of the most important ones.

D.2 Stateless Definitions

Axiomatic definitions allow us to define global variables with associated invariants. For instance, a monotonically increasing function f could be defined as follows.

$$\begin{array}{|l} f : \mathbb{Z} \rightarrow \mathbb{Z} \\ \hline \forall\, x : \mathbb{Z} \bullet f(x) > x \end{array}$$

Types	
$\mathbb{Z}$	The set of integers.
$\mathbb{N}$	The set of natural numbers.
$[X]$	Defines X as a primitive set.
$\mathbb{P}\,X$	The power set of X.
$X \times Y$	The cross product of sets X and Y.
$X \rightarrow Y$	A function with domain X and range Y.
$X \nrightarrow Y$	A partial function with domain X and range Y.
Functions	
$\mathrm{dom}\, f$	The domain of function f.
$\mathrm{ran}\, f$	The range of function f.
$a \,\mathsf{div}\, b$	The division of integer a by integer b.
$*, +, -$	The usual arithmetical operations of multiplication, sum and subtraction, respectively.
$X \setminus Y$	The set X minus the set Y.
$\cup, \cap$	Set union and intersection, respectively.
$X \oplus Y$	Assumes X and Y to be binary relations and results in a binary relation that contains: all pairs of set X whose first element is not a first element in any pair in Y; plus all pairs of Y (i.e., pairs in Y override pairs in X).
θS	A tuple of values corresponding to the variables of schema S.
Predicates	
$x \,\underline{R}\, y$	(x, y) is in the relation R.
$\forall\, x_1 : X_n, \ldots, x_n : X_n \mid F \bullet P$	For all $x_1, \ldots, x_n$ of types $X_1, \ldots, X_n$ such that F holds, it must be the case that P holds as well.
$\exists\, x_1 : X_n, \ldots, x_n : X_n \mid F \bullet P$	There exists some $x_1, \ldots, x_n$ of types $X_1, \ldots, X_n$ such that F holds, and for which P holds as well.
$<, \leq, =, \neq, \geq, >$	The usual comparison predicates among integers.
$\in, \subset, \subseteq$	The usual set containment predicates.
$\wedge, \vee, \neg, \Rightarrow, \Leftrightarrow$	The usual logical connectives.

Table D.1: Some important types, functions and predicates of the Z Notation.

Enumerations are convenient ways of specifying finite domains. For example, the *Error* enumeration below defines two possible kinds of error.

$$Error ::= IllegalInput \mid OutOfMemory$$

D.3 State Schemas

In Z, states are specified using *schemas*. A schema has three main parts:

- A name;
- Variables declarations;
- Invariants over the variables.

For instance, to define a positive number we could specify a schema *PositiveNumber* with a variable x, of type $\mathbb{Z}$ (i.e., an integer), and an invariant that specifies that x must be greater than zero, as follows.

PositiveNumber

$$x : \mathbb{Z}$$

$$x > 0$$

A schema, then, is a tuple of variables whose elements must obey certain restrictions. A schema can also *import* other schemas. By so doing, the schema gains all the variables and the invariants of the imported schemas. For instance, to specify a positive and even number, we could import the schema we have just defined and add the parity invariant, as follows.

PositiveEvenNumber

PositiveNumber

$$\exists\, k : \mathbb{Z} \bullet x = 2 * k$$

This is equivalent to the following schema.

PositiveEvenNumber

$$x : \mathbb{Z}$$

$$x > 0$$

$$\exists\, k : \mathbb{Z} \bullet x = 2 * k$$

We can also employ a schema as a type. For example, to define a triple of different positive numbers, we could have this schema:

$$
\begin{array}{|l}
\hline
\textit{PositiveTriple} \\
a : \textit{PositiveNumber} \\
b : \textit{PositiveNumber} \\
c : \textit{PositiveNumber} \\
\hline
a.x \neq b.x \\
b.x \neq c.x \\
c.x \neq a.x \\
\hline
\end{array}
$$

Notice how we selected the x variable of each schema in order to state the invariants.

D.4 Operation Schemas

Schemas can also describe transitions between states, which in Z we call *operations*. In order to do so, there are some conventions that the schema must follow. Thus, besides the basic structure that we described for states, an operation also presents the following characteristics:

- A declaration of which schemas are undergoing a transition;
- Input variables are followed by a question mark (e.g., $x?$) and output variables are followed by an exclamation mark (e.g., $x!$);
- Variables before the transition are denoted by their usual names, while variables after the transition are denoted by their names followed by a prime (e.g., x').

Consider the following operation, which increases the value of a positive number.

$$
\begin{array}{|l}
\hline
\textit{PositiveIncrementOp} \\
\Delta \textit{PositiveNumber} \\
\textit{increment}? : \mathbb{Z} \\
\textit{sum}! : \mathbb{Z} \\
\hline
\textit{increment}? > 0 \\
x' = x + \textit{increment}? \\
\textit{sum}! = x' \\
\hline
\end{array}
$$

The $\Delta PositiveNumber$ declaration states that this is an operation that changes the value of a *PositiveNumber* state schema. The *increment*? variable is an input. There is an associated invariant that requires this input to be positive. This is called a *pre-condition*, because it is something that must be true before the operation is applied. The effect of the operation, in turn, is given by *post-conditions*. In this case, the post-conditions state that the value of x after the operation will be the previous value plus the increment. Finally, an invariant states that the output variable *sum*! will also hold the new value of x.

D.5 Schema Calculus

It is possible to compose complex schemas out of simple ones by using the *schema calculus*, which provide a number of logical operators for this. For example, let us specify what happens when the pre-condition for the operation *PositiveIncrementOp* fails.

$$
\begin{array}{|l}
\hline
\textit{PositiveIncrementOpError} \\
\Delta \textit{PositiveNumber} \\
\textit{increment}? : \mathbb{Z} \\
\textit{error}! : \textit{Error} \\
\hline
\textit{increment}? \leq 0 \\
x' = x \\
\textit{error}! = \textit{IllegalInput} \\
\hline
\end{array}
$$

This merely states that if the increment specified is less than or equal to zero, an error is generated and the value of x is not changed. We can then compose the two operations using the disjunction connective of the schema calculus.

$$T_PositiveIncrementOp \mathrel{\widehat{=}} PositiveIncrementOp \lor PositiveIncrementOpError$$

This states that either one operation or the other will take place. Notice that their preconditions are complementary. This means that every possible case for the input variable has been considered. Operations that have this property are called *total* (thus the "$T_$" prefix). Whenever possible, it is interesting to have such total operations.

Other connectives of the schema calculus include the usual logical connectives, such as conjunction ($\wedge$), implication ($\Rightarrow$), and negation ($\neg$).

D.6 Refinement

Specifications can be written at various levels of abstraction. The more abstract a specification is, the less restrictions it imposes, and the more choices an implementer has. Given a specification, it is possible to make a more concrete one by adding new restrictions to it, but in a way that the original specification still holds. Such a concrete specification is called a *refinement* of the original one. More formally, a predicate C refines a predicate A if, and only if, C implies A (Jacky, 1996).

When refining stateless definitions or state schemas without changing their variables, this simple definition suffices. However, in Z two more specialized kinds of refinements exist, namely, operation and data refinement. The former is concerned with making schema operations more precise, and the latter aims at making abstract data types more concrete by transforming variables in more detailed data structures (e.g., in order to find a suitable programming construct for implementation). In this thesis, of these two kinds we only employ operation refinement, thus here we only consider it.

Following Spivey (1992), for an operation C_{op} to refine an operation A_{op} two things are required. First, whenever the pre-conditions of A_{op} are true, the pre-conditions of C_{op} must be true as well. That is to say, C_{op} must be applicable at least in the same situations in which A_{op} is applicable, but it may also be applicable in other cases. Second, if the pre-conditions of A_{op} hold and C_{op} is applied, then at least the post-conditions specified by A_{op} must hold (besides any extra post-conditions defined by C_{op}). In other words, whenever both operations are applicable, C_{op} can have more effects than A_{op}, but not less.

Definition D.1 (Operation Refinement)**.** *Let C_{op} and A_{op} be schema operations over state State, with inputs $x_1? : X_1, \ldots, x_m? : X_m$ and outputs $y_1! : Y_1, \ldots, y_n! : Y_n$. Then C_{op}* refines *A_{op} if, and only if, the following conditions hold:*

- $\forall\, State;\ x_1? : X_1;\ \ldots;\ x_m? : X_m \bullet \mathrm{pre}(A_{op}) \Rightarrow \mathrm{pre}(C_{op})$

- $\forall\, State;\ State';\ x_1? : X_1;\ \ldots;\ x_m? : X_m;\ y_1! : Y_1;\ \ldots;\ y_n! : Y_n \bullet \mathrm{pre}(A_{op}) \wedge C_{op} \Rightarrow A_{op}$

where $\mathrm{pre}(Op)$ *denotes the pre-conditions of operation schema* Op*.*

In the previous section we saw how the schema calculus can be used to decompose a specification in simpler parts. However, it is important to note that the schema calculus has a serious limitation: it is not monotonic with respect to refinement. That is to say, if two schemas (e.g., A and B) are connected in a schema calculus expression (e.g., $A \wedge B$), it cannot be guaranteed that refinements of these schemas (e.g., A_{ref} and B_{ref}) can be used to refine the expression (e.g., it may be the case that $A \wedge B$ is not refined by $A_{ref} \wedge B_{ref}$). Therefore, one must be specially careful when refining Z specifications that employ the schema calculus (Groves, 2002).

APPENDIX E

π-calculus Overview

This appendix presents a brief account of the π-calculus . Our objective is not to teach the calculus, but merely to quickly recall the notions that we employ to accomplish our work. The definitions we present are adapted from Parrow (2001), which the reader might also find useful as an introduction to the calculus.

The π-calculus is a process algebra designed to model interaction and mobility of *processes*[1]. To do so, it provides an algebraic language in which to write such processes, as well as a mathematical framework that interprets them in terms of Labeled Transition Systems (LTS). Let us then begin by defining what an LTS is.

Definition E.1 (Labelled Transition System)**.** *A* labeled transition system *is a tuple* $\langle S, L, \rightarrow \rangle$ *such that:*

- S *is a set of* states*;*
- L *is a set of* labels*;*
- $\rightarrow \in S \times L \times S$ *is a* transition *relation.*

Moreover, let $s_1, s_2 \in S$ *and* $l \in L$*. Then, if* $\langle s_1, l, s_2 \rangle \in \rightarrow$*, we also denote this fact by writing* $s_1 \xrightarrow{l} s_2$*. The opposite fact, in turn, is denoted by* $\neg\ (s_1 \xrightarrow{l} s_2)$*.*

[1] In the literature, π-calculus processes are often called "agents". We avoid using this terminology in order to do not confuse it with our own notion of agents.

Processes are written by using *names* to create prefixes and by using several operators to combine such prefixes. These prefixes represent *events*[2].

Definition E.2 (π-calculus Process)**.** *Let a, x, y, $x_1, \ldots, x_n$ and $y_1, \ldots, y_n$ be names. Then a* π-calculus process *is an expression defined by the following syntax.*

Prefixes			
α	$::=$	$\overline{a}\langle x\rangle$	*Output*
		$a(x)$	*Input*
		τ	*Internal*
Processes			
P	$::=$	$\boldsymbol{0}$	*Nil*
		$\alpha.P$	*Prefix*
		$P + P$	*Choice*
		$P \mid P$	*Parallel composition*
		$(\nu x)P$	*Restriction*
		$[x = y]P$	*Match*
		$[x \neq y]P$	*Mismatch*
		$!P$	*Parallel replication*
		$A(y_1, \ldots, y_n)$	*Identifier*
Definitions			
		$A(x_1, \ldots, x_n) = P$	*Process definition* $\quad (i \neq j \Rightarrow x_i \neq x_j)$

Given a a process P, we denote the set of its *bound names* by $bn(P)$, and of its *free names* by $fn(P)$. Moreover, we denote by $\mathscr{P}^{\pi}$ the set of all processes.

Names often need to change over the course of a process execution. This is achieved using *substitution functions.*

Definition E.3 (Substitution Function)**.** *Let $x_1, \ldots, x_n$ and $y_1, \ldots, y_n$ be names, and P be a process. Then a* substitution function

$$\{x_1, \ldots, x_n / y_1, \ldots, y_n\}$$

maps process P into P', such that in P':

- *For all $y_i \in fn(P)$, x_i will substitute y_i in P';*
- *Alpha-conversion is performed in order to prevent that $x_i \in bn(P')$.*

Moreover, we denote the application of the substitution function on P as:

$$P\{x_1, \ldots, x_n / y_1, \ldots, y_n\}$$

[2]In the literature, such events are often called "actions". Again, we avoid using this terminology in order to prevent confusion with our own notion of actions.

It is often the case that processes which are syntactically different should have the same behavior. To model this, the calculus provides a *structural congruence* relation, which defines equivalences that can be determined by syntax alone. This is useful, in particular, to fully define the replication operator.

Definition E.4 (π-calculus Structural Congruence)**.** *Let P, Q and R be arbitrary π-calculus processes. Then the* structural congruence *relation $\equiv$ is the smallest relation that satisfies the following axioms.*

- *If P and Q are variants by alpha-conversion, then $P \equiv Q$*
- $P \mid Q \equiv Q \mid P$, $(P \mid Q) \mid R \equiv P \mid (Q \mid R)$, $P \mid \boldsymbol{0} \equiv P$
- $P + Q \equiv Q + P$, $(P + Q) + R \equiv P + (Q + R)$, $P + \boldsymbol{0} \equiv P$
- $!P \equiv P \mid !P$
- *Scope extension laws:*
  - $(\nu x)\boldsymbol{0} \equiv \boldsymbol{0}$
  - $(\nu\, x)(P \mid Q) \equiv P \mid (\nu\, x)Q$ *if* $x \notin fn(P)$
  - $(\nu\, x)(P + Q) \equiv P + (\nu\, x)Q$ *if* $x \notin fn(P)$
  - $(\nu\, x)(\nu\, y)P \equiv (\nu\, y)(\nu\, x)P$
  - $(\nu\, x)[u = v]P \equiv [u = v](\nu\, x)P$ *if* $x \neq u$ *and* $x \neq v$
  - $(\nu\, x)[u \neq v]P \equiv [u \neq v](\nu\, x)P$ *if* $x \neq u$ *and* $x \neq v$

The behaviour of processes is given by an operational semantics. That is to say, a number of rules that define how algebraic expressions should be translated to LTSs.

Definition E.5 (π-calculus Operational Semantics)**.** *Let P, P', Q and Q' be processes, α be a prefix, and a, x and u be names. Then the* operational semantics of the π-calculus *is given by the following rules.*

$$\frac{P' \equiv P \quad P \xrightarrow{\alpha} Q \quad Q' \equiv Q}{P' \xrightarrow{\alpha} Q'}\ \text{STRUCT} \qquad \frac{}{\alpha.P \xrightarrow{\alpha} P}\ \text{PREFIX}$$

$$\frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'}\ \text{SUM} \qquad \frac{P \xrightarrow{\alpha} P' \quad bn(\alpha) \cap fn(Q) = \emptyset}{P \mid Q \xrightarrow{\alpha} P' \mid Q}\ \text{PAR}$$

$$\frac{P \xrightarrow{a(x)} P' \quad Q \xrightarrow{\overline{a}\langle u\rangle} Q'}{P \mid Q \xrightarrow{\tau} P'\{u/x\} \mid Q'}\ \text{COM} \qquad \frac{P \xrightarrow{\alpha} P' \quad x \notin \alpha}{(\nu x)P \xrightarrow{\alpha} (\nu x)P'}\ \text{RES}$$

$$\frac{P \xrightarrow{\overline{a}(x)} P' \quad a \neq x}{(\nu x)P \xrightarrow{\overline{a}\nu x} P'}\ \text{OPEN} \qquad \frac{P \xrightarrow{\alpha} P'}{[x = x]P \xrightarrow{\alpha} P'}\ \text{MATCH}$$

$$\frac{P \xrightarrow{\alpha} P' \quad x \neq y}{[x \neq y]P \xrightarrow{\alpha} P'}\ \text{MISMATCH}$$

Notice that on all of these definitions, prefixes can have only one parameter. It is possible, however, to have prefixes with multiple parameters (the so called *polyadic* π-calculus) and define them in terms of these simple ones. It is this polyadic notation that we use in this thesis.

APPENDIX F

Extended Abstracts

This appendix contains longer versions of the thesis' abstracts in English, in Portuguese and in French.

F.1 Extended Abstract

Multi-Agent Systems (MASs) can be used to model phenomena that can be decomposed into several interacting agents, which exist within an environment. In particular, they can be used to model human and animal societies, for the purpose of analysing their properties by computational means. This thesis addresses the problem of automated analysis of a particular kind of such social models, namely, those based on behaviourist principles, which contrasts with the more dominant cognitive approaches found in the MAS literature. The hallmark of behaviourist theories is the emphasis on the definition of behaviour in terms of the interaction between agents and their environment. In this manner, not merely reflexive actions, but also learning, drives, and emotions can be defined. The thesis proposes a verification technique that investigates such MASs by means of guided simulations. This is achieved by modelling the evolutions of an MAS as a transition system (implicitly), and the property to be verified as another transition system (explicitly). The former is derived (on-the-fly) from a formal specification of the MAS's environment. The latter, which we call a ***simulation purpose***, is used both to verify the property and to guide the simulation. In this way, only the states that are relevant for the property in question are actually simulated. Algorithmically, this corresponds to building a synchronous product of these two transitions systems on-the-fly and using it to operate a simulator. Figure F.1 shows the most important elements of the proposed approach. In what follows we sum-

SIMULATION PURPOSE

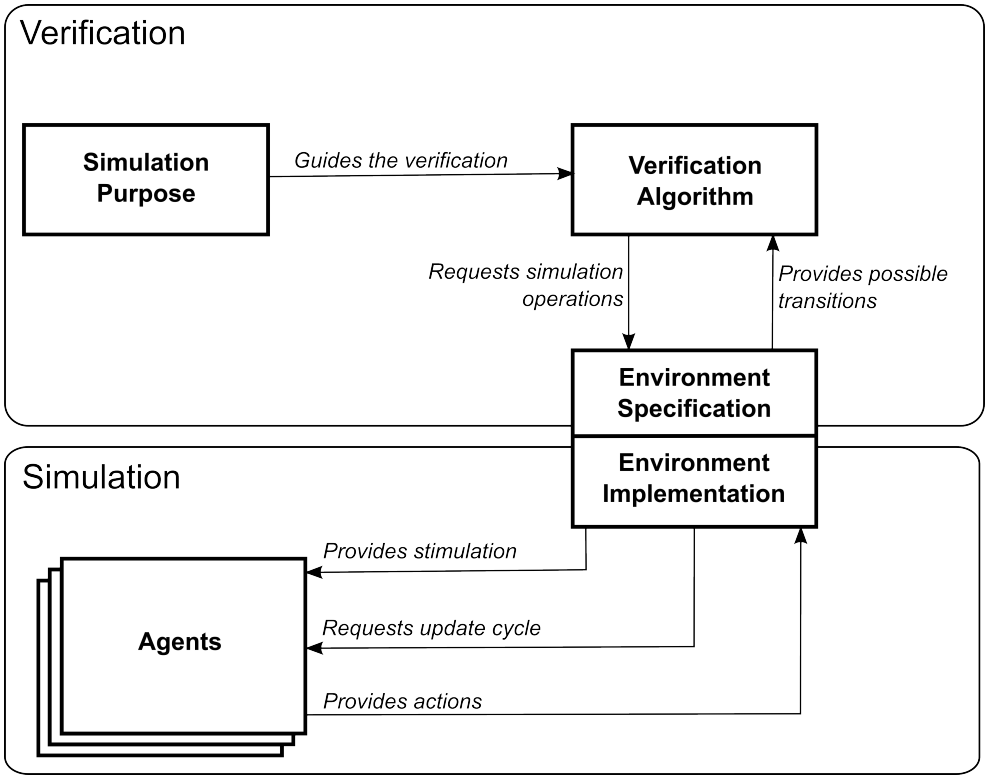


Figure F.1: Verification and simulation elements interaction. Notice, in particular, the important role that the environment has in relating verification and simulation. It acts as a coordinator which, on the one hand, formally defines what can be done, while on the other hand requests actual simulator operations.

marize the several parts of this work, taking special care to show how one goes from informal ideas to formalizations, and then to the actual implementation.

Agents

To describe an MAS, one needs specific notions of agents and environments. With respect to agents, much work has been done in trying to understand and model so-called *intelligent* and *cognitive* agents. These approaches focus largely on what constitute rational decisions, specially in the case of agents with limited computing capabilities (e.g., all of us). The *Beliefs-Desires-Intentions* (BDI) architecture (Bratman, 1987; Cohen and Levesque, 1990; Rao and Georgeff, 1995) is a well-known example.

Behaviour of organisms, however, is sometimes better described in different

terms. A dog does not reason that it will die if it does not eat[1]; rather, it has a drive to seek food when hungry. If it has learned that whenever his master whistles he provides food, the dog will salivate at the sound of the whistle – without thinking. These observations suggest that a different focus in agent modelling is possible. In this thesis we consider such a model, based on the psychology theory known as Behaviour Analysis (Skinner, 1953). In this theory, the actions of agents are seen as the result of past stimulation and certain innate parameters according to behavioural laws. The focus is not in mental qualities such as the nature of reason, but merely in the prediction and control of behaviour by means of environmental stimulation. This point of view, though classical within psychology, is scarce in MAS literature. As a contribution in this sense, this thesis introduces the ***Behaviourist Agent Architecture***.

Behaviourist Agent Architecture

This architecture defines the main parts of agents so that they comply with such behaviourist principles. Its structure follows core principles of Behaviour Analysis, which we organize in five classes: (i) stimulus conditioning; (ii) respondent behaviour (i.e., reflexes); (iii) operant behaviour; (iv) drives; and (v) emotions. These elements work in a coherent manner in order to allow adaptive and learning behaviour.

The Z Notation is used to formalize the architecture. This provision ensures that it is defined in a precise and compositional form. The benefits of precision are evident. But compositionality should also be valued, for it allows each part of the specification to be examined and modified separately, and thus allows further progresses to be made upon it. Indeed, thanks to the possibility of refining Z specifications, one may add new refinements to the architecture in order to specialize it.

The architecture is implemented in Java. In order to instantiate an agent, it suffices to create a new `Organism` object and initialize it with a special kind of XML file, in which several agent parameters must be defined (e.g., the stimuli it recognizes, the actions it can perform, the reflexes).

In a complete formal development approach, this Java program should be proved correct with respect to the Z specification (e.g., by means of formal refinements). This was not done in this thesis, where the main formal effort has been devoted to formally guided simulations and the related verification algorithms, which are carefully shown to be correct. However, the Java implementation of the agent architecture follows very closely the structure of its Z specification, and thus it is likely to be correct.

[1] Assuming, of course, that dogs cannot foresee their own deaths in the same way that we humans can.

Environments

In comparison with agents, environments of MASs have received comparatively very little attention, as the survey of Weyns *et al.* (2005) points out. The environment model of Ferber and Müller (1996) is one exception. In this thesis we propose the ***Environment Model for Multi-Agent Systems (EMMAS)***, which is designed to work with the ***Behaviourist Agent Architecture***. Since the psychology theory from which we draw from puts great emphasis in the relation between agents and their environment, this is a quite important aspect of the MASs being considered.

ENVIRONMENT MODEL FOR MULTI-AGENT SYSTEMS (EMMAS)
BEHAVIOURIST AGENT ARCHITECTURE
AGENT PROFILES

In **EMMAS**, agents are represented by ***agent profiles***. Such a profile assigns an identifier to an agent and defines the stimuli it recognizes as well as the actions it can emit. This provides the necessary information to define an **EMMAS** environment. Nonetheless, in order to execute the resulting MAS (e.g., to simulate it), it is necessary to have an underlying implementation for these **agents profiles**, such as the one we considered above.

The aim of the MASs used in this thesis is to model social systems that exist in the real world, where agents have a physical position in space. One way to address this, often used in MASs simulations, is to provide a simplification of the physical space, such as a two dimensional grid in which each cell is a possible position. However, in **EMMAS** we adopt a different abstraction. Instead of representing the physical position of agents, we represent their relationships. That is to say, the MAS is viewed as a social network. In this manner, we are able to focus on the social relations among agents, which may be quite independent of the physical conditions to which they are subject to. For example, the fact that agent ag_1 can reward agent ag_2 by praising him holds independently of whether they are in the same room or communicating through the Internet.

ACTION TRANSFORMERS

Given the behaviourist point of view that we adopt, these relationships are modelled by defining how the actions of an agent are transformed in stimuli for other agents by means of ***action transformers***. An agent is related to another if it can stimulate the other in this manner. This intermediary element between the action of an agent and the stimulation received by another is justified by the fact that an action can have several different concurrent consequences. Indeed, the same action by an agent can be perceived as different stimuli by different agents.

EMMAS allows the dynamic creation and destruction of these **action transformers**. The importance of this is twofold. First, it allows the specification of phenomena in which the relation among agents change as they age. Second, it allows specification of several possible network structures for the same environment (i.e., the description of a class of social networks, and not one

particular social network).

Although the relationships between agents can change, agents themselves cannot be created nor destroyed in **EMMAS**. A reason for this is that, to be created, an agent following the **Behaviourist Agent Architecture** must be parametrized *a priori* in a detailed manner before becoming part of the MAS. But since in **EMMAS** agents are represented as **agent profiles**, this parametrization can not be done within **EMMAS** specifications themselves. It is nevertheless possible to emulate creation and destruction of agents in **EMMAS**. To do so, it suffices to define a pool of agents, initially not related to any other, and then manipulate how their actions affect the environment (and thus other agents). An agent whose actions are entirely ignored and which receives no stimuli is effectively irrelevant in the MAS, and therefore can be considered as non-existing.

An environment defines the context in which the agents exist, which is more than merely setting initial conditions. It includes behaviours pertaining to the environment itself, which may be executed either in response to an agent's actions or independently of any such action. From the simulation and verification point of view adopted in this thesis, this environmental context can be seen as an experimental setup, which defines all the possible experiments that can be performed in the agents. One may provide **environment operations** in **EMMAS** in order to define such an experimental setup.

Given an MAS meant for simulation and verification, a crucial point concerns how it evolves. That is to say, how it changes from one state to another until the end of the procedure being applied (e.g., a simulation). Since in this thesis we explicitly separate the formalizations of agents and environments, we are led to give different accounts for the evolutions of each:

- *Agents states.* The **Behaviourist Agent Architecture** defines how any agent state is transformed in a new agent state by the provision of environmental stimulation. This definition is specified using the Z Notation.
- *Environment states.* **EMMAS** specifications have a semantics in terms of the π-calculus process algebra Milner (1999); Parrow (2001). Hence, each state of the environment corresponds to a π-calculus expression P, together with contextual information (a triple called ***environment status***) that constrain the possible successors to the state:

ENVIRONMENT STATUS

$$(P, \langle \mathit{Stimulation}, \mathit{Response}, \mathit{Literals} \rangle)$$

The environment states have contextual information (functions Stimulation, Response, and set Literals) that relate them to the agents states, thereby

incorporating all the observable information relevant to the MAS. That is to say, environment states are global states in so far as the observable evolutions of the MAS are concerned. It suffices then to provide a semantics for **EMMAS** in order to have the semantics of the MAS itself.

EMMAS Semantics

The semantics of **EMMAS**, as said above, is given in terms of the π-calculus process algebra. Process algebras are typically employed to describe concurrent systems. They are good at succinctly describing behaviours relevant to inter-process communication. The particular choice of π-calculus as a theoretical foundation is motivated by some of its features, which together make it a distinguished formalism among existing such algebras. First, it takes communication through channels as a primitive notion, which makes it a natural choice for representing networks. Second, it allows for dynamic modification, which makes the creation and destruction of connections between agents possible. Third, it provides a convenient representation for broadcast behaviour through its replication operator. Finally, it has few operators and a simple operational semantics. Owing to this operational semantics, we can give an ***annotated transition system (ATS)*** that denotes all the possible evolutions of the MAS.

ANNOTATED TRANSITION SYSTEM (ATS)

The semantics of **EMMAS** is actually given in two stages, by considering: (i) a syntactical translation of **EMMAS** into π-calculus expressions through a **translation function** $[\,]_\pi$; (ii) a mathematical foundation which relates π-calculus events to the stimuli and actions of agents in a transition system. The π-calculus translation of (i), through its operational semantics, provides an over-approximation of the desired behaviour, which is then made precise using the restrictions provided by (ii). By this method, we are able to define an ***environment ATS*** (possibly infinite) that establishes the possible states and transitions for any particular environment specification.

ENVIRONMENT ATS

The semantics thus achieved is general and is not tied to any particular application, not even simulation. For the purposes of the verification technique, however, it will be necessary to carry out stage (ii) in a slightly more specific manner, so that the result can be used in a simulator. In short, the difference is that we introduce a new event in the **environment ATS**, called *commit*, which makes the computation of runs more efficient and explicit during simulations (an example is given below).

To implement **EMMAS**, we have developed a π-calculus library in Java, which allows the construction of the required π-calculus processes and the application of its operational semantic rules. This library follows the definition of π-calculus closely, which allows a direct mapping between π-calculus

elements and their implementation.

Each sequence of states and events (called a **run**) in an **environment ATS** corresponds to one possible global evolution of the MAS. By considering different **runs**, one investigate different possibilities of global evolutions of the MAS, and therefore one may look systematically for certain properties of interest. This is the basis on which relies the verification technique developed in this thesis.

As an example of such a **run**, consider the following one:

$$s_1 \overset{!beginning_u^0}{\rightarrow} s2 \overset{!beginning_v^0}{\rightarrow} s_3 \overset{!commit}{\rightarrow} s_4 \overset{?emit_a^0}{\rightarrow} s_5$$

where s_1, s_2, s_3, s_4 and s_5 are states of the **environment ATS**, and $!beginning_u^0$, $!beginning_v^0$ and $?emit_a^0$ denote events relative to an agent identified by 0, stimuli u and v and an action a (i.e., stimulation and action events concerning an agent). During a simulation, this run is interpreted as follows: first, schedule events $!beginning_u^0$ and $!beginning_v^0$, but do not pass them yet to agent 0. Then, since a $!commit$ is found, all scheduled events are taken into account. In the case of the two events we have scheduled, it means that the agent implementation will be notified that it is now receiving stimuli u and v. Then the event $?emit_a^0$ happens, and this means that in state s_4 the agent implementation has notified the simulator that agent identified by 0 is actually emitting action a.

Simulation Purposes and Satisfiability Relations

One models an MAS in order to study its properties. In this thesis, we propose a way to do so by formulating hypotheses about the MAS and checking whether they hold or not (e.g., "every time the agent does X, will it do Y later?"). If a hypothesis does not hold, it means that either the hypothesis is false or the MAS has not been correctly specified. The judgement to be made depends of our objectives in each particular circumstance. Are we trying to discover some law about the MAS? In this case, if a hypothesis that represents this law turns out to be false, it is the hypothesis that is incorrect, not the MAS. Are we trying to engineer an MAS that obey some law? In this case we have the opposite, a falsified hypothesis indicates a problem in the MAS. This view is akin to that found in empirical sciences, in which scientists investigate hypotheses and make judgements in a similar manner. In this respect, the main difference is that the empirical scientist studies the natural world *directly*, while we are concerned with *models* of nature in the form of MASs.

In this thesis, such a hypothesis is defined by specifying a **simulation purpose** and a satisfiability relation. If the MAS satisfies the specified **simulation purpose** with respect to the desired satisfiability relation, then the

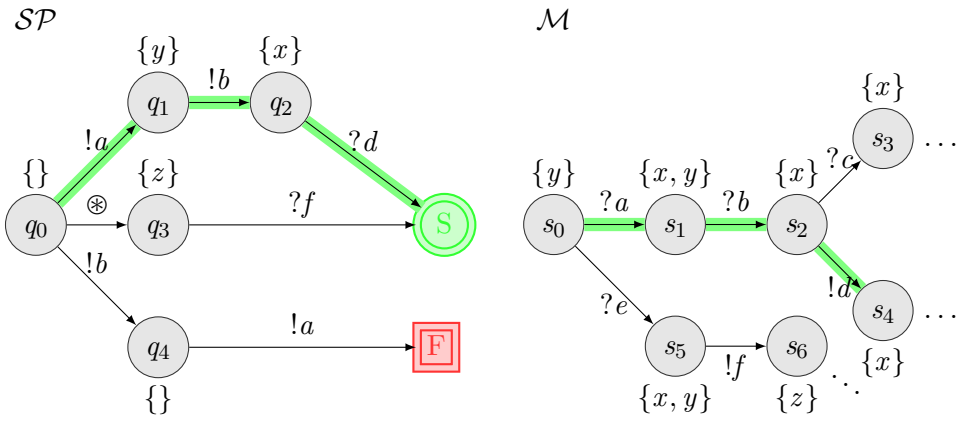


Figure F.2: Only the shaded **runs** in the **ATS** $\mathcal{M}$ and the **simulation purpose** $\mathcal{SP}$ can **synchronize**. States are annotated with propositions and transitions with events. There are rules that determine how events and states **synchronize**. The state labelled with S is *Success*, and the one labelled with F is *Failure*. The dots $(\ldots)$ denote that $\mathcal{M}$ continues beyond the states shown (it is possibly infinite).

hypothesis is corroborated. Otherwise, it is falsified. The idea of using such a **simulation purpose** is inspired by the TGV (Jard and Jéron, 2005) approach to model-based testing, in which formal *test purposes* are used to select relevant test cases. Here, a formal **simulation purpose** is used to select relevant simulations executions, though of course the criteria of relevance, among other technicalities, are quite different.

Formally, a **simulation purpose** is an **ATS** subject to further restrictions. In particular, it is finite and defines two special states, *Success* and *Failure*. All **runs** that lead to *Success* denote desirable simulations, whereas all that lead to *Failure* denote undesirable ones.

SYNCHRONOUS PRODUCT

The satisfiability relations, in turn, require the introduction of another technical definition, namely, the notion of ***synchronous product***. Given an **ATS** $\mathcal{M}$ that models an MAS, and a **simulation purpose** $\mathcal{SP}$, their **synchronous product** (denoted by $\mathcal{SP} \otimes \mathcal{M}$) is another **ATS** in which every **run** represents a possible evolution of $\mathcal{M}$ which has been allowed by the $\mathcal{SP}$. Each state in $\mathcal{SP} \otimes \mathcal{M}$ takes the form of (q, s), where s is a state of $\mathcal{M}$, and q is a state of $\mathcal{SP}$. Figure F.2 shows an example of $\mathcal{SP}$, $\mathcal{M}$ and **runs** that **synchronize** to form $\mathcal{SP} \otimes \mathcal{M}$.

These **runs** may terminate in a state (q, s) where $q = \mathit{Success}$ or $q = \mathit{Failure}$, meaning that the **run** in question is desirable or undesirable, respectively. Different satisfiability relations are defined, namely:

FEASIBILITY

- ***Feasibility***: $\mathcal{SP}$ is **feasible** with respect to $\mathcal{M}$ if there is at least one

run in $\mathcal{SP} \otimes \mathcal{M}$ which terminates in a state (q, s) such that $q = \mathit{Success}$. There are weak and strong variants of this.

- ***Refutability***: $\mathcal{SP}$ is **refutable** with respect to $\mathcal{M}$ if there is at least one **run** in $\mathcal{SP} \otimes \mathcal{M}$ which terminates in a state (q, s) such that $q = \mathit{Failure}$. There are weak and strong variants of this. REFUTABILITY

- ***Certainty***: $\mathcal{SP}$ is **certain** with respect to $\mathcal{M}$ if all **runs** in $\mathcal{SP} \otimes \mathcal{M}$ terminate in a state (q, s) such that $q = \mathit{Success}$. CERTAINTY

- ***Impossibility***: $\mathcal{SP}$ is **impossible** with respect to $\mathcal{M}$ if all **runs** in $\mathcal{SP} \otimes \mathcal{M}$ terminate in a state (q, s) such that $q = \mathit{Failure}$. IMPOSSIBILITY

Verification Algorithms

These satisfiability relations define conditions that one may require of the MAS, but they do not specify the computation to be used. Granted, they are rather operational definitions, but the details of how to perform the **synchronous product** are abstracted away. It is therefore also necessary to have algorithms to check whether they hold or not. Moreover, despite the fact that there are several satisfiability relations, their verification is carried out in a similar manner. All the algorithms have the following main characteristics:

- They perform a depth-first search on the **synchronous product** of $\mathcal{SP}$ and $\mathcal{M}$.
- The search has a maximum depth, depth_{max}.
- $\mathcal{SP} \otimes \mathcal{M}$ is computed on-the-fly (i.e., it is not computed *a priori*; rather, at each state, the algorithm calculates the next states necessary to continue), because $\mathcal{M}$ itself is obtained on-the-fly from the π-calculus expressions present in each of its states.
- A simulator interface is assumed to exist. This is used to control the simulation execution, including the possibility of storing simulator states and backtracking to them later.

What the search is looking for is what changes from one algorithm to another. In the case of **feasibility**, the objective is to find one **run** that leads to *Success*, thus showing an example of how to perform a successful simulation. In **certainty**, it is to find one **run** that leads to *Failure*, and therefore provides a counter-example (if no such **run** is found, it means that **certainty** holds with respect to the available observations). The other satisfiability relations are checked analogously to these. Furthermore, if the verification of a

satisfiability relation requires a search depth greater than $depth_{max}$, then the result of the algorithm is *inconclusive*, instead of *Success* or *Failure*.

The complexities of the algorithms must be given considering the fact that they perform a depth-first search on a possibly infinite transition system that is built on-the-fly (i.e., $\mathcal{SP} \otimes \mathcal{M}$) and only up to a maximum depth (i.e., $depth_{max}$). This means that the complexities must be given mainly in terms of $depth_{max}$ and the maximum branching factor (i.e., the maximum number of possible successors of any state), instead of the number of states and transitions in the complete transition system, as it would be the case if all of these states and transitions were to be explored. Moreover, since states in $\mathcal{M}$ are actually computed from an **EMMAS environment** specification, the complexities of these computations must be taken in account as well. These characteristics lead to many parameters to be accounted for in the statement of the complexities. The complete development is given in the thesis. In a few words, the complexity in space is polynomial with respect to the size of the **environment** and other parameters, and the complexity in time is exponential with respect to $depth_{max}$.

What is important in this technique is that, once given a **simulation purpose**, it chooses which simulations to execute automatically and in a systematic manner, instead of depending on a user to guide and inspect the simulation manually, thereby exploring the possible simulations more efficiently, even if inconclusively. Moreover, the algorithms are carefully justified to be correct according to precise notions of soundness and completeness.

Simulator

These algorithms have been implemented in a tool called Formally Guided Simulator (FGS), which works as follows. It keeps a repository of components, among which is the Java implementation of the **Behavioural Agent Architecture**. To request the verification of an MAS, three kinds of XML files must be provided to FGS:

- One or more parameterizations to the agents present. These parameterizations are used to instantiate the implementation of the **Behaviourist Agent Architecture**.
- One scenario description, in which the agents are declared to exist, and the **EMMAS** specification is provided.
- One experiment description, in which a **simulation purpose** and the satisfiability relation to be checked are specified.

Once FGS is invoked, it first uses the scenario description to instantiate the **Behavioural Agent Architecture**, so that the relevant agents become available to the simulator. It then transforms the **EMMAS** specification provided in the scenario description in a π-calculus expression. This expression is implemented directly by instantiating classes from the π-calculus simulation library according to the implementation of the **translation function**. This corresponds to the initial state of the MAS to be simulated.

FGS then reads the experiment description in order to obtain the **simulation purpose** and the satisfiability relation to be checked. The **simulation purpose** is implemented trivially, since its structure is very simple. The satisfiability relation, in turn, determines which verification algorithm should be used.

After all this has been done, FGS merely executes the requested verification algorithm. To calculate the next states in the **ATS** implementation, it uses the π-calculus operational semantics rules which are implemented in the π-calculus library. The state of the agents are changed and inspected by manipulating the objects that instantiate them. This information is used to annotate states, and apply the relevant constraints.

There is a special event, *commit*, which signals that agents should receive stimulation and provide behavioural responses. This is implemented by FGS as follows. Events are stored as they are found in a **run** of the **ATS**, but are not delivered to the agents. When a commit event is found, then the stored events are delivered to the relevant agents, and they may also change whether they are emitting or not emitting their actions.

If the appropriate options have been set, FGS will output information about every synchronization made in the **synchronous product**, which allows the visualization of the simulations. When the algorithm terminates, the verdict is shown, and a **run** is displayed as well if relevant (e.g., the **feasible run** that led to *Success*).

Conclusion

This thesis proposes a way to model a class of MASs, as well as a related framework for their simulation and verification. This framework is based on formal descriptions of agents and their environments, and formal definitions of **simulation purposes**. These elements are used to perform simulations in a systematic and guided manner in order to determine whether certain precisely defined satisfiability relations hold or not. It is thus a significant step forward in bridging the gap between simulation and verification techniques. It must be said that some transformations (mainly related to programming the simulator) are not completely verified. Nonetheless, the thesis brings a novel and wide

framework for some automated and formally-based analyses of MASs.

F.2 Resumo Estendido

Sistemas Multi-Agentes (SMAs) podem ser usados para modelar fenômenos que podem ser decompostos em vários agentes que interagem dentro de um ambiente. Em particular, eles podem ser usados para modelar sociedades humanas e animais, com a finalidade de analisar as suas propriedades através de meios computacionais. Esta tese aborda o problema de análise automatizada de um tipo particular de tais modelos sociais, a saber, aqueles baseados em princípios comportamentalistas, os quais contrastam com as abordagens cognitivas dominantes na literatura sobre SMAs. A principal característica das teorias comportamentalistas é a ênfase na definição do comportamento em termos da interação entre os agentes e seu ambiente. Desta forma, não apenas ações reflexivas, mas também aprendizagem, motivações, e emoções podem ser definidas. A tese propõe uma técnica de verificação que investiga tais SMAs por meio de simulações guiadas. Isso é feito através da modelagem das evoluções de um SMA como um sistema de transição (implicitamente), e da propriedade a ser verificada como um outro sistema de transição (explicitamente). O primeiro é derivado (em tempo de execução, *on-the-fly*) de uma especificação formal do ambiente do SMA. O segundo, que chamamos de ***propósito de simulação***, é utilizado tanto para verificar a propriedade quanto para guiar a simulação. Dessa forma, somente os estados que são relevantes para a propriedade em questão são simulados. Algoritmicamente, isto corresponde a construir um produto síncrono desses dois sistemas de transição em tempo de execução e usá-lo para operar um simulador. A Figura F.3 mostra os elementos mais importantes da abordagem proposta. A seguir resumimos as várias partes deste trabalho, tomando cuidados especiais para mostrar como se vai de idéias informais para formalizações, e depois para a implementação real.

PROPÓSITO DE SIMULAÇÃO

Agentes

Para descrever um SMA, é preciso ter noções específicas tanto de agentes quanto de ambientes. Com relação aos agentes, muito trabalho tem sido feito na tentativa de entender e modelar os chamados agents *inteligents* e *cognitivos*. Essas abordagens focam em grande parte na natureza das decisões racionais, especialmente no caso de agentes com capacidades computacionais limitadas (por exemplo, todos nós). A arquitetura BDI (*Beliefs-Desires-Intentions*) (Bratman, 1987; Cohen and Levesque, 1990; Rao and Georgeff, 1995) é um exemplo bem conhecido.

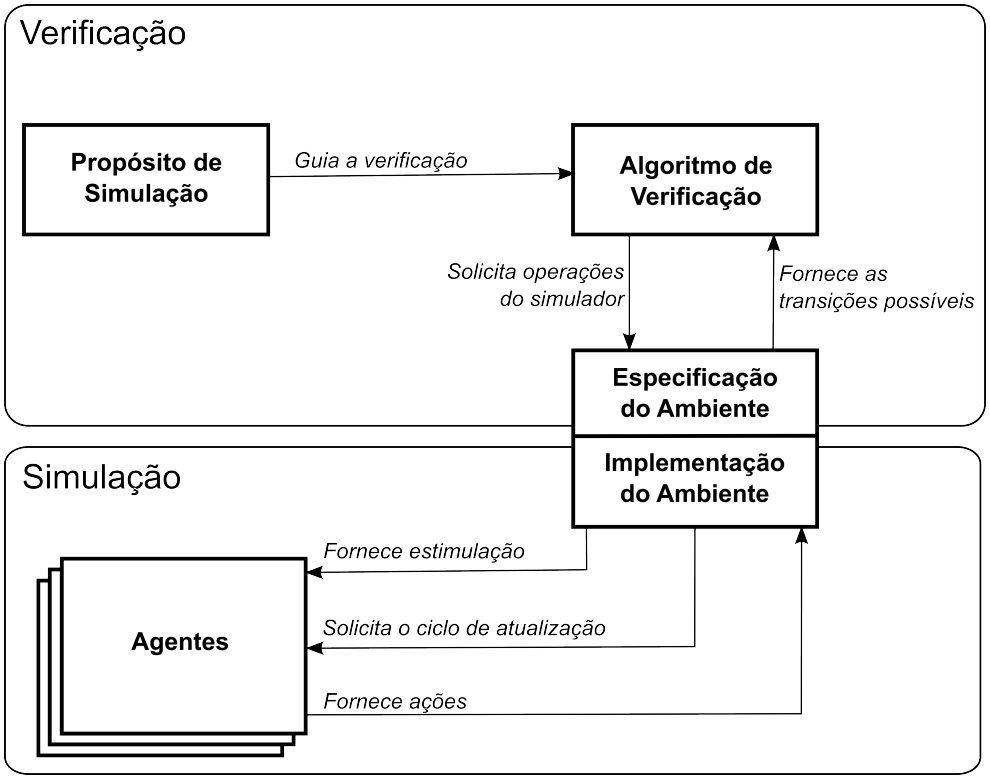


Figure F.3: Interação entre os elementos de verificação e simulação. Note, em particular, a importante posição ocupada pelo ambiente ao relacionar a verificação à simulação. Ele atua como um coordenador que, por um lado define formalmente o que pode ser feito, e por outro requisita operações concretas ao simulador.

O comportamento dos organismos, no entanto, às vezes é melhor descrito em termos diferentes. Um cão não raciocina que ele irá morrer se não comer[2]; ele meramente tem uma motivação para procurar comida quanto sente-se com fome. Se o cão aprendeu que sempre que seu dono assobia ele fornece alimento, o cão irá salivar na presença desse som – sem pensar. Estas observações sugerem que um foco diferente na modelagem de agentes é possível. Nesta tese consideramos tal modelo, baseado numa teoria de psicologia conhecida como Análise do Comportamento (Skinner, 1953). Nessa teoria, as ações dos agentes são vistas como o resultado da estimulação passada e de certos parâmetros inatos de acordo com leis comportamentais. O foco não é em qualidades mentais, tal como a natureza da razão, mas simplesmente a previsão e o controle de comportamentos por meio da estimulação ambiental. Esse ponto de vista, embora clássico dentro da psicologia, é escasso na literatura de SMAs. Como

[2]Assumindo, é claro, que os cães não podem prever suas próprias mortes da mesma forma que nós humanos podemos.

Arquitetura de Agente Comportamentalista

uma contribuição neste sentido, esta tese introduz a ***Arquitetura de Agente Comportamentalista*** (*Behaviourist Agent Architecture*).

Essa arquitetura define os principais componentes de um agente para que ele se enquadre nos principios comportamentalistas visados. Sua estrutura segue princípios centrais da Análise do Comportamento, que nós organizamos em cinco classes: (i) condicionamento de estimulos; (ii) reflexos; (iii) comportamento operante; (iv) motivações; e (v) emoções. Esses elementos trabalham de modo coerente, a fim de permitir comportamentos adaptativos e aprendizagem.

A Notação Z é usado para formalizar a arquitetura. Isso garante que uma espefificação precisa e composicional. Os benefícios de precisão são evidentes. Mas a composicionalidade também deve ser valorizada, pois permite que cada parte da especificação possa ser examinada e modificada separadamente. Portanto, permite que progressos sejam feitos com base no que é fornecido incialmente. De fato, graças à possibilidade de refinar as especificações Z, pode-se adicionar novos refinamentos para a arquitetura, a fim de especializá-la.

A arquitetura é implementada em Java. Para instanciar um agente, basta para criar um novo objeto `Organism` e inicializá-lo com um tipo especial do arquivo XML, em que os diversos parâmetros do agente devem ser definidos (por exemplo, os estímulos que ele reconhece, as ações que pode realizar, os reflexos).

Em uma abordagem de desenvolvimento completament formal, este programa Java deveria ser provado correto com relação à especificação Z (por exemplo, por meio de refinamentos formais). Isso não foi feito nesta tese, na qual o principal esforço formal foi dedicado às simulações formalmente guiadas simulações e aos algoritmos de verificação correspondentes, que são cuidadosamente justificados. No entanto, a implementação Java da arquitetura segue de muito perto a estrutura da sua especificação Z, e assim é provável que esteja correta.

Ambientes

Modelo de Ambiente para Sistemas Multi-Agentes
Arquitetura de Agente Comportamentalista

Em comparação com agentes, ambientes de SMAs têm recebido pouca atenção, como o estudo de Weyns *et al.* (2005) ressalta. O modelo de ambiente de Ferber and Müller (1996) é uma exceção. Nesta tese propomos o ***Modelo de Ambiente para Sistemas Multi-Agentes*** (*Environment Model for Multi-Agent Systems – EMMAS*), que é projetado para trabalhar com a ***Arquitetura de Agente Comportamentalista***. Dado que a teoria piscológica da qual nos valemos coloca grande importância na relação entre agentes e seus ambientes, um tal modelo explícito de ambiente é um aspecto importante a ser considerado.

No **EMMAS**, os agentes são representados por ***perfis de agente*** (*agent profiles*). Um tal perfil atribui um identificador ao agente e define os estímulos que ele reconhece, assim como as ações que ele pode emitir. Isto fornece as informações necessárias para definir um ambiente **EMMAS**. No entanto, a fim de executar o SMA resultante (e.g., para simulá-lo), é necessário ter uma implementação subjacente para esses **perfis de agente**, como por exemplo a que consideramos acima.

PERFIS DE AGENTE

O objetivo dos SMAs utilizados nesta tese é modelar sistemas sociais que existem no mundo real, onde os agentes têm uma posição física no espaço. Uma maneira de resolver isso, freqüentemente utilizado em simulações de SMAs, é fornecer uma simplificação do espaço físico, tal como uma grade bidimensional em que cada célula é uma posição possível. No entanto, no **EMMAS** adotamos uma abstração diferentes. Ao invés de de representarmos a posição física dos agentes, representamos apenas seus relacionamentos. Isto é, o SMA é visto como uma rede social. Dessa forma, somos capazes de nos concentrar sobre as relações sociais entre os agentes, que podem ser consideravelmente independentes das condições físicas às quais estão sujeitos. Por exemplo, o fato de que o agente ag_1 pode recompensar outro agente ag_2 elogiando-o pode ocorrer de várias maneiras: na mesma sala ou comunicando-se através da Internet, por exemplo.

Dado o ponto de vista comportamentalista que adotamos, essas relações são modeladas através da definição de como as ações de um agente são transformados em estímulos para outros agentes, por meio de ***transformadores de ação*** (*action transformers*). Um agente está relacionado a outro se ele pode estimulá-lo dessa maneira. Esse elemento intermediário entre a ação de um agente e o estímulo recebido por outro é justificada pelo fato de que uma ação pode ter diversas conseqüências simultâneas. De fato, a mesma ação de um agente pode ser percebida como diferentes estímulos por diferentes agentes.

TRANSFORMADORES DE AÇÃO

EMMAS permite tanto a criação quanto a destruição dinâmica desses **transformadores de ação**. A importância disso é dupla. Em primeiro lugar, isso permite a especificação de fenômenos em que a relação entre os agentes muda à medida em que envelhecem. Em segundo lugar, esse recurso permite a especificação de várias estruturas de rede possíveis para o mesmo ambiente (ou seja, a descrição de uma classe de redes sociais, e não uma rede social particular).

Embora as relações entre os agentes possam mudar, os agentes em si não podem ser criados nem destruídos num ambiente **EMMAS**. A razão é que, para ser criado, um agente definido de acordo com a **Arquitetura de Agente Comportamentalista** deve ser parametrizado *a priori* de forma detalhada antes de se tornar parte do SMA. Dado que numa especificação **EMMAS** os agentes são representados como **perfis de agente**, essa parametrização não pode ser feita dentro da especificação do ambiente. No entanto, é possível

imitar a criação e a destruição de agentes no **EMMAS**. Para isso, basta definir um conjunto de agentes, inicialmente não relacionadas a quaisquer outros, e então manipular como suas ações afetam o ambiente (e, portanto, outros agentes). Um agente cujas ações são inteiramente ignoradas e que não recebe estímulos é efetivamente irrelevante no SMA e, portanto, pode ser considerada como não-existentes.

Um ambiente define o contexto no qual os agentes existem, que é mais do que simplesmente a definição das condições iniciais. Ele inclui comportamentos pertencentes ao próprio ambiente, que podem ser executadas seja em resposta a ações de um agente, seja de forma independente de qualquer ação. Do ponto de vista da simulação e da verificação adotado nesta tese, esse contexto ambiental pode ser visto como um configuração de experimento, que define todas as experiências possíveis que podem ser realizadas com os agentes. É possível especificar **operações de ambiente** em uma especificação **EMMAS** a fim de definir uma tal configuração experimental.

Dado um SMA para simular e verificar, o modo como ele evolui é um ponto crucial. Isto é, como ele muda de um estado para os próximos até o final do procedimento a ser aplicado (e.g., uma simulação). Uma vez que nesta tese separamos explicitamente as formalizações de agentes e ambientes, somos levados a descrever as evoluções de cada um de modo diferente:

- *Estados dos agentes.* A **Arquitetura de Agente Comportamentalista** define como qualquer estado do agente é transformado em um novo estado levando-se em conta a estimulação recebida do ambiente. Tais definições são especificadas usando a Notação Z.

- *Estados do ambiente.* Especificações **EMMAS** têm uma semântica em termos da álgebra processos π-calculus (Milner, 1999; Parrow, 2001). Assim, cada estado do ambiente corresponde a uma expressão P do π-calculus, juntamente com a informação contextual, numa tripla chamada de ***status do ambiente*** (*environment status*), que limita os sucessores possíveis para o estado:

STATUS DO AMBIENTE

$$(P, \langle \mathit{Stimulation}, \mathit{Response}, \mathit{Literals} \rangle)$$

Os estados do ambiente possuem informação contextual (funções Stimulation, Response, e conjunto Literals) que os relaciona aos estados dos agentes, integrando assim todas as informações observáveis relevantes para o SMA. Isto é, os estados do ambiente são estados globais no que diz respeito às evoluções observáveis do SMA. Basta, portanto, fornecer uma semântica para o **EMMAS** a fim de obter-se uma semântica para o próprio SMA.

Semântica do EMMAS

A semântica do **EMMAS**, como afirmado acima, é dada em termos da álgebra de processos π-calculus. Álgebras de processos são normalmente empregadas na descrição de sistemas concorrentes. Elas são boas em descrever sucintamente os comportamentos relevantes para comunicações inter-processos. A escolha particular do π-calculus como fundamento teórico é motivada por algumas de suas características, que juntas tornam-o um formalismo particularmente útil entre tais álgebras. Primeiro, ele define comunicação por canais como uma noção básica, o que o torna uma escolha natural para a representação de redes. Segundo, ele permite a modificações dinâmicas, o que torna a criação e destruição de conexões entre agentes possível. Terceiro, ele fornece uma representação conveniente para o comportamento de difusão através de seu operador de replicação. Finalmente, ele tem poucos operadores e uma semântica operacional simples. Devido a essa semântica operacional, pode-se definir um ***Sistema de Transição Anotado*** (*Annotated Transition System – ATS*) que denota todas as evoluções possíveis do SMA.

SISTEMA DE TRANSIÇÃO ANOTADO

A semântica do **EMMAS** é dada em duas etapas: (i) uma tradução sintática de especificações **EMMAS** em expressões π-calculus usando uma ***função de tradução*** $[\,]_\pi$; (ii) uma fundação matemática que relaciona eventos do π-calculus a estímulos e ações dos agentes em um sistema de transição. A tradução para o π-calculus de (i), através de sua semântica operacional, fornece uma sobre-aproximação do comportamento desejado, que é então tornada precisa pelas restrições estipuladas em (ii). Dessa forma, somos capazes de definir um ***ATS de ambiente*** (*environment ATS*), possivelmente infinito, que estabelece os possíveis estados e transições para qualquer especificação de ambiente.

FUNÇÃO DE TRADUÇÃO

ATS DE AMBIENTE

A semântica assim alcançada é geral e não está vinculada a nenhuma aplicação em particular, nem mesmo à simulação. Para os fins da técnica de verificação, no entanto, será necessário realizar o estágio (ii) de uma forma um pouco mais específica, para que o resultado possa ser usado em um simulador. Em poucas palavras, a diferença é que nós introduzimos um novo evento no **ATS de ambiente**, chamado *commit*, que torna a computação das execuções mais eficiente e explícita durante as simulações (um exemplo é dado abaixo).

Para implementar o **EMMAS**, desenvolvemos um biblioteca em Java para o π-calculus, a qual permite a construção dos processos π-calculus necessários e a aplicação das regras de sua semântica operacional. Essa biblioteca segue de perto a definição formal do π-calculus, o que permite um mapeamento direto entre os elementos do π-calculus e suas implementações.

Cada seqüência de estados e eventos, chamada de **execução** (*run*), em um **ATS de ambiente** corresponde a uma possível evolução global do SMA. Considerando diferentes **execuções**, temos as diferentes possibilidades de evolução

global do SMA, e, portanto, podemos buscar sistematicamente as propriedades de interesse. Essa é a base sobre a qual a técnica de verificação desenvolvida nesta tese repousa.

Tomemos como exemplo a seguinte **execução**:

$$s_1 \overset{!beginning_u^0}{\rightarrow} s2 \overset{!beginning_v^0}{\rightarrow} s_3 \overset{!commit}{\rightarrow} s_4 \overset{?emit_a^0}{\rightarrow} s_5$$

onde s_1, s_2, s_3, s_4 e s_5 são os estados do **ATS de ambiente**, e $!beginning_u^0$, $!beginning_v^0$ e $?emit_a^0$ denotam eventos relativos a um agente identificado por 0, estímulos u e v e uma ação a (i.e., eventos de estimulação e ação com relação a um agente) Durante uma simulação, essa **execução** é interpretado da seguinte forma: primeiro, escalonar os eventos $!beginning_u^0$ e $!beginning_v^0$, mas não passá-los ainda ao agente 0. Então, como um $!commit$ é encontrado, todos os eventos escalonados são processados. No caso dos dois eventos que escalonamos, isso significa que a implementação do agente será notificada que ela está recebendo os estímulos u e v. Em seguida, o evento $?emit_a^0$ acontece, e isso significa que no estado s_4 a implementação do agente notificou o simulador de que o agente identificado por 0 está emitindo a ação a.

Propósitos de Simulação e Relações de Satisfazibilidade

SMAs são modelados de modo a poder-se estudar as suas propriedades. Nesta tese, propomos uma maneira de fazer isso através da formulação de e da verificação de hipóteses sobre o SMA (e.g., "sempre que o agente fizer X, ele fará Y depois?"). Se uma hipótese não não é confirmada, isso significa ou que a hipótese é falsa ou que o SMA não foi especificado corretamente. O julgamento a ser feito depende do objetivos em cada circunstância particular. Estamos tentando descobrir alguma lei sobre o SMA? Nesse caso, se uma hipótese que representa tal lei é falsa, é a hipótese que está incorreta, não o SMA. Estamos tentando projetar um SMA obedeça a alguma lei? Nesse caso, temos o oposto, uma hipótese falseada indica um problema no SMA. Esse ponto de vista é semelhante ao encontrada nas ciências empíricas, em que os cientistas investigam hipóteses e estabelecem julgamentos de forma semelhante. A principal diferença é que o cientista empírico estuda o mundo natural *diretamente*, enquanto que nós estamos preocupados com *modelos* da natureza na forma de SMAs.

Nesta tese, tais hipóteses são definidas especificando-se um **propósito de simulação** e uma relação de satisfazibilidade. Se o SMA satisfaz o **propósito de simulação** especificado no que diz respeito à relação satisfazibilidade desejada, então a hipótese é corroborada. Caso contrário, é falseada. A idéia de usar um tal **propósito de simulação** é inspirada na abordagem TGV (Jard and Jéron, 2005) de testes baseados em modelos (*model-based testing*),

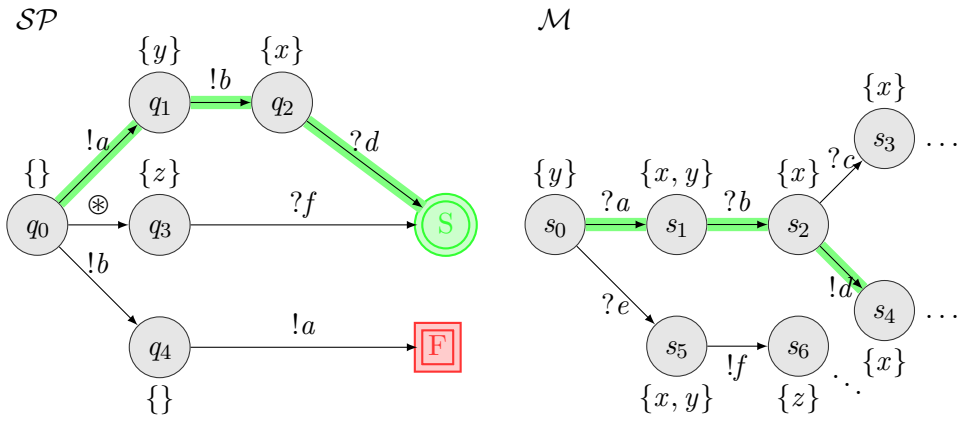


Figure F.4: Somente as **execuções** enfatizadas no **ATS** $\mathcal{M}$ e no **propósito de simulação** $\mathcal{SP}$ podem **sincronizar**. Estados são anotados com proposições e transições com eventos. Existem regras que determinam como os eventos e estados **sincronizam**. O estado rotulado com S é o *Success*, e o marcado com um F é o *Failure*. Os pontos (...) denotam que $\mathcal{M}$ continua além dos estados mostrados (e é possivelmente infinito).

na qual *propósitos de teste* formais são usados para selecionar casos de teste relevantes. Aqui, um **propósito de simulação** formal é utilizado para selecionar as simulações relevantes, embora evidentemente os critérios de relevância, entre outros aspectos técnicos, sejam bastante diferentes.

Formalmente, uma **propósito de simulação** é um **ATS** sujeito a novas restrições. Em particular, ele é finito e define dois estados especiais, *Success* (sucesso) e *Failure* (fracasso). Todas as **execuções** que levam a *Success* denotam simulações desejáveis, enquanto que todas que levam a *Failure* denotam simulações indesejáveis.

As relações satisfazibilidade, por sua vez, requerem a introdução de uma outra definição técnica, a saber, a noção de ***produto síncrono***. Dado um **ATS** $\mathcal{M}$ que modela um SMA, e um **propósito de simulação** $\mathcal{SP}$, o **produto síncrono** deles (denotado por $\mathcal{SP} \otimes \mathcal{M}$) é um outro **ATS** em que cada **execução** representa uma evolução possível de $\mathcal{M}$ autorizada por $\mathcal{SP}$. Cada estado em $\mathcal{SP} \otimes \mathcal{M}$ assume a forma (q, s), onde s é um estado de $\mathcal{M}$ e q é um estado de $\mathcal{SP}$. A Figura F.4 mostra um exemplo de $\mathcal{SP}$, $\mathcal{M}$ e das **execuções** que **sincronizam** para formar $\mathcal{SP} \otimes \mathcal{M}$. PRODUTO SÍNCRONO

Essas **execuções** podem terminar em um estado (q, s) onde $q = Succes$ ou $q = Failure$, o que significa que a **execução** em questão é desejável ou indesejável, respectivamente. Definimos diferentes relações satisfazibilidade, a saber:

- ***Factibilidade*** (*Feasibility*): $\mathcal{SP}$ é **factível** com relação a $\mathcal{M}$ se houver FACTIBILIDADE

pelo menos uma **execução** em $\mathcal{SP} \otimes \mathcal{M}$ que termina em um estado (q, s) tal que $q = \mathit{Success}$. Há variantes fraca e forte disso.

REFUTABILIDADE

- ***Refutabilidade*** (*Refutability*): $\mathcal{SP}$ é **refutável** com relação a $\mathcal{M}$ se houver pelo menos uma **execução** em $\mathcal{SP} \otimes \mathcal{M}$ que termina em um estado (q, s) tal que $q = \mathit{Failure}$. Há variantes fraca e forte disso.

CERTEZA

- ***Certeza*** (*Certainty*): $\mathcal{SP}$ é **certo** com relação a $\mathcal{M}$ se todas as **execuções** em $\mathcal{SP} \otimes \mathcal{M}$ terminarem em estados (q, s) tais que $q = \mathit{Success}$.

IMPOSSIBILIDADE

- ***Impossibilidade*** (*Impossibility*): $\mathcal{SP}$ é **impossível** com relação a $\mathcal{M}$ se todas as **execuções** em $\mathcal{SP} \otimes \mathcal{M}$ terminarem em estados (q, s) tais que $q = \mathit{Failure}$.

Algoritmos de Verificação

Essas relações de satisfazibilidade definem as condições que podemos exigir do SMA, mas elas não especificam a computação a ser utilizada. Embora sejam definições bastante operacionais, os detalhes de como construir o **produto síncrono** não são dados. Portanto, é também necessário ter algoritmos para verificar se elas são ou não válidas. Ademais, apesar do fato de que existem várias relações de satisfazibilidade, a verificação de todas elas é realizada de forma semelhante. Todos os algoritmos possuem as seguintes características principais:

- Eles executam uma busca em profundidade no **produto síncrono** de $\mathcal{SP}$ e $\mathcal{M}$.
- A busca tem uma profundidade máxima, depth_{max}.
- $\mathcal{SP} \otimes \mathcal{M}$ é computado em tempo de execução (i.e., *on-the-fly* – ele não é computado *a priori*; em cada estado, o algoritmo calcula os estados sucessores necessários para continuar), pois o próprio $\mathcal{M}$ é obtido em tempo de execução das expressões do π-calculus presentes em cada um de seus estados.
- Supõe-se que o simulador fornece uma interface. Ela é usada para controlar a execução da simulação, inclusive a possibilidade de se armazenar estados do simulador para retornar a eles mais tarde.

O que muda de um algoritmo para o outro é a **execução** buscada. No caso da **factibilidade**, o objetivo é encontrar uma **execução** leve a *Success*, mostrando assim um exemplo de como realizar uma simulação bem sucedida. Por outro lado, na relação de **certeza**, o objetivo é encontrar uma **execução**

que leve a *Failure* e, portanto, forneça um contra-exemplo (se não houver tal **execução**, isso significa que a relação de **certeza** é verdadeira com relação às observações disponíveis). As demais relações de satisfazibilidade são verificadas de forma análoga. Além disso, se a verificação de uma relação de satisfazibilidade exige uma profundidade de busca superior a $depth_{max}$, então o resultado do algoritmo é *inconclusivo*.

A complexidade dos algoritmos deve ser dada considerando-se o fato de que eles executam uma busca em profundidade em um sistema de transição possivelmente infinito que é construído em tempo de execução (i.e., $\mathcal{SP} \otimes \mathcal{M}$) e apenas até uma profundidade máxima (i.e., $depth_{max}$). Isso significa que a complexidade deve ser dada principalmente em termos de $depth_{max}$ e o fator de ramificação máxima (i.e., o número máximo de possíveis sucessores de qualquer estado), em vez do número de estados e transições no sistema de transição completo, como seria o caso se todos esses estados e transições fossem ser explorados. Ademais, como os estados de $\mathcal{M}$ são computados a partir de uma especificação de **ambiente EMMAS**, a complexidade de tais computações deve ser levada em conta também. Essas características implicam na inclusão de diversos parâmetros nos cálculos das complexidades. O desenvolvimento completo desses cálculos é dado na tese. Em poucas palavras, a complexidade no espaço é polinomial com relação ao tamanho do **ambiente** e outros parâmetros, e a complexidade no tempo é exponencial com relação a $depth_{max}$.

O importante nessa técnica é que, uma vez dado um **propósito de simulação**, ela escolhe automatica e sistematicamente quais simulações executar, em vez de depender de um usuário para guiar e inspecionar a simulação manualmente. Isso torna a exploração das simulações possíveis mais eficiente, mesmo que por vezes inconclusiva. Ademais, mostramos cuidadosamente que os algoritmos são corretos com relação a noções precisas.

Simulador

Esses algoritmos foram implementados em uma ferramenta chamada Simulador Formalmente Guiado (*Formally Guided Simulator* – FGS), que funciona da seguinte forma. Ele mantém um repositório de componentes, entre os quais a implementação em Java da **Arquitetura de Agente Comportamentalista**. Para solicitar a verificação de um SMA, três tipos de arquivos XML devem ser fornecidos ao FGS:

- Uma ou mais parametrizações para os agentes presentes. Essas parametrizações são utilizadas para instanciar a implementação da **Arquitetura de Agente Comportamentalista**.

- Uma descrição de cenário, na qual a existência dos agentes é declarada, e uma especificação **EMMAS** é fornecida.

- Uma descrição de experimento, na qual um **propósito de simulação** e a relação de satisfazibilidade são especificados.

Uma vez que o FGS tenha sido invocado, ele primeiro usa a descrição do cenário para instanciar a **Arquitetura de Agente Comportamentalista**, de modo que os agentes relevantes tornem-se disponíveis para o simulador. Em seguida, ele transforma a especificação **EMMAS** fornecida na descrição do cenário em uma expressão do π-calculus. Essa expressão é implementada diretamente por classes instanciadas da biblioteca de simulação do π-calculus de acordo com a implementação da **função de tradução**. Isto corresponde ao estado inicial do SMA a ser simulado.

O FGS, em seguida, lê a descrição de experimento, a fim de obter o **propósito de simulação** e a relação de satisfazibilidade a ser verificada. O **propósito de simulação** é implementado trivialmente, pois sua estrutura é muito simples. A relação de satisfazibilidade, por sua vez, determina qual algoritmo de verificação deve ser usado.

Depois de que tudo isso foi feito, o FGS apenas executa o algoritmo de verificação solicitado. Para computar os estados sucessores na implementação do **ATS**, ele utiliza as regras da semântica operacional do π-calculus disponíveis na biblioteca de simulação do π-calculus. O estado dos agentes são alterados e inspecionados através da manipulação dos objetos que os instanciam. Essa informação é usada para anotar os estados, e aplicar as restrições relevantes.

Há um evento especial, *commit*, que sinaliza que os agentes devem receber estimulação e fornecer respostas comportamentais. Isso é implementado pelo FGS da seguinte maneira. Eventos são armazenados conforme eles são encontrados em uma **execução** do **ATS**, mas não são entregues aos agentes. Quando um evento *commit* é encontrado, então os eventos armazenados são entregues aos agentes apropriados, os quais, por sua vez, também podem mudar as ações que estão emitindo.

Se certas opções foram definidas, o FGS irá imprimir informações sobre cada sincronização feita durante a construção do **produto síncrono**, o que permite a visualização das simulações. Quando o algoritmo termina, o veredito é mostrado e uma **execução** apropriada é apresentada (e.g., a **execução factível** que levou a *Success*).

Conclusão

Esta tese propõe uma maneira de se modelar uma classe de SMAs, bem como um arcabouço relacionado para a sua simulação e verificação. Esse arcabouço baseia-se em descrições formais de agentes e seus ambientes, e definições formais de **propósitos de simulação**. Esses elementos são utilizados para se realizar simulações de forma sistemática e guiada, a fim de determinar se certas relações de satisfazibilidade precisamente definidas são ou não verdadeiras. Trata-se, portanto, de um passo significativo para aproximar técnicas de simulação e de verificação formal. É preciso dizer que algumas transformações (principalmente relacionadas à programação do simulador) não são completamente verificadas. Apesar disso, a tese traz um arcabouço novo e amplo para analizar SMAs de modo automatizado e formalmente guiado.

F.3 Résumé Étendu

Les Systèmes Multi-Agents (SMAs) peuvent être utilisés pour modéliser les phénomènes décomposables en plusieurs agents en interaction, qui co-existent au sein d'un environnement. En particulier, ils permettent de modéliser les sociétés humaines et animales et d'analyser leurs propriétés par des moyens informatiques. Cette thèse aborde le problème de l'analyse automatisée d'un type particulier de tels modèles sociaux, à savoir ceux fondés sur les principes comportementalistes (*behaviourist*), qui contrastent avec les approches cognitives plus dominante dans la littérature SMA. La caractéristique des théories comportementalistes est l'accent mis sur la définition du comportement en termes d'interaction entre les agents et leur environnement. De cette manière, non seulement des actions réflexives, mais aussi l'apprentissage, les motivations et les émotions peuvent être définies. La thèse propose une technique de vérification qui examine de tels SMAs en utilisant des simulations guidées. Ceci est réalisé en modélisant les évolutions d'un SMA comme un système de transition (implicitement), et la propriété qui doit être vérifiée comme un autre système de transition (explicitement). Le premier est dérivé (à la volée) à partir d'une spécification formelle de l'environnement du SMA. Le deuxième, que nous appelons un ***objectif de simulation*** (*simulation purpose*), est utilisé à la fois pour vérifier la propriété et pour guider la simulation. De cette façon, seuls les états qui sont pertinents pour la propriété en question sont en réalité simulés. Algorithmiquement, cela correspond à la construction à la volée d'un produit synchrone de ces deux systèmes de transitions et à son utilisation pour faire fonctionner un simulateur. La Figure F.5 montre les éléments les plus importants de l'approche proposée. Dans ce qui suit, nous résumons les différentes parties de ce travail, en prenant soin de montrer comment on passe des idées informelles à la formalisation, et puis à la mise

OBJECTIF DE SIMULATION

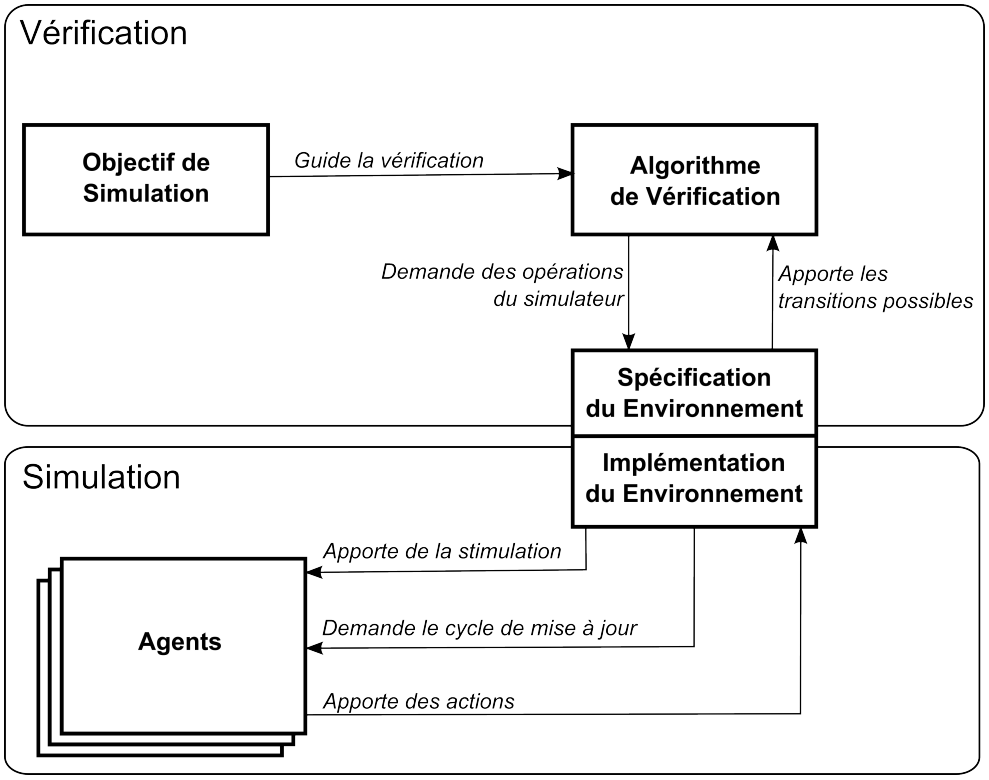


Figure F.5: Interaction entre les éléments de vérification et de simulation. L'environnement a un rôle important pour mettre la vérification en rapport avec la simulation. Il fonctionne comme un coordinateur qui, d'une part, définit formellement ce qu'on peut faire, tandis que d'autre part il active les opérations de simulation réelles.

en oeuvre effective.

Agents

Pour décrire un SMA, nous avons besoin des notions spécifiques d'agents et d'environnements. En ce qui concerne les agents, beaucoup de travail a été fait dans le domaine des agents dits *intelligents* et *cognitifs*. Ces approches se concentrent largement sur ce qui constitue les décisions rationnelles, spécialement dans le cas des agents avec des capacités de calcul limitées (par exemple, les êtres humains). L'architecture BDI (*Beliefs-Desires-Intentions*) (Bratman, 1987; Cohen and Levesque, 1990; Rao and Georgeff, 1995) est un exemple bien connu.

Cependant, le comportement des organismes est parfois mieux décrit de façon

différente. Un chien ne fait pas le raisonnement qu'il va mourir s'il ne mange pas[3]; en revanche, il dispose d'une motivation à chercher de la nourriture quand il a faim. S'il a appris que chaque fois que son maître siffle il lui fournit de la nourriture, le chien va saliver au son du sifflement – sans raisonner. Ces observations suggèrent qu'une orientation différente en matière de modélisation d'agent est possible. Dans cette thèse, nous considérons un tel modèle, basé sur la théorie psychologique connue comme l'Analyse Comportementale (*Behaviour Analysis*) (Skinner, 1953). Dans cette théorie, les actions des agents sont vues comme le résultat d'une stimulation passée et certains paramètres innés, en fonction des lois comportementales. L'accent n'est pas dans les qualités mentales telles que la nature de la raison, mais seulement dans la prédiction et le contrôle des comportements à partir de la stimulation environnementale. Ce point de vue, bien que classique au sein de la psychologie, est rare dans la littérature SMA. En guise de contribution en ce sens, cette thèse présente l'***Architecture d'Agent Comportementaliste*** (*Behaviourist Agent Architecture*).

Architecture d'Agent Comportementaliste

Cette architecture définit les aspects essentiels des agents afin qu'ils soient conformes aux principes comportementalistes. Sa structure suit les principes fondamentaux de l'analyse comportementale, que nous organisons en cinq classes: (i) conditionnement de stimulus; (ii) les réflexes; (iii) le comportement opérant; (iv) les motivations; et (v) les émotions. Ces éléments travaillent de manière cohérente afin de permettre l'adaptation et l'apprentissage des comportements.

La notation Z est utilisé pour formaliser l'architecture. Cela assure que celle-ci est définie dans une forme précise et compositionnelle. Les avantages de la précision sont évidents. Mais la compositionnalité devrait également être valorisée, car elle permet à chaque partie de la spécification d'être examinée et modifiée séparément, et donc permet qu'elle soit améliorée. En effet, grâce à la possibilité de raffinement des spécifications Z, on peut ajouter de nouveaux raffinements à l'architecture afin de la spécialiser.

L'architecture est implémentée en Java. Afin d'instancier un agent, il suffit de créer une nouvel objet `Organism` et de l'initialiser avec un type spécial du fichier XML, dans lequel plusieurs paramètres de l'agent doivent être définis (par exemple, les stimuli qu'il reconnaît, les actions qu'il peut effectuer, les réflexes).

Dans une démarche de développement formel complet, ce programme Java devrait être prouvé correct par rapport à la spécification Z (par exemple, au moyen des raffinements formels). Cela n'a pas été fait dans cette thèse, où l'effort formel principal a été consacrée a les simulations formellement guidées et les algorithmes de vérification correspondants, qui sont vérifiés. Cependant,

[3]En supposant, évidement, que les chiens ne peuvent pas prévoir leur propre mort de la même manière que nous, les humains, le pouvons

l'implémentation Java de l'architecture d'agent suit de très près la structure de sa spécification Z, et donc a de grandes chances d'être correcte.

Environnements

En comparaison avec les agents, les environnements des SMA ont reçu peu d'attention, comme c'est remarqué par l'étude de Weyns *et al.* (2005). Le modèle d'environnement de Ferber and Müller (1996) est une exception. Dans cette thèse, nous proposons le modèle ***Environnement pour Systèmes Multi-Agents*** (*Environment Model for Multi-Agent Systems – EMMAS*), conçu pour fonctionner avec l'**Architecture d'Agent Comportementaliste**. Étant donné que la théorie psychologique de base que nous utilisons accorde une grande importance aux rapports entre les agents et leur environnement, une telle modélisation explicite de l'environement est un aspect très important à prendre en compte.

ENVIRONNEMENT POUR SYSTÈMES MULTI-AGENTS

PROFILS D'AGENT

Dans **EMMAS**, les agents sont représentés par les ***profils d'agent*** (*agent profiles*). Un tel profil attribue un identifiant à un agent et définit les stimuli qu'il reconnaît ainsi que les actions qu'il peut émettre. Cela fournit les informations nécessaires pour définir un environnement **EMMAS**. Néanmoins, afin d'exécuter le SMA résultant (e.g., pour le simuler), il est nécessaire d'avoir une implémentation sous-jacente pour ces **profils d'agents**, tel que celle que nous avons considérée ci-dessus.

Le but des SMA utilisés dans cette thèse est de modéliser les systèmes sociaux qui existent dans le monde réel, où les agents ont une position physique dans l'espace. Une façon de résoudre ce problème, souvent utilisée dans les simulations des SMAs, est de fournir une simplification de l'espace physique, comme par example une grille à deux dimensions dans lequel chaque cellule est une position possible. Cependant, dans **EMMAS** nous adoptons une abstraction différents. Au lieu de représenter les position physique des agents, nous représentons leurs relations. C'est-à-dire, le SMA est considéré comme un réseau social. De cette manière, nous sommes en mesure de nous concentrer sur les relations sociales entre les agents, qui peuvent être tout à fait indépendantes de la condition physique auxquelles ils sont soumis. Par exemple, le fait que un agent ag_1 récompense un autre agent ag_2 en le complimentant peut se passer de plusieurs manières : dans la même salle ou à en communiquant par l'Internet.

Etant donné le point de vue comportementaliste que nous adoptons, ces relations sont modélisées en définissant comment les actions d'un agent sont transformées en stimuli pour d'autres agents par l'intermédiaire de ***transformateurs d'action*** (*action transformers*). Un agent est lié à un autre s'il peut stimuler l'autre de cette manière. Cet élément intermédiaire entre l'action

TRANSFORMATEURS D'ACTION

d'un agent et la stimulation reçu par un autre est justifié par le fait qu'une action peut avoir plusieurs conséquences simultanées. En effet, la même action par un agent peut être perçue comme des stimuli différents par des agents différents.

EMMAS permet la création et la destruction dynamique de ces **transformateurs d'action**. L'importance de cela est double. Tout d'abord, ça permet la spécification de phénomènes dans lesquels le rapport parmi les agents change à mesure qu'ils vieillissent. Deuxièmement, ça permet la spécification de plusieurs structures de réseau possibles pour le même environnement (i.e., la description d'une classe de réseaux sociaux, et non pas un seul réseau social).

Bien que les relations entre les agents puissent changer, les agents eux-mêmes ne peuvent pas être créés ni détruits dans **EMMAS**. Une raison à cela est que, pour être créé, un agent suivant l'**Architecture d'Agent Comportementaliste** doit être paramétrée *a priori* d'une manière détaillée avant de devenir partie du SMA. Mais puisque dans **EMMAS** les agents sont représentés comme des **profils d'agent**, cette paramétrisation ne peut pas être faite au sein des spécifications **EMMAS** elles-mêmes. Il est néanmoins possible d'émuler la création et la destruction des agents dans **EMMAS**. Pour le faire, il suffit de définir un pool d'agents, d'abord isolés, et ensuite de manipuler comment leurs actions affectent l'environnement (et donc d'autres agents). Un agent dont les actions sont entièrement ignorées et qui ne reçoit pas de stimuli peut être considéré comme non-existant.

Un environnement définit le contexte dans lequel les agents existent, ce qui est plus que fixer les conditions initiales. Il inclut les comportements relatifs à l'environnement lui-même, qui peuvent être exécutés en réponse aux actions d'un agent ou indépendamment d'une telle action. Du point de vue de simulation et de vérification adopté dans cette thèse, ce contexte de l'environnement peut être considéré comme un dispositif expérimental, qui définit toutes les expériences possibles qui peuvent être effectuées avec les agents. Il est possible de spécifier des ***opérations d'environnement*** (*environment operations*) dans **EMMAS** afin de définir un tel dispositif expérimental.

OPÉRATIONS D'ENVIRONNEMENT

Si nous avons un SMA à simuler et vérifier, un point crucial concerne la façon dont il évolue. C'est-à-dire, comment il change d'un état à un autre jusqu'à la fin de la procédure appliquée (e.g., une simulation). Etant donné que dans cette thèse nous séparons explicitement les formalisations des agents et des environnements, nous tenons compte de l'évolution de chacun de façon différente:

- *Les états des agents.* L'**Architecture d'Agent Comportementaliste** définit comment chaque état d'agent est transformé en un état nouveau

en prenant en compte des stimulations environnementales. Ces définitions sont spécifiés en utilisant la notation Z.

- *Lest états du environnement.* Les spécifications **EMMAS** ont une sémantique en termes de l'algèbre de processus π-calculus (Milner, 1999; Parrow, 2001). Ainsi, chaque état de l'environnement correspond à une expression P du π-calculus, avec des informations contextuelles, dans un triplet appelé ***statut de l'environnement*** (*environment status*), qui limitent les successeurs possibles de l'état:

STATUT DE L'ENVIRONNEMENT

$$(P, \langle \mathit{Stimulation}, \mathit{Response}, \mathit{Literals} \rangle)$$

Les états de l'environnement ont des informations contextuelles (fonctions Stimulation, Response, et ensemble Literals) qui les mettent en rapport avec les agents, incorporant ainsi toutes les informations observables pertinentes du SMA. C'est-à-dire, les états d l'environnement sont des états globaux en ce que concerne les évolutions observables du SMA. Il suffit alors de fournir une sémantique pour **EMMAS** afin d'avoir la sémantique du SMA lui-même.

Sémantique d'EMMAS

La sémantique d'**EMMAS**, comme nous avons déjà dit ci-dessus, est donnée en termes de l'algèbre de processus π-calculus. Les algèbres de processus sont généralement employées pour décrire les systèmes concurrents. Elles sont bonnes pour décrire succinctement les comportements pertinents à la communication inter-processus. Le choix particulier du π-calculus comme un fondement théorique est motivé par certaines de ses caractéristiques qui le distinguent parmi les algèbres de processus existantes. D'abord, il considére la communication à travers les canaux comme une notion primitive, ce qui en fait est un choix naturel pour représenter les réseaux. Deuxièmement, il permet des modifications dynamiques, ce qui rend la création et la destruction des connexions entre les agents possibles. Troisièmement, il fournit une représentation pratique pour la diffusion grâce à son opérateur de réplication. Enfin, il a peu d'opérateurs et une sémantique opérationnelle simple. Grâce à cette sémantique opérationnelle, nous pouvons donner un ***Système de Transition Annoté*** (*Annotated Transition System* – ATS) qui dénote toutes les évolutions possibles du SMA.

SYSTÈME DE TRANSITION ANNOTÉ

La sémantique d'**EMMAS** est effectivement donnée en deux étapes : (i) une traduction syntaxique d'**EMMAS** dans des expressions π-calculus à travers une ***fonction de traduction*** (*translation function*) $[\,]_\pi$; (ii) un ensemble de définitions qui lie les événements du π-calculus aux stimuli et actions des agents dans un système de transition. La traduction vers le π-calculus de (i), à

FONCTION DE TRADUCTION

travers sa sémantique opérationnelle, fournit une sur-approximation du comportement désiré, qui est ensuite rendu précis grâce aux restrictions prévues par (ii). Par cette méthode, nous sommes en mesure de définir un ***ATS d'environnement*** (*environment ATS*), peut-être infini, qui établit les états et transitions possibles pour n'importe quel spécification **EMMAS**.

ATS D'ENVIRONNEMENT

La sémantique ainsi obtenue est d'ordre général et n'est pas liée à une application particulière, pas même la simulation. Pour les fins de vérification, cependant, il sera nécessaire de procéder à l'étape (ii) d'une façon un peu plus précise, de sorte que le résultat puisse être utilisé dans un simulateur. En bref, la différence est que nous introduisons un nouvel événement dans le **ATS d'environnement**, appelé *commit*, ce qui rend le calcul des exécutions plus efficace et plus explicite lors des simulations (un exemple est donné ci-dessous).

Pour mettre **EMMAS** en oeuvre, nous avons développé une bibliothèque pour le π-calculus en Java, ce qui permet la construction des processus π-calculus et l'application des règles de sa sémantique opérationnelle. Cette bibliothèque suit la définition formelle du π-calculus de très près, ce qui permet une correspondance directe entre les éléments du π-calculus et leur mise en oeuvre.

Chaque séquence d'états et d'événements, appelé une **exécution** (*run*), dans un **ATS d'environnement** correspond à une évolution globale possible du SMA. En considérant des **exécutions** différentes, nous avons les différentes possibilités d'évolutions globales du SMA, et donc nous pouvons chercher systématiquement les propriétés d'intérêt. Ceci est la base sur laquelle repose la technique de vérification développée dans cette thèse.

Soit l'exemple suivant d'une telle **exécution** :

$$s_1 \overset{!beginning_u^0}{\rightarrow} s2 \overset{!beginning_v^0}{\rightarrow} s_3 \overset{!commit}{\rightarrow} s_4 \overset{?emit_a^0}{\rightarrow} s_5$$

où s_1, s_2, s_3, s_4 et s_5 sont des états de l' **ATS d'environnement**, et $!beginning_u^0$, $!beginning_v^0$ et $?emit_a^0$ dénotent des événements relatifs à un agent identifié par 0, stimuli u et v et une action a (i.e., événements de stimulation et d'action concernant un agent). Lors d'une simulation, cette **exécution** est interprétée de la façon suivante: d'abord, noter les événements $!beginning_u^0$ et $!beginning_v^0$, mais ne pas les passer encore à l'agent 0. Ensuite, comme un $!commit$ est trouvé, tous les événements programmés sont pris en compte. Dans le cas des deux événements que nous avons prévus, cela signifie que l'implémentation de l'agent 0 sera avisée qu'il est maintenant en train de recevoir les stimuli u et v. Finalement, l'événement $?emit_a^0$ arrive, et cela signifie que dans l'état s_4 l'implémentation de l'agent 0 a avisé le simulateur qu'il est en train d'émettre l'action a.

Objectifs de Simulation et Relations de Satisfaction

On modélise les SMAs afin d'étudier leurs propriétés. Dans cette thèse, nous proposons un moyen de le faire en formulant des hypothèses sur le SMA et en vérifiant si elles tiennent ou pas (e.g., "chaque fois que l'agent fait X, va-t-il faire Y plus tard?"). Si une hypothèse ne tient pas, cela signifie soit que l'hypothèse est fausse, soit que le SMA n'a pas été correctement spécifié. Le jugement dépend de notre objectif dans chaque circonstance particulière. Essayons-nous de découvrir une loi sur le SMA? Dans ce cas, si une hypothèse qui représente cette loi s'avère être fausse, c'est l'hypothèse qui est inexacte, et non le SMA. Essayons-nous de concevoir un SMA qui obéisse à une loi? Dans ce cas, nous avons l'inverse, une hypothèse falsifiée indique un problème dans le SMA. Cette vue est similaire à celle trouvée dans les sciences empiriques, dans lesquelles les scientifiques étudient des hypothèses et font des jugements d'une manière semblable. À cet égard, la différence principale que nous sommes concernés par des *modèles* de la nature sous la forme des SMAs, alors qu'eux étudient le monde naturel *directement*.

Dans cette thèse, une telle hypothèse est définie par la spécification d'un **objectif de simulation** et une relation de satisfaction. Si le SMA satisfait l'**objectif de simulation** spécifiée par rapport a la relation de satisfaction désirée, l'hypothèse est corroborée. Sinon, elle est falsifiée. L'idée d'utiliser un tel **objectif de simulation** est inspiré par l'approche de TGV (Jard and Jéron, 2005) pour les tests de logiciel basés sur des modèles (*model-based testing*), dans lequel des *objectifs de test* formels sont utilisés pour sélectionner les cas de test pertinents. Ici, une **objectif de simulation** formel est utilisé pour sélectionner les exécutions de simulations pertinentes, bien que les critères de pertinence, entre autres technicités, soient assez différents.

Formellement, un **objectif de simulation** est un **ATS** soumis à de nouvelles restrictions. En particulier, il est fini et définit deux états spéciaux, le *Success* (succès) et le *Failure* (échec). Tous les **exécutions** qui mènent au *Success* dénotent des simulations souhaitables, alors que celles qui conduisent à *Failure* dénotent des simulations indésirables.

PRODUIT SYNCHRONE

Les relations de satisfaction, à leur tour, nécessitent encore l'introduction d'une autre définition technique, à savoir, la notion de ***produit synchrone*** (*synchronous product*). Soit un **ATS** $\mathcal{M}$ qui modélise un SMA, et un **objectif de simulation** $\mathcal{SP}$. Leur **produit synchrone** (écrit $\mathcal{SP} \otimes \mathcal{M}$) est un autre **ATS** dans lequel chaque **exécution** représente une évolution possible de $\mathcal{M}$ qui a été autorisée par $\mathcal{SP}$. Chaque état dans $\mathcal{SP} \otimes \mathcal{M}$ prend la forme de (q, s), où s est un état de $\mathcal{M}$, et q est un état de $\mathcal{SP}$. La Figure F.6 montre un exemple de $\mathcal{SP}$, $\mathcal{M}$ et des **exécutions** qui **synchronisent** pour former $\mathcal{SP} \otimes \mathcal{M}$.

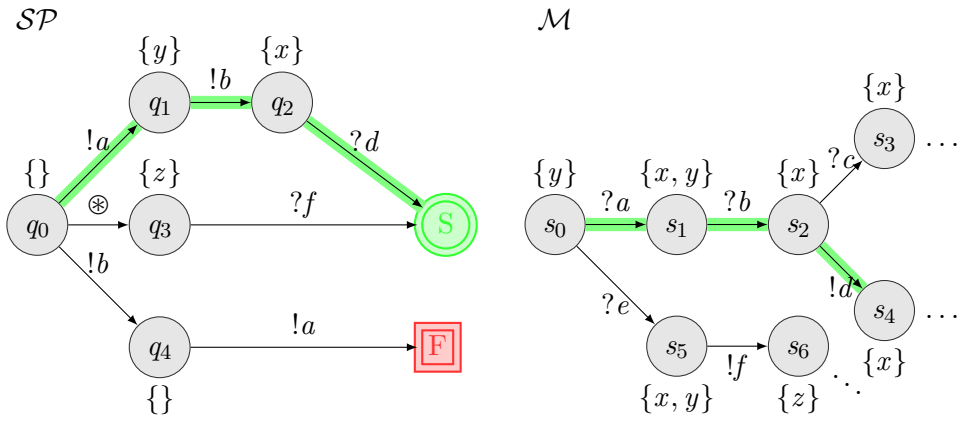


Figure F.6: Seules les **exécutions** ombrées dans l'**ATS** $\mathcal{M}$ et l'**objectif de simulation** $\mathcal{SP}$ peuvent se **synchroniser**. Les états sont annotés avec des propositions et les transitions avec des événements. Il y a des règles qui déterminent comment les événements et les états se **synchronisent**. L'état étiqueté avec S est le *Success*, et celui marqué avec F est le *Failure*. Les points (...) dénotent que $\mathcal{M}$ continue au-delà des états représentés (il est peut-être infini).

Ces **exécutions** peuvent se terminer dans un état (q, s) où $q = Succes$ ou $q = Failure$, ce qui signifie que l'**exécution** en question est désirable ou indésirable, respectivement. Nous définissons plusieurs relations de satisfaction, à savoir:

- ***Faisabilité*** (*Feasibility*): $\mathcal{SP}$ est **faisable** par rapport à $\mathcal{M}$ si il y a au moins une **exécution** dans $\mathcal{SP} \otimes \mathcal{M}$ qui se termine dans un état (q, s) tel que $q = Success$. Il existe des variantes faible et forte de ceci. FAISABILITÉ
- ***Réfutabilité*** (*Refutability*): $\mathcal{SP}$ est **réfutable** par rapport à $\mathcal{M}$ s'il y a au moins une **exécution** dans $\mathcal{SP} \otimes \mathcal{M}$ qui se termine dans un état (q, s) tel que $q = Failure$. Il existe des variantes faible et forte de ceci. RÉFUTABILITÉ
- ***Certitude*** (*Certainty*): $\mathcal{SP}$ est **certain** par rapport à $\mathcal{SP}$ si tous les **exécutions** dans $\mathcal{SP} \otimes \mathcal{M}$ finissent dans des états (q, s) tels que $q = Success$. CERTITUDE
- ***Impossibilité*** (*Impossibility*): $\mathcal{SP}$ est **impossible** par rapport à $\mathcal{M}$ si tous les **exécutions** dans $\mathcal{SP} \otimes \mathcal{M}$ finissent dans des états (q, s) tels que $q = Failure$. IMPOSSIBILITÉ

Algorithmes de Vérification

Ces relations de satisfaction définissent les conditions que nous pouvons exiger du SMA, mais elles ne précisent pas le calcul à utiliser. Certes, ce sont plutôt

des définitions opérationnelles, mais les détails concernant la construction du **produit synchrone** ne sont pas donnés. Il est donc nécessaire d'avoir des algorithmes afin de vérifier si elles sont valides ou pas. En outre, malgré le fait qu'il y a plusieurs relations de satisfaction, leur vérification est réalisé d'une manière semblable. Tous les algorithmes ont les caractéristiques principales suivantes:

- Ils effectuent une recherche en profondeur (*depth-first search*) sur le **produit synchrone** de $\mathcal{SP}$ et $\mathcal{M}$.
- La recherche a une profondeur maximale,$depth_{max}$.
- $\mathcal{SP} \otimes \mathcal{M}$ est calculée a la volée (c'est à dire, il n'est pas calculé *a priori*; à chaque état, l'algorithme calcule les états suivants nécessaires pour continuer), car $\mathcal{M}$ est lui-même obtenu à la volée, a partir de l'expression π-calculus présente dans chacun de ses états.
- Une interface avec le simulateur est supposée exister. Elle est utilisée pour contrôler l'exécution de la simulation, y compris la possibilité de stocker les états du simulateur et d'y retourner plus tard.

Ce qui change d'un algorithme à l'autre est l'**exécution** recherchée. Dans le cas de **faisabilité**, l'objectif est de trouver une **exécution** qui mène à *Success*, montrant ainsi un exemple de comment effectuer une simulation à succès. En revanche, pour la relation de **certitude**, le but est de trouver une **exécution** qui mène à *Failure*, et fournit donc un contre-exemple (si aucune telle **exécution** est trouvé, cela signifie que la relation de **certitude** tient par rapport aux observations disponibles). Les autres relations de satisfaction sont vérifiées de manière similaire à celles-ci. Par ailleurs, si la vérification d'une relation de satisfaction exige une profondeur de recherche supérieur à $depth_{max}$, alors le résultat de l'algorithme est *non-concluant*.

La complexité des algorithmes doit être donnée en considérant le fait qu'ils effectuent une recherche en profondeur sur un système de transition éventuellement infini qui est construit à la volée (i.e., $\mathcal{SP} \otimes \mathcal{M}$) et que jusqu'à une profondeur maximale (i.e., $depth_{max}$). Cela signifie que la complexité doit être donnée principalement en termes de $depth_{max}$ et le maximum du facteur de branchement (i.e., le nombre maximal de successeurs possibles d'un état), au lieu du nombre d'états et les transitions dans le système complet, comme ce serait le cas si tous ces états et les transitions étaient à explorer. En outre, étant donné que les états à $\mathcal{M}$ sont en fait calculées à partir d'une spécification d'**environnement EMMAS**, la complexité de ces calculs doit être prise en compte aussi. Ces caractéristiques introduisent de nombreux paramètres dans la complexité. Le développement complet est donné dans la thèse. En quelques mots, la complexité en espace est polynomiale par rapport à la taille

de l'**environnement** et d'autres paramètres, et la complexité en temps est exponentielle par rapport à $depth_{max}$.

Ce qui est important dans cette technique est qu'elle choisit les simulations à exécuter automatiquement et de manière systématique, au lieu de dépendre d'un utilisateur pour guider et d'inspecter la simulation manuellement. Cela rendre l'exploration des simulations possibles plus efficace, même si on a parfois des résultats non concluants. Par ailleurs, les algorithmes sont avérés être corrects selon des notions précises.

Simulateur

Ces algorithmes ont été mis en oeuvre dans un outil appelé Simulateur Formellement Guidé (*Formally Guided Simulator* – FGS), qui fonctionne de la façon suivante. Il maintient une collection de composants, parmi lesquels l'implémentation Java de l'**Architecture d'Agent Comportementaliste**. Pour demander la vérification d'un SMA, trois types de fichiers XML doivent être fournis au FGS:

- Une ou plusieurs paramétrisations des agents présents. Ces paramétrages sont utilisés pour instancier l'implémentation de l'**Architecture d'Agent Comportementaliste**.
- Une description du scénario, dans lequel les agents sont déclarés d'exister, et la spécification **EMMAS** sont fournies.
- Une description d'expérience, dans laquelle un **objectif de simulation** et la relation de satisfaction à vérifier sont spécifiées.

Une fois qu'on démarre FGS, il utilise d'abord la description du scénario pour instancier l'**Architecture d'Agent Comportementaliste**, de sorte que les agents concernés deviennent disponibles pour le simulateur. Il transforme ensuite la spécification **EMMAS** fournie dans la description du scénario en une expression du π-calculus. Cette expression est mise en oeuvre directement par l'instanciation des classes à la bibliothèque de simulation du π-calculus en fonction de l'application de la **fonction de traduction**. Cela correspond à l'état initial du SMA à simuler.

FGS lit alors la description de l'expérience afin d'obtenir l'**objectif de simulation** et de savoir quelle relation de satisfaction dois être vérifiée. L'**objectif de simulation** est implémenté trivialement, puisque sa structure est très simple. La relation de satisfaction, à son tour, détermine l'algorithme de vérification qui doit être utilisé.

Ensuite, FGS exécute simplement l'algorithme de vérification demandé. Pour calculer les états suivant dans l'implémentation du **ATS**, il utilise les règles de la sémantique opérationnelle du π-calculus, disponibles aussi dans la bibliothèque de simulation du π-calculus. Les états des agents sont modifiés et inspectés en manipulant les objets qui les instancient. Cette information est utilisée pour annoter les états, et pour appliquer les contraintes pertinentes.

Il y a un événement spécial, *commit*, qui signale que les agents doivent recevoir des stimulations et fournir des réponses comportementales. Ceci est implémenté par FGS de la manière suivante. Les événements sont stockés au fur et à mesure qu'ils sont trouvés dans une **exécution** du **ATS**, mais ils ne sont pas livrés aux agents. Quand un événement *commit* est trouvé, alors les événements stockés sont livrés aux agents appropriés, et ils peuvent aussi changer les actions qu'ils sont en train d'emmètre.

Si certains options ont été fixées, FGS va afficher des informations a propos de chaque synchronisation faite dans les **produit synchrone** lors des simulation, ce qui permet de les visualiser. Lorsque l'algorithme se termine, le verdict est montré, et une **exécution** approprié est affichée aussi (e.g., le **exécution faisable** qui a conduit au *Success*).

Conclusion

Cette thèse propose une façon de modéliser une classe de SMAs, ainsi qu'un cadre correspondant pour effectuer leur simulation et leur vérification. Ce cadre est fondé sur des descriptions formelles des agents et de leur environnement, ainsi que des définitions formelles des **objectifs de simulation**. Ces éléments sont utilisés pour effectuer des simulations de manière systématique et guidée afin de déterminer si certaines relations de satisfaction précisément définies sont valides ou pas. C'est donc une étape importante pour combler le fossé entre les techniques de simulation et de vérification. Il faut dire que certaines transformations (principalement liées à la programmation du simulateur) ne sont pas complètement vérifiée. Néanmoins, la thèse apporte un cadre nouveau et d'une portée considérable pour analyser les SMAs de façon automatisée et guidées formellement.

Glossary

Agent An entity capable of autonomous behaviour which exists within an environment.

Agent Architecture An abstract, structured and integrated description of a class of agents.

Annotated Transition System An extension of Labelled Transition Systems developed in this thesis in which states can also be labelled ("annotated"). See Definition 6.2.

Architecture Depends on the context. See either Agent Architecture or Software Architecture.

Behaviour Analysis A branch of behaviourism created and developed mainly by Burrhus Frederic Skinner from the decade of 1930 onwards.

Behavioural Agent Architecture The Agent Architecture developed in this thesis, which follows core ideas from the Behaviour Analysis tradition.

Behaviourism A movement which postulates that observable and measurable behaviour is the only legitimate object of study within Psychology.

Drive A temporary modification of behaviour which is regulated by either the provision or the deprivation of particular stimuli. For instance, thirst and hunger.

Emotion A temporary modification of behaviour which is not explained by any Drive. For instance, depression and frustration.

Environment The entity in which agents exist and through which they communicate. See Definition 5.6.

Experiment The specification of the Simulation Strategy to adopt.

Experimenter A human that sets up and performs experiments. In this thesis, the experimenter is the user who defines the scenarios to be simulated and runs the simulator in order to address some particular question.

Formal Method Any software development method characterized by the use of abstract models and rigorous mathematical tools.

Formal Specification Any mathematical description of some system, intended to be precise and unambiguous.

Formal Verification Any technique to formally check whether a system satisfies some formal specification.

Labelled Transition System A tuple composed by a set of states and labelled transitions. Such representations are often used to provide the Operational Semantics of Process Algebras. See Definition E.1.

Model Checking A formal verification technique which consists in systematically, explicitly and (usually) exhaustively investigating the State-Space of a system in order to determine its conformance with respect to some property. It is, therefore, a technique that operates directly on semantics, and not on syntax.

Multi-Agent System A system composed by agents that exist and interact within an environment.

Operant Behaviour The class of behaviour in which behaviours are learned through either reinforcement or punishment. A behaviour thus learned establishes an association between an organism's action and a consequent stimulus, but the strength of this relation can be changed over time.

Operational Semantics A style of providing semantics of languages in which the meaning of a sentence is given in terms of the changes it produces in some structure. This is often achieved by a hierarchical system of rules which specify changes in terms of pre- and post-conditions.

Process Algebra Any Formal Method whose specifications are given in terms of expressions composed of primitive elements and operators ruled by certain laws. That is to say, specifications that resemble ordinary algebra. Process algebras are usually employed to specify the communication capabilities of concurrent systems.

Reflex A fixed behaviour (i.e., not learned) which depends on an eliciting stimulus to happen.

Respondent Behaviour The class of behaviours formed by Reflexs.

Scenario The specification of the initial configuration of a Simulation Model.

Simulation The execution of a simulation model.

Simulation Model An abstract and executable description of some Target System which can be subject to simulation.

Simulation Purpose A particular kind of Annotated Transition System suitable to the specification of desired and undesired simulations. These structures are used in this thesis as the properties to be verified w.r.t. a system. See Definition 7.8.

Simulation Strategy An algorithm used to perform simulations.

Simulator A software that given a executable model produces sequences of states of this model.

Software Architecture An abstract, structured and integrated description of a class of programs.

Software Component A unit of software which is developed to be composed later with other components, and which depends on some underlying infrastructure.

State-Space The set of all possible states for a system. It is often formalized as a Labelled Transition System. In this thesis we employ our Annotated Transition System for this purpose.

Target System That which one wishes to model. That is to say, the object of the real world which one wishes to model in order to study it in an abstract form.

Utility The perceived value of a stimulus.

Acronyms

ATS Annotated Transition System.

BDI Belief-Desire-Intention.

EMMAS Environment Model for Multi-Agent Systems.

FGS Formally Guided Simulator.

LTS Labelled Transition System.

MAS Multi-Agent System.

SP Simulation Purpose.

XML Extensible Markup Language.

Bibliography

Abrial(1996) J.R. Abrial. *The B-book: assigning programs to meanings.* Cambridge University Press, 1996. Cited in page 28

Alam *et al.***(2009)** Shah Jamal Alam, Ruth Meyer and Emma Norling. A model for HIV spread in a south african village. In Nuno David and Jaime Simão Sichman, editors, *Multi-Agent-Based Simulation IX*, pages 33–45. Springer-Verlag, Berlin, Heidelberg, 2009. Cited in page 23

Alloway(2005) Tom Alloway. *Sniffy: The Virtual Rat, Pro Version 2.0.* Wadsworth Publ. Co., Belmont, CA, USA, 2005. Cited in page 19

Anderson *et al.***(2004)** John R. Anderson, Daniel Bothell, Michael D. Byrne, Scott Douglass, Chruistian Lebiere and Yulin Qin. An integrated theory of mind. *Psychological Review*, 111(4):1036 – 1060, 2004. Cited in page 16, 40

Baier and Katoen(2008) Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking.* The MIT Press, 2008. Cited in page 24, 45, 128

Balbi *et al.***(2010)** S. Balbi, P. Perez and C. Giupponi. An agent-based model to explore scenarios of adaptation to climate change in an alpine tourism destination. In *3rd World Congress on Social SimulationComplex Systems (WCSS 2010), September 6-9, 2010, Kassel, Germany.* 2010. Cited in page 23

Bansal *et al.***(2008)** Vidit Bansal, Ramachandra Kota and Kamalakar Karlapalem. System issues in multi-agent simulation of large crowds. In Luis Antunes, Mario Paolucci and Emma Norling, editors, *Multi-Agent-Based Simulation VIII*, pages 8–19. Springer-Verlag, Berlin, Heidelberg, 2008. Cited in page 23

Barbuceanu and Fox(1995) Mihai Barbuceanu and Mark S. Fox. COOL: A language for describing coordination in multi agent systems. In *ICMAS*, pages 17–24, 1995. Cited in page 21

Barringer *et al.***(2004a)** Howard Barringer, Allen Goldberg, Klaus Havelund and Koushik Sen. Rule-based runtime verification. In Bernhard Steffen and Giorgio Levi, editors, *VMCAI*, volume 2937 of *Lecture Notes in Computer Science*, pages 44–57. Springer, 2004a. ISBN 3-540-20803-8. Cited in page 27

Barringer *et al.***(2004b)** Howard Barringer, Allen Goldberg, Klaus Havelund and Koushik Sen. Program monitoring with LTL in EAGLE. In *IPDPS*. IEEE Computer Society, 2004b. ISBN 0-7695-2132-0. Cited in page 27

Batagelj and Mrvar(1998) Vladimir Batagelj and Andrej Mrvar. Pajek – program for large network analysis. *Connections*, 21(1), 1998. Cited in page 31

Bauer *et al.***(2006)** Andreas Bauer, Martin Leucker and Christian Schallhart. Monitoring of real-time properties. In S. Arun-Kumar and Naveen Garg, editors, *FSTTCS*, volume 4337 of *Lecture Notes in Computer Science*, pages 260–272. Springer, 2006. ISBN 3-540-49994-6. Cited in page 27

Bauer *et al.***(2007)** Andreas Bauer, Martin Leucker and Christian Schallhart. The good, the bad, and the ugly, but how ugly is ugly? In Oleg Sokolsky and Serdar Tasiran, editors, *RV*, volume 4839 of *Lecture Notes in Computer Science*, pages 126–138. Springer, 2007. ISBN 978-3-540-77394-8. Cited in page 26

Bearman *et al.***(2004)** Peter S. Bearman, James Moody and Katherine Stovel. Chains of affection: The structure of adolescent romantic and sexual networks. *American Journal of Sociology*, 110(1):44–91, 2004. Cited in page 23

Benerecetti *et al.***(1998)** Massimo Benerecetti, Fausto Giunchiglia and Luciano Serafini. Model checking multiagent systems. *Journal of Logic and Computation*, 8(3):401–423, 1998. Cited in page 30, 32

Bergstra and Klop(1984) J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information and Control*, 60(1-3):109 – 137, 1984. Cited in page 27

Bhargavan *et al.***(2002)** Karthikeyan Bhargavan, Carl A. Gunter, Insup Lee, Oleg Sokolsky, Moonjoo Kim, Davor Obradovic and Mahesh Viswanathan. Verisim: Formal analysis of network simulations. *IEEE Trans. Softw. Eng.*, 28(2):129–145, 2002. ISSN 0098-5589. URL `http://dx.doi.org/10.1109/32.988495`. Cited in page 23, 31

Biere *et al.***(1999)** Armin Biere, Alessandro Cimatti, Edmund M. Clarke and Yunshan Zhu. Symbolic model checking without BDDs. In *TACAS*, pages 193–207, 1999. Cited in page 24

Biere *et al.***(2003)** Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman and Yunshan Zhu. Bounded model checking. *Advances in Computers*, 58:118–149, 2003. Cited in page 24

Bordini *et al.***(2004)** Rafael Bordini, Michael Fisher, Willem Visser and Michael Wooldridge. Verifiable multi-agent programs. In Mehdi Dastani, Jürgen Dix and Amal El Fallah-Seghrouchni, editors, *Programming Multi-Agent Systems*, volume 3067 of *Lecture Notes in Computer Science*, pages 72–89. Springer Berlin / Heidelberg, 2004. Cited in page 32

Bordini *et al.***(2003)** Rafael H. Bordini, Michael Fisher, Carmen Pardavila and Michael Wooldridge. Model checking AgentSpeak. In *Proceedings of the second international joint conference on autonomous agents and multiagent systems*, AAMAS'03, pages 409–416, New York, NY, USA, 2003. ACM. ISBN 1-58113-683-8. doi: http://doi.acm.org/10.1145/860575.860641. URL `http://doi.acm.org/10.1145/860575.860641`. Cited in page 32

Bordini *et al.***(2007)** Rafael H. Bordini, Michael Wooldridge and Jomi Fred Hübner. *Programming Multi-Agent Systems in AgentSpeak using Jason (Wiley Series in Agent Technology)*. John Wiley & Sons, 2007. ISBN 0470029005. Cited in page 15

Borgatti *et al.***(2002)** S. P. Borgatti, M. G. Everett and L. C. Freeman. UCINet for Windows: Software for social network analysis. *Harvard Analytic Technologies*, 2006, 2002. Cited in page 31

Bosse and Gerritsen(2008) Tibor Bosse and Charlotte Gerritsen. Agent-based simulation of the spatial dynamics of crime: on the interplay between criminal hot spots and reputation. In Lin Padgham, David C. Parkes, Jörg P. Müller and Simon Parsons, editors, *AAMAS (2)*, pages 1129–1136. IFAAMAS, 2008. ISBN 978-0-9817381-1-6. Cited in page 23, 31

Bosse *et al.***(2009)** Tibor Bosse, Catholijn M. Jonker, Lourens van der Meij, Alexei Sharpanskykh and Jan Treur. Specification and verification of dynamics in agent models. *Int. J. Cooperative Inf. Syst.*, 18(1):167–193, 2009. Cited in page 31, 50, 56

Bratman(1987) Michael E. Bratman. *Intention, Plans, and Practical Reason*. 1987. Cited in page 4, 15, 37, 39, 412, 422, 434

Brinksma and Tretmans(2001) Ed Brinksma and Jan Tretmans. Testing transition systems: An annotated bibliography. In Franck Cassez, Claude Jard, Brigitte Rozoy and Mark Ryan, editors, *Modeling and Verification of Parallel Processes*, volume 2067 of *Lecture Notes in Computer Science*, pages 187–195. Springer Berlin / Heidelberg, 2001. Cited in page 24

Briot *et al.***(2010)** Jean-Pierre Briot, Alessandro Sordoni, Eurico Vasconcelos, Vinicius Sebba Patto, Diana Adamatti, Marta Irving, Gustavo Melo and Carlos Lucena. *Design of an Artificial Decision Maker for a Human-based Social Simulation – Experience of the SimParc Project*, pages 17–35. Presses Universitaires Blaise-Pascal, 2010. Cited in page 23

Brooks(1986) Rodney A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2(1), 1986. Cited in page 20

Brooks(1991) Rodney A. Brooks. Intelligence without representation. *Artif. Intell.*, 47(1-3):139–159, 1991. ISSN 0004-3702. doi: http://dx.doi.org/10.1016/0004-3702(91)90053-M. Cited in page 20

Burch *et al.***(1990)** Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings, Fifth Annual IEEE Symposium on Logic in Computer Science, 4-7 June 1990, Philadelphia, Pennsylvania, USA*, pages 428–439. IEEE Computer Society, 1990. Cited in page 24

Burse(2000) Jan Burse. Quicksilver: A component-based environment for agent-based computer models and simulations. 2000. Cited in page 53

Cardelli and Gordon(1998) Luca Cardelli and Andrew D. Gordon. Mobile ambients. In *Proceedings of the First International Conference on Foundations of Software Science and Computation Structure*, pages 140–155, London, UK, 1998. Springer-Verlag. Cited in page 27

Catania(1998) Charles A. Catania. *Learning*. Prentice Hall, 1998. Cited in page 16, 84, 200

Chen and Roşu(2005) Feng Chen and Grigore Roşu. Java-MOP: A monitoring oriented programming environment for java. In *Proceedings of the Eleventh International Conference on Tools and Algorithms for the construction and analysis of systems (TACAS'05)*, volume 3440 of *LNCS*, pages 546–550. Springer-Verlag, 2005. Cited in page 27

Chen and Roşu(2007) Feng Chen and Grigore Roşu. MOP: An Efficient and Generic Runtime Verification Framework. In *Object-Oriented Programming, Systems, Languages and Applications(OOPSLA'07)*, pages 569–588. ACM press, 2007. Cited in page 27

Clarke(2008) Edmund Clarke. The birth of model checking. In Orna Grumberg and Helmut Veith, editors, *25 Years of Model Checking*, volume 5000 of *Lecture Notes in Computer Science*, pages 1–26. Springer Berlin / Heidelberg, 2008. URL `http://dx.doi.org/10.1007/978-3-540-69850-0_1`. Cited in page 24

Clarke and Emerson(1981) Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs Workshop*. Springer-Verlag, 1981. Cited in page 23

Clarke *et al.***(2001)** Edmund M. Clarke, Armin Biere, Richard Raimi and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001. Cited in page 24

Clarke *et al.***(1999)** Edmund M. Clarke, Jr., Orna Grumberg and Doron A. Peled. *Model checking*. MIT Press, Cambridge, MA, USA, 1999. ISBN 0-262-03270-8. Cited in page 24

Cohen and Levesque(1990) Philip R. Cohen and Hector J. Levesque. Intention is choice with commitment. *Artif. Intell.*, 42(2-3):213–261, 1990. Cited in page 4, 15, 412, 422, 434

Cormen *et al.***(2001)** Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001. Cited in page 50

da Silva(2005) Lourival Paulino da Silva. A formal model for the fifth discipline. *Journal of Artificial Societies and Social Simulation*, 8(3):6, 2005. ISSN 1460-7425. URL `http://jasss.soc.surrey.ac.uk/8/3/6.html`. Cited in page 30

da Silva and de Melo(2007) Paulo Salem da Silva and Ana C. V. de Melo. A simulation-oriented formalization for a psychological theory. In Matthew B. Dwyer and Antonia Lopes, editors, *FASE 2007 Proceedings*, volume 4422 of *Lecture Notes in Computer Science*, pages 42–56. Springer-Verlag, 2007. Cited in page 34

da Silva and de Melo(2008) Paulo Salem da Silva and Ana C. V. de Melo. Reusing models in multi-agent simulation with software components. In Müller Padgham, Parkes and Parsons, editors, *Proc. of 7th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2008)*, volume 2, pages 1137 – 1144. International Foundation for Autonomous Agents and Multiagent Systems, 2008. Cited in page 52

da Silva and de Melo(2011a) Paulo Salem da Silva and Ana C. V. de Melo. A formal environment model for multi-agent systems. In Jim Davies, Leila Silva and Adenilso Simao, editors, *Formal Methods: Foundations and Applications*, volume 6527 of *Lecture Notes in Computer Science*, pages 64–79. Springer Berlin / Heidelberg, 2011a. Cited in page 41

da Silva and de Melo(2011b) Paulo Salem da Silva and Ana C. V. de Melo. On-the-fly verification of discrete event simulations by means of simulation purposes. In *Proceedings of the 2011 Spring Simulation Multiconference (SpringSim'11)*. The Society for Modeling and Simulation International, 2011b. Cited in page 46

Dean *et al.***(2000)** Jeffrey S. Dean, George J. Gumerman, Joshua M. Epstein, Robert L. Axtell, Alan C. Swedlund, Miles T. Parker and Steven McCarroll. *Understanding Anasazi culture change through agent-based modeling*, pages 179–205. Oxford University Press, Oxford, UK, 2000. Cited in page 23

Denise *et al.***(2004)** A. Denise, M.-C. Gaudel and S.-D. Gouraud. A generic method for statistical testing. In *ISSRE '04: Proceedings of the 15th International Symposium on Software Reliability Engineering*, pages 25–34, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2215-7. URL `http://dx.doi.org/10.1109/ISSRE.2004.2`. Cited in page 25

Denise *et al.***(2008)** Alain Denise, Marie-Claude Gaudel, Sandrine-Dominique Gouraud, Richard Lassaigne, Johan Oudinet and Sylvain Peyronnet. Coverage-biased random exploration of large models and application to testing. Technical Report 1494, LRI, Université Paris-Sud XI, June 2008. Cited in page 25

d'Inverno and Luck(2003) Mark d'Inverno and Michael Luck. *Understanding Agent Systems*. Springer, 2003. Cited in page 30, 41, 56

d'Inverno *et al.***(1997)** Mark d'Inverno, David Kinny, Michael Luck and Michael Wooldridge. A formal specification of dMARS. In *Agent Theories, Architectures, and Languages*, pages 155–176, 1997. Cited in page 15, 30

Doi *et al.***(2005)** Takuo Doi, Yasuyuki Tahara and Shinichi Honiden. IOM/T: an interaction description language for multi-agent systems. In *Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*, AAMAS '05, pages 778–785, New York, NY, USA, 2005. ACM. Cited in page 21, 30

Doran and Palmer(1995) J. Doran and M. Palmer. *The EOS project: Integrating two models of palaeolithic social change*. 1995. Cited in page 23

Downing *et al.***(2001)** Thomas Downing, Scott Moss and Claudia Pahl-Wostl. Understanding climate policy using participatory agent-based social simulation. In Scott Moss and Paul Davidsson, editors, *Multi-Agent-Based Simulation*, volume 1979 of *Lecture Notes in Computer Science*, pages 127–140. Springer Berlin / Heidelberg, 2001. Cited in page 23

d'Inverno and Saunders(2005) Mark d'Inverno and Rob Saunders. Agent-based modelling of stem cell self-organisation in a niche. In Sven A. Brueckner, Giovanna Di Marzo Serugendo, Anthony Karageorgos and Radhika Nagpal, editors, *Engineering Self-Organising Systems*, volume 3464 of *Lecture Notes in Computer Science*, pages 52–68. Springer Berlin / Heidelberg, 2005. Cited in page 23

Epistein and Axtell(1996) Joshua M. Epistein and Robert Axtell. *Growing Artificial Societies: Social Science from the Bottom Up*. The MIT Press, 1996. Cited in page 23, 40

Eubank *et al.***(2004)** Stephen Eubank, Hasan Guclu, V. S. Anil Kumar, Madhav V. Marathe, Aravind Srinivasan, Zoltn Toroczkai and Nan Wang. Modelling disease outbreaks in realistic urban social networks. *Nature*, 429(6988):180–184, 2004. Cited in page 23

Ferber and Müller(1996) J. Ferber and J.-P Müller. Influences and reactions: a model of situated multiagent systems. In *Proceedings of ICMAS'96 (International Conference on Multi-Agent Systems)*. AAAI Press, 1996. Cited in page 5, 20, 30, 43, 44, 56, 414, 424, 436

Ferber(1999) Jacques Ferber. *Multi-agent systems – An introduction to distributed artificial intelligence*. Addison-Wesley, 1999. Cited in page 6, 13, 21, 31, 43, 44, 56

Fernandez *et al.***(1992)** Jean-Claude Fernandez, Laurent Mounier, Claude Jard and Thierry Jéron. On-the-fly verification of finite transition systems. *Form. Methods Syst. Des.*, 1(2-3):251–273, 1992. ISSN 0925-9856. URL `http://dx.doi.org/10.1007/BF00121127`. Cited in page 25

Ferster and Skinner(1957) Charles B. Ferster and Burrhus Frederic Skinner. *Schedules of reinforcement*. Appleton-Century-Crofts, 1957. Cited in page 244

Finkbeiner and Sipma(2004) Bernd Finkbeiner and Henny Sipma. Checking finite traces using alternating automata. *Form. Methods Syst. Des.*, 24(2):101–127, 2004. ISSN 0925-9856. Cited in page 26

FIPA(2003) FIPA. Fipa modeling: Interaction diagrams. Technical report, FIPA, 2003. URL `http://www.auml.org/auml/documents/ID-03-07-02.pdf`. Cited in page 21, 30

Freitas and Cavalcanti(2006) A. F. Freitas and A. L. C. Cavalcanti. Automatic Translation from *Circus* to Java. In J. Misra, T. Nipkow and E. Sekerinski, editors, *FM 2006: Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 115 – 130. Springer-Verlag, 2006. Cited in page 27

Frigg and Hartmann(2009) Roman Frigg and Stephan Hartmann. Models in science. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Summer 2009 edition, 2009. Cited in page 22

Gaudel(1995) Marie-Claude Gaudel. Testing can be formal, too. In *Proceedings of the 6th International Joint Conference CAAP/FASE on Theory and Practice of Software Development*, TAPSOFT '95, pages 82–96, London, UK, 1995. Springer-Verlag. ISBN 3-540-59293-8. URL `http://portal.acm.org/citation.cfm?id=646619.697560`. Cited in page 24

Gaudiano and Phone(1997) Paolo Gaudiano and Carolina Chang Phone. Adaptive obstacle avoidance with a neural network for operant conditioning: Experiments with real robots. In *CIRA '97: Proceedings of the 1997 IEEE International Symposium on Computational Intelligence in Robotics and Automation*, page 13, Washington, DC, USA, 1997. IEEE Computer Society. ISBN 0-8186-8138-1. Cited in page 19

Geilen(2001) Marc Geilen. On the construction of monitors for temporal logic properties. *Electr. Notes Theor. Comput. Sci.*, 55(2), 2001. Cited in page 26

Gilbert and Bankers(2002) Nigel Gilbert and Steven Bankers. Platforms and methods for agent-based modeling. *Proceedings of the National Academy of Sciences of the United States*, 99(Supplement 3), 2002. Cited in page 6, 22

Goldberg and Havelund(2005) Allen Goldberg and Klaus Havelund. Automated runtime verification with eagle. In Ulrich Ultes-Nitsche, Juan Carlos Augusto and Joseph Barjis, editors, *MSVVEIS*. INSTICC Press INSTICC Press, 2005. ISBN 972-8865-22-8. Cited in page 27

Groves(2002) Lindsay Groves. Refinement and the Z schema calculus. *Electronic Notes in Theoretical Computer Science*, 70(3):70 – 93, 2002. REFINE 2002, The BCS FACS Refinement Workshop (Satellite Event of FLoCl 2002). Cited in page 405

Halim and Seck(2011) Ronald Apriliyanto Halim and Mamadou Diouf Seck. The simulation-based multi-objective evolutionary optimization (SIMEON) framework. In *Proceedings of the 2011 Spring Simulation Multiconference (SpringSim'11)*. The Society for Modeling and Simulation International, 2011. Cited in page 29

Hannoun *et al.***(2000)** Mahdi Hannoun, Olivier Boissier, Jaime Sichman and Claudette Sayettat. MOISE: An organizational model for multi-agent systems. In Maria Monard and Jaime Sichman, editors, *Advances in*

Artificial Intelligence, volume 1952 of *Lecture Notes in Computer Science*, pages 156–165. Springer Berlin / Heidelberg, 2000. Cited in page 22

Henein and White(2005) Colin M. Henein and Tony White. Agent-based modelling of forces in crowds. In Paul Davidsson, Brian Logan and Keiki Takadama, editors, *Multi-Agent and Multi-Agent-Based Simulation*, volume 3415 of *Lecture Notes in Computer Science*, pages 173–184. Springer Berlin / Heidelberg, 2005. Cited in page 23

Herrnstein(1970) R. J. Herrnstein. On the law of effect. *Journal of the Experimental Analysis of Behavior*, 13:243 – 266, 1970. Cited in page 37

Hoare(1985) C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985. Cited in page 27

Holzmann(2003) Gerard Holzmann. *The Spin model checker: primer and reference manual*. Addison-Wesley Professional, first edition, 2003. ISBN 0-321-22862-6. Cited in page 32

Hutchison(2010) William R. Hutchison. A behavior analytic paradigm for adaptive autonomous agents. 2010. Available online at `http://www.behavior.org/resources/320.pdf`. Last retrieved in August 29, 2010. Cited in page 19

Hwang(2005) Moon-Ho Hwang. Generating finite-state global behavior of reconfigurable automation systems: DEVS approach. In *Proceedings of the IEEE International Conference on Automation Science and Engineering, 2005*. IEEE, 2005. Cited in page 29

Ingrand *et al.***(1992)** Francois F. Ingrand, Michael P. Georgeff and Anand S. Rao. An architecture for real-time reasoning and system control. *IEEE Expert: Intelligent Systems and Their Applications*, 7(6):34–44, 1992. ISSN 0885-9000. URL `http://dx.doi.org/10.1109/64.180407`. Cited in page 15

ISO/IEC(2002) ISO/IEC. Information technology – Z formal specification notation – Syntax, type system and semantics. Technical Report ISO/IEC 13568:2002(E), ISO/IEC, 2002. Cited in page 28, 35, 399

Jacky(1996) Jonathan Jacky. *The way of Z: practical programming with formal methods*. Cambridge University Press, New York, NY, USA, 1996. ISBN 0-521-55976-6. Cited in page 28, 35, 399, 404

Jakubow(2007) J. J. Jakubow. Review of the book sniffy the virtual rat pro version 2.0. *Journal of the Experimental Analysis of Behavior*, 87:317 – 323, 2007. Cited in page 19

Jard and Jéron(2005) Claude Jard and Thierry Jéron. TGV: theory, principles and algorithms: A tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems. *Int. J. Softw. Tools Technol. Transf.*, 7(4):297–315, 2005. ISSN 1433-2779. URL `http://dx.doi.org/10.1007/s10009-004-0153-x`. Cited in page 7, 25, 49, 56, 147, 418, 428, 440

Kakas *et al.***(2008)** Antonis Kakas, Paolo Mancarella, Fariba Sadri, Kostas Stathis and Francesca Toni. Computational logic foundations of KGP agents. *Journal of Artificial Intelligence Research*, 33:285–348, 2008. Cited in page 15

Kim *et al.***(2001)** M. Kim, S. Kannan, I. Lee, O. Sokolsky and M. Viswanathan. Java-MaC: a run-time assurance tool for java programs. In *Runtime Verification 2001 proceedings*, volume 55 of *ENTCS*. Elsevier Science Publishers, 2001. Cited in page 27, 31

Klugl and Puppe(1998) Franziska Klugl and Frank Puppe. The multi-agent simulation environment SeSAm. In *Proceedings of the Simulation in Knowledge-based Systems workshop*, 1998. Cited in page 22, 54

Laird *et al.***(1987)** John E. Laird, Allen Newell and Paul S. Rosenbloom. SOAR: an architecture for general intelligence. *Artif. Intell.*, 33(1): 1–64, 1987. ISSN 0004-3702. URL `http://dx.doi.org/10.1016/0004-3702(87)90050-6`. Cited in page 16, 40

Lea(1999) Doug Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley Longman Publishing Co., Inc., 1999. See also `http://www.cs.kent.ac.uk/projects/ofa/jcsp/`. Cited in page 27

Lee *et al.***(1997)** D. Lee, A. N. Netravali, K. K. Sabnani, B. Sugla and A. John. Passive testing and applications to network management. In *ICNP '97: Proceedings of the 1997 International Conference on Network Protocols (ICNP '97)*, page 113, Washington, DC, USA, 1997. IEEE Computer Society. ISBN 0-8186-8061-X. Cited in page 25

Li *et al.***(2011)** Xiaobo Li, Hans Vangheluwe, Yonglin Lei, Hongyan Song and Weiping Wang. A testing framework for DEVS formalism implementations. In *Proceedings of the 2011 Spring Simulation Multiconference (SpringSim'11)*. The Society for Modeling and Simulation International, 2011. Cited in page 29

Lichtenstein *et al.***(1985)** Orna Lichtenstein, Amir Pnueli and Lenore D. Zuck. The glory of the past. In *Proceedings of the Conference on Logic of Programs*, pages 196–218, London, UK, 1985. Springer-Verlag. ISBN 3-540-15648-8. Cited in page 27

Lomuscio *et al.***(2009)** Alessio Lomuscio, Hongyang Qu and Franco Raimondi. MCMAS: A model checker for the verification of multi-agent systems. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification*, volume 5643 of *Lecture Notes in Computer Science*, pages 682–688. Springer Berlin / Heidelberg, 2009. URL `http://dx.doi.org/10.1007/978-3-642-02658-4_55`. Cited in page 32, 56

Luke *et al.***(2004)** Sean Luke, Claudio Cioffi-Revilla, Liviu Panait and Keith Sullivan. MASON: A new multi-agent simulation toolkit. 2004. `http://cs.gmu.edu/~eclab/projects/mason/`. Cited in page 6, 22, 53, 56

Matarić(1998) Maja J. Matarić. Behavior-based robotics as a tool for synthesis of artificial behavior and analysis of natural behavior. *Trends in Cognitive Sciences*, 2(3):82 – 86, 1998. Cited in page 20

McCarthy *et al.***(2008)** James McCarthy, Tony Sabbadini and Sonia Sachs. Multi-agent model of technological shifts. In Luis Antunes, Mario Paolucci and Emma Norling, editors, *Multi-Agent-Based Simulation VIII*, volume 5003 of *Lecture Notes in Computer Science*, pages 112–127. Springer Berlin / Heidelberg, 2008. Cited in page 23

McCarthy(1958) John McCarthy. Programs with common sense. 1958. Cited in page 14

McCarthy(2008) John McCarthy. The well-designed child. *Artif. Intell.*, 172(18):2003–2014, 2008. ISSN 0004-3702. doi: http://dx.doi.org/10.1016/j.artint.2008.10.001. Cited in page 39

McDowell(2004) J. J. McDowell. A computational model of selection by consequences. *Journal of the Experimental Analysis of Behavior*, 81:297 – 317, 2004. Cited in page 19, 80

McDowell *et al.***(2006)** J. J McDowell, Paul L. Soto, Jesse Dallery and Saule Kulubekova. A computational theory of adaptive behavior based on an evolutionary reinforcement mechanism. In *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 175–182, New York, NY, USA, 2006. ACM. Cited in page 19, 41

McIlroy(1968) M.D. McIlroy. Mass produced software components. In P.Naur and B. Randel, editors, *NATO Conference on Software Engineering*. NATO Science Commitee, 1968. Cited in page 29

Miller(1956) George A. Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological Review*, 63:81–97, 1956. Cited in page 37

Milner(1999) Robin Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999. Cited in page 6, 10, 27, 42, 45, 116, 128, 415, 426, 438

Minar *et al.***(1996)** N. Minar, R. Burkhart, C. Langton and M. Askenazi. The Swarm simulation system: A toolkit for building multi-agent simulations. 1996. Working Paper 96-06-042. Cited in page 6, 22, 53, 56

Mysore V. *et al.***(2005)** Gill O. Mysore V., Daruwala R.S., Antoniotti M., Saraswat V.and and Mishra B. Multi-agent modeling and analysis of the brazilian food-poisoning scenario. In *Proceedings of Generative Social processes, Models, and Mechanisms Conference – Argonne National Laboratory & The University of Chicago*, 2005. Cited in page 23, 31

Nau *et al.***(2004)** Dana Nau, Malik Ghallab and Paolo Traverso. *Automated Planning: Theory & Practice*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004. ISBN 1558608567. Cited in page 41

Neisser(1967) Ulric Neisser. *Cognitive psychology*. Appleton-Century-Crofts, 1967. Cited in page 15, 37

North *et al.***(2006)** Michael North, Nick Collier and Jerry R. Vos. Experiences creating three implementations of the Repast agent modeling toolkit. *ACM Transactions on Modeling and Computer Simulation*, 16(1):1–25, 2006. http://repast.sourceforge.net/. Cited in page 6, 22, 31, 53, 56

North *et al.***(2005)** M.J. North, T.R. Howe, N.T. Collier and R.J. Vos. The repast simphony runtime system. 2005. `http://repast.sourceforge.net/`. Cited in page 53

Okuyama *et al.***(2005)** Fabio Y. Okuyama, Rafael H. Bordini and Antnio Carlos da Rocha Costa. ELMS: An environment description language for multi-agent simulation. In D. Weyns et al., editor, *Environments for Multi-Agent Systems*, volume 3374 of *Lecture Notes in Artificial Intelligence*, pages 91–108. Springer-Verlag, 2005. Cited in page 21, 43, 54, 56

Oliveira(2005) Marcel Vinícius Medeiros Oliveira. *Formal Derivation of State-Rich Reactive Programs using* Circus. PhD thesis, University of York, 2005. Cited in page 27

Parrow(2001) Joachim Parrow. An introduction to the pi-calculus. In J. A. Bergstra, Alban Ponse and Scott A. Smolka, editors, *Handbook of Process Algebra*, pages 479–543. Elsevier, 2001. Cited in page 407, 415, 426, 438

Peschanski and Hym(2006) F. Peschanski and S. Hym. A stackless runtime environment for a pi-calculus. In *Proceedings of the 2nd international*

conference on Virtual execution environments, pages 57–67. ACM, 2006. Cited in page 27, 52

Pierce and Turner(1997) Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. In G. Plotkin, C. Stirling and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 1997. Cited in page 27, 52

Popper(1959) Karl Popper. *The Logic of Scientific Discovery*. 1959. Cited in page 9

Queille and Sifakis(1982) Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the 5th Colloquium on International Symposium on Programming*. Springer-Verlag, 1982. Cited in page 23

Quenum *et al.***(2006)** José Ghislain Quenum, Samir Aknine, Jean-Pierre Briot and Shinichi Honiden. A modeling framework for generic agent interaction protocols. In *DALT*, pages 207–224, 2006. Cited in page 21

Rao(1996) Anand S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In *MAAMAW '96: Proceedings of the 7th European workshop on Modelling autonomous agents in a multi-agent world : agents breaking away*, pages 42–55, Secaucus, NJ, USA, 1996. Springer-Verlag New York, Inc. ISBN 3-540-60852-4. Cited in page 15, 41, 43

Rao and Georgeff(1993) Anand S. Rao and Michael P. Georgeff. A model-theoretic approach to the verification of situated reasoning systems. In *IJCAI*, pages 318–324, 1993. Cited in page 32

Rao and Georgeff(1995) Anand S. Rao and Michael P. Georgeff. BDI agents: From theory to practice. In Victor R. Lesser and Les Gasser, editors, *ICMAS*, pages 312–319. The MIT Press, 1995. ISBN 0-262-62102-9. Cited in page 4, 15, 39, 56, 412, 422, 434

Russell and Subramanian(1995) Stuart Russell and Devika Subramanian. Provably bounded-optimal agents. *Journal of Artificial Intelligence Research*, 2:575–609, 1995. Cited in page 15

Russell and Norvig(2002) Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, second edition, 2002. Cited in page 14, 18, 19, 20, 40

Sampaio *et al.***(2002)** Augusto Sampaio, Jim Woodcock and Ana Cavalcanti. Refinement in circus. In Lars-Henrik Eriksson and Peter Lindsay, editors, *FME 2002:Formal Methods—Getting IT Right*, volume 2391 of *Lecture Notes in Computer Science*, pages 1–15. Springer Berlin / Heidelberg, 2002. Cited in page 28

Schruben(2010) Lee Schruben. Simulation modeling for analysis. *ACM Trans. Model. Comput. Simul.*, 20(1), 2010. Cited in page 28

Shoham(1993) Yoav Shoham. Agent-oriented programming. *Artif. Intell.*, 60(1):51–92, 1993. ISSN 0004-3702. URL `http://dx.doi.org/10.1016/0004-3702(93)90034-9`. Cited in page 15, 39

Sichman *et al.***(1998)** Jaime Simão Sichman, Rosaria Conte, Yves Demazeau and Cristiano Castelfranchi. A social reasoning mechanism based on dependence networks. In Michael N. Huhns and Munindar P. Singh, editors, *Readings in agents*, pages 416–420. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998. Cited in page 22

Simon(1996) Herbert A. Simon. *Sciences of the Artificial.* The Mit Press, third edition, 1996. ISBN 0262691914. Cited in page 16, 36

Skinner(1948) Burrhus Frederic Skinner. 'Superstition' in the pigeon. *Journal of Experimental Psychology*, 38:168 – 172, 1948. An online version can be obtained at `http://psychclassics.yorku.ca/Skinner/Pigeon/`. Last retrieved in December 25, 2011. Cited in page 36

Skinner(1953) Burrhus Frederic Skinner. *Science and Human Behavior.* The Free Press, 1953. Cited in page 5, 16, 74, 413, 423, 435

Sowa(1987) John F. Sowa. *Semantic Networks.* Wiley, 1987. An updated version of this article can be found online on `http://www.jfsowa.com/pubs/semnet.htm`. Last retrieved in December 25, 2011. Cited in page 40, 67

Spivey(1992) J. M. Spivey. *The Z notation: a reference manual.* Prentice Hall, second edition, 1992. Cited in page 404

Stocker *et al.***(2001)** Rob Stocker, David G Green and David Newth. Consensus and cohesion in simulated social networks. *Journal of Artificial Societies and Social Simulation*, 4(4), 2001. URL `http://jasss.soc.surrey.ac.uk/4/4/5.html`. Cited in page 23

Szyperski(1999) Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley, second edition, 1999. Cited in page 29

Tobias and Hofmann(2004) Robert Tobias and Carole Hofmann. Evaluation of free java-libraries for social-scientific agent based simulation. *Journal of Artificial Societies and Social Simulation*, 7(1), 2004. Cited in page 22

Touretzky and Saksida(1997) David S. Touretzky and Lisa M. Saksida. Operant conditioning in skinnerbots. *Adapt. Behav.*, 5(3-4): 219–247, 1997. ISSN 1059-7123. URL `http://dx.doi.org/10.1177/105971239700500302`. Cited in page 18, 38, 40, 56

Tretmans(2008) Jan Tretmans. Model based testing with labelled transition systems. In Robert M. Hierons, Jonathan P. Bowen and Mark Harman, editors, *Formal Methods and Testing*, volume 4949 of *Lecture Notes in Computer Science*, pages 1–38. Springer, 2008. ISBN 978-3-540-78916-1. Cited in page 25

Tsvetovat and Latek(2009) Maksim Tsvetovat and Maciej Latek. Dynamics of agent organizations: Application to modeling irregular warfare. In Nuno David and Jaime Simão Sichman, editors, *Multi-Agent-Based Simulation IX*, pages 60–70. Springer-Verlag, Berlin, Heidelberg, 2009. Cited in page 23

van der Hoek and Wooldridge(2003) Wiebe van der Hoek and Michael Wooldridge. Towards a Logic of Rational Agency. *Journal of IGLP*, 11 (2):133–157, 2003. Cited in page 32

Visser *et al.***(2003)** Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park and Flavio Lerda. Model checking programs. *Automated Software Engineering*, 10:203–232, 2003. ISSN 0928-8910. URL `http://dx.doi.org/10.1023/A:1022920129859`. Cited in page 32

Wang and Wysk(2008) Jianrui Wang and Richard A. Wysk. A π-calculus formalism for discrete event simulation. In *WSC '08: Proceedings of the 40th Conference on Winter Simulation*, pages 703–711. Winter Simulation Conference, 2008. Cited in page 30

Wasserman *et al.***(1994)** Stanley Wasserman, Katherine Faust, Dawn Iacobucci and Mark Granovetter. *Social Network Analysis: Methods and Applications*. Cambridge University Press, 1994. Cited in page 22, 31, 42

Watkins(1989) Christopher John Cosnish Hellaby Watkins. *Learning from Delayed Rewards*. PhD thesis, Cambridge University, 1989. Cited in page 19, 39

Watson(1913) John B. Watson. Psychology as the behaviorist views it. *Psychological Review*, (20):158–177, 1913. Cited in page 16

Weiss(1999) Gerhard Weiss, editor. *Multiagent systems: a modern approach to distributed artificial intelligence*. MIT Press, Cambridge, MA, USA, 1999. ISBN 0-262-23203-0. Cited in page 4, 13

Weyns *et al.***(2005)** Danny Weyns, H. Van Dyke Parunak, Fabien Michel, Tom Holvoet and Jacques Ferber. Environments for multiagent systems: State-of-the-art and research challenges. In Danny Weyns *et al.*, editor, *Proceedings of the 1st International Workshop on Environments for Multi-agent Systems (Lecture Notes in Computer Science, 3374)*, pages 1–47. Springer, 2005. Cited in page 5, 20, 107, 414, 424, 436

Wilensky(1999) U. Wilensky. NetLogo. 1999. URL `http://ccl.northwestern.edu/netlogo/`. Cited in page 22, 54

Woodcock and Cavalcanti(2001) J. C. P. Woodcock and A. L. C. Cavalcanti. A concurrent language for refinement. In A. Butterfield and C. Pahl, editors, *IWFM'01: 5th Irish Workshop in Formal Methods*, BCS Electronic Workshops in Computing, 2001. Cited in page 28, 44

Woodcock and Davies(1996) Jim Woodcock and Jim Davies. *Using Z: Specification, Refinement, and Proof.* Prentice Hall, 1996. ISBN 0139484728. Cited in page 28, 35, 399

Wooldridge(2009) Michael Wooldridge. *An Introduction to MultiAgent Systems.* Wiley Publishing, 2nd edition, 2009. Cited in page 4, 13, 14, 18, 20

Wooldridge *et al.***(2006)** Michael Wooldridge, Marc-Philippe Huget, Michael Fisher and Simon Parsons. Model checking for multiagent systems: the MABLE language and its applications. *International Journal on Artificial Intelligence Tools*, 15(2):195 – 225, 2006. Cited in page 32, 56

Zeigler *et al.***(2000)** Bernard P. Zeigler, Herbert Praehofer and Tag G. Kim. *Theory of Modeling and Simulation.* Academic Press, 2 edition, 2000. Cited in page 28

Index

π-calculus, 27, 407
  implementation, 193
  implementation optimization, 193
  labelled transition system, 407
  operational semantics, 409
  process, 408
  simulation, 192
  structural congruence, 409
  use in EMMAS, 109

action, 75
  base level, 76
  conflict, 75
  conflict resolution, 77
  history, 76
agent, 14, 59
  architecture, 14, 33, 59
  behavioural principles, 16
  mentalistic principles, 36
algorithm, *see* verification algorithm
annotated transition system, 127, 128, 139
  event, 128
  input event, 128
  output event, 128
  run, 129
  simulation purpose, *see* simulation purpose
ATS, *see* annotated transition system

BDI, *see* Beliefs-Desires-Intentions
behaviour analysis, 59
  computational models, 38
  history, 16
behaviour elimination, 210
behaviour shaping, 244
behaviour-based robotics, 20
behavioural response, 73, 77
  elicitation, 95
  emission, 79, 92
  regulation, 99, 101
  scheduling, 77
  termination, 79
  update, 79
behaviourism, 16, 243
Behaviourist Agent Architecture, 59, 62
  implementation, 189
Beliefs-Desires-Intentions, 15
Burrhus Frederic Skinner, 16

certainty, 152
classical conditioning, 67, 200
cognitive psychology, 15
common sense, 14
consistency, 153

drive, 95, 96
  deprivation, 97
  satiation, 97

EMMAS, 108, 190
  semantics, 106, 130, 140
  translation function, 108
emotion, 98
  anger, 90, 99
  depression, 90, 99
  frustration, 87, 99

environment, 20, 105
examples
    multi-agent, 210
    single agent, 200
experiment, 190
experimentation, 9, 104, 243

feasibility, 151
FGS, 187

impossibility, 153
informativeness, 153
intelligence, 14
ioco relation, 25
Ivan Pavlov, 200

layered architecture, 244
literal, 129
logic, 14

magnitude, 63
model checking, 23, 50
model-based testing, 24
monitor, 26
motivation, 95
multi-agent system, 4

on-the-fly, 150
operant, 83
    conditioning, 88
    discrimination, 88
    elimination, 87
    emission, 92
    extinction, 88
    implication, 85
    negative punishment, 90
    negative reinforcement, 89
    positive punishment, 90
    positive reinforcement, 89
operant behaviour, 83
operant chaining, 206
operant conditioning chamber, 200
operant utility, *see* utility
organism, 62
    purpose, 83

passive testing, 25
practical rationality, 15
process algebra, 27
psychology, 15
punishment, 90

Q-learning, 19

random testing, 25
refinement, 28, 404
reflex, 92, 93
    adjustment, 95
    elicitation, 95
refutability, 152
reinforcement, 89
respondent behaviour, 92

satisfiability relation, 151
scenario, 190
schedules of reinforcement, 244
simulation, 22
simulation purpose, 139, 147, 190
simulator, 22, 154, 187
    architecture, 188
    input format, 385
    interface, 154
    parameters, 398
Skinner, *see* Burrhus Frederic Skinner
Skinner box, 200
social network, 9, 113, 216, 224, 232
software architecture, 188
software components, 189
SP, *see* simulation purpose
state-space, 5
stimulation, 65, 71
stimulus, 65
    conditioning, 70
    equivalence, 68
    implication, 67
stimulus utility, *see* utility
subsumption architecture, 20
synchronization, 148
    complementary event, 148
    event synchronization, 149

state synchronization, 149
synchronous product, 150
system model, 139

test purpose, 25
testing, 24, 49

utility, 16, 69
operant, 85
regulation, 96, 99, 100
utility function, 69

verdict, 152
verification, 23, 31, 139
runtime, 26
verification algorithm, 154, 155
certainty, 162
completeness, 166
complexity, 179
correctness, 179
cycles, 160
feasibility, 155
impossibility, 162
refutability, 162
soundness, 167, 172–174
termination, 161, 176

Z Notation, 28, 30, 62, 399