Goal-driven agents for the tolerance of unforeseen faults. A safety net for the programmers

Costin Caval



THÈSE DE DOCTORAT DE
l'UNIVERSITÉ PIERRE ET MARIE CURIE

Spécialité

Informatique

École doctorale Informatique, Télécommunications et Électronique (Paris)

Présentée par

Costin CAVAL

Pour obtenir le grade de

DOCTEUR de l'UNIVERSITÉ PIERRE ET MARIE CURIE

Sujet de la thèse :

Agents Dirigés par des Buts pour la Tolérance aux Fautes Imprévues

Un Filet de Sécurité pour les Programmeurs

Goal-Driven Agents for the Tolerance of Unforeseen Faults

A Safety Net for the Programmers

soutenue le 31 mai 2016

devant le jury composé de :

Mme. Amal El Fallah Seghrouchni,	Professeur Université Pierre et Marie Curie,	Directrice de thèse
M. Cyrille Enderli,	Ingénieur Thales,	Invité
Mme. Adina-Magda Florea,	Professeur Politehnica de Bucarest,	Rapporteur
Mme. Zahia Guessoum,	Maître de Conférences Université de Reims,	Examinateur
M. René Mandiau,	Professeur Université de Valenciennes,	Rapporteur
M. Patrick Taillibert,	Ingénieur,	Invité
M. Laurent Vercouter,	Professeur INSA de Rouen,	Examinateur

ABSTRACT

While fault tolerance is hot topic in software development, there are situations when potential faults can be omitted by the near-exhaustive identification and handling methods employed by "classic" approaches. Examples range from cases where the complexity hides faults from a rigorous development process, to cases where due to cost and time constraints on the fault tolerance effort, risks are assumed, either consciously or not. The main question this thesis addresses is "How should software be developed in order to be tolerant to *unforeseen faults*?", i.e. faults that were not covered in the implementation.

The first contribution of this thesis is a development framework – design, language and platform requirements – for producing software that is tolerant to *unforeseen faults*. We show that the use of a multi-agent architecture with goal-driven agents has numerous benefits for the confinement of errors and the subsequent system recovery. We propose language requirements that constrain the programmers in order to limit some of the possible faults and in the same time localise the areas where other faults can be present. The execution platform for the written code also needs to be adapted to take advantage of the resulting structure and trigger the necessary reparation, dependency handling and reconfiguration reactions in case of fault manifestations at runtime. We illustrate the approach by discussing the design and implementation of an application based on a well known multi-agent protocol (the CNP). For this, we propose an adapted agent-oriented programming language (ALMA+) and the corresponding platform. Just as a trapeze artist's "safety net", the use of our approach provides last resort mechanisms complementing the "classic" fault tolerance methods for improving the robustness of software applications.

The second contribution of the thesis focuses on the way goal-driven agents are programmed. The issue is that many approaches to cognitive agent modelling permit the agent developers to interleave the levels of plans and goals. This is possible through the adoption of new goals inside plans. These goals will have plans of their own, and the definition can extend on many levels. From a software development point of view, the resulting complexity can render the agents' behaviour difficult to trace, due to the combination of elements from different abstraction levels, i.e. actions and goal adoptions. This has a negative effect on the development process when designing and debugging agents. We thus propose a change of approach that aims to provide a more comprehensible agent model with benefits for the ease of engineering and the fault tolerance of agent systems. This is achieved by imposing a clear separation between the reasoning and the acting levels of the agent. The use of goal adoptions and actions on the environment inside the same plan is therefore forbidden. Our approach is illustrated in two agent-based applications: a maritime patrol application developed at Thales Systèmes Aéroportés (Thales Airborne Systems) and an ambient intelligence deployment software. We argue that by constraining the agent model we gain in clarity and traceability therefore benefiting the development process and encouraging the adoption of agent-based techniques in industrial contexts.

RÉSUMÉ EN FRANÇAIS

Dans le cadre de la tolérance aux fautes dans le développement logiciel, il y a des situations où des fautes potentielles peuvent être omises par les méthodes d'identification et de traitement quasi-exhaustif employées par les approches « classiques ». Les exemples vont des cas où la complexité cache les fautes même en présence d'un processus de développement rigoureux, à des cas où, en raison des contraintes en termes de coûts et de temps sur les démarches de tolérance aux fautes, des risques sont assumés, consciemment ou pas. La principale question que cette thèse aborde est « Comment le logiciel devrait être développé afin qu'il soit tolérant aux *fautes imprévues* ? », c'est à dire les fautes qui ne sont pas couvertes dans la mise en œuvre.

La première contribution de cette thèse est l'élaboration d'un cadre de développement – des exigences pour la conception, le langage de programmation et les outils employés – pour produire des logiciels tolérants aux *fautes imprévues*. Nous montrons que l'utilisation d'une architecture multi-agent avec des agents dirigés par des buts a de nombreux avantages pour le confinement des erreurs et la récupération ultérieure du système. Nous proposons des exigences au niveau du langage de programmation ayant pour but de contraindre les programmeurs afin de limiter certaines des fautes possibles et dans le même temps de localiser les zones où d'autres fautes peuvent être présentes. La plate-forme d'exécution doit également être adaptée pour tirer parti de la structure résultante et déclencher la réparation nécessaire, gérer les interdépendances des composants et la reconfiguration en cas de manifestation de fautes à l'exécution. Nous illustrons l'approche en étudiant la conception et la mise en œuvre d'une application reprenant un protocole multi-agent bien connu (le CNP). Pour cela nous proposons un langage de programmation orientée agent (ALMA+) adapté et la plate-forme correspondante. Tout comme le « filet de sécurité » d'un trapéziste, l'utilisation de notre approche fournit des mécanismes de dernier recours en complément des méthodes de tolérance aux fautes « classiques » pour améliorer la robustesse des applications logicielles.

La deuxième contribution de la thèse concerne la manière de programmer les agents dirigés par des buts. Le problème est que de nombreuses approches pour la modélisation des agents cognitifs autorisent les développeurs à entrelacer les niveaux des plans et des buts. Ceci est possible grâce à l'adoption de nouveaux buts à l'intérieur des plans. Ces buts ont leurs propres plans, et la définition peut s'étendre sur plusieurs niveaux. Du point de vue du développement logiciel, la complexité résultante peut rendre le comportement des agents difficilement traçable, en raison de l'entrelacement d'éléments de différents niveaux d'abstraction, à savoir les actions et les adoptions de buts. Ceci a un effet négatif sur le processus de développement lors de la conception et du débogage des agents. Nous proposons un changement d'approche qui vise à fournir un modèle d'agent plus compréhensible pour faciliter le travail des ingénieurs et augmenter la tolérance aux fautes des systèmes d'agents. Ceci est réalisé en imposant une séparation claire entre les niveaux de raisonnement et d'action des agents. L'utilisation des adoptions de buts et des actions sur l'environnement dans le même plan est désormais interdite. Notre approche est illustrée dans deux applications à base d'agents : une application de patrouille maritime développée à Thales Systèmes Aéroportés et une application de déploiement de logiciels dans le domaine de l'intelligence ambiante. En contraignant le modèle d'agent nous gagnons en lisibilité et traçabilité, avec un bénéfice pour le processus de développement. Cela aide aussi à l'adoption de techniques à base d'agents dans des contextes industriels.

ACKNOWLEDGMENTS

This thesis is the accomplishment of a voyage that saw the author start as an unpolished stone and work his way through many experiences to become who he is today. The chosen subject turned out to be more challenging than anticipated, but with perseverance and the help and goodwill of the many wonderful people whom I had the chance to meet and work with, the appropriate "tools" were in place for the project to be successfully completed. The vastness of the possible choices helped me learn how to filter out potential paths and choose the ones that are relevant and feasible given the time constraints. This gave me the opportunity to improve my autonomy with respect to decision making and work independently when needed. Working between the university and industry was an opportunity to study the two sometimes different perspectives and try to always find the best solution.

I would like to start by thanking the members of my defence jury for their appreciation of my work, their challenging questions and valuable feedback. I want to acknowledge my thesis reviewers, Professor Adina-Magda Florea and Professor René Mandiau, for their thorough and insightful reviews on my work. I would like to add a particular word for Ms. Florea who was been my mentor and supervisor during the bachelor and masters dissertations and opened me to the doors that led to this PhD. I would like to thank Professor Laurent Vercouter for his kind words and presiding the jury, and Professor Zahia Guessoum for her all her advice, both during the PhD and the defence.

I would like to express my gratitude towards my PhD Supervisor Professor Amal El Fallah Seghrouchni whose mentorship, trust, support, guidance and advice helped me learn, evolve and get through even the most difficult parts of the project. I would like to thank Patrick Taillibert, with whom I first discovered the subject that become my PhD and who then dedicated a part of his free time to our collaboration during my PhD, for his insights and the way he challenged and enriched my ideas, with an ever-present attention to details. I would also like to thank Cyrille Enderli for his understanding and care during my time at Thales.

I cannot express enough gratitude towards my parents Cristiana and George for the way they raised me and supported me through this all. Brenda deserves a special mention, for not only being the best sister one can have, but also a great friend. I would like to single out Magda for being by my side and being so understanding and empowering. I would also like to thank Ileana, Christian, Simona, Marc and Cristi for being my family away from home. Thank you Dorina for all your support.

I had the chance to have many great colleagues during my years at Thales and LIP6, whom I can unfortunately not mention all by name. I would like to thank Emmanuel, Kei, Kevin, Olivier and Thomas for their support during my final months at LIP6. A special thanks to Cédric Herpson and Sylvain Ductor for their advice and especially their invaluable input for the preparation of my defence. From Thales, I would like to note Hadrien, Hugo, Ludovic, Pierre-Yves and Romain for the good times we had in our lab, as well as Alex, Emilie, Fanny, Nico, Sophie, Will and many others from the Youth Employee Society who helped me feel at home in the company and discover many interesting industrial projects. I also want to thank Andrei Ciortea, Mihai Trascau, Andreea Urzica, Nicolas Vidal for the discussions we had and their advice and support at different stages of my journey. Many thanks to my friends Radu and Anca for our late night talks, their support and the enriching experiences we shared together. Alex, Alex (yes, there are more), Andrei, Bogdan, Cristi, Tudor and Vlad, thank you for your friendship.

A great thanks to all of you out there who had a direct or indirect, known or unknown, wanted or accidental, minuscule or enormous contribution to my work, my life and my personal growth during these unforgettable years.

CONTENTS

LIST OF ACRONYMS

AAMAS	Autonomous Agents and Multi-Agent Systems International Conference
AmI	Ambient Intelligence
ATMS	Assumption-based Truth Maintenance System
BDI	Belief, Desire, Intention
CNP	Contract Net Protocol
COMPS	system components set
DAG	Directed Acyclic Graph
EMAS	Engineering Multi-Agent Systems
FMEA	Failure Mode and Effects Analysis
GPS	Goal-Plan Separation
GPT	Goal-Plan Tree
GPU	Graphics Processing Unit
HTN	Hierarchical Task Network
IE	Inference Engine
JTMS	Justification-based Truth Maintenance System
LIP6	Laboratoire d'Informatique de Paris 6
MAS	Multi-Agent System
MBD	Model-Based Diagnosis
MDS	Mission Data System
MEA	Means-end analysis
OBS	observations set
RT	Reasoning Thread
SD	system description set
TMS	Truth Maintenance System
TPN	Time Petri Net
TSA	Thales Airborne Systems/Thales Systèmes Aéroportés
UAV	Unmanned Aerial Vehicle

Part I

INTRODUCTION AND STATE OF THE ART

1 INTRODUCTION

1.1 RAISON D'ÊTRE

> **Interviewer**: "HAL, you have an enormous responsibility on this mission, in many ways perhaps the greatest responsibility of any single mission element. You're the brain, and central nervous system of the ship, and your responsibilities include watching over the men in hibernation. Does this ever cause you any lack of confidence?"
> **HAL**: "Let me put it this way, Mr. Amor. The 9000 series is the most reliable computer ever made. No 9000 computer has ever made a mistake or distorted information. We are all, by any practical definition of the words, foolproof and incapable of error."
>
> — *2001: A Space Odyssey*, 1968. Film.

While the fault-free software application has always been a desideratum for programmers, project managers and users alike, and many fault tolerance techniques have pushed for ever more reliable software, from the occasional high profile catastrophe to the more common mundane annoyances, reality has proven that perfection is rarely achievable. Needless to say that the fictional dialogue above precedes the computer's transition towards a villain of the film, due to high-level behaviour faults. For a real example we turn to the notorious Ariane 5 crash [73] which was caused by a software bug coupled with a system-level fault. Less critical software is even more prone to encounter such unforeseen circumstances: smartphone users may be more accustomed to the occasional application crash. This thesis tackles the question of how to build programs that are less vulnerable to faults not taken into consideration in their implementation.

Nowadays, computers are given more and more important tasks, departing from their original "computational duties" from the days when the Enigma Machine was being broken, to control and decision tasks for various uses in factories, cars, planes etc. and one day, as HAL[1] in the dialogue above, spacecraft to take us to the stars. As the role of the human diminishes – e.g. Unmanned Airborne Vehicles ("drones") and autonomous cars – the responsibility and importance of the software increase. Even more less critical electronic devices surround, accompany and support us in our daily lives, from computers, to smartphones and now ubiquitous computing devices[2], all interconnected and run by software. While the stakes are different, all these need to offer a corresponding degree of assurance with respect to their correct functioning.

The autonomy of devices, from the manufacture's and well as the user's point of view, is the fact that these devices function without any or with only minimal human inputs. This also means that in case of faults, they are expected to be able

1 HAL, **H**euristically programmed **AL**gorithmic computer, is the artificial intelligence that controls the systems in the film's spacecraft and interacts with the crew.

2 One can now buy smart power plugs that allow the monitoring and control of power consumption, light bulbs that are controlled via the Internet and even smart kettles that boil the water exactly when your phone tells them you need your coffee.

to continue functioning without much disruption. The recent example the Rosetta Mission [37] for exploring the comet Churyumov–Gerasimenko is evocative for not only is it out of the reach of any repair team, but communication time (up to 52 minutes in each direction at the farthest point during the mission) and bandwidth were also extremely limiting for most earth-based measures.

Most of these systems exhibit complexity in a form or another, either in their internal design – e.g. a plane with its subsystems: avionics, radar, communications etc. – or through their distributed nature – e.g. the devices in a connected home or an internet-based application – or both – e.g. a fleet of drones. As these applications grow in complexity, the task of rendering them fault tolerant becomes more and more challenging. A method for the fault tolerance that is frequently used in the industry is the Failure Mode and Effects Analysis (FMEA) [107] which requires listing all possible faults and specifying the handling of each, which is difficult in open and complex systems. Software testing does not provide a perfect validation either, as it is notoriously known "to show the presence of bugs, but never to show their absence!" [32]. On top of these susceptibilities, real life projects are subject to project management constraints – i.e. taking into account system criticality, costs, time to market etc. When the stakes are high, the margin of error is narrow, and this comes with a price: exhaustive tests and evaluations for complex systems take many man-hours and require highly specialised workforce. This can result in an increased chance to omit faults.

Proof and verification techniques are also gaining recognition (they are, under specific conditions, allowed even for avionics software certifications DO-178C [78]) but they require an enormous amount of work (e.g. to the 2.2 man years required for developing a microkernel, [63] needed another 11 man years for the proof).

Once they are identified, faults are either (1) removed altogether through changes in design, (2) provided with tolerance mechanisms (e.g. redundancy) or (3) accepted as possible, depending on the risk versus cost analysis. This means that even in critical systems, there is always a calculated, albeit very small, probability of failure: for example a 1 in 10 billion probability of failure for the Boeing 777 flight computer is defined in its requirements [118]. Fault tolerance can be expensive and may involve trade-offs for the original system (e.g. five-point harnesses are seatbelts used in automotive racing for better safety, but are too cumbersome to use on a day to day basis). When time is an issue, for example in prototyping or just when the time to market criterion prevails, fault tolerance may be given less importance. Furthermore, component re-usability and validation, while largely beneficial, do not always guarantee a smooth ride, as proven by the Ariane 501 accident [73] cited before, where a reused piece of equipment failed due to the different running context in which it was used. Reliability is therefore a matter of assumed risks.

At runtime, these accepted faults, together with the unidentified ones, can produce errors that the system is not prepared for – *unanticipated* errors – and that can cause catastrophic results, or just unpleasant experiences for users. We call these faults *unforeseen faults*. The concept emerged while discussing with engineers working with well-established methods such as FMEA. As stated before, these methods require the designers to identify *all* possible faults and prepare for *all* possible errors linked to these faults. The engineers' questions were: "What happens if we overlook a fault case? How can we improve the behaviour of the system in such situations?".

When even compilers are shown to contain errors [117], we can try to change the point of view to a higher level and focus on results, using a "let it crash" [2]

approach for lower level components, while ensuring higher level controls and recovery solutions.

As software becomes more and more complex, better development tools are needed to cope with the increased risk of errors, from the models, languages and methodologies, to platforms and development environments. More code often means more "opportunities" for errors, or, in the form of the software engineers' joke paraphrasing Einstein's formula:

$$E = m \cdot c^2 \leftrightarrow \text{Errors} = \text{more} \cdot \text{code}^2$$

The code for handling errors can be source of further errors itself. Furthermore, exception handling was shown to be treated lightly by programmers [16], who often use generic catchers and do not provide recovery measures – only logging and then terminating the execution. A means for runtime and as well as user generated exceptions to be caught and handled automatically is therefore needed in order to ease the task of programmers and result in more dependable applications.

Over the years, in parallel with the need for more and more complex applications, the evolution of software has been accompanied by a constant preoccupation for reliability. From early languages to Assembler and then Java and beyond, the programming paradigm and language evolution has been a constant string of abstractions that gave programmers more power while often limiting their possibility to make mistakes. An example is the abolition of `goto` for its facilitation to create difficultly readable "spaghetti code" [113]. Agent oriented programming follows this trend in offering a higher level of abstraction, coupled with the framework and tools for developing modular and distributed software, thus supporting the development of complex software [8]. The modularity of agents and the loose coupling associated with their message-only communications recommend them for the development of dependable programs. However, this paradigm has two tricks up its sleeves which we argue are at least as important for the dependability of the resulting systems: goal-directed agents and autonomous agents.

Using agent goals helps structure the human programmer's thoughts and improves the design process, facilitating development methodologies [71]. Furthermore, when used inside agent design, goals also guide the agents' behaviour at runtime. The goals' property to describe the desired outcome make them well suited for tolerating faults [91]: as long as the goal's satisfaction condition is not fulfilled, regardless of the reason, the goal is not successful and the agent may try again. This means that even errors that were not caught can be masked, as long as they impact a goal's outcome. Another property that is interesting for the fault tolerance is that achieved goals can act as checkpoints for the agent behaviour: they provide intermediary verifications and can be used for roll-back strategies in case of errors.

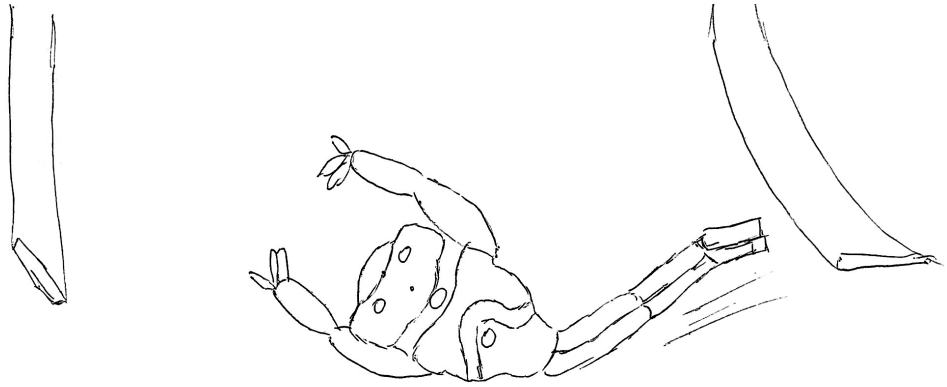
While a lot of interest goes into building autonomous systems, we aim at guiding the programmers' attention to a different perspective: what does it mean to build agents that interact with autonomous agents? Autonomy can be perceived from the outside as "the right to say *no*" [86]. From this perspective, designing code that can interact with autonomous agents means not taking their predictability for granted, thus being ready for any situation, such as a message reply that is not sent because the agent decided not to do so. In reality, the lack of reply can be due to an agent decision, overload or even because the message was lost, but the important aspect here is that an agent should not work under the supposition that its peers will necessarily reply to any request. The resulting systems are therefore

more loosely coupled and more robust. Note that this state of mind is beneficial in multiple contexts, including open systems and systems which are developed by different teams, also called heterogeneous.

When pursuing this line of thought, a programmer can doubt any data inputs – e.g. from other agents, sensors, other modules, human operators etc. Ideally, one would be able to evaluate the validity of data before acting, but most of the times, even as humans, we are forced to reason and act based on assumptions. For example one may prepare for the summer holidays by buying the plane tickets and booking hotels, "knowing" that there are no constraints at work that can prevent her for leaving. Then, at the last minute, she finds out about a meeting that had been scheduled long before the vacation plans without the person's knowledge prevent her for leaving, she needs to evaluate the situation and take the necessary measures. Maybe finding another colleague for the meeting is possible to continue with the original vacation plans, or maybe rebooking the outbound flight is enough. In the worst case, one may just accept that the money were lost and there is nothing left to do then go to work. The idea here is that as humans we are able to adapt because most of the time our beliefs are, consciously or not, assumptions that allow us to act and we can reconsider our actions if at a later moment these assumptions are contradicted. In a context where unforeseen faults are acknowledged to appear, having the possibility to work with assumptions is therefore an important feature, as pursuing a course of action may need to be stopped due to an error in the system. Tools for handling assumptions were already proposed for applications dealing with uncertainty, e.g. diagnosis [62], and recently included in the ALMA agent programming language [29] which we used in our work.

So far we have introduced the unforeseen fault issue and a few existing techniques that provide good properties for the fault tolerance.

This thesis proposes a development approach with the aim of producing programs that are implicitly tolerant to residual faults – unforeseen faults. Our approach covers the used paradigm, through the programmer's state of mind – the ideas that guide her design and programming – to the language and platform used.



<— safety net here —>

1.2 WEAVING A NET

In trapeze[3] shows, one or more artists perform various acrobatics including jumps at a considerable height. While the shows are usually well rehearsed and the artists take all the necessary precautions, given the difficulty of their acts and the risks involved, a *safety net* is usually placed below them. The net, which is not normally involved in the actual shows, provides a last resort for unexpected situations: an artist falls due to a miscalculated jump, slipping etc. Let us note the following three characteristics of the safety net:

- it is a *last resort* solution, being complementary to other, usually more desirable, techniques. For example the trapezist would be better off skipping a difficult move and continuing the show than falling into the net. In the same time, having the net also allows novices to try the trapeze without much risk.
- it is a *generic* solution covering many cases, from a trapeze breaking to a trapezist sneezing, missing a jump and falling.
- it is *not intrusive*: the net does not interfere with the show.

There are many parallels that can be drawn between the trapeze show and computer software. Organising the show – e.g. deciding on the number and attributes of artists, the choreography, the safety measures and so on – corresponds to the software engineering effort. Using many artists in a single show may be likened to the use of the multi-agent paradigm: they offer more expressiveness for a more complex show and in case one or more of them encounters a problem, the others can most likely still continue without. Let us now focus on the three safety net properties listed above, in the context of our approach for the tolerance to unforeseen faults:

- a solution meant for faults that escaped the used fault tolerance mechanisms – i.e. the unforeseen faults – acts as a *last resort* solution;
- an unforeseen fault's characteristics are obviously unknown, so it is important that the provided handling solution be *generic*;
- as a general software development objective and because the solution's use will be very limited (as unforeseen faults are meant to manifest rarely), it is important for it to be light and *not intrusive*.

3 A *trapeze* is "a short bar hanging high up in the air from two ropes, which acrobats use to perform special movements" (cf. http://dictionary.cambridge.org/dictionary/british/trapeze).

Note that the subtitle of this thesis is "A Safety Net for the Programmer" and this implies that the aim is not to save particular components of the final application, but the overall system functionalities. This implies that (1) particular components may be allowed to crash for the benefit of the overall application and (2) the recovery of the application after a local incident is very important as well. The second point especially may go against the first intuition of the metaphor that may make one think that once the artist fell into the safety net, the story is over. In the context of a show with an audience, however, recovery is as important for the overall show as it is for the particular artist: the show would surely stop in case someone actually got badly injured, while with the safety net, that artist can – depending on the reason of his or her fall – even go back and continue to contribute to an overall successful show.

Tolerating unforeseen faults is concerned with two aspects: (1) the offline preparation phase – where the system is engineered by designers and/or programmers following our requirements and using the tools we provide – which then results in (2) a safety net behaviour at runtime.

The idea of the thesis is that the programmer is able to focus on the definition of the behaviour of the system and assign any effort he or she desires to the fault tolerance, with our approach providing a supplementary level of robustness, a safety net. For this, the programmer is required to write goal-driven agents that are meant to interact with autonomous components – agents or other types of entity. The programmer is guided through programming constraints to define the agents in a certain way – including writing reparation code in specific locations in the code – so that the provided error recovery mechanisms – including the hypothesis-based use of inputs – can act as a safety net.

The result is a program which has its own fault tolerance mechanisms, but can also tolerate faults that were not normally covered by these "classic" mechanisms. In the best case scenario, the resulting error is masked by the goal driven agent that has the necessary plans and resources to reach its goals, despite the manifestation of the fault. In the worst case, the language structure guides the behaviour towards a correct shut-down or mission abort, keeping within the specifications.

Let us switch now from the point of view of the programmer to the point of view of the fault tolerance engineer. Our work will need to focus on the latter in order to provide the programmer with the desired safety net effect. The goal is that the programmer can produce fault tolerant software without even realising it by simply defining the functional aspects of the program using our tools and complying with our requirements.

We will study fault tolerance using a three phase approach to produce the desired safety net behaviour. First, an error needs to be *detected*. As we are aiming at unforeseen faults, the detection needs to be implicit. For example, code "crashes" (e.g. segmentation faults, divisions by zero) are usually detected by the runtime environment without any input from the programmer. However, as known faults can benefit from our safety net too, we do include the means for a programmer to call our mechanisms willingly, just like a trapezist that decides to let go and fall in the safety net.

Following the error detection, the *confinement* phase has the goal to limit the propagation of the error in the system from the point of detection. Here we take advantage of the modular architecture provided by agents together with their goals and plans. The third and most complex phase concerns the *recovery* of the system. First, other agents and plans that may have been impacted already – e.g. transmit-

ted possibly corrupted data – are identified and informed of the detection. Then, any available reparation code, coupled with the power of the goals to reconfigure provide the means to bring the system towards a nominal behaviour. The goals also provide a last resort for undetected errors as the goal verification conditions can trigger the recovery even without a detection event, just because the condition was not satisfied.

Going back to the metaphor, the true safety net in our approach is the agent-goal-plan architecture that facilitates confinement and offers the necessary base for recovery, this placing the goal-driven agents at the centre of our approach. This also means that the level of granularity is important: smaller agents with shorter plans and more specific goals create a net with a finer and more resistant mesh.

Another very important role will be played by the programming language which, together with the other design requirements will ensure the necessary elements are in place for the moment when the unforeseen fault manifests. This is like a person who surveys the preparations for the show to ensure that the acrobats will be working above the safety net and they know what to do once they fall into it – e.g. get back in the show or leave the stage gracefully.

This work does not claim to surpass or overthrow well established fault tolerance approaches – especially in the world of critical systems for which more mature propositions are required –, but to explore a new possible path. Furthermore, specific solutions for specific types of fault will most likely provide better optimised results than our generic approach, hence the complementarity of our work.

Limitations

A problem in fault tolerance is that errors are not necessarily detected at the moment when they are produced, they may spread throughout the system just as an unknown computer virus that spreads to many machines and only later its manifestation (e.g. deleting files) is triggered by a condition (e.g. a specific date or a command). Similarity, errors that are below a detection threshold can spread to many components until their detection. Our dependency handling step which is part of the recovery phase aims to limit the impact of propagating errors. However, given that the main focus of this work is not the detection and the means discussed are usually generic and aimed at unforeseen faults, our work may be even more susceptible to such issues.

Also, while we use tools from the diagnostic domain, we are not attempting to diagnose the cause of an error due to the complexity of the task.

As our work has a methodological component, the actual design and programming work is very important for obtaining the desired results. While we provide tools and requirements, the final responsibility lies with the authors of the design and code. Furthermore, just as an acrobat may rely too much on the safety net, a programmer may end up taking too many risks and end up too often on the mechanisms proposed in this work. Design and code verification – e.g. through peer review – would therefore be needed to ensure the safety net requirements are complied with.

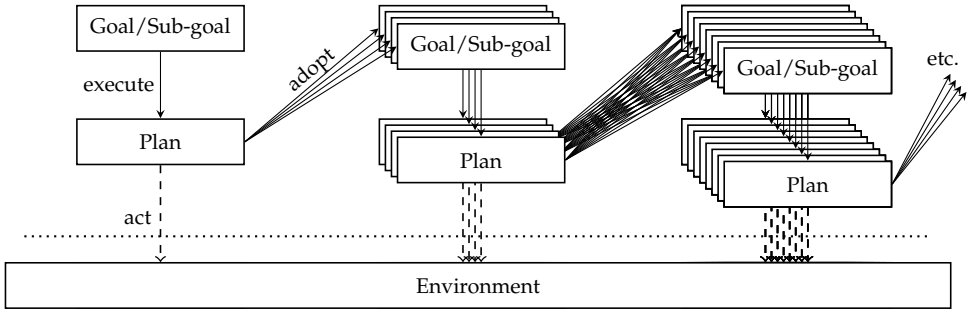


Figure 1: Agent complexity when goals are adopted in plans acting on the environment

1.3 SEPARATING REASONING FROM ACTING

In our pursuit for fault tolerant systems, we chose goal-driven agents for the good properties offered by this paradigm for our safety net. However, we considered that the actual use of goals and plans in many platforms can be improved.

The purpose of an agent is usually to act on the environment, which is done through its plans. Actions can involve the use of actuators, but they also cover the sending of messages[4]. However, in practice, various works [49, 109] and programming frameworks (Jason [11], Jadex [14] etc.) employ a model where plans can also adopt new goals, often termed *sub-goals*. A goal can thus have multiple possible plans, whose success depends on the achievement of their respective sub-goals and this can extend on many levels (Fig. 1). Note however that the successful completion of a plan does not necessarily guarantee the achievement of a goal, as goals can have success and failure conditions [99].

While it may be straightforward to design in this way, the fact that in a plan (1) actions on the environment – i.e. with effects "outside" of the agent – and (2) goal adoptions – i.e. with effects on the, possibly long-term, reasoning and behaviour of the agent – are used together in the same structure can have adverse effects on the resulting agents: low intelligibility during design, difficult traceability during execution and poor reusability afterwards.

This recursive construction has the advantage of using already existing Belief, Desire, Intention (BDI) building blocks and can help abstract certain aspects of an agent's behaviour offering the possibility to define the agent in a top-down approach. However, it also creates a structure which is difficult to trace, especially when actions occur at any level, and whose depth may be unpredictable. Important aspects in the behaviour of an agent might be hidden from the eyes of a developer or code reviewer due to this intricate design. One might always wonder whether the current plan is a terminal one or whether the model continues with further sub-goals. Given that the adoption of a goal usually implies a new reasoning process with an automaton and further plans, the goal adoption should not be treated the same as an atomic action.

For a change of perspective, let us take the example of the army as a clear-cut multi-level organisation. A soldier executes the orders (goals) given from "above" but cannot make high level decisions. Strategies and new objectives (goal adoptions) are decided by the higher ranks. This is due to the separation of responsibilities and competences, as well as the soldier's limited view of the situation. In a similar way, an agent's goals should not be mixed with the acting. This would

4 We do not consider belief revision to be an action.

also allow plans to have limited interdependencies, just as the soldier has a limited view of the situation, with benefits on complexity and fault confinement. A similar analogy can be made with other hierarchical human organisations such as companies, where the management decides – either on a single or at multiple levels – before requiring the workers to perform the required tasks. Needs that can arise have to be discussed with the manager or managers, who can then decide to take new measures, just as an agent's reasoning would adopt new goals. While small companies with a "flatter" hierarchy can cope with certain issues faster, complex organisations have proven to benefit from this hierarchical composition[5].

Agent oriented development methodologies such as Tropos [44] and Prometheus [114] have top-down approaches where they start with system level characteristics to then "descend" towards agent goals before defining plans and other low level details. Implementing agent systems modelled using methodologies such as these would also be more natural if reasoning and acting were more clearly separated.

Several works [27, 53, 99, 115] have argued for the interest of using declarative *goals-to-be* together with procedural *goals-to-do*, for decoupling goal achievement (the "to be" part) from plan execution (the "to do" part), giving the agents their *pro-activeness*, but also better flexibility and fault tolerance. Taking this delimitation a step further, we argue for the interest of separating a level where goal reasoning takes place – managing goal adoptions, dependencies, conflict resolution – from an action level where the agent interacts with its peers and environment.

While at runtime it is useful and even inevitable to alternate between reasoning and acting, we argue that these already conceptually distinct levels should be kept separate when designing agents.

To address these issues we propose a subtle change in the agent modelling that simplifies the agent representation by requiring the actions on the environment to be separated from the goal adoptions. We call the approach *Goal-Plan Separation (GPS)*. As shall be seen, the direct consequence of this separation is the structuring of the agent into two levels: one concerned with goals and one concerned with actions.

Publication

Our work on the subject produced the GPS approach (Part III of the thesis) was presented in the Engineering Multi-Agent Systems (EMAS) 2014 Workshop and later published in a reviewed form in a Springer volume dedicated to the Workshop [19]. We later applied the GPS approach for papers to be presented at AAMAS [81] and EMAS [80], both in May 2016.

5 Note: while we are presenting examples of organisations with many people, our scope remains the design of the reasoning of a single agent, which would thus correspond to the army or the company as a whole.

1.4 DEFINITIONS AND WORKING HYPOTHESES

> "The elevation was probably not under 11,000 feet [...]. At the place where we slept water necessarily boiled, from the diminished pressure of the atmosphere, at a lower temperature than it does in a less lofty country; the case being the converse of that of a Papins digester. Hence the potatoes, after remaining for some hours in the boiling water, were nearly as hard as ever. The pot was left on the fire all night, and next morning it was boiled again, but yet the potatoes were not cooked. I found out this, by overhearing my two companions discussing the cause; they had come to the simple conclusion, that the cursed pot [which was a new one] did not choose to boil potatoes."
>
> — Charles Darwin, *The Voyage of the Beagle*, originally published in 1839.

THESIS CONCEPTS Let us now define the main concepts that we will be using in the thesis. The descriptions of the following dependability-related concepts are based on the work by Avizienis et al. [4], while the agent-specific and other concepts are based on the author's own view, as well as other works, as specified. The two definitions are part of the author's contribution.

A **(service) failure** is a situation in which the service no longer performs as required by its functional specifications (which usually has a functionality as well as performance component). An **error** is a deviation of the external state of the system, so a service failure is a succession of errors.

Definition 1. *An **unanticipated error** is an error for which no specific handling exists in a system.*

An unanticipated error is therefore left to the platform and will probably cause a component failure, or "crash".

An *exception* is special situation that is signalled by an invoked operation to its caller, that is then permitted or even required to react to this condition [45]. In software development, exceptions are often used to indicate and treat error cases. The common verbs used for generating and handling an exception are "throw" and "catch".

The determined or hypothetical cause of an error is called a **fault**. A fault may manifest to become active and produce an error, or remain "dormant".

Definition 2. *An **unforeseen fault** is a fault that was not covered when building a system.*

As stated before, a fault can be unforeseen because (1) it was not identified at all or (2) it was identified during design time but was consciously ignored (e.g. due to high costs, low risk etc.).

Examples of unforeseen faults:

1. residual code error ("bug"), uncaught exceptions: "segmentation fault", division by zero etc.;
2. system error: an error code interpreted as data (Ariane 5 [73]);
3. hidden variables: when Darwin's men were unable to cook potatoes as they were not aware of the influence of the altitude on the boiling point of water (as in the quote above);

4. unconsidered situation: an important computer for the system in question stops (for example the power cable is disconnected).

The purpose of **fault tolerance** is that no service failures occur despite the presence of faults. A system is said to be in a **degraded mode** when due to partial failures, it can only provide a subset of its services. In this case we say a system's functionality or performance suffered a partial failure. The **coverage** of a fault tolerance technique is the measure of its effectiveness.

While many different definitions of the concept exist, depending on the application domain and field of computer science research, we define a software **agent** as a clearly delimited software entity that does not share memory with other entities and communicates through messages.

A Multi-Agent System (MAS) is "a set of software agents that interact to solve problems that are beyond the individual capacities or knowledge o each individual agent" [86]. In this work, we adopt the distinction between *(intelligent) agents* whose behaviour is autonomous and pro-active, and *artifacts* [93] which are the tools or services used by the agents.

The use of agents for designing and programming systems can be referred to as a paradigm and the resulting program structure is sometimes called a *multi-agent architecture*.

A belief is an agent's momentary representation of a particular characteristic of the environment or itself. Beliefs correspond to variables in "classic" programming languages but they usually have another component than their value which deals with belief revision, for example generating events when the belief value changes in Jadex [14] or keeping that belief's justification in ALMA [29] (see Sec. 2.4).

A **goal** is the state that an agent wishes to bring about. While this is a generic definition, we shall see in Sec. 2.3 that goals can be used to represent agent pro-activity in different ways: by requiring a plan to be executed without regard to the system state, maintaining a system state etc.

A **goal-driven agent** is an agent built to pursue explicit goals. BDI agents are goal-driven agents.

A **plan** corresponds to the sequence of operations that the agent can use to pursue a goal.

For an agent, the **environment** is comprised of everything else that is exterior to that agent. An **action** is an interaction emanating from an agent towards its environment (e.g. sending a message or using an actuator such as a robotic arm, as long as this latter is considered included in the agent rather than a stand-alone artifact). Operations internal to the agents – belief writes – are not considered actions.

THESIS CONTEXT This work is concerned with systems whose size and complexity or needs for distribution can justify the use of a multi-agent architecture.

The thesis focuses on the improvement of the development process in order to obtain the desired runtime behaviour in the presence of faults. It is important to delimit our work – which is concerned with fault tolerance – from other reliability-related domains such as safety, security, robustness, reliability, even availability (defined and compared in [4]).

1.5 THESIS STRUCTURE

This first part of the thesis continues with a state of the art (Chapter 2) covering elements of fault tolerance that we relate to the unforeseen faults (Sec. 2.1), as well as to agents (Sec. 2.2). We then discuss goal-driven agents in Sec. 2.3. We conclude the state of the art by introducing ALMA, an agent language built for agents dealing with uncertainty and which, as we will see, incorporates elements that are of interest for the tolerance of unforeseen faults (Sec. 2.4).

Part II of the thesis is dedicated to the safety net approach for the tolerance of unforeseen faults. Chapter 3 details our safety net approach for the tolerance of unforeseen faults. In this chapter we detail the 3 phases for fault tolerance introduced above: detection, confinement and recovery. We first present various methods that can be used for each phase, and then present our choices and contributions for each. This allows us to distinguish between the design, the programming language and the technical aspects of our approach, which we concentrate into 10 principles. We then continue in Chapter 4 with the extension of the ALMA agent programming language [29] in order to make it and its platform compliant with the safety net principles. This then allows us to illustrate the safety net at work in an implemented example using a scenario based on the Contract Net Protocol (CNP) (Chapter 5).

In Part III we describe our contribution to agent design: the Goal-Plan Separation approach. In Chapter 6 we describe the approach and then, in Chapter 7 we provide an example of implementation. In Chapter 8 we present two applications on which we experimented the GPS approach.

In Part IV we discuss the conclusions of this thesis and the perspectives that this work offers.

SETTING

This work has been carried out at Thales Airborne Systems/Thales Systèmes Aéroportés (TSA), Elancourt, France, and at the Laboratoire d'Informatique de Paris 6 (LIP6) at Pierre et Marie Curie University. We built, among other, on heritage from the PhD work of Sylvain Dekoker on the ALMA hypothetical reasoning-based agent programming language [28, 29], Katia Potiron on fault tolerance in multi-agent systems and autonomy [85, 86] and Caroline Chopinaud on norms and autonomy [23, 24]. An internship by Xavier Jean offered the first ideas on the tolerance to unforeseen faults, before the author's own internship on the matter preceded this PhD.

This work was supported through a CIFRE[6] grant from the ANRT[7].

6 *Convention Industrielle de Formation par la REcherche.*

7 *Association Nationale de la Recherche et de la Technologie.*

2

STATE OF THE ART

This thesis aims to propose solutions for programmers to help them produce systems that are tolerant to unforeseen faults. The first part of this chapter (Sec. 2.1) is dedicated to fault tolerance methods that can be related to the concept of "unforeseen fault". We then discuss (in Sec. 2.2) fault tolerance in the context of multi-agent systems, before going into more details regarding the goal-driven agent representations (Sec. 2.3). We end this chapter with the description of ALMA, an agent programming language designed for working under uncertainty and which has several fault tolerance properties.

2.1 THE TOLERANCE OF UNFORESEEN FAULTS

We start by revisiting the concept descriptions from Sec. 1.4, based on [4].

THREATS A *fault* is the judged or hypothesised cause of an error. A fault is dormant until manifested into an error. *Errors* are one or more abnormal external states of a system and can lead to failures if not contained and handled properly. A *failure* describes the functioning of a system that does not perform according to its specifications. For example, a car's flat tire is an error, with the nail that the car ran over being the fault and the failure being the impossibility to continue the trip. However, as this situation was foreseen when the car was built, there is usually a spare wheel that can replace the damaged one thus avoiding the failure. Note that these three concepts are a matter of perspective, as for example the failure of an agent can be only an error at Multi-Agent System (MAS) level, provided that the overall system continues to perform according to its specifications.

DEPENDABILITY Dependability is the ability of a system "to avoid failures that are more frequent or more severe than is acceptable" [4]. The means to build dependable systems can be split into four categories:

- *fault prevention* for preventing the faults from appearing in the system in the first place;
- *fault tolerance* for the good reaction to faults at runtime;
- *fault removal* for reducing the number and severity of faults;
- *fault forecasting* for estimating the characteristics (type, consequences etc.) of future faults.

A widespread method for building dependable systems consists in listing all the possible faults and then handling them, depending on the situation, by either eliminating them from the finished product (i.e. fault prevention), or specifying the desired behaviour in case they manifest (i.e. fault tolerance) [112]. Various formalisations of this approach have been proposed, e.g. Failure Mode and Effects Analysis (FMEA) [92, 107], used by NASA among others. With this method, systems are studied in a bottom-up manner beginning with the most basic components and

then moving to sub-system level, with each identified fault described with respect to its impact. Afterwards, the appropriate measures can be taken when building the finished product.

This type of approach, more or less formal, is susceptible to missing faults as it is mostly based on the knowledge and expertise of the system designers and developers, as well as on the hypothesis that the system is completely known, which is often difficult to guarantee for complex, open or evolving systems.

FAULT TOLERANCE The purpose of fault tolerance is to avoid system failures in the presence of faults. The focus is thus on the behaviour of the system at runtime, even though the means, e.g. goals to achieve and plans to support these goals, are put in place at design time. This is complementary to other approaches such as fault prevention and fault removal that are usually addressed at design time only. As stated before, we aim at complementing the classic approaches [112] where *all faults* are identified during design and development, and are either removed altogether or provided with specific handlers. This complementarity is needed because complex systems that act in open environments can encounter unforeseen situations, but our approach can also help lower fault tolerance-related costs by providing means for creating an "implicit" fault tolerance. The idea is thus to provide a "safety net" for the programmer.

REDUNDANCY In [47], Gärtner makes two important observations. First of all, he notes that despite the existence of fault tolerance mechanisms, there is always the possibility that the seriousness of faults result in a failure. This is no surprise as there is always a limit in the coverage and strength of these mechanisms and so is the case for the trapezist's safety net. The second observation is that in order to achieve fault tolerance, a form of redundancy is always required. He distinguishes redundancy in *space* (e.g. a second inertial system in a rocket, a duplicated agent in a MAS, but also the parity bit in transmissions) from redundancy in *time*, i.e. executing the same computation again (for example in rollback recovery mechanisms which take the system back to a previous state and then re-execute [36]). With our aim to provide implicit fault tolerance through our safety net approach, we will need to find a paradigm that encompasses features of redundancy, while in the same time encouraging the programmers to introduce redundancy themselves, ideally without concern for specific faults. We shall see that the execution mechanisms behind the Multi-Agent Systems with goal driven agents that we use in this work already contain elements of redundancy in time.

SELF-HEALING In "The Vision of Autonomic Computing" [58], Kephart and Chess speak of the need of a new approach for relieving the humans from the burden of managing, optimising, diagnosing, repairing etc. the ever more complex systems that are being created. They even make an analogy with "the autonomic nervous system [which] governs our heard rate and body temperature, thus freeing our conscious brain from the burden of dealing with these and many other low-level, yet vital, functions" [58]. The similarity to our work comes from the fact that we too aim to free the programmer from the burden of consciously dealing with fault tolerance. One of the *self-** axes envisaged for autonomic computing is *self-healing* [87], which shares many characteristics with fault tolerance, while sometimes focusing more on recovery.

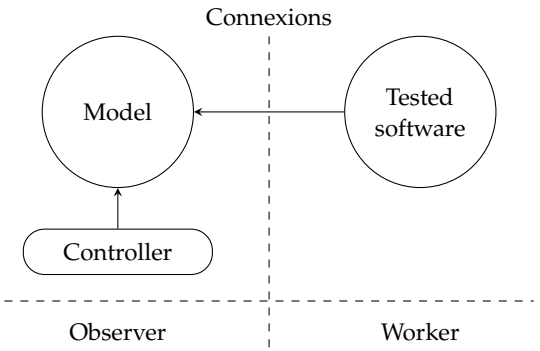


Figure 2: The principle of the observer method

RESILIENCE Another related concept is *resilience* [72], which in a general definition "is the ability to successfully accommodate unforeseen environmental perturbations or disturbances", or, closer to our field of study "the persistence of dependability when facing changes".

ANTIFRAGILE SYSTEMS Recently, a new design approach for engineering *antifragile* [55] systems was proposed. The objective is to produce systems that become stronger when subjected to "stress", just as a muscle does. The new direction comes as a consequence of the observation that current requirements-based systems result in "fragile" systems, in the sense that they eventually break under stress. To avoid that, the idea is to provide the systems with properties that permit them to adapt to unexpected situations. The authors argue that these properties are also required for achieving true autonomy, rather than just autonomy expressed through higher levels of automation. The antifragile systems, however, often have a learning component and fall outside the scope of our current work.

Let us now introduce examples of works that can be used to increase system robustness in the presence of unforeseen faults.

2.1.1 *The Observer*

Diaz et al. [31] propose a system in which distributed systems can be verified with respect to the specifications of a previously defined model. Their method is called "the observer" and is an evolution of a concept originally proposed in [5], initially conceived for parallel rather than distributed systems. The observer method requires the existence of a model of the system and a means of verifying the behaviour by accessing certain of the system's internal states or events. This can be done, for example, by monitoring the messages exchanged by the system components [31] or even by using physical connections in the case of electronic circuits [70]. The result is a two-part layout, as seen in Fig. 2: the working part of the system, marked "worker", and the "observer". This provides a sort of "minimal" redundancy (as opposed to a complete redundancy which would use two copies of the "worker") used to detect deviations from the specifications. This method is based on the assumption that the system is either functioning correctly, or it is producing visibly incorrect outputs, in other words the errors are observable.

The model serves as a reference for the correct functioning of the system, thus allowing the observer to detect errors at runtime, including any errors caused by unforeseen faults. This model can take the form of a Petri net, as is the case in

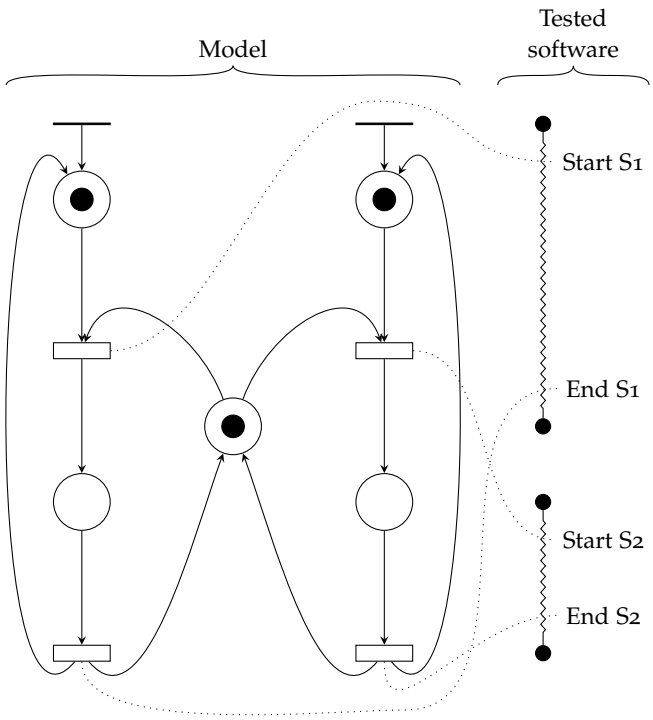


Figure 3: Petri net with corresponding observed code

the cited paper by Diaz et al. The example in Fig. 3 shows two code sequences that must not be executed simultaneously. In the Petri net (ignoring the dotted lines and the "tested software" part) there are four transitions and five places, with the initial tokens presented in the figure. Due to the non-determinism of Petri nets, the central token can be used to trigger a transition either to the left or to the right, but not both at the same time. Assuming that the token is passed to the left (the network is symmetrical anyway), the only transition that can fire next is the one at the lower left, which will produce a token in the central place again. The two transitions at the left are thus always forced to fire consecutively, before giving the other transitions the opportunity to fire as well. In Fig. 3, the dotted lines indicate the connections for the observer method, with the start and end of each code sequence each linked to a transition in the Petri net. If, for example, S2 starts before the end of S1, the observer will find it impossible to fire the corresponding transition, thus indicating a deviation from the expected behaviour.

The levels of detail for the model and the observations are fixed taking into account the fact that more observations can often lead to a more accurate detection of defects, but can also increase the execution time.

This type of approach can detect faults we call "unforeseen" because it observes deviations in the execution of a system without needing to know the cause, but the need for a correct model (often validated by other means which are generally quite expensive) makes it difficult to apply.

2.1.2 *Anomaly detection*

The extensive review by Chandola et al. [20] on the detection of anomalies drew our attention due to the similarities to our subject. The study in concerned with

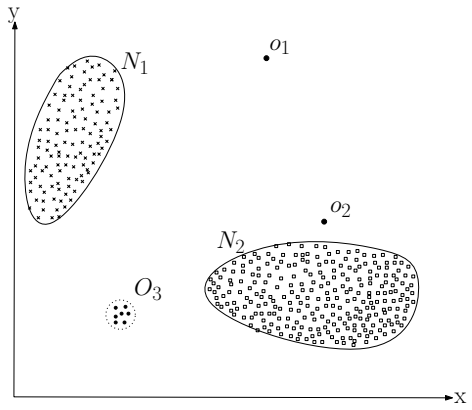


Figure 4: Point anomaly example (example from [20])

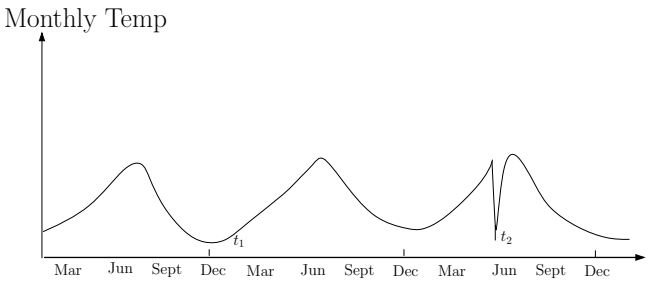


Figure 5: Context anomaly example from [20]

the anomalies, defined as samples from data collections that do not conform to the expected or desired behaviour. In Fig. 4 for example, both points O_1 and O_2, as well as the group O_3 can be considered as anomalies with respect to the normal areas N_1 and N_2. In particular, these are "point anomalies". This type of anomaly can be used for example for detecting bank fraud, where unusual transactions can be used to raise the alarm. They can also be used to identify noise in data sets.

Another type of abnormality is the "contextual anomaly", as in Fig. 5, where temperatures t_1 and t_2 are equal, but the context indicates a problem in the case of t_2 (e.g. a winter temperature in July). This type of anomaly is often used for time series.

A third type of anomaly identified in the review mentioned above is the "collective anomaly", exemplified with a heartbeat example in Fig. 6. In this case, the anomaly is represented by a subset of the data instances.

Essentially, the anomaly detection problem is a classification problem: for each instance in the data set, its class must be decided between "normal" or "abnormal". For this type of problem, three types of approach are commonly used:

1. The supervised anomaly detection uses data that is already labelled/classified to decide on new instances. This approach is expensive and cannot guarantee the coverage of all possible cases.

2. The semi-supervised detection uses only instances labelled "normal". As it does not involve a knowledge of the possible cases of failure, this approach has broader applications, such as the fault detection for space craft [42]. From this perspective, this approach is also the closest to ours, because we too are

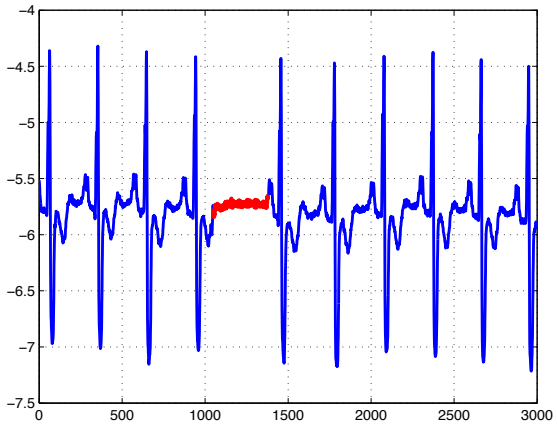


Figure 6: Collective anomaly example from [20]

considering the use of a description of the normal (or nominal) behaviour for the detection of errors.

3. The unsupervised anomaly detection does not use any already classified instances and works on the assumption that the abnormalities are much rarer than the normal instances.

The description of the anomaly detection problem for Internet attacks is also worth mentioning in relation to the concept of unexpected fault. In the work of Kruegel and Vigna [68], the detection of anomalies is based on models of "normal" behaviour of the users. Applications are then used to interpret the behavioural deviations of the users, which are judged as malicious activity. This approach is complementary to the detection of misuse, where descriptions of known attacks are used to identify attacks from the flow of monitored events. This is therefore the detection of *unexpected* events, which brings this method closer to our proposal.

The interest in discussing the anomaly detection here is that it can show a model for the detection of unexpected behaviours, similar to a certain point to our problem. While we are currently not concerned with data deviations as presented here, it can be envisaged for future work to integrate such tools at the detection level of our approach.

2.1.3 *TibFit and Chameleon*

TIBFIT TibFit [67] is a protocol for the tolerance of arbitrary faults in wireless sensor networks. It uses a trust index for quantifying the reliability of each sensor in the network. The index is a real number between zero and one calculated through a learning process. The process starts with the maximal confidence level (of one) and then increases or decreases the value (while remaining between zero and one) of this level depending on the accuracy of the evaluated sensor behaviour over time. The index is used to determine the validity of these sensors in a weighted vote that is designed to help the system achieve a consensus by giving more credit to sensors that have proven reliable in the past. To enable the use of a voting protocol, each event is assumed to be located within the range of a well-defined group of sensors. After each assessment, the indices of the relevant sensors are updated. This approach allows the protocol to cover various cases of errors such as temporary or permanent sensor failures, as well as malicious sensors. TibFit is an interesting ex-

ample for us because it allows components of a distributed system – and therefore potentially agents of a multi-agent system – to tolerate arbitrary faults collectively through a vote.

CHAMELEON Chameleon [56] is a collection of tools for fault tolerance in a networked environment – several applications working together – and provides three types of entity. *Daemons* are attached to each node of the system for communication and local support for other entities of Chameleon. Then, entities called *ARMORs* are used for the implementation of specific fault tolerance techniques (voter ARMOR, for example). The third type of entity consists of *managers* who are used for the supervision of the Chameleon system. Although not a multi-agent architecture itself, this distributed network for fault tolerance is interesting for our study because:

- its ARMORs are tools that are available for use in different applications, resulting in different levels of reliability. This flexibility allows on the one hand, choosing from different tools those that are better suited to the current application, and on the other hand adjusting the balance between the computation speed and the level of fault tolerance – knowing that normally adding additional tools increases the computation cost.
- one objective of this infrastructure is to provide fault tolerance for "off the shelf" applications, which means that this system is transparent for the application developers, a feature we would also like to offer through our safety net approach. In other words, we aim to minimise the level of intrusion of our fault tolerance mechanisms so that the programmer can focus on desired system behaviours. This transparency for applications means that the Chameleon tools are responsible for the tolerance to faults that are unexpected for the developers of the final system.

This last statement brings us to an important observation on the concept of "unforeseen fault" with respect to the point of view differences. Taking the example of Chameleon, the faults that are "unforeseen" for application developers are not necessarily unexpected for those who have implemented the underlying platform and tools. Similarly, we will see that for our safety net approach we provide mechanisms that can be considered as "foreseen" faults without it being a violation of the concept of "unforeseen fault" for the final developer.

2.1.4 *Mission Data System*

The system complexity and the constraints linked to the fact that the response time can be very large in space missions determined NASA to use a goal-based control system instead of the usual commands [91]. The essential difference identified by the authors of the paper between a command and a goal is that a command is linked to a moment in time and does not easily allow the verification of its permanent effects. This also makes it difficult to verify the conflicts between different commands. In the proposed system, called Mission Data System (MDS), goals are represented as constraints on state variables over time intervals. Then, the verification of conflicts and inconsistencies is reduced to a comparison between the constraints on shared variables and their time intervals. Taking the example of a drone, if a command to "avoid hazardous area" is launched, it will be easier to find

a conflict with "follow target X" when the target enters the danger zone if the two are represented as constraints (the position of the drone and the next movement), compared to the case when the two are represented as individual commands. In the same time, if the area is classified as "dangerous" between the instants t1 and t2, the verification can also conclude that there is no conflict if for example the target comes in the area after time t2. The actions to perform are deducted from the differences between the current state of the variables and the desired state.

Fault tolerance is included in the system naturally as error conditions are treated in the same way as normal states. Moreover, the states do not need to be explicitly and accurately described, as it suffices to define them only with respect to the observable main states. One possibility that is closer to our concept of "unforeseen fault" is to describe the "normal" behaviour or "acceptable" of the system. An error is detected when the observations on the system do not match the expected behaviours. We consider this natural inclusion of fault tolerance in the design specifications useful for reliable systems and it can be used to handle cases of "unforeseen faults". Compared to our approach, the MDS does not discuss the distribution into several entities – the multi-agent design in our case – which we consider very important for fault tolerance.

2.1.5 *Recovery Blocks*

In the early days of software fault tolerance, an enriched program [54] and system [88] structure was proposed for allowing error detection and recovery. The idea is to include regular tests on the outcome of program execution and include alternative solutions for the situations when the original code did not produce the expected results. For this, the programs are segmented into *recovery blocks*. The normal code becomes a succession *primary blocks* which are each tested using an *acceptance test*. For each primary block, one or more *alternate blocks* are provided for the situations when the acceptance tests fail. Should the test of a primary block fail, the *alternate blocks* are executed one by one until the test is successful. If none of the alternate blocks for a primary block produce the required results, control is passed at a higher level where similar measures may apply. Each alternate block is applied as if the previous blocks of the same recovery block were never applied. To ensure this property, all non-local variables[1] are tagged when modified using a boolean flag, while their original values are stored in a stack. When a primary or alternate block fails, any modifications that it operated on these variables are undone. Should more specific recovery measures be needed, dedicated procedures can be defined and triggered by the same mechanisms as the automatic variable recovery.

The types of errors that are covered by this technique are generic and of interest for our approach:

- errors in the block that are detected by the acceptance tests,
- failure to terminate, caught by a timeout,
- detection inside the block by an implicit error detection mechanism (e.g. division by zero),
- the failure of an inner recovery block.

1 Variables that are local to the blocks are not concerned.

Similarly to our approach, the authors note, however, the complementarity to other fault tolerance techniques as "errors which are expected to be sufficiently frequent that special handling would be appropriate can perhaps be regarded as normal program conditions rather than unforeseeable errors".

The advantage of the recovery blocks approach is that it provides redundancy of design, which ensures better fault tolerance than replication which has more chances to produce the same errors as the original execution.

Algorithmus 1 : The *recovery block* structure

```
recovery block A
    acceptance test AT
    recovery block AP
        program
    recovery block AQ
        program
```

When multiple processes are involved, *conversations*, which are recovery blocks spanning two or more processes, are used to avoid a domino recovery effect. For the conversation to be complete, all the processes must satisfy their respective acceptance tests.

There is also an important multi-level aspect, as the failure of a block is treated at the next level, therefore enhancing the overall fault tolerance in complex applications. This means that even hardware errors may be masked by the application of the recovery block structure at higher levels.

2.1.6 *A Case for Automatic Exception Handling*

Cabral and Marques [16] offer an insight in the way exceptions are used in Java and .NET and conclude that exceptions are treated lightly by the programmers:

- generic exceptions that are difficult to properly handle and recover are thrown;
- generic catching mechanisms are provided, resulting in a poor recovery (causing the program to continue in a corrupt state). There are even cases when errors are not caught at all, allowing the program to crash even from minor errors;
- providing "proper" exception handling decreases productivity and can have negative effects on the overall software development project;
- providing "proper" exception handling can be challenging and even contribute to the introduction of new errors.

They go on to make "A Case for Automatic Exception Handling"[2] [17], drawing a parallel with the introduction of garbage collections and memory allocation. The idea is to improve software quality and robustness by better covering exception cases and also ease the programmer's task by minimising their error-handling inputs.

Their solution combines exceptions with an execution similar to the recovery blocks approach discussed here in Sec. 2.1.5. The programmers have the possibility

2 The actual name of the publication.

to let the platform handle exceptions or provide specific handlers. The platform handling, however, is ensured through exception-specific actions – which can include throwing a new exception to be handled by the higher level, i.e. the caller of the caller – provided by the programmer in a separate configuration file. This helps diminish the programmer's task when writing the bulk of the application but still requires his or her involvement and concern for specific, foreseen, cases. At runtime, an execution section producing an exception can be ran multiple times, each time applying a different handler, until recovery is successful or the last handler – "Log&Abort" – is reached. A transactional model ensures that after each exception handler is executed the application state is restored to the initial condition so that the code can be ran again.

Their study on exceptions shows that there are cases where fault handling is poorly done and can result in a system crash or even continuing in an inconsistent state. This means that even errors that were foreseen – for example because the language would normally force the programmers to provide a specific handler – become unforeseen as they are not treated or not treated correctly in the finished application. Furthermore, there are also situations when the programmers could use the aid of the platform for handling certain types of error. Our goal is to provide a development framework (platform, language and design requirements) that allow the programmer to rely on the platform for the automatic handling of at least some of the runtime exceptions. The safety net in this case is used in a conscious manner by the programmer who either throws exceptions knowing that they are handled by our mechanisms, or simply does not provide generic, empty or possibly wrong handlers, knowing that the platform will take care of the concerned exceptions. Note, however, that as the authors of the cited studies, we too acknowledge the limits of providing a completely generic mechanism for handling exceptions, thus we need to integrate in the language the necessary features that facilitate the recovery, e.g. goals with satisfaction tests.

2.1.7 *Defensive Programming*

The software engineering technique called *defensive programming* requires the programmers to systematically cover all possible cases, even if this may seem redundant. While this technique does bring robustness benefits, it does so by relying heavily on the judgement of the programmer who is forced to add numerous tests to ensure the correct values for all variables. More tests means more code and this comes with the increased risk of errors. This technique is thus outside the scope of our work but constitutes an interesting example of expensive and yet not guaranteed fault tolerance technique.

2.1.8 *Design by Contract and Executable Specifications*

DESIGN BY CONTRACT The contract programming paradigm was introduced by Meyer with the Eiffel programming language [74, 75]. The idea is to require the programmer to systematically specify the conditions to check, but without the complexity of the defensive programming approach. These conditions (annotations) are assertions, to which the programmer associates a truth value and which have their own semantics (not necessarily the same as the language). In general, this semantics corresponds to boolean expressions with first order logic quantifiers. This program-

ming paradigm is used not only to systematically test during the execution (and thus in a way provide a means to elegantly perform defensive programming), but also to analyse the code. One can indeed, in certain cases, link the contracts to an automatic prover or a static analysis tool. There are three types of assertion:

- Precondition: verified before an operation, for example a function call, which will not be performed if the assertion is not valid;
- Postcondition: verified after an operation;
- Invariant: is an assertion that needs to hold permanently during the entire program execution or more locally (e.g. in a loop).

Contract programming is a popular paradigm as it increases the robustness of software and also reduces the debugging time. Various programming languages contain an annotation facility in order to comply with the paradigm, for example SPARK [7] for annotating Ada code.

EXECUTABLE SPECIFICATIONS Another approach is represented by the use of executable specifications [41] for increasing software reliability. The goal in this case is to identify errors and deviations in the development process from the user intent in order to correct them early in the application life-cycle. More recently, Samimi et al. [98] extend the application of the executable specifications to runtime, thus obtaining a use similar to the contracts, in an approach called *Plan B*. The specifications are used to check the postconditions after executions. In case of failure in the execution (through a `RunTimeException`, e.g. an `ArrayIndexOutOfBounds` or `NullPointerException` in Java) or if the postconditions are not as required, instead of halting the execution, the execution falls back on the specifications which are used to try and provide an alternative solution. The authors aim to:

- increase software reliability by introducing redundancy through the specifications and catching the error states in order to handle them using the specifications. The advantage is twofold: the imperative and more efficient implementation (in Java in their case) is used for the actual execution, followed by a verification and possibly an attempt at recovery through the more computationally expensive specifications.
- improve the developer's experience by not requiring him or her to program the specific cases. In case they occur, an exception is thrown which causes the execution to fall back on the specifications.

These approaches rely on the programmer for the verifications but are both more refined than the defensive programming and can cover unforeseen faults. Furthermore, the Plan B approach, similarly to the recovery blocks above, also provides mechanisms for attempting to recover in case of error.

2.1.9 *Let It Crash*

Erlang [2, 3] was conceived for concurrent applications with a large number of threads and with high availability requirements. Software written in Erlang has a reputation for being very reliable, for example the network switch AXD301 [10] with a "nine 9" reliability.

Erlang is a dynamically typed functional language whose particularity is in the way it handles processes. It contains a library of very light threads which only communicate by messages on which timeouts can be set.

Erlang processes can also send messages on their functioning, for example when ending their execution: `{'EXIT', Existing_Process_Id, Reason}`. In case the value of `Reason` is "`normal`", the execution is considered correct and the message is ignored. Otherwise, it is seen as an error message and the concerned thread can either handle through a `catch` block, or finish sending the same signal. While error messages are standardised in many languages (e.g. Java, C#, Prolog), the exit signal standardisation is specific to Erlang. By default, a thread always sends an exit signal to its parent thread. Two processes can also be linked together, so that one of them is informed of the ending of the other.

The designers of Erlang encourage the programmers to not catch exit signals unless these are required by the specifications, thus leaving the processes apply their default handling when the case is not covered by the specifications. This approach is called "let it crash" [2] and is a characteristic we are intending to integrate in our safety net as well.

2.1.10 *The Mercury Programming Language*

Mercury [51] issued from the observation that even if Prolog was more expressive than the imperative programming languages of the 1990s, it was not much used by companies. The two main arguments the creators of Mercury give are:

- the Prolog compilers do not detect enough compilation errors,
- the programs written in Prolog are sensibly slower than the ones written in imperative languages.

Mercury is a strongly typed language, proposing a more evolved typing system than Prolog. It also has a means for analysing the input/output modes of predicates (i.e. the state of instantiation of variables of a predicate) and a determinism analyser (to identify the number of potential outputs of a predicate). These verifications increase both the reliability of software by helping avoid certain runtime errors and the execution speed (e.g. no backtracking is performed on a deterministic predicate). However, this is done through language restrictions, in particular on the constructions that are outside the scope of the first order logic, e.g. the "`cut`".

A compromise is thus required between the restrictions imposed on the programmer and the ease of programming in a language. For the tolerance to unforeseen faults, we need to keep the chosen language usable, expressive and in the same time include restrictions to guide the programmer towards more reliable code.

2.2 FAULT TOLERANCE WITH AND FOR AGENTS

In order to discuss fault tolerance in the domain of Multi-Agent System, let us first present a couple of definitions of the term "agent".

AGENTS While we gave a definition of agents focusing on the practical aspect of delimiting their memory and defining their communication in Sec. 1.4, we cite here the very wide definition of the concept by Ferber [39]:

> We call agent a physical or abstract entity

a. which is capable of acting in an environment,
b. which can communicate directly with other agents,
c. which is driven by a set of tendencies (as individual goals, satisfaction or even survival functions that it seeks to optimise),
d. which has its own resources,
e. which is able to perceive (but in a limited manner) its environment,
f. which has only a partial representation of this environment (and possibly none),
g. which has capabilities and provides services,
h. which can possibly reproduce,
i. whose behaviour tends to satisfy its objectives, taking into account the available resources and capabilities, depending on its perceptions, representations and the communications that it receives.

Wooldridge [116], focusing on computer systems, defines an agent as: "a computer system that is situated in some environment, and that is capable of autonomous action in this environment in order to meet its design objectives.". The same author continues by pointing out three characteristics of agents:

- *reactivity* – they can can perceive and react to those perceptions, in line with their design objectives;
- *proactiveness* – they are able to take the initiative in order to satisfy their design objectives;
- *social ability* – they can interact with other agents.

AGENTS AND FAULT TOLERANCE Generally speaking, multi-agent systems are seen as having one advantage and one disadvantage for fault tolerance [48]:

+ they are naturally modular, which is a very important characteristic for the dependability of software;
- it is difficult to guarantee a deterministic behaviour due to their autonomy and asynchronism.

We continue this section by describing a few approaches that can be used for the fault tolerance when using MAS.

2.2.1 *A Perspective on Exceptions in Multi-Agent Systems*

Platon et al. [82, 83] distinguish the "classical" programming exceptions – they call *continuity* exceptions – from the *rupture* exceptions (in the sense of MAS) by defining the latter as "the evaluation *by* the agent of a perceived event as *unexpected*". While we do not use the word "unexpected" in the same sense, we do acknowledge their perspective on the fact that exceptions are a matter of agent perspective rather than a given "external" event. Their definition takes into account agent autonomy, as the agent perspective is used to decide whether an event is an exception or not, contrary to the classic definition where the program is forced to react to any exception thrown by a called operation. The agent architecture they propose is

thus focused on evaluating the agent perceptions (e.g. messages) with respect to the local expectations and relevance in order to decide if they need to be treated as exceptions. This architecture does not suffice for our purposes, because our concept of *unforeseen fault* covers both their rupture and continuity exceptions, as we take into consideration issues that appear at different levels, both at programming and system level. Furthermore, they base their exception detection and handling on mechanisms that require conscious designer involvement – e.g. providing a specific knowledge base for certain interactions – which makes the exceptions "foreseen" from our perspective.

One of their proposed directions of research for exception handling [82] is automatically enriching the agent context at runtime with information that are to be used in case of exceptions. As shall be seen in Chapter 3, this corresponds to one of our contributions for handling dependencies.

2.2.2 *Communication Standards for Agent Fault Tolerance*

Numerous works on the fault tolerance in the MAS were concerned with the creation of standards in the communication languages, with languages such as AgenTalk [69] and COOL [6]. One of the objectives was to force the programmer to specify the behaviour in case a language primitive failed, for example because of an agent death. The communication language FT-ACL [33] was proposed in 2006 as a communication standard. An agent supporting FT-ACL must contain a *facilitator* acting as a mediator between an agent and the other agents. The facilitator is in charge of sending and receiving messages and thus the errors that occur at low level (e.g. physical communication error). It communicates with a *failure detector* which monitors the good functioning of the other agents, in particular for detecting agent deaths.

2.2.3 *Replication*

Another research direction for the fault tolerance of the MAS is the management of the replication [38]. The idea is to introduce proxys between agents and the rest of the system in order to make the replication transparent with respect to the other agents. These works were influenced by the similar approaches in distributed systems, in particular the N-version programming. These techniques were extended to take into account the importance of each agent in the MAS [46]. The relative importance of an agent with respect to the others is determined dynamically by a reactive agent associated with each problem-solving agent and whose role is to monitor the communications. The idea is then to only replicate the agents that are evaluated as "critical", thus improving system performance. This method has the advantage of allowing the system to dynamically adapt to any conditions it can encounter, which are thus *unforeseen*.

2.2.4 *Detecting Errors Through Agent Disagreement*

The idea of Socially Attentive Monitoring (SAM) [57] is that an error will manifest through a disagreement between the agents, for example on the value of a belief or even a goal to adopt. SAM is based on a social psychology theory, in which an agent turns to the other agents in order to look for errors inside itself as well as the

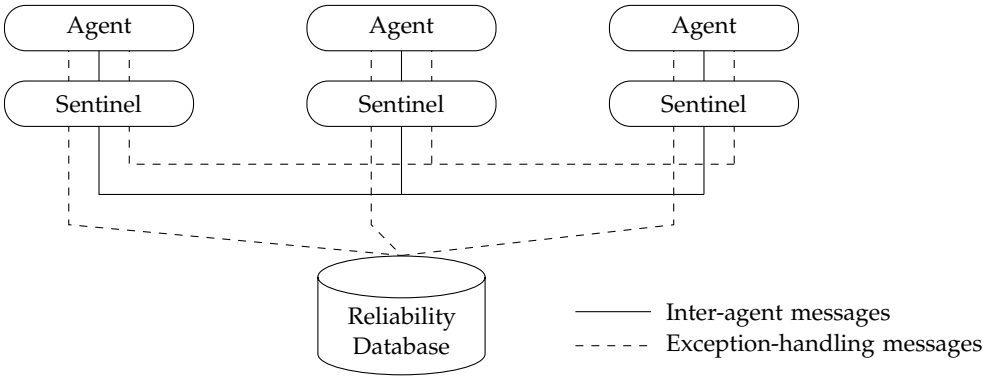


Figure 7: Sentinel architecture example from [65]

others. Once an error is found, the other agents of the group are announced and a diagnosis can be attempted. The problem with this approach is that it requires an explicit representation of the group and the possible interactions, which makes it difficult to adopt in dynamic architectures.

2.2.5 *The Sentinels*

For fault tolerance in MAS, Haegg [48] proposes adding special agents called "sentinels" whose sole purpose is to monitor the behaviour of problem-solving agents. Sentinels can intervene, if necessary, by choosing alternative problem solving methods for the agents, deciding to exclude failing agents, change the parameters of the agents, or even to refer to human operators. Communication in this case is a form of broadcast which facilitates the monitoring work of the sentinels. An example described by Haegg presents a distributor and several customers in the context of the electricity market. Sentinels are introduced into the system with the aim of monitoring its correct functioning and the respect of the different market rules. If for example a customer permanently offers contracts that are below certain threshold prices, the sentinels can notify the human operators or advise the distributor to avoid this agent.

A sentinel can be provided for each agent [64] upon its creation, in order to control the agent's behaviour with respect to the its nominal behaviour. If an exception is identified, the sentinels use dedicated diagnostic and reparation agents to recover the system. A proposed alternative [65, 100, 101] is to interpose a sentinel between each agent and the rest of the system to better control its state, allowing in the case of [65] to improve the speed of detection for the death of an agent. The proposed architecture is shown in Fig. 7, where messages between agents are represented using the solid line, while messages for handling exceptions are represented by the dashed line. A "reliability database" is used to centralise all information on the exceptions detected thus far.

The sentinels of Shah et al. [100] provide each agent with a service similar to the observers described in Sec. 2.1.1. To communicate with other agents, an agent passes each message through its sentinel, which not only transmits it to the recipient, but also checks it with respect to the "ideal" behaviour of the agent. This approach uses multiple knowledge bases, such as the one for the behaviour of the agents and the one for diagnostic rules. The sentinels are responsible for diagnosing the causes of exceptions, regardless of their source: an undesirable behaviour detected by the sentinels themselves or an exception received from their agents.

The diagnosis is made based on local knowledge, but also by making queries to other sentinels.

While this approach is powerful for monitoring multi-agent systems, its implementation remains within the the scope of foreseen faults, since the detection system is based on the knowledge of abnormal situations. In addition, in order to develop systems that are tolerant to unforeseen faults, we are not interested in diagnosis. Indeed, finding the origin of the fault may require an important computing time and also a more specific knowledge of the application domain, which may conflict with the concept of unexpected fault.

It is important to note that the observers and the sentinels are usually generic and offered by the platform, thus being tools that can be formally validated and reused. The advantage of comparing the functioning of a system with the ideal functioning indicated by its model lies in the fact that errors can be detected without the designer's concern for specific errors, making it a good detection mechanism for errors caused by unforeseen faults.

2.2.6 *Norms. Trust and Reputation*

The need to handle system complexity and the level of abstraction of agents led researchers to propose organisational approaches to MAS designs, with the purpose of organising and regulating the agents. Norms are used in these organisations to regulate the agent behaviour and avoid unwanted behaviours, regardless of their cause, e.g. unforeseen emergent behaviours [24].

Trust and reputation [97] are another pair of concepts the computer science community has borrowed from human societies. For agents, they can be used to keep track of the agent interactions and penalise agents that are perceived as misbehaving, regardless of the actual cause, including thus what we call "unforeseen" faults. The concept of trust is also used in TibFit, as presented in Sec. 2.1.3.

2.2.7 *Agent Autonomy for Robust Agents*

In the domain of multi-agent systems, *autonomy* is a special property that presents both positive and negative aspects for the fault tolerance [86]. Autonomy is often cited among an agent's defining characteristics. It implies the ability to make one's decisions without external intervention and adapt to changing conditions.

A first interpretation of this property – and the reason why it is often marketed as a desirable characteristic of agents, but also robots and other systems – presents it as positive for the fault tolerance: autonomous agents are more independent with respect to others, and should therefore be more resilient.

However, a more in-depth evaluation can give a more nuanced result. When *other* agents are autonomous, they can be seen as "black boxes" that can "refuse" interactions so their peers have to take these into consideration. Autonomous agents can thus be perceived as being less reliable by their peers, with behaviours that can be seen as unpredictable by their human designers or controllers too. For this reason, in [86] it is argued that the classic fault classification needs to be extended to include what others perceive as faults caused by autonomy. The bright side of this observation is that it can be included in the design of agents with benefits on their robustness. While difficult to quantify, this makes an important guideline to give to system designers: "an agent should not depend on a single second agent".

From this point of view, ideally for the fault tolerance it would be for the agent to be completely independent, but that would result in one or several single-agent systems, which defeats the purpose of the MAS in the first place. But applying the rule in moderation results in agents that are less prone to be negatively impacted by errors or failures in their peers.

This gives a totally different approach from the tightly coupled classic systems where designers suppose that the other components work according to their specifications. The result is more flexible systems where errors are well confined and agents are designed taking into consideration the possible autonomy of their peers and thus their possible "refusal" to cooperate – for whatever reason, either motivated by their own goals, or an agent or communication error. While there is an obvious limit to the number of reconfiguration alternatives for action an agent can have, it is important for it not to behave erratically or block in case none of them responds as expected.

2.3 GOAL-DRIVEN AGENTS

In the field of intelligent agents, goal-driven agents are used extensively due to their pro-activity, adaptability and similarity between their abstract representation and the human reasoning. The original model for these agents is called Belief, Desire, Intention (BDI) [90]. BDI agents are enticed with *beliefs* to cover their view of the world, a reason for their behaviours in the form of *desires* or *goals*, and a description of the means to act, in the form of *plans* or *intentions*. Different implementations [1, 11, 14] require various characteristics such as goal preconditions, postconditions or satisfaction conditions, as well as ways to handle conflicts and other execution issues.

Goal-driven agents allow designers and programmers to separate the objectives of an agent from the means to achieve those objectives. As seen in Sec. 2.1.4, this characteristic of goals makes them particularly interesting for systems where autonomy and fault tolerance are especially important issues: in space missions. Dalpiaz et al. [26] propose a four-step cycle: monitor, diagnose, reconcile and compensate, that is based on goal-driven agents for detecting errors and reconfiguring, promoting the use of goal-driven agents for fault tolerance. We cite three of the most important advantages offered by goal-driven agents:

1. at runtime, flexibility to adapt to the given situations by choosing the appropriate means depending on the execution context;
2. at runtime, the possibility to retry achieving a goal in case the means initially applied were not successful;
3. at design-time, the facility of designating high-level objectives that are then decomposed into lower level ones and eventually actions [44, 114].

As shall be seen in Part II of this thesis on the safety net approach, the first two advantages are particularly useful for augmenting agent robustness. The third advantage is more relevant for the GPS approach discussed in Part III.

While we focus on a plan-centred approach, we note the use of goals for interaction-centred approaches, always in the interest of robustness and agent autonomy [13, 22].

WHERE DO GOALS AND PLANS COME FROM? Our work is based on the idea that goals and plans are available for the agent execution when needed. For simplicity reasons, we use plans and goals that are already written by the programmers when the program is executed. However, our work should be compatible with other means of procuring goals and plans, such as motivational goal generating [50] and planning [21, 99], including dynamic plan revision [12]. The advantage of pre-existing plans is that they can make the agent behaviour more predictable and the computational cost is lower.

2.3.1 *Describing Goals*

GOAL PROPERTIES In order to better define the abstract concept of goal, Winikoff et al. [115] identify a series of desired properties of goals:

- *persistent* – goals should only be dropped for a good reason;
- *unachieved* – goals should not be adopted if they are already achieved, and they should be dropped when the desired state is achieved;
- *possible* – impossible goals should not be pursued;
- *consistent* – adopted goals should not be in conflict between them. However, an agent can have conflicting goals, as long as they are not pursued in the same time (e.g. a robot whose desires can be to "plug in to recharge" and "work outside");
- *known* – the agent should know its goals in order to be able to reason based on these goals.

GOAL TYPES Among various classifications of goals proposed in the literature, goals can be procedural when the goal is to execute actions, or it can be declarative when it describes the state sought [115]. Van Riemsdijk et al. [94] also point out the distinction between system goals which represent high level goals that the system as a whole needs to accomplish and which are different from the individual goals of its constituent actors [71]. When adding a temporal aspect to the description of goals, they can be [15, 94, 115]:

- *achievement* goals that relate to the common understanding of the word "goal", as "something has to be achieved";
- *maintenance* goals that aim to preserve a certain condition, either by reacting to its changing, or by pro-actively acting to avoid a foreseen menace;
- *perform* goals that are goals to execute actions;
- *query* or *test* goals that are goals to own a certain information.

PARTIAL GOAL SATISFACTION In this thesis, we treat only binary goal outcomes (success or failed), but goal satisfaction can also be evaluated using a more refined approach: a *progress metric* [95]. Partial goal satisfaction could be integrated with our model by enforcing the coverage of the whole range of possible values for the progress metric used. For example for a Surveillance goal, instead of specifying success and fail behaviours, it could be interesting to estimate the percentage of the

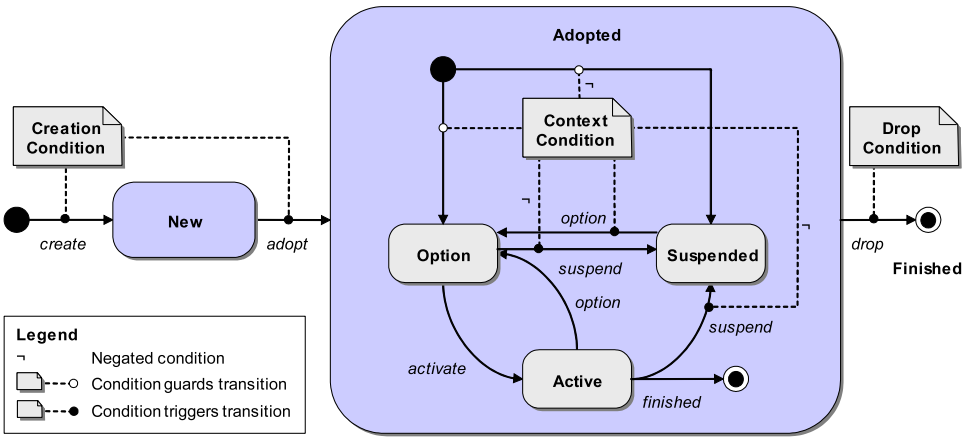


Figure 8: Goal automaton example from [15]

assigned area that was covered and to use thresholds for the desired behaviours: less than 30% would be considered a mission failure with the area announced as *unsafe*, a coverage between 30 and 80% would require a call for backup to finish the job, while a coverage of more than 80% would be considered a success. Note that this does not concern the intermediary stages such as those that are handled by the goal automata, but final goal failures, i.e. when all alternatives have been tried and no positive outcome resulted.

2.3.2 *The Goal Life-Cycle*

In the original BDI model proposed by Rao and Georgeff [90], the "matching" between goals and plans is assured through a cycle that considers the options for *desires*, deliberates on them to update the existing *intentions* and then executes the actual actions. In more practical approaches – e.g. Fig. 8 by Braubach et al. [15] or Fig. 9 by Thangarajah et al. [111], described into more detail in [49] – automata are used to handle the life-cycle of goals from their adoption to the appropriate plan selection and execution.. The automaton generally uses different parameters and conditions to decide when and how to adopt goals, handle conflicts and most importantly goal retries. Note how in Fig. 9 the goal state transitions depend on the goal type, with the particular case of the maintenance goal which has specific transitions related to its particular state *Waiting*.

The goal automaton proposed by Braubach et al. [15] presents a goal state labelled "New" with a "Creation condition" acting as a triggering condition for the goal before the adoption and the actual goal life-cycle. This state, together with the condition are at the level we are concerned with for our GPS approach. A goal that was defined for the agent is considered to be in the "New" state, as opposed to a goal that can for example be received from the exterior or generated through the agent reasoning. Only when such a goal is received does it pass into the "New" state. All the goals discussed in the examples in our work can be considered already in this state.

In Fig. 24 in Sec. 3.3.3 we give our version of an automaton for achievement goals, whose implementation we discuss in Appendix A.

WHEN TO GIVE UP: COMMITMENT STRATEGIES A question that needs to be asked is how much an agent should insist in pursuing a goal, as various conditions

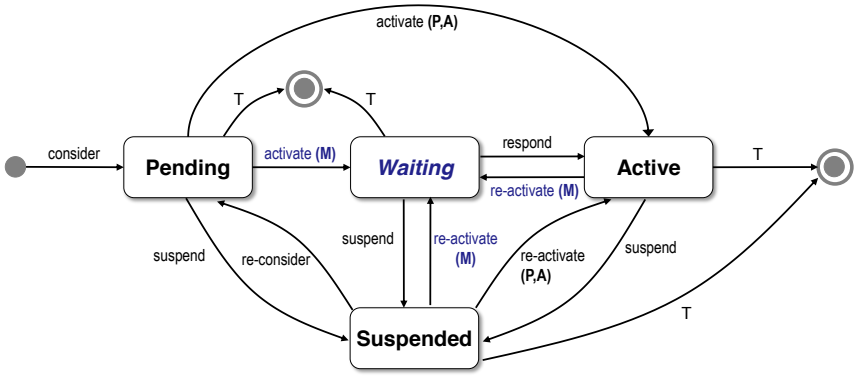


Figure 9: Goal automaton example from [111], with specific transitions depending on goal type: A - Achievement goal, P - Perform goal and M - Maintenance Goal, and $\top$ corresponding to a goal ending: drop, abort, succeed or fail.

that depend on the context (e.g. no route to the destination), agent definition (e.g. the goal is too ambitious with respect to the means in place) or existing errors (either anticipated or not) can make achieving the goal impossible or just more difficult. As Winikoff et al. note in [115], using declarative goals in BDI agents helps decouple plan failure from goal failure, meaning that a goal is not be dropped only because its plan failed. This is a very important property of the goal model that we will base our error recovery on.

Here are three strategies for the level of commitment of an agent to its goals, based on [89]:

1. *blind* commitment, where an agent pursues a goal until achievement;
2. *single-minded* commitment, where an agent pursues a goal as long as it estimates that it is still achievable;
3. *open-minded* commitment, where an agent pursues a goal as long as the goal is still desirable, is still valid.

In practice, different degrees of commitment can be used, for example allowing an agent goal to be aborted when a certain failure condition is true [96]. Sardina and Padgham [99] propose a solution whose "flexible" commitment is placed between the single-minded and open-minded strategies.

2.3.3 *Reasoning on Agent Goals*

SEPARATING REASONING FROM ACTING As introduced in Sec. 1.3, in Part III of this thesis we present the Goal-Plan Separation (GPS) approach for designing goal-driven agents in which actions are well delimited from the part of the agent concerned with the goals – adopting, handling, reasoning etc. – which we will call *the goal reasoning level*.

The aspect of the Goal-Plan Separation that handles goal reasoning is situated at what Harland et al. [49] and Thangarajah et al. in earlier works [108, 109] call *agent deliberation level*. This is where agent goals are *considered*, which constitutes the point where goals start their life-cycle. It is the same level where *top level commands* are issued to interfere with the goal life-cycle, e.g. when deciding to drop or suspend the goal. As the cited authors point out, goal deliberation can deal with issues such as goal prioritisation, resource management and even user intervention.

These aspects are beyond the scope of our work but can be considered for future developments of our approach. We note, however, that in [49], changes in the goal state have preference over any executing plans, which is an important detail to consider for our GPS approach.

The arguments for planning in BDI agents at goal level employed by Sardina and Padgham [99] offer more reasons for the existence of the goal reasoning level (be it "hardcoded", created through planning or other means) that the GPS approach aims to delimit: *"(a) important resources may be used in taking actions that do not lead to a successful outcome; (b) actions are not always reversible and may lead to states from which there is no successful outcome; (c) execution of actions take substantially longer than "thinking" (or planning); and (d) actions have side effects which are undesirable if they turn out not to be useful"*. All these advocate for an agent that behaves strategically and pro-actively rather than react based on a limited context, and it is at goal reasoning level that such a strategic reasoning is possible.

In [84] Pokahr et al. address the issue of *goal deliberation*. An important difference from our work on GPS is that they only consider goals that are already adopted, while we are concerned with where exactly goals are adopted in the agent definition. Their work focuses on goal interactions, i.e. when goals interfere positively or negatively with each other, and they base their proposed strategy on the extension of the definition of goals. They include for example inhibition arcs that block the adoption of a certain goal or type of goal when another goal is adopted. We do not include such a specific facility in the goal definition, but, as shall be seen in the example in Chapter 5, we can achieve the desired effect using the available agent reasoning mechanisms. This concept is not equivalent but rather included in our goal reasoning level as they consider only goals that have already been adopted.

Klenk et al. [66] introduce an approach called Goal-Driven Reasoning (GDR) based on the hypothesis that their agents are functioning in a complex and dynamic environment which is difficult (if not impossible) to model completely. Their work deals with planning issues when the agent is confronted with new, unexpected situations. They aim to complement the "knowledge engineering" (design work put into defining goals, the environment model etc.) needed for agents, which is somewhat similar to our goal of complementing programmer work on fault tolerance. They employ an agent model called Autonomous Response to Unexpected Events (ARTUE) and which uses a Hierarchical Task Network (HTN) planner to produce plans and their expected results. "Discrepancy detection" is then used to identify any deviation from the expected results, which then is evaluated by a "explanation generation" system that produces hypotheses on the possible cause. The discrepancy can lead to new goals being created to solve problems or take advantage of opportunities. The new goals can influence the goal management as they may modify which goals are currently executing. While we are similarly interested in the good runtime behaviour in the presence of unforeseen difficulties, their work focuses on external, where ours is concerned with internal issues. Furthermore, while their work is based on having a planner, we speak of the possibility of using one, but we focus on readily-written plans (as in Jadex, Jason etc.). We also note they use an Assumption-based Truth Maintenance System (ATMS)[3] for handling hypotheses, which, as shall be seen, is also part of our solution.

3 Discussed here in Sec. 2.4.2.

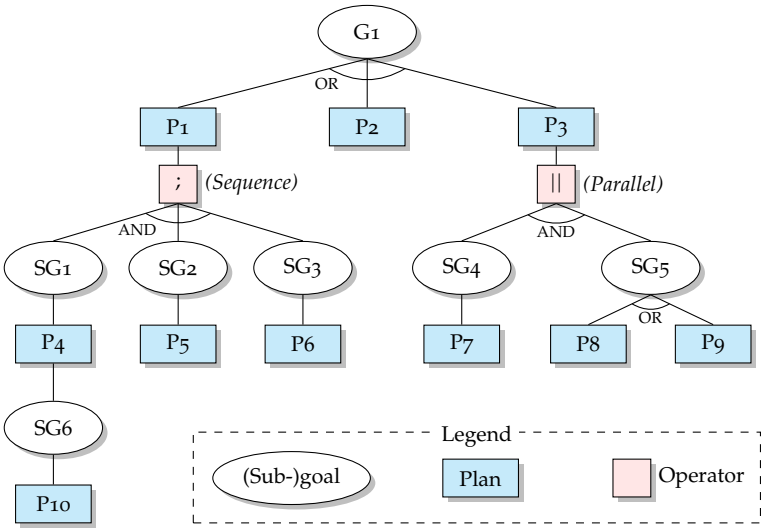


Figure 10: An example of Goal-Plan Tree (GPT)

2.3.4 *The Goal-Plan Tree*

Thangarajah [108, 109] formalises the representation of the agent model in the form of an AND-OR tree: the Goal-Plan Tree (GPT). Goals are *OR* nodes since their child nodes, the plans, offer alternative solutions and only one plan suffices for the achievement of a goal. Plans on the other hand are *AND* nodes in order to denote the obligation to achieve all the adopted sub-goals for a successful plan execution. Furthermore, two operators are added to the plan node, to indicate either that the goals have to be achieved in sequence (;) or in parallel (||). A generic example which illustrates all these is given in Fig. 10. Here, the GPT using the two operators spreads in depth across several levels. Note that there can be more than one tree for a given agent, in other words more than one root goal. This model is used more as an analysis than a development tool as it represents execution traces: the same agent can have different GPTs depending on the specific goals adopted during a certain execution.

The goal-plan trees have been used in various works for representing agent specifications and as a basis for further treatments. In [109] GPTs are used to gather resource requirements called summary information and identify possible goal interactions. This is due to the hierarchical structure of the tree where summary information can be propagated upwards towards the root of the tree. Further works on the subject [49] reuse the model to illustrate their operational semantics for the goal life-cycle. However, Shaw et al. propose different approach for handling goal interactions using Petri nets [103] and constraints [104] instead of GPTs.

Singh et al. [105] use learning for plan selection in BDI agents. They also use GPTs to describe the agents and even note briefly that only *"leaf plans interact directly with the environment"*, which is consistent objective for the GPS approach. This allows for a representation where, given the results – i.e. *success* or *fail* – of the executions of all leaf nodes, the success or failure of the root node is decided by simply propagating these logic values in the AND-OR tree. This shows a benefit of separating agent reasoning from acting, for, if actions were included in intermediary plans, even if all sub-goals of a plan were achieved, the plan would not necessarily cause the achievement of its parent goal. The GPT is therefore already a simplification of the system, as it uses the rather strong hypothesis that there are no perturbations, such

as the one in the afore-mentioned case, in the AND-OR tree. Another example of "perturbation" in the propagation of success values in the tree can be the use of specific achievement and failure conditions for each goals [76, 99].

2.4 ALMA: AN AGENT LANGUAGE FOR DEPENDABLE AGENTS

We will now describe the ALMA agent programming language[4] whose main goal is to permit writing agents capable of interacting with autonomous agents, in particular to adapt to their unpredictability. As a consequence, the designers of the language placed a strong emphasis on working with assumptions. As we argue in Sec. 4.1, the language's built-in fault tolerance features and other characteristics (e.g. the use of assumptions, modularity) recommend it for our safety net approach and therefore the implementations for this thesis (both for the safety net and the GPS) were done in ALMA.

2.4.1 *ALMA Motivations*

Agents, just as humans, are often pressured into acting without complete information on their current context. They therefore need to base their reasoning and acting on assumptions that may later be proven wrong. Inconsistent assumptions can be used when reasoning: "supposing the weather is nice tomorrow, I'm calling my friends to organise a barbecue; in parallel, supposing it rains, I'm borrowing this projector so that we can all see a film indoors". The agent (or person) will most certainly end up doing only one of the actions and cancel the other when the assumptions will be confirmed, but the advantage is that all cases were well covered.

Some facts are known to be true. We can thus write $true \Rightarrow fact$ which reads "fact is true under all circumstances". However, in case newer information contradicts the previously "sure" fact, the agent may find itself in the impossibility to continue functioning. The statement "water boils at 100° C" may seem a sure fact, until one is exposed to a different atmospheric pressure, for example due to altitude, resulting in the contradiction of this "sure" fact. A 1800s sailor or a robot with too strict rules would be seriously confused by such an event. It may thus be useful for the sailor and the agent to be able to take a step back and consider the options, possibly taking into consideration both contradicting facts and being able to discard any of them depending on other information and criteria.

Both these examples show the utility of assumptions and as well as the capability to function in multiple, possibly inconsistent contexts, something that we humans are able to do.

When taking this perspective, facts can be based on other facts and assumptions: $f_1 \wedge \ldots f_n \Rightarrow fact$. For example instead of having $true \Rightarrow temperature = 10$, we would have $sensor \Rightarrow temperature = 10$. This means that when we add $sensor' \Rightarrow temperature = -5$, the agent can conclude that $sensor$ and $sensor'$ cannot be both correct simultaneously and it can even continue reasoning in both hypotheses in parallel: either $sensor$ or $sensor'$ is right.

For an agent, the environment in which it exists and acts can be [29]:

1. More or less observable, in terms of completeness, accuracy and of the information available to the agent.

4 An in-house language developed in collaboration between TSA and LIP6.

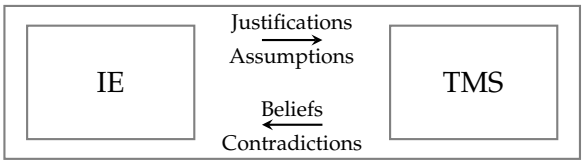


Figure 11: Problem Solver = Inference Engine + Truth Maintenance System (from [40])

2. Deterministic or not, in the sense that an action always produces the same results.

3. Static or dynamic, depending on whether the agent is the only one who can produce changes in it.

In most real applications, the environment is partially observable, not deterministic and dynamic, so the use of assumptions is useful to allow for the agent to act without having all the data. This also has a positive effect on the fault tolerance, as the agent is more flexible and less likely to crash due to a contradiction.

Following this rationale, the ALMA agent programming language [29] was developed with the purpose of creating agents able to reason and act under uncertainty. Particular care was given to the fault tolerance characteristics of the language and its platform, as we shall see in the following.

ALMA can be used for defining agents with respect to their behaviour, decisions, use of beliefs and communications. The language aims at a high level of abstraction as the sole action an agent can perform is sending a message[5], with actuators and sensors being represented as artefacts that are exterior to the agent and with which the agent interacts only through messages. This is an important aspect as it facilitates the abstraction of hardware errors which can be handled by specific means in their own artefacts.

The result is an imperative agent programming language whose central point is the way beliefs are handled. It uses a rule-based inference engine together with a consistency checker to represent assumptions and complex reasoning, while various fault tolerance elements are present both at language and platform level. ALMA was implemented on top of Prolog, so, as we will describe later on, it shares certain characteristics with the language, notably regarding data types.

As we used ALMA as a base for our work and experimentations for both main contributions of this thesis (Parts II and III), we will now describe the language into more detail. In order to fully understand ALMA, we need to first describe the underlying mechanisms that make its belief and assumption management possible: the problem solver composed of an inference engine – truth maintenance system duo.

2.4.2 *Problem Solvers and Truth Maintenance Systems*

The problem solver model on which ALMA is based is composed of two parts (Fig. 11): an Inference Engine (IE) focused of drawing inferences and a Truth Maintenance System (TMS) focused on storing and managing beliefs, assumptions and contexts [40, Chapter 6].

5 In this acception of the term "action", we do not include belief writing. However, note that other internal "debugging" actions such as console writes are possible in the ALMA language.

THE INFERENCE ENGINE A pattern directed inference system (PDIS) [40, Chapter 4] works with two types of data:

1. declarative data in assertions (facts, which when justified are also called "beliefs");
2. procedural data in rules: (rule < trigger > . < body >), where the body can be an assertion, another rule etc. Rules are applied indefinitely, remain in the database forever and are order-independent.

The inference engine uses forward-chaining: it applies rules on existing assertions to produce new ones and it ensures the rules are applied until the system reaches quiescence[6] for each new rule and assumption added. It is important to note that the order of application of rules is not deterministic.

THE TRUTH MAINTENANCE SYSTEM Generally speaking, a TMS allows the problem solver to:

1. identify responsibility for conclusions, in order to provide more convincing solutions (e.g. why the proposed reparation) and more useful explanations (e.g. why the system does not work, not just that it does not). This is done by tracing the justifications for beliefs.
2. recover from inconsistencies by tracing backwards justifications to find the sources of bad conclusions.
3. avoid unnecessary repetitions of computations and reasoning by maintaining a cache of inferences.
4. guide backtracking by indicating assumptions that lead to contradiction, thus identifying early the branches whose exploration is futile.
5. manipulate assumptions and use default reasoning, i.e. reasoning based on insufficient information (e.g. Tweety is a bird therefore it flies, unless proven contrary). The TMS allows assumptions to be retracted gracefully.

While it offers these advantages, a TMS is not a solution for any system. For example, if the rules used are inexpensive to apply and the system does not have a very large number of rule applications, then storing the results of their application may not be justified as the TMS might actually slow down the resulting problem solver. Also, the rigid form of rationality imposed by the TMS in which rules are added forever in the database and no facts can be simply retracted can be a problem for some purposes.

In a problem solver, the inference engine interprets data and decides which rules are applied. These are then given to the TMS in the form of justifications. The TMS stores all data in form of nodes, which it cannot semantically understand. Its job is to maintain a dependency network between these nodes and reply to queries from the problem solver regarding justifications, contradictions[7] etc. At a given moment, these nodes can be believed to be true or not by the problem solver, depending on the existence of a valid justification for them at that moment. The dependency network is a bipartite graph containing assertions (facts, beliefs) and justifications.

6 Quiescence: state in which there are no more rules that can produce new beliefs

7 It is the TMS that detects contradictions and informs the IE.

		P	
		In	Out
¬P	In	Contradiction	¬P
	Out	P	Don't know

Figure 12: Representing a node and its negation

The set of justifications grows monotonically as there are usually no means to retract a justification. However, the set of enabled (believed) assumptions is always subject to change.

The TMSs work with definite clauses, which, by associating each node a propositional symbol, allow us to write the justification of a node n in the following manner:

$$\neg x_1 \vee \cdots \vee \neg x_m \vee n$$

which is equivalent to

$$x_1 \wedge \cdots \wedge x_m \Rightarrow n$$

A node of the TMS can either be *believed* or *enabled*, "in", or not believed, "out". Given a set of justifications $\mathcal{J}$ and a set of enabled assumptions $\mathcal{A}$, a node x is labelled "in" if it logically follows from $\mathcal{J} \cup \mathcal{A}$ and "out" otherwise. This, however, does not imply its logical value, as see in Fig. 12. Note that in order to represent a negation, a new node corresponding to the negated assumption needs to be created.

Let us consider for example a *belief* "Sky colour = blue". "Sky colour" is an attribute, while "blue" is its value. A belief can be "in" because: (1) is was declared to be always true, in which case it is called an *premise*, (2) it is justified *by itself*, in which case it is an *assumption* or (3) justified by other beliefs through rules. As depicted in Fig. 12, there is a clear difference between a fact that is not believed, i.e. not known to be true, and a belief that is known to be false.

A special place is given to contradictions, which can be explicitly introduced through rules whose right-hand side is *false* ($\bot$). When such a rule is activated, measures need to be taken by retracting assumptions. If there are no assumptions that can be retracted, an error is produced (it means that we reached a case where $true \Rightarrow \bot$). While the TMS does not explicitly allow negation of nodes, this can be achieved for example for a node n by creating another node $\neg n$ and adding the justification $n \wedge \neg n \Rightarrow \bot$.

THE ASSUMPTION-BASED TRUTH MAINTENANCE SYSTEM The Assumption-based Truth Maintenance System (ATMS) [40, 59, 61] provides a tool for handling hypotheses and multi-context applications. This means that, as opposed to other TMS like the Justification-based Truth Maintenance System (JTMS)[8], the ATMS can handle multiple inconsistent contexts in parallel, with a node being in the same time "in" and "out" in different contexts. For example when reasoning in the same time on the hypothesis that tomorrow it rains and tomorrow it will be sunny, an agent will be able to use either hypothesis independently but will not be allowed to work with both in the same context, as this would cause a contradiction. The belief that tomorrow will be sunny will be "in" for one of the two contexts and "out" for the other. The ATMS handles facts linked through rules, all in propositional logic.

Definition 3. *A set of assumptions is called an environment.*

8 The JTMS maintains a single consistent justification context at a time.

A belief holds in an environment if when all the assumptions in the environment are enabled, the belief is enabled. A *nogood* is an environment in which a contradiction holds. A *consistent* environment is one which is not a nogood.

Definition 4. *All the facts that can be deduced from an environment form a context.*

Definition 5. *The label of a fact comprises all the minimal consistent environments that support that fact.*

An environment is minimal with respect to a belief if removing any of its assumptions causes the belief to no longer be supported. There are two special cases to consider:

1. the empty label, corresponding to a disabled belief;
2. the label of a premise, which signifies that the premise holds in any environment. Nodes that are not premise nodes can have the same label if they ultimately depend only on premises. Therefore, labels contain only assumptions (no premises).

Contradictions must be entered by the inference engine. This can be produced by an application specific rule (e.g. $\mathtt{Netherlands} \wedge \mathtt{mountain} \Rightarrow \bot$) or an automatic rule (e.g. as shall be seen, normal beliefs in ALMA are governed by an "unique value" rule $\mathtt{belief}(B1, V1) \wedge \mathtt{belief}(B1, V2) \wedge V1 \neq V2 \Rightarrow \bot$). When the contradiction becomes justified, the ATMS updates the labels of its nodes to ensure that there are no environments that can result in the contradiction, nogoods.

As stated before, the rules are handled monotonously, meaning that once they are added, they cannot be removed. However, their application could be controlled by adding an "applicable" assumption to the left-hand side of the rule, for example $\mathtt{applicable}(\mathtt{rule}) \wedge \mathtt{left} \Rightarrow \mathtt{conclusion}$. When using this instead of a rule $\mathtt{left} \Rightarrow \mathtt{conclusion}$ the programmer can later prevent the rule's application by adding the contradiction $\mathtt{applicable}(\mathtt{rule}) \Rightarrow \bot$ to cause the assumption to be retired.

2.4.3 *Parenthesis on Model Based Diagnosis*

One of the applications of the ATMS is for Model-Based Diagnosis (MBD) [62]. We chose to present this as an example because we will use elements inspired by MBD in our safety net approach (Sec. 3.3.1).

In MBD, a system's correct behaviour is described using three sets of logical formulae:

- system components set (COMPS)
- system description set (SD)
- observations set (OBS)

The idea is that in case the actual observations OBS is not consistent with the system description SD, the diagnosis will attempt to identify the "guilty" components among the elements of COMPS. The ATMS serves to handle dedicated assumptions ($\mathtt{ok}(X)$ in our examples) that correspond to components that are functioning correctly. In the example in Fig. 13, three resistors are connected to each other as part of a simple circuit. In Fig. 14 we introduce the three MBD sets using rules which

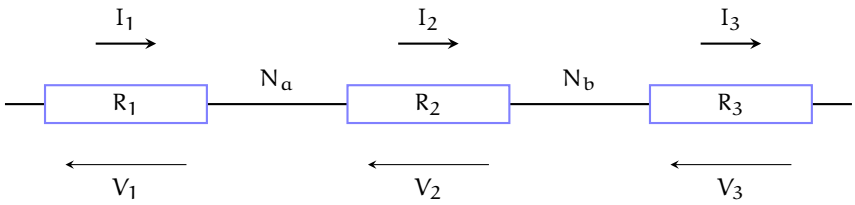


Figure 13: System example for diagnosis: three 100Ω resistors

$$\begin{aligned}
\text{COMPS} &= \{R_1, R_2, R_3, N_a, N_b\} && (1)\\
\text{OBS} &= \{V_1 = 10, V_3 = 10, V_2 = 12\} && (2)\\
\text{SD} &= \{\\
&\{ok(R_1) \Rightarrow V_1 = R_1 * I_1\} && (3)\\
&\cup\\
&\{ok(R_2) \Rightarrow V_2 = R_2 * I_2\} && (4)\\
&\cup\\
&\{ok(R_3) \Rightarrow V_3 = R_3 * I_3\} && (5)\\
&\cup\\
&\{ok(N_a) \Rightarrow I_1 = I_2\} && (6)\\
&\cup\\
&\{ok(N_b) \Rightarrow I_2 = I_3\} && (7)\\
&\}
\end{aligned}$$

Figure 14: The MBD sets for the resistor example

allow the introduction of the specific formulae depending on whether the corresponding components are functioning correctly or not. The underlying logics in not important, as long as the system has access to a mechanism for proving the consistency of a formula.

The system is faulty if the assumption that all components COMPS are functioning correctly conflicts with the actual observations OBS.

$$\text{Faulty system} \Leftrightarrow \text{SD} \cup \text{OBS} \cup \{ok(c) | c \in \text{COMPS}\} \text{ is inconsistent}$$

A diagnostic is a subset Δ of COMPS such that, supposing that the components in Δ are functioning incorrectly and the others correctly reinstates the consistency of the set of relations:

$$\Delta \subseteq \text{COMPS} \text{ is a diagnostic} \Leftrightarrow \text{SD} \cup \text{OBS} \cup \{\neg ok(c) | c \in \Delta\} \cup \{ok(c) | c \in \text{COMPS} \setminus \Delta\} \text{ is consistent}$$

While proving the consistency is difficult, the fact that the $ok(X)$ predicates are similar to assumptions bring us back to the concept of *nogood* introduced in the previous section on the ATMS. For determining a diagnostic, it suffices to compute the minimal sets that make the system inconsistent (i.e. the nogoods) and then combining them by calculating the *hitting sets* (sets containing at least one element from each nogood).

$$NG \subseteq COMPS \text{ is a nogood} \Leftrightarrow$$
$$SD \cup OBS \cup \{ok(c) | c \in NG\} \text{ is inconsistent}$$

In diagnosis, the *conflict set* is comprised of the components which cannot all be working correctly. A *candidate set* is an assumption on how the error was produced, so it contains assumptions (ok(COMPONENT)) that when considered false can explain the observed symptom or error.

$$Candidate \cap Conflict_i \neq \emptyset, \forall i$$

The purpose of diagnosis is to find candidate set, which is often difficult due to the fact that many possible interpretations can explain an error. A minimality condition is often used to filter out unnecessary candidates, as the hypothesis is usually that the probability that multiple components fail simultaneously is usually very low.

In the example above, the given values result in two nogoods:

- $\{R_1, R_2, N_a\}$ because rules 3, 4 and 6 are inconsistent ($I_1 = 0.1A$ is different from $I_2 = 0.12A$);
- $\{R_2, R_3, N_b\}$ because rules 4, 5 and 7 are inconsistent ($I_2 = 0.12A$ is different from $I_3 = 0.1A$).

Computing the hitting sets in this case produces the list of minimal candidates, each of which can explain the observed symptoms:

$$\{R_2\}, \{R_1, R_3\}, \{R_1, N_b\}, \{R_3, N_a\}, \{N_a, N_b\}$$

This method can take into consideration multiple failures, but if the minimality condition is applied, the only remaining candidate is $\{R_2\}$. Otherwise, a means to filter the candidates is needed, for example using heuristics.

As we see in [62], the ATMS is useful when measurements are made on the system in order to help narrow down the conflict sets in the diagnosis process.

2.4.4 *The Programming Language*

ALMA agents are defined using Reasoning Threads (RTs), also called plans in the original work, which are Directed Acyclic Graph (DAG) structures using 4 types of transition node (Fig. 15): *wait* (*perception*), *decision*, *action* and *add rules* (*reasoning*), to which *start* and *end* nodes are added. Each transition node has a specific role, with one or more possible transitions towards the next node, each guarded by a condition on the current context, e.g. a specific event waited by the perception node. The language structure ensures that for each RT, all nodes are reachable from the *start* node and the *end* node can be reached from any node.

An ALMA agent starts with an initial RT which can then launch other RTs which will execute in the same time. The programmer has the possibility to require an RT to wait for the completion of a child RT if necessary. Parallelism is obtained through the sequential execution of RT segments that can belong to different RTs. The parallelism is thus handled in a safe way thanks to "critical" execution sections in which an RT is sure to be the only one having access to the agent memory. The critical sections are delimited by *wait* nodes. This ensures that between two *wait*

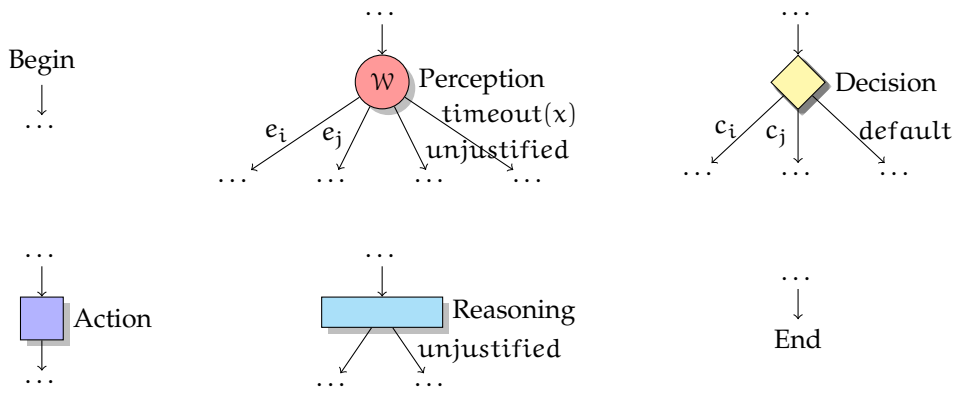


Figure 15: The original ALMA nodes: perception (wait), decision, action, reasoning (add rules) and the Begin and End terminal nodes. The compulsory branches represented for the corresponding nodes are: `unjustified`, `timeout(x seconds)` and `default`.

nodes an RT has exclusive access to the agent memory, thus avoiding consistency issues linked to parallelism. This design choice is based on the hypothesis that RTs are written with a cooperative state a mind, in that one RT will not monopolise the use of resources by avoiding *wait* nodes and blocking the rest of the agent[9].

RTs can recursively call on instances of themselves, thus eliminating the need for iterative loops (which are actually not allowed in ALMA). RTs can be passed parameters when they are instantiated which, while circumventing the agent memory and beliefs, can facilitate certain tasks. Besides from helping keep RTs short, the recursiveness could be coupled with an input variable validation (e.g. typing, or more precise, on values) applied to the RT parameters to detect errors early.

THE WAIT NODE A *perception* node waits for an event. It can have multiple transitions marking different event types:

- a message was received (of a specific nature or not);
- another RT finished;
- a belief is given a (specific or not) value;
- a timeout (with respect to the current *wait* node) occurred. This is a compulsory transition for this type of node, thus avoiding the situation when an agent blocks in a waiting state (e.g. for a message that was lost);
- the RT is no longer justified (unjustified), another compulsory transition marking the fact that the context of the present node is no longer consistent (as described by the RT context below).

THE DECISION NODE A *decision* node is equivalent to an `if-elseif-else` where the final `else` (marked `default`) is compulsory, thus ensuring that there is always

9 The default task management in JACK [1] is very similar to the solution chosen in ALMA, as an agent's tasks are executed sequentially from a queue until finishing or reaching a `@wait_for` or `@sleep` statement, in which case the task moves back to the queue. The possibility of a task to block the execution (as in ALMA) is acknowledged by the authors. An alternative round-robin based manager aims to overcome this by allowing a maximum predefined number of task steps (Java code counts as one step) to execute for each task turn.

a transition that can be crossed. The transition that is triggered corresponds to the first condition that is valid. A condition is a conjunction of elements of various types:

- beliefs, used to verify the existence of a certain belief value, but also to query for the current value of a belief;
- a boolean function, which, due to the Prolog-based implementation, is currently possible only through a Prolog predicate.

For example, a decision can be (Fig. 16):

$$\begin{aligned} &\texttt{node\_name decision} \\ &\qquad \texttt{node\_climb\_more} \;<<\; \texttt{belief}(\texttt{altitude}, X) \wedge \texttt{goal}(X < 3000) \\ &\qquad \texttt{node\_do\_nothing} \;<<\; \texttt{default} \end{aligned}$$

where the "`goal`" keyword marks a predicate that can also be defined outside the ALMA code. This offers multiple possibilities, while keeping the overall ALMA code readable.

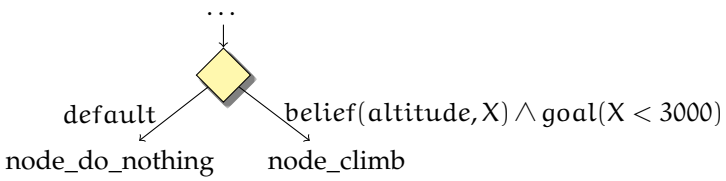


Figure 16: *Decision* node example

THE ACTION NODE An *action* node is given a list of actions to perform and only has a single possible transition. An action can be:

- sending a message;
- creating an RT;
- a `for each` containing actions;
- demanding the migration of the agent;
- a printing action.

THE REASONING NODE In ALMA, the agent reasoning is represented by a set of rules which control the beliefs and assumptions. Rules and assumptions are added to the rule base using the *reasoning* nodes (namely `add rules` in the language).

The node has two possible transitions:

- the normal continuation after the rules are added;
- *unjustified* (compulsory), activated in the event that the rules added negatively affected the preconditions of this node (as in the case of the similar transition in the *wait* node, see the description of the RT context below).

RULES The agent knowledge in ALMA is represented through rules. Rules are presented as clauses of the form:

$$p_1 \wedge \cdots \wedge p_n \Rightarrow c_1 \wedge \cdots \wedge c_m$$

Concretely, rules have the form $decision \Rightarrow belief\_conjunction$, where $decision$ is as defined for the *decision* node, e.g.:

$$belief(weather, cloudy) \wedge belief(temp, X) \wedge goal(X < 0) \Rightarrow belief(clothes, ski\ jacket)$$

Once added, the rules continue independently of their parent RT, cannot be retracted and are applied permanently. This means that nothing is lost in the agent memory, but beliefs may end up disabled, depending on their justifications.

Note that the use of $true \Rightarrow p$ rules (corresponding to the declaration of premises) is closer to "classical" imperative programming and limits the inference engine's task. The risk with such premises is that they may produce system level contradictions, in other words a set of rules that ca be reduced to $true \Rightarrow \bot$, i.e. the "universal environment" is contradicted, in which case there is currently no choice but to stop the agent.

ASSUMPTIONS Assumptions are beliefs that are considered enabled but do not have a justification. This allows them to be disabled in case they are either directly contradicted or they serve as premises in a chain of rules that result in a contradiction.

THE ATMS IN ALMA By now we have seen many similarities in the use of rules and assumptions with the problem solvers described above. ALMA thus uses an inference engine and an ATMS for reasoning and handling assumptions, beliefs and rules. With the strong emphasis in ALMA on assumptions and reasoning with rules, the ATMS plays a central role in the programming language. While its reasoning capabilities can be used explicitly, e.g. adding rules with the purpose of performing multi-context reasoning, it is also incorporated in the language in two ways: (1) beliefs are used by an RT (in *decision* and *wait* nodes) are added to the RT's context as seen below, and (2) beliefs are written using rules, so justifications are kept for each of them.

Note that due to the multi-context possibilities of the ATMS, an assumption can be in the same time enabled and disabled, depending on the context of reference.

EXECUTION CONTEXT At every moment an RT is characterised by an execution context comprised of all the facts used by that RT together with the context that its parent RT had when it created the RT. In other words, the context of an RT represent the preconditions corresponding to the execution up to the current node of that RT.

We say that a belief is *unjustified* (no longer justified) when its label is empty. A context is unjustified when at least one of its beliefs is no longer unjustified. An RT is unjustified when its context becomes unjustified. Note that even if justifications change, a context can remain justified as long as all its beliefs remain justified. Take the example of a family going to the supermarket to buy a cake. While driving there, they realise that they also need milk, but decide to cut down on their sugar intake and not buy the cake after all. The reasons are now different, but they are still going to the supermarket.

The context contains beliefs that were used during the current execution trace. These beliefs were therefore enabled at a certain moment. As an *unjustification* is reached when at least one of the beliefs in the context is no longer justified, this is equivalent to saying that the current execution context is contradicted, in other words, a contradiction ($\perp$) can be deduced using the assumptions in the execution context and the existing rules. An agent cannot function in the presence of inconsistencies so it is important to be able to identify these situations.

EXECUTION CONTROL The context can be used to stop the execution of an RT in case it is no longer justified. In ALMA, this results in the RT entering a special reparation mode [28]. This is ensured by the *unjustified* branch that is obligatory for two types of node: (1) the *wait* node, because this is where another RT can modify agent beliefs and (2) the *add_rules* node as the RT can cause an unjustification is achieved when adding a rule itself. The advantage is that this format forces the programmer to consider the reparation required at that specific point in the code, thus providing a more precise and better response than any other generic reparation means to such situations. Furthermore, the precise reason why the reparation is triggered is not known, thus keeping in line with the idea of unforeseen fault studied in the current thesis.

This mechanism allows for the execution of RTs to be controlled and neatly stopped in case this is wanted (e.g. use a rule to willingly retract an assumption that allowed for a certain set of actions to be pursued) or a situation that was not foreseen changes the justifications of current RTs (e.g. update a belief only to realise that it contradicts something else), with the result being that the execution continues as specified in the reparation *unjustified* branches.

DATA TYPES HANDLED IN THE RTS Given that ALMA is written on top of Prolog, the data types that are handled based on the Prolog types: symbolic atoms, numbers (floats and integers) and compound terms (predicate style, for example $\mathtt{car(ford, mustang, 1967)}$, lists and strings). As stated before, parameters can be passed to RTs. These parameters, as well as other values from beliefs and messages are handled in the form of intra-RT variables. They are useful for example when creating the content of a message to send, as seen in Fig. 17. These are single assignment variables, which means that after they are assigned a value, any other assignation attempt acts as a verification: if a new value is proposed for the variable, the statement returns false, which constitutes a useful feature for avoiding unwanted variable changes.

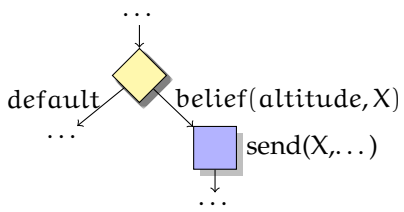


Figure 17: Possible variable use

DATA TYPES STORED IN BELIEFS The same type of data is handled by rules and stored in beliefs. Note, however, that while rules can use variables for example for

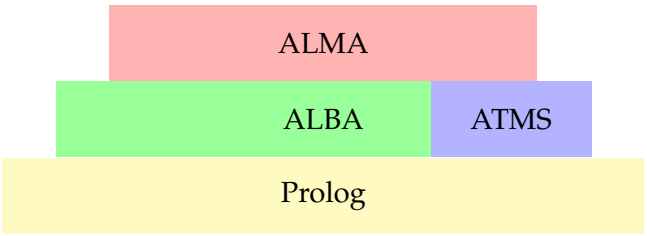


Figure 18: The ALMA framework

computing the value of a belief, beliefs are functional terms, they can only store values, but no variables.

BELIEF TYPES IN ALMA In ALMA there are currently three types of belief which are based on several variations of attribute-value pairs:

1. $belief(Name, Value)$ – for a simple "single assignment" belief, which cannot have two different values in a same context (otherwise, a contradiction is reached, which puts the concerned RTs in the unjustified state).
2. $set(Name, Value)$ – for representing beliefs with multiple possible values. Note that these are monotonous: no retract of value is allowed.
3. $belief(Name, Value, Timestamp)$ – for storing variables that change over time (e.g. state of a system). The particularity of this type of belief is that each time it is read at a moment T_{read}, an implicit persistence assumption is created to guarantee that the value did not change since the last known timestamp $T_{reference}$ corresponding to the current known value. If a late update arrives with a timestamp T_{update} with $T_{reference} < T_{update} < T_{read}$, the use of the value at T_read is unjustified, thus avoiding inconsistencies. This illustrates the use of assumptions for improving fault tolerance.

TECHNICAL ASPECTS As so often is the case with agent languages, ALMA comes with its own execution platform (called ALBA [30]) that ensures all lower level functionalities, from communications to agent creation, timing etc. and which, in turn, is implemented in Prolog (Fig. 18).

The platform is completely distributed, as each agent is executed in a separate Prolog instance with its own ALBA module, thus ensuring a clear separation with respect to agent memory and helping confine any errors at agent level. The responsibility of inter-agent parallelism is therefore passed to the operating system.

Through Prolog-based interfaces, ALMA can interact with any other language, a feature used for example to develop a means to use Java-based graphical interfaces.

A MAS initialisation and management system using a plain text file combined with a graphical interface is available for ALMA. This system also provides a yellow pages service. Figure 19 shows the ALMA agent architecture, together with the MAS Management Agent, distinguishing between the elements to be written by the programmer and the ones provided by the platform. Agent migration was also studied for ALMA but this is not relevant for our work.

Note that ALMA, while built on top of Prolog, is an imperative language. However, as seen in the case of decision conditions, Prolog code can be easily used inside ALMA. The "external" code is not limited to Prolog, as other languages can be interfaced with and used inside decisions and rules, thus giving many possibilities to program designers.

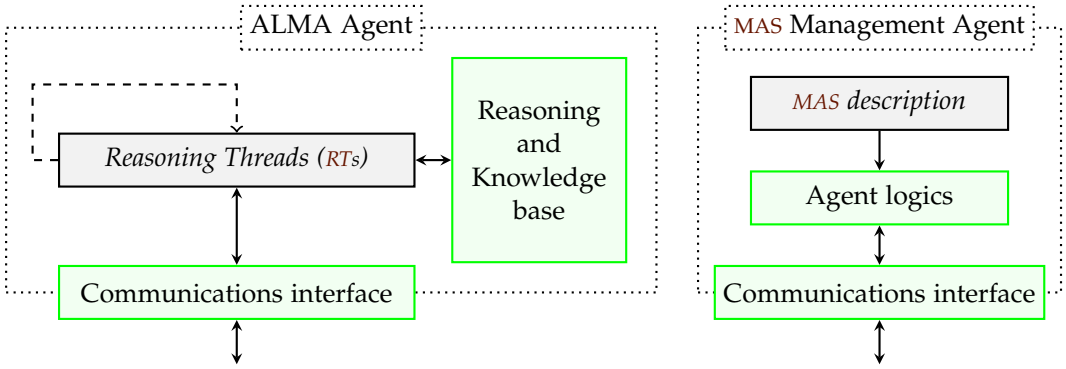


Figure 19: Defining an ALMA MAS, with normal agent architecture to the left and the MAS Management Agent (which includes the *yellow pages* service) to the right. Black rectangles represent the components provided by the programmer (RTs and the MAS description), with the rest (green) being provided by the platform. The dashed line represents the execution flow, while full lines are for information exchanges (messages, belief reads, rule writes).

CONCLUSIONS As can be seen, ALMA is an agent programming language that allows for the development and execution of agents with a specific assumption-based handling of their memory.

As it is a research prototype, several aspects of the language need improvement, for example the variable and belief types which need to be extended to enhance the language expressibility and thus the programming experience. Ameliorations in the memory management also need to be envisioned.

Nevertheless, ALMA incorporates many interesting features that make this language useful for use and further research, in particular for the development of fault tolerant software:

- the clear and cycle-less RTs provide a good programming base for modular code segments.
- the control of *wait* nodes through the compulsory *timeout* branches helps avoid the situations of infinite waiting.
- the execution control through contexts and the *unjustified* branch allow for programs that take into consideration inconsistencies to be written. The same mechanism also gives more control to programmers on the execution of programs, as they can willingly trigger reparations in case they require to do so (for example in case they detect another type of error, as we shall see in Sec. 4.3.1).
- the integration of assumptions and *unjustified* branches allows the agents to function under uncertainty and still be able, to a certain degree, to reconsider their behaviour in case the assumptions are no longer supported.
- the timestamped beliefs allow the program to take into account inconsistencies caused by messages arriving late.
- the *default* branch in the *decision* nodes helps avoid unexpected situations.
- the use of single assignment variables and beliefs improves the control on variable changes.

- the execution platform based on independent Prolog threads enforces the clear separation between agents thus contributing to the confinement of errors.

With a strong Prolog heritage, ALMA allows the execution of non-ALMA code only inside decisions and the left-hand side of rules, and that in a functional programming fashion: while not producing side effects. The actual variable writes are done indirectly trough the rules added by the programmer. The idea in ALMA is to have memory writes done only through *reasoning* nodes and the application of rules, while interactions with the environment are possible only via messages towards other agents or artefacts, the latter possibly having actuator capabilities. The "external" code executed inside decisions and rules is therefore without side effects, thus limiting unwanted and unforeseen interactions. Furthermore, as we shall see, this structure provides a good basis for safety measures that help create a fault tolerant language.

While rules provide a declarative component, the language in itself is imperative, thus, depending on the needs of the programmer, the written code can be more imperative (even only using premises and assumptions with no rule implications and the rest in the RT code) or declarative (using mostly rules).

ALMA is therefore an interesting programming language for our safety net approach as it proposes specific fault tolerance properties while in the same time allowing the use of the agent paradigm with its own advantages and fault tolerance mechanisms.

2.5 CONCLUSION

In this chapter, we presented various approaches that can be used for augmenting the fault tolerance of a software system, with a particular interest in works which are close to the concept of "unforeseen fault". We showed different uses of behavioural models, such as the observer, which can indicate, when compared with the observed facts, behaviour anomalies possibly caused by unforeseen faults. We also saw TibFit, a way to control the sensor information through a voting mechanism, as well as the agent-based use of norms and reputation, all of which can be used to increase the robustness in the presence of faults in distributed systems. Programming language elements were also discussed, from the issues of exception handling and defensive programming, to the "let it crash" philosophy and design by contract. While generic enough to cover unforeseen faults, many of these approaches are still introduced by the designers and programmers with the explicit purpose of handling errors. Our working hypothesis on the existence and nature of faults makes these approaches unsuitable, as long as they are not included into the design and programming approach in a way that does not purposefully aim faults. There are, however, also characteristics that are worth considering for our cause, including for example the idea of taking into consideration the autonomy of other components or agents, as well as the various fault tolerance elements of ALMA, such as the use of assumptions and the control of the execution through contexts.

Goals in agent systems, together with the Mission Data System (MDS), a goal-based control system, as well as the lower level recovery blocks and even design by contract serve the same purpose of using predefined objectives to detect possible deviations and keep the system within its specified limits. These objective-based

means are more powerful than the other language level error detection means that are used to produce exceptions, due to the fact that in their definition, a default means to treat abnormal situations is usually included – e.g. a goal whose plan does not finish successfully can retry automatically. As shall be seen later in this thesis, this characteristic makes these approaches particularly interesting for integrating in a solution for unforeseen faults.

A common error handling pattern is usually present, starting from a detection event, to which the system then reacts to recover. The isolation, or confinement, of the error is also an important aspect, especially in large and distributed systems.

This section concludes the state of the art of this thesis. Next we are ready to introduce our contribution to fault tolerance: the safety net approach.

Part II

CONTRIBUTION TO THE FAULT TOLERANCE

The Safety Net Approach

3

A SAFETY NET APPROACH TO FAULT TOLERANCE

> "A fool throws a rock in a lake and a hundred wise men cannot pull it out."
>
> Proverb

OBJECTIVE The objective of this thesis is to provide the means for assisting programmers in producing programs that in the presence of unforeseen faults have a more controlled and appropriate behaviour than a "brutal" stop or crash. For this, the programmers do not need to be aware of the fault tolerance measures, being either uninvolved or unconsciously involved in the implementation of the safety net mechanisms (depending on the requirements of each mechanism, as shall be seen). In the extreme case, the programmer writing his or her code without any concern for the fault tolerance but respecting our requirements would produce software that exhibits characteristics of fault tolerance thanks to the safety net approach.

CONTRIBUTION To achieve the desired fault tolerance properties, our contribution addresses 3 software development factors by requiring:

i. a set of *design requirements*: program using autonomous goal-driven agents, containing a certain level of redundancy and with a consideration for system granularity;

ii. a *programming language* that guides the programmer to specify all possible cases (e.g. providing a behaviour both in case a goal succeeds and fails, timeouts), as well as reparation measures and also improves fault coverage (probability of error detection);

iii. a *software platform* that provides a series of facilities for the fault tolerance: confinement of errors, dependency tracking.

In this chapter, we present how these are defined and show how their combination provides two levels of fault tolerance, each aiming to handle more and more subtle errors:

1. the first level is aimed at errors that generate exceptions which are not treated by the programmer;

2. the second level is aimed at errors that do not cause exceptions but prevent a software component from achieving its objective. This can happen due to an undetected local error in the component or even an undetected error inside another cooperating component.

As we aim to tolerate unforeseen faults, i.e. faults that were not included by the programmer in the system design, our approach is complementary to other fault tolerance methods. The aim of the safety net approach is thus to increase the overall system fault coverage.

THE FAULT TOLERANCE PERSPECTIVE In order to determine the necessary framework that will result in the safety net, we shall now discuss the issue of unforeseen faults from a fault tolerance perspective. Questions that are asked at this stage relate to the ability of the approach to catch errors, limit their propagation and then recover the system functionalities. As the objective of our work is to improve the fault tolerance of programs, we are concerned with the runtime manifestation of these faults, so a first phase to consider is the detection of deviations from the nominal states or behaviours, i.e. errors. Recovery measures can then be taken to attempt to compensate and eventually mask these errors. As the proverb above suggests[1], it is often much easier to do a mistake than to undo it. In what follows, we shall see that recovering from an error is a more complex task than the other phases of our approach. Detection can occur in a different component than the one in which the fault originated, a phenomenon exacerbated by our fault model, as we are interested in faults that were not aimed by specific mechanisms. For this reason we are also interested in means to confine the error propagation.

The chosen three phase approach on fault tolerance follows the reactions to a fault chronologically, from (1) detecting the error, through (2) its confinement and to (3) system recovery. As shall be seen, while the faults that we aim to tolerate are unforeseen, the reaction of the system needs to be triggered by an error detection event. To facilitate confining the error to a limited part of the system, as well as the recovery process, a good modularity is required. Finally, recovery is performed in three steps: (a) identification of the concerned components, (b) reparation of these components and (c) reconfiguration of the system. To illustrate these steps we use the metaphor of a boat or ship to compare it to the system being programmed. The boat, just as the system, has a purpose: to get a certain load from one point to the other. While there are many ways in which it can fail, the one we are interested in is the boat taking on water and eventually sinking, thus not reaching its destination.

The analysis of these three phases of fault tolerance will then allow us to return to the software development perspective and list the requirements with respect to the used tools and the programmer's state of mind and actions. In the following chapters, the discussion will therefore switch between the fault tolerance and the software development perspectives (Fig. 20).

3.1 EXPECTING THE UNEXPECTED: ERROR DETECTION

UNFORESEEN BY WHOM? The concept of unforeseen fault might seem paradoxical as one might question whether not all faults are unforeseen. This is not the case: while the moment when a fault manifests is usually not known, the actual faults or fault classes are normally identified. This is because the classic approach when building a system is to foresee everything that can go wrong and either design the system in order to avoid those situations (i.e. fault removal) or include corresponding behaviours in case the faults do manifest (i.e. fault tolerance). It is thus with respect to this identification that faults can be unforeseen. The goal is to provide a framework that allows for faults to be overlooked by the programmers – either due to time or cost constraints, either by actual design errors – without disastrous consequences: they are still covered, but by the safety net. However, we will see

1 Obviously, the programmers' mistakes would be different in nature, but the issue remains: how can they be reversed or repaired?

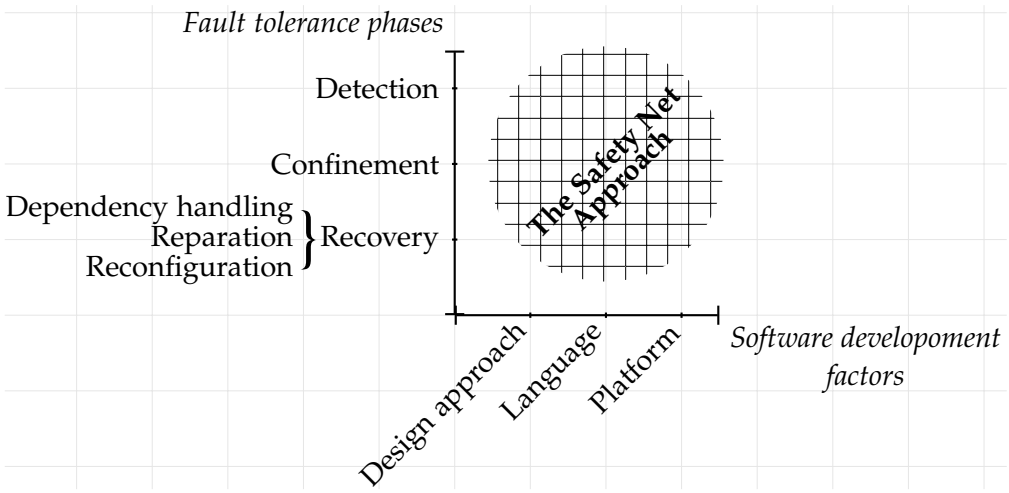


Figure 20: The two discussion axes for the safety net approach

that certain types of fault and errors are discussed here, but this is done from our perspective – that of the safety net designer.

FROM FAULT TO ERROR Note here that the distinction between fault and error is very important: while the fault is unforeseen, when it manifests as an illegal state it becomes an error whose manifestation can and should be detected. We call *unanticipated errors* the errors for which no specific handling was provided by the programmer of a system. We use the concept of unanticipated error as a manifestation of the unforeseen fault in the system state. We do not need to take into consideration the cases where unforeseen faults end up being successfully covered by fault tolerance mechanisms aimed at other types of known fault, thus not producing an unanticipated error. However, a badly handled error – be it foreseen or not – should be handled by the safety net. For example if a system reacts to an error by restarting a component which ends up in the same erroneous state, resulting in repeated (possibly infinite) restarts, the other components should eventually decide to give up – e.g. due to timeouts –, even if no other symptoms are visible.

APPROACH Our objective from the perspective of the detection phase is to provide mechanisms that allow the coverage of the fault tolerance of the resulting system to be improved without the conscious involvement of the programmer. For a boat or ship, an error would be having water inside, while the fault could be anything from a hole in the hull to too much cargo or very bad weather. A non purposeful detection that would be close to the idea of unforeseen fault is the moment when the cook goes to the galley to pick up potatoes and finds them floating in a half a meter of water. This would clearly indicate that there is a problem, without knowing its cause. Placing the storage room in an area that is at risk would be a way to ensure that a problem is detected early enough through the "cook method". For the cook, this would be an unexpected fault. For the ship designer, if positioned in this manner with a detection purpose, it would be a normal, foreseen, fault. If, however, the layout was made due to design rules that the ship designer was required to apply without specific a fault tolerance purpose, then this would constitute tolerance to unforeseen faults by design. In this thesis we position ourselves at the level of definition of the design requirements, while the ship designer

would be the programmer and the "cook method" would correspond to a runtime behaviour. In software, the *error detection* is an event produced to indicate that an abnormal state of the system was observed.

We continue this section by listing a few techniques that can be used for detecting errors, with a focus on language techniques and unanticipated errors. The idea of this thesis is to provide a framework that allows the developers to build fault tolerance into their systems without this being a conscious action. With the detection being a vital component of the approach, we aim at mechanisms that, while seemingly helping specify the normal behaviour of the system, they also set clear boundaries that are not to be crossed. Once these boundaries are crossed, the other components of our approach are called into action to provide the confinement and recovery. When they involve language features thus concerning the human programmers, these mechanisms need to be sufficiently acceptable – difficult and complicated constructs and the languages that impose them end up being avoided by the programmers. Also, these need to be light as requiring more code to be written increases the risk of introducing more faults. The aim is to avoid, for example, ending up in a defensive programming (see Sec. 2.1.7) mindset where the programmer is required to permanently verify all the possible values and cases, regardless of the other tests and guarantees provided by other components, thus moving away from the issue of unforeseen faults. Lining all the hull of a ship with sensors to detect any crack or breach would be the equivalent of defensive programming – expensive and difficult to maintain. What we aim for is having simpler error-focused mechanisms, for example sensors for water in the lowest levels of the ship, as that is where water would end up regardless of the location of the breach, if any.

Depending on the scope of the detection mechanisms, we distinguish 2 levels:

1. exception-based detection mechanisms with a focus on lower level errors such as those concerning programming bugs and data,

2. objective-based detection mechanisms that are aimed at higher level errors that do not trigger exceptions but prevent, nevertheless, the objective of the respective component of being reached.

3.1.1 *Exception-Based Detection*

Exceptions are a powerful tool for signalling error detections in modern programming languages. Any exceptions (or errors, depending on the vocabulary of the chosen language) that would be thrown at runtime by the execution environment and not caught by any programmer-specified mechanism can cause the program to crash. This is specifically the type of error that our safety net aims to cover. Naive well-known and yet dreaded examples are the division by zero in Java (among others) and segmentation fault errors in C. These uncaught exceptions can be generated by the language and platform, or even thrown but not handled by the programmer. In the following, we describe a few language characteristics that can contribute to the detection unanticipated errors.

(1) DATA TYPING Data typing is present in many mainstream programming languages. For the tolerance of unforeseen faults, what is interesting here are the runtime rather than compile-time verifications provided by such mechanisms (even though the compile-time verifications do contribute to the fault removal during

development), for example the content of a received message not being of the expected type.

(2) SINGLE ASSIGNMENT VARIABLES A more specific technique is the use of single assignment variables that can help identify unwanted variable changes in languages like Prolog[2]. The `X = 5` statement translates to "the variable X is assigned the value 5" if it previously did not have an assigned value, or "test if X has the value 5" if the variable already had a value. This can help identify unwanted and unexpected value changes, even in cases when this simply guides the execution elsewhere, such in Prolog where such a situation is considered normal and does not produce an exception. Single assignment, especially when coupled with an exception generating mechanism, forces the programmer into a state of mind where he or she is more aware of the values used and their changes than when using destructive assignation (where variables content can be overwritten at will). These variables also cause an exception to be thrown in case their use is attempted before their first assignation.

(3) CONSTRAINT PROGRAMMING Constraint programming [52] also provides interesting properties for the detection of unanticipated errors. While they are primarily used for specific problem solving, constraints also allow for the detection of unanticipated errors, for example when the inputs lead to impossible solutions regardless of the cause of the incoherence. It is this type of unexpected implicit detection that helps increase the fault coverage of systems without the programmer being directly concerned by errors that we are interested in for the tolerance of unforeseen faults.

(4) DATA ANOMALIES AND (5) INCONSISTENCIES In a wider sense, constraints can be used to signal errors in various situations such as anomalies in data streams, e.g. the value corresponding to the altitude of an aircraft varies too abruptly (see Sec. 2.1.2), and inconsistencies in the existing data, e.g. $city = Dubai \wedge weather = snowing$ (as shall be seen in Sec. 4.3.1).

3.1.2 *Objective-Based Detection*

(6) "CONTRACTS" AND (7) RECOVERY BLOCKS VERIFICATIONS Certain mechanisms are used to evaluate the degree of success of an execution (in terms of duration, produced results etc.), for example the "contracts" proposed by Bertrand Meyer (see Sec. 2.1.8) and the verifications proposed in the recovery blocks approach (see Sec. 2.1.5) can help identify errors without aiming specific faults.

(8) GOAL VERIFICATIONS (BDI) The recovery blocks approach marks the transition towards techniques that are based on state evaluations but are also an integrating part of another – fault tolerance oriented or not – mechanism and thus do not produce an exception as the cases before. The (8) BDI agent model provides another detection mechanism, which interestingly enough is implicit to the model. The model requires plans to be executed for the achievement of goals. If anything goes wrong during the execution of a plan (i.e. a fault manifests and the plan execution is not as expected), this may or may not result in an error detection event.

2 The `final` variables in Java have a similar signification but focused on compile-time verifications.

If an error event is generated, it will have to be handled by the mechanisms that are intended for this – either provided by the programmer or by the safety net. However, if no error event is produced, there is another way the system can overcome the fault manifestation: the goal satisfaction conditions may not be satisfied, in which case the faulty plan execution will not be considered valid – despite the lack of other signals – and the execution will continue as if the plan failed. As errors are defined as deviations from the normal state, we can consider that the non-satisfaction of a goal by the purposeful plan execution is an error. Note, however, that this error is not signalled through an exception as before, but through a goal-related event. Depending on the available resources, the same or another plan may be attempted, or the goal may fail, passing the treatment at goal level where a reaction to the goal failure may be specified. Let us consider the example of a simple drone that is asked to reach a position (x, y) (this is the goal) and it evaluates that it needs to fly for an hour in a certain direction to reach that location (this is the plan). If during its advancement varying wind speeds and slow it down and its advancement is less than expected, the fact that it did not reach the desired GPS (Global Positioning System) position after the execution of the first plan will cause the drone to execute a new plan to compensate for the error.

(9) TIMEOUTS Message exchanges can be seen as sure (e.g. in tightly coupled systems), when the reply characteristics, both in terms of time and content (i.e. the protocol) are "guaranteed" by design. At the other end of the spectrum, a defensive programming approach would check for each and every characteristic, both content-wise and time-wise. In the same idea, we propose a perspective of the programmer where he or she is forced to see the other components as autonomous, which in this context means that they cannot be fully trusted with their replies. This brings us to the error corresponding to (9) the lack of an expected reply from another entity. The non-reply, as explained in [86] in the case of agents, can be either caused by an error in the concerned entity or a communication error, but also by a decision of that entity. Either way, it constitutes an abnormal situation for the observing entity. This case, as well as other situations when a system waits for an event can be protected from a possible and unexpected blocking by imposing systematic maximum waiting times, *timeouts*, as it is done for example in ALMA (Sec. 2.4). The risk here is that when a programmer is forced to provide a timeout value that is difficult to evaluate or useless, he or she can use a very large – practically infinite – value that defeats the purpose of the mechanism. This, however can be identified when the code is reviewed by a peer. Another issue with such techniques is that they risk imposing "hardcoded" values for elements that may need more flexibility. Note that while we describe the timeouts as objective-based detection mechanisms, they can also be used to generate exceptions.

CONCLUSIONS Goals, constraints and other techniques such as requiring the programmer to specify timeouts for specific events (e.g. message replies) are thus important for the tolerance of unforeseen faults as they can constrain the programmer to specify these as normal program execution without this being a voluntary fault tolerance measure. Therefore, for the tolerance of unforeseen faults, a specific programming language, either totally new or an extension of an existing one, incorporating these elements will be required. In Chapter 4 we propose a programming language for the tolerance of unforeseen faults and in Sec. 4.3.1 we describe the specific detection mechanisms linked to the language characteristics.

All the detection mechanisms described here are agent-centred: there is either a local error detection, or an error corresponding to the non-fulfilment of the local objective. In our work we do not currently consider system level errors (such as errors that are not detectable at agent level) and errors that an agent detects in a different agent (e.g. "you sent me the wrong data, *you* are experiencing an error"). More on this in our perspectives for future work (Sec. 9.1).

There are thus various mechanisms that correspond to our needs and can be introduced into the framework being used. Let us now see how the system reacts once an error is detected.

3.2 AVOIDING FURTHER ERROR PROPAGATION: CONFINEMENT

THE PROBLEM Ideally, an error is detected in the same moment it occurs and at its source, but this is often not the case. In absence of specific mechanisms, errors freely propagate in the system both before and after being detected. This is especially true when we take into consideration errors that resulted from faults that were not foreseen at design time and thus not targeted by any specific detection mechanisms.

For example when a simple wooden boat without any compartments or separations is pierced by a rock, there is no way to prevent the water from spreading everywhere[3], so the boat may gradually fill up with water and eventually sink. In software systems, many errors concern data: either actual corrupted or incorrect data, or unwanted, illegal or erroneous data manipulation. These are then involved in interactions – data exchanges, actions etc. – that easily propagate the error to other parts of the system and to the environment. In a program where memory is freely shared, an error in a part of the system can impact any other part and is very difficult to control and trace. A sensor that is used outside its capabilities (e.g. an accelerometer subjected to greater accelerations than it can measure) can produce an erroneous value that is then treated by the navigation system that produces an incorrect order. If we imagine this happening in a monolithic layout with shared memory and all these components interacting with many other variables and components, a detection in the middle of the navigation procedure would largely be useless since all the other components would already be "infected" by the error. So the problem is that errors can freely propagate inside large software systems.

Furthermore, in complex, as well as in open systems, there is also the fact that the actions of parts of the system, sometimes even created by different programmers, are not always known and guaranteed, thus increasing the risk of unwanted interactions and error propagations.

There is therefore a need to constrain interactions to specific cases between clearly delimited components. This would help limit the propagation of errors, while in the same time give the possibility to control the interactions and stop the propagation upon the detection of an error. This would also offer the possibility to monitor or trace (as we shall see in the recovery phase) and even to add tests on these interactions.

For the purpose of the tolerance to unforeseen faults, it is important that the confinement is implicitly obtained through the used paradigm, language and architecture, and not a purposeful fault tolerance-oriented effort from the programmer.

3 Except from starting to bail the water overboard and hope to be efficient while also continuing to operate the boat (e.g. steer and maybe row).

APPROACH The *confinement* phase of the safety net aims at ensuring that once an error occurs and is detected, it can be restrained to a limited part of the system. This is addressed in two ways: (1) programs are written with a built-in modularity and (2) in case of an error detection, the reaction is to isolate, to "quarantine", the concerned components. Using an image from the world of shipping, instead of the simple layout of small boats, large ships (and submarines) are built with compartments that can be cut away from the rest of the ship in case of a hull breach, in order to limit the flooded areas and maximise the chances that the ship remains afloat. This means that the ship may need to give up the use of some of its compartments, but it will be able to keep functioning. In software systems, modularity implies the construction of the programs using more or less tightly-coupled components (e.g. procedures, objects, agents, packages).

An unanticipated error caught by the safety net mechanisms signals a situation that was not explicitly handled by the programmer, be it an exception-based or an objective-based error. Given the safety net is generic, the reaction in this case needs to be cautious. While the source or the impact of the error are not known, the detection event is a clear indication of an element impacted by the error and thus the reaction is to take measures on the concerned component. In the case of exception-based detection, the component where the exception occurred would be stopped, a similar approach to the "let it crash" described in Sec. 2.1.9. As we consider that the goal-based detection evaluates the system state only after the execution of a plan, the plan would not need to be stopped. However, as shall be seen in Sec. 4.3.2, if the evaluation itself produces an exception, then the goal needs to be aborted. Modularity and, as we describe below, granularity are thus very important for facilitating this first reaction to the error. The result is the protection of the system from further propagating the error[4], e.g. by writing corrupted data, sending corrupted messages or performing unjustified actions.

For the purpose of confinement, the software components need to be loosely coupled, so that they can be easily isolated in case of error. The solution is therefore to use a modular representation as a base paradigm for the programmers using the safety net approach. The other safety net mechanisms would then be in charge of catching unanticipated errors and stopping the concerned components. To ensure the required modularity, we aim at restricting memory sharing between modules and use communication through message exchanges, which are some of the main characteristics of the multi-agent paradigm.

CONFINEMENT THROUGH THE MULTI-AGENT PARADIGM The multi-agent paradigm corresponds to a new level of abstraction, in line with what has always been a source of progress in computer science: restrictions on the possibilities of the designer, coupled with new language abstractions. This was the case of the structured programming introduced mainly through the elimination of the "goto", but also the object oriented programming that offered a central role to data, until then seen only as memory space. The lack of shared memory and, consequently, the agents' obligation to only communicate through messages is one of the defining characteristics of the multi-agent paradigm and this creates a highly modular structure. We are not putting the agent paradigm into question, but rather place it into the context of fault tolerance, with a focus on three of the paradigm's characteris-

4 Unfortunately, as we discuss at the end of this chapter, there is a possibility that the error appeared elsewhere and may have affected and even may still be affecting and propagating to other components which, given the existing information, may be impossible to identify.

tics: (1) information sharing with no common memory, (2) communication through messages, as well as (3) the choice of a proper level of granularity. As agents share no memory and are clearly separated software entities, any error is inherently confined to the agent concerned and its propagation can be traced through the agent's communications and interactions with its peers. The distribution that is inherent to the multi-agent paradigm provides another level of confinement: the applications can be deployed on multiple machines, thus taking advantage of the hardware and operating system protections as well.

Changing the point of view to a higher level of abstraction, when designing agents, the concept of autonomy can help enhance the system-level confinement. As described in Sec. 2.2.7, autonomy is an agent's ability to make its own decisions, but from the outside it can be seen as the possibility to refuse even legitimate requests from its peers, "the right to say *no*" [86]. Designing agents that are meant to interact with autonomous agents makes them less susceptible to being affected by errors or even failures of their peers, thus contributing to confinement. While this requires a certain change of point of view from the programmer, it is primarily not focused on the fault tolerance, but rather on the "normal" functioning.

GRANULARITY Modularity in general comes with questions of *granularity*, the choice of component size, which also implies the number of used components in a given system. Granularity depends on the particularities of the chosen paradigm and the needs of the application and is often balanced (consciously or not) to meet the cost-benefit targets of each project. For example if two components exchange a lot of information, they may be better off merged. Likewise, if each component requires the use of a virtual machine, the number of components might need to be kept under control to avoid slowing down the host machine needlessly. When it comes to confinement, granularity plays a key role. Intuitively, having a system made up of only 2 similarity-sized components means that in case of error, the confinement options are limited to isolating one of the two relatively large components. If, on the contrary, the system is made up of many smaller components, there is more margin for confinement and the isolation will hopefully[5] concern only one or a few components representing a more limited part of the overall system.

Agent granularity is important for fault tolerance since the more the agents, the finer the confinement. The downside is that if the agents are not well chosen, resources (time, bandwidth, processing power) are wasted on excessive communication. Thus, depending on what each agent requires for its execution (generic elements such as modules to include, memory allocation etc., but also specific data requirements), there is a certain "reasonable" size for the agent. For the safety net approach, however, the number of agents needs to be high in order to facilitate the confinement of and recovery from errors.

CONFINEMENT THROUGH GOALS AND PLANS Given these different constraints, there is also a need for an intra-agent modularity. The agent representation that we require for the recovery phase (specifically the reconfiguration part described in the next section), involves writing agents using goals and plans which can also serve to produce a modular structure. While writing an agent with a single goal and plan would not be of much help for the confinement, breaking up the agent behaviour in multiple independent, related or even hierarchically connected

5 If the detection occurred early enough.

goals and their corresponding plans provides an excellent intra-agent modularity. This is similar to an employee working on multiple projects in the same time. If he or she encounters a problem in one of those projects, chances are that only that one is reconsidered or even dropped.

The added advantage of this representation is that stopping a plan, even unexpectedly, is natively handled by the goal-directed architecture which requires the execution to continue with the evaluation of the goal satisfaction condition and, depending on the result and programmer-provided specifications, acceptance of the goal failure or retries (more on the goal life-cycle in Sec. 3.3.3).

However, because plans usually have shared access to an agent's memory, their confinement is more difficult to realise compared to the case of agents. Just as in programming languages such as Java, the widespread use of public variables is frowned upon because they facilitate uncontrolled data exchanges, data exchanges between plans should be limited for the purpose of confinement. Similarly to the agent-message case, if two plans exchange a lot of information, it may be useful to merge them.

There is therefore a balance to be struck with respect to the granularity between the number and size of agents, and the number and size of their goals and plans. While at first each agent can be assigned a single goal, as we can see in various methodologies that use goal decomposition (e.g. Tropos [44]), in the end it is preferable to decompose that goal into more refined sub-goals. As the authors of the recovery blocks note (Sec. 2.1.5), such a structure opens up possibilities for confinement because failure at a level can be treated at higher levels. In our case, we favour the format in which higher level goals adopt plans whose role is to manage the adoption of lower level goals that refine the behaviour to eventually apply action plans. This model is described in detail in Part III of this thesis. The multi-level approach helps increase the chances that the system tolerates the unanticipated (or not) error, as error handling can manifest both horizontally – in the same plan with more or less "classic" fault tolerance mechanisms – or vertically – through the goals at various levels and their plans. Another advantage is that by representing the system through agents which are themselves made of multiple goals with plans, the overall level of granularity with respect to plans becomes lower as the size of these latter can be kept small. The consequence is that the number of interactions and dependencies of each plan is also limited, thus facilitating confinement.

UNRECOVERABLE ERRORS In the previous section we discussed detection methods. Plan-level confinement concerns errors that can be caught and handled locally, e.g. an illegal variable value. However, when it comes to unrecoverable errors, i.e. errors that are too serious for their component to continue executing (e.g. an `OutOfMemoryError` or other errors in Java which are meant to indicate a serious condition that usually requires the Virtual Machine to exit), the plan-level confinement cannot cope with the situation and it is at agent level that the error has to be handled, thus protecting the other agents from a general crash. In our case, this can mean for example that an error in a plan causing the agent to run out of memory would only impact that specific agent.

PLATFORM-LEVEL CONFINEMENT Confinement also needs to be ensured by the platform with respect to the environment where the software is executed, with the operating system being protected from the running components, making sure that a defect component cannot block the machine or cause it to crash.

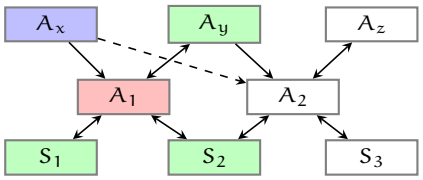


Figure 21: Multi-agent example, with the solid lines indicating current interactions and the dashed line the reconfiguration of agent A_x

CONCLUSION A strict modularity is therefore required, which, while constraining the programmer, provides the required elements for confining errors. This modularity also helps master system complexity. There is an issue of granularity to be considered and the "reasonable" level is not obvious to determine, in the trade-off between too few modules with diminished advantage for the fault tolerance, and too many modules whose overhead is too costly. For the safety net approach, the system must be comprised of a considerable number of agents, each represented using multiple goals and plans.

The solution we propose is to require designing using goal-directed multi-agent systems executed on a platform that ensures confinement between the agents as well as with respect to the the operating system. Upon the detection of an error, the platform also ensures the directly concerned module or modules (e.g. plans, goals, agents) stop immediately.

ILLUSTRATIVE EXAMPLE Let us consider the example of a robot subsystem that takes decisions based on data from various sensors. The subsystem is comprised of a camera, a sonar and computing device, including a Graphics Processing Unit (GPU) for image processing. Among the objectives of the subsystem, we focus on the objective to provide localisation information for the robot.

For confinement, the subsystem is structured in a hierarchy of agents, as illustrated in Fig. 21:

- the resources: a camera S_1, a sonar S_3 and the GPU-based image processing unit S_2.
- the functional agents: A_1 for localising the robot using the camera S_1 and processing unit S_2, and A_2 for localising using the sonar S_3 and the same processing unit S_2.
- the control agents: A_x, A_y and A_z for handling high level decisions and dispatching orders to the functional agents.

In Table 1 we present each agent with a single goal and its corresponding plan or plans. However, as discussed above, each agent could and should have more than one goal.

Figure 21 shows an arbitrary state of the multi-agent system with A_1 executing $P_{1.1}$ for A_x and A_y, using S_1 and S_2, when the plan crashes for an unknown reason. Owing to the chosen agent-based architecture and the robust implementation at platform-level, the crash is confined only to the respective plan and the rest of the agent, as well as the other agents, continue functioning. We will continue the discussion on this example after introducing the recovery measures.

Table 1: The goals and plans for each agent from Fig. 21

Agent	Goal	Plan	Plan description
A_x	G_x	$P_{x.1}$	Localise using the camera
		$P_{x.2}$	Localise using the sonar
A_y	G_y	$P_{y.1}$	Localise using the camera
		$P_{y.2}$	Localise using the sonar
A_z	G_z	$P_{z.1}$	Localise using the sonar
A_1	G_1	$P_{1.1}$	Use the camera and processing unit to localise
A_2	G_2	$P_{2.1}$	Use the sonar and processing unit to localise
S_1	G_{S1}	$P_{S1.1}$	Provide camera image (one plan instance per demand)
S_2	G_{S2}	$P_{S2.1}$	Process location information (one plan instance per demand)
S_3	G_{S3}	$P_{S3.1}$	Provide sonar data (one plan instance per demand)

As shall be seen in the next section, we take advantage of the modular structure defined in this section for the system recovery in multiple ways: interactions are monitored to help cut dependencies in case of error and the goals of the agents provide pro-active reconfiguration.

Also, both at multi-agent and agent level, as shall be seen in Sec. 3.3.1, the delimitation of these confinement components allows us to take into consideration the domino effect of errors causing successive components to fail. These errors can spread either from plan to plan (usually through shared beliefs) inside an agent or from agent to agent (through messages) at MAS level.

3.3 SYSTEM RECOVERY

PURPOSE The role of fault tolerance is to help overcome errors that occur during system execution, so once an error is detected and its propagation is limited, the system needs to recover, provided it has the necessary means. *Recovery* is the phase in which the system adapts to the error in an attempt to mask it and return a functioning that is normal or degraded but within specifications.

Going back to the ship metaphor, after water was detected and the concerned compartments were sealed, other measures need to be taken in order to ensure that the ship and its functions are preserved so that it can fulfil its mission, e.g. deliver cargo to its destination, limit the material losses by reaching another port or even abandon the ship (which would correspond to an orderly system shutdown). The event raises multiple questions. How serious is the situation? Can the ship continue its mission or an evacuation will be required? Can any of the compartments or their content be recovered? What functions of the ship are impacted by this situation? For example maybe the power supply is now endangered. Similarity, the software system that deals with an unforeseen fault needs to react properly and attempt to keep providing its services.

OVERVIEW In our ship scenario, the first reaction would be to inform all the functions that are concerned by the "quarantined" areas, so that they can react to the situation, if needed. For example if a part of the power supply of the ship is in

a flooded room, the engineer in charge of the power on board would need to be identified and informed. In a software system, this would correspond to identifying the agents or plans that depend on a failing one and sending them an error signal. This *dependency handling* step ensures a fast reaction once an error was detected and confined. The agents or plans that receive this signal would be able to react in order to limit the further deterioration due to the error, for example by stopping early a protocol that involved a failing agent. On the ship, the engineer could decide to stop all power in the affected areas in order to avoid any short circuit or fire. This is the *reparation* step. Once the system was brought to a safer state, the continuation of the service providing needs to be considered. The engineer would check for alternative power sources and depending on the available resources and the severity of the accident, provide full power restoration or only a fraction of the power for the critical components. This is the *reconfiguration* phase, where a software system would attempt to fulfil its objectives, in our case agent goals, despite the perturbation caused by the error.

As discussed above, the paradigm that we are using for our work is goal-driven agents. Therefore, the main software "components" that are concerned by these recovery steps are agent plans. In Chapter 4 we will see how other components are integrated into our approach due to the specificities of the chosen programming language, in particular the reasoning rules. In the same way, other applications of the approach may need to include different types of component or even dependencies.

3.3.1 *Dependency Handling*

THE PROBLEMS Errors occur during the execution of a program, which often implies components interacting between them. When a component encounters an unanticipated error, it is abruptly stopped by the confinement reaction, which, while possibly harmless, leaves nevertheless two possible problems for the components it interacted with. These components might have been "infected" through the propagation of corrupted data before the detection. Even if this were not the case, these components may still find themselves in the middle of a complex interaction that is no longer justified because of the error. In our case, this could be an agent plan sending a call for proposals and then failing while its peers prepare proposals, or a plan requesting an expensive computation and then crashing, in both cases leaving pointless and possibly costly consequences.

The problems that are aimed at by this step appear thus:

1. when the component (plan) concerned by the error detection propagated the error to other components (plans) prior to the detection;

2. when the component (plan) concerned by the error detection was involved in an interaction that is no longer justified.

APPROACH The concerned components need to be identified and informed in case of unanticipated error. Using this information, the components (plans) can take the necessary measures to adapt, for example by repairing, restarting or even avoiding the use of all these components any further. The role of *dependency handling* is thus (1) to trace the dependencies between the components of the system in order to be able (2) to trigger recovery measures (reparations and reconfigura-

tions, which are described in the next subsection) when an unanticipated error is detected.

WHAT IS TRACED A first question to ask here is which were the outputs of the concerned component, as either (1) they can constitute unfinished activities (e.g. incomplete data, partially executed actions from a plan, a request that does not need a reply any more), or (2) they can actually have propagated the error (e.g. incorrect data sent, actions that were performed for the wrong reasons). As these examples show, outputs (and inputs) can take different forms, from writing to (and reading from) the local memory (variables, beliefs etc.), to data exchanges (e.g. through messages) and even interactions with the physical world through sensors and actuators. We call *dependency* the directed relation between a component that is the source of an interaction and the component that is the target or recipient of that interaction. The second component is thus said to depend on the first. In case of error in a component of the system, the other components that depend on the first may find themselves acting or reasoning in an incorrect context. Using the multi-agent paradigm simplifies this step as agents communicate only using messages. In the next chapters we will see other dependencies that are traced once a specific language and platform are chosen. From the point of view of a component $\mathcal{C}$ where an error was detected, dependency handling is concerned with the outgoing dependencies, "downstream" in the flow of data, for which the outputs of $\mathcal{C}$ help identify the components which may find it useful to know that $\mathcal{C}$ was quarantined.

SOLUTION The other question for the dependency handling step is how to trigger reparation and reconfiguration actions in the affected components once they are identified. The solution we propose is to work under assumptions: for example when a message is received, it is used under the assumption that its source was working correctly. If later its source is discovered to have been compromised, this means that the use of the message is no longer justified, in which case measures may need to be taken in the receiving entity. Using a MBD[6]-inspired representation (see Sec. 2.4.3 in the State of the Art or directly [62] for more on the subject), we can create a *dependency context* for executing components. This rule-based context, similar to the RT context in ALMA, would contain for all concerned components and at any moment during the execution the dependencies that justify that component's continuation. If any of these dependencies is later proven to no longer be supported (for example because it encountered an unanticipated error), the components which based their execution on that dependency, in other words the components whose dependency contexts contain that dependency, are no longer justified. Depending in the component, this can mean simply stopping, retracting or placing it into reparation state, as discussed in the following recovery step in Sec. 3.3.2.

EXAMPLE In Fig. 22, the concerned components are agent plans. The context is formed using rules that build on the assumption that all the components are functioning normally (rule 10 creates an *ok(Pl)* assumption for each component Pl). The context is expanded through the incoming dependencies which in the example are the read beliefs and the received messages (rules 11 and 14). The successive context states are represented through the second parameter of the predicate. As we discuss below, these rules produce a graph of dependencies between all interacting

6 Model-Based Diagnosis.

$$\begin{aligned}
\text{COMPS} &= \{\text{Pl} | \text{Pl} \in \text{Plans}\} && (8)\\
\text{OBS} &= \{\text{test}(\text{Pl}) = \text{error}\} && (9)\\
\text{SD} &= \{\\
&\{\text{ok}(\text{Pl}) \Rightarrow \text{context}(\text{Pl}, 0)\} && (10)\\
&\cup\\
&\{\text{context}(\text{Pl}, n) \wedge \text{read}(\text{Bel}) \Rightarrow \text{context}(\text{Pl}, n+1)\} && (11)\\
&\cup\\
&\{\text{context}(\text{Pl}, n) \Rightarrow \text{write}(\text{Bel})\} && (12)\\
&\cup\\
&\{\text{write}(\text{Bel}) \Rightarrow \text{read}(\text{Bel})\} && (13)\\
&\cup\\
&\{\text{context}(\text{Pl}, n) \wedge \text{rcv}(\text{M}, \text{A}_{\text{from}}) \Rightarrow \text{context}(\text{Pl}, n+1)\} && (14)\\
&\cup\\
&\{\text{context}(\text{Pl}, n) \Rightarrow \text{send}(\text{M}, \text{A}_{\text{to}}, \text{Pl.Agent})\} && (15)\\
&\cup\\
&\{\text{ok}(\text{comm}(\text{A}_{\text{from}} - \text{A}_{\text{to}})) \wedge\\
&\quad \text{send}(\text{M}, \text{A}_{\text{to}}, \text{A}_{\text{from}}) \Rightarrow \text{rcv}(\text{M}, \text{A}_{\text{from}})\} && (16)\\
&\}
\end{aligned}$$

Figure 22: An example of MBD-inspired representation for tracing dependencies generated through beliefs and messages between agent plans

components. In its original form, the model would include a rule stating that given the system description with all components functioning correctly, the expected *test* value is okay:

$$context(Pl, n) \Rightarrow test(Pl, n) = okay \tag{17}$$

In case of error, a contradiction would be reached due to the inconsistency between the "observed" $test(Pl, n) = error$ and the expected value of the observation. The result would be a *nogood*, a list of components that cannot be all functioning correctly, a first step for diagnosing the source of the error. As we do not focus on the diagnosis, we use the previous rules in a different manner, by replacing rule 17 with rule 18 below which can be interpreted in the following manner: "if an error occurs in plan Pl, the assumption that the plan is justified cannot be true":

$$ok(Pl) \Rightarrow test(Pl) = okay \tag{18}$$

Note that the right side fact $test(Pl) = okay$ is a prediction rather than an observation, so it is not declared in the OBS set. Its role in this model is to produce a contradiction once the error is detected.

FUNCTIONS OF THE DEPENDENCY CONTEXT This model has two uses:

1. in case an unanticipated error is detected in the current component (plan in this example), the correct functioning assumption is disabled and all the component's outputs are invalidated;

2. in case an unanticipated error is detected in a component on which the current component depends (e.g. a plan that wrote a variable used by the current plan or sent a message used by the current plan), the current component's context will be invalidated and it will be forced to enter the reparation state discussed in the next subsection.

This is therefore a mechanism that can be used for maintaining a trace of system dependencies and triggering recovery when unanticipated errors are detected.

Depending on the chosen language and its characteristics, different components and interactions can be concerned by this tracing mechanism, as shall be seen in Chapter 4.

Our approach here is to try to rapidly isolate the error and "keep the boat afloat" rather than look for the guilty components, so we are not aiming to use this mechanism for diagnosis as well – more on this at the end of the chapter.

THE DEPENDENCY GRAPH The structure of dependencies can be seen as a directed graph $\mathcal{G} =< N, E >$ dynamically created during execution, with:

- $N = \{Components\}$ the nodes: in this case the plans, but they can also other components, depending on the chosen level of granularity and base model;

- $E = \{Dependencies\}$ the directed edges: in this case through belief accesses and message exchanges.

The "downstream" approach of the dependency handling step corresponds to propagating a signal to the nodes that are reachable in the graph from the node where the unanticipated error was detected.

SYSTEM-LEVEL VIEW This system-wide representation of the dependencies needs to be stored and handled. A solution would be to keep a unique service for monitoring all dependencies and triggering reparations in case of unanticipated error. Such a service would suffer from a high demand as storing all dependencies for all agents would be very communication intensive. Due to real-life constraints (such as system complexity, communication delays etc.), keeping this omniscient, i.e. complete and up to date, dependency graph is not feasible. More in line with the multi-agent approach would be to keep a local version of the dependency graph in each agent and use it in case of error detection. This would allow local reparations to be triggered more easily as communications would be minimal. The two aspects of dependency handling need to be discussed in the context of this distribution: the representation of the dependency graph and the propagation of the error signal.

AGENT PERSPECTIVE How much of the system should the graph stored in an agent cover? As the nodes we are considering, the plans, are at a finer grain than the agents, the agent would have to store at least the local dependency graph corresponding to its plan interactions. The question is then if it is desirable for agents to share their dependency graphs with other agents. To do so, a protocol would have to be implemented for exchanging dependency information either with each normal message, in the form of meta-data, or independently from the normal agent interactions. These exchanges could include (a) only the parts of the graph that directly concern the current node (i.e. all the nodes from which the current node can be reached and thus which could cause the current node to repair), or (b) all known dependencies of the involved nodes. These strategies would involve sharing all the internal inter-plan dependencies of each agent, which would be verbose and possibly unnecessary, as agents usually communicate with other agents and not with specific plans in those agents. Furthermore, the exchanges would become more and more difficult as the graph would expand over time. We therefore prefer a lighter alternative: (c) to have each agent manage only its own local graph but take into consideration the incoming and outgoing dependencies from messages.

SIGNAL PROPAGATION When an unanticipated error is detected, how is the signal propagated in the system given the agents only have a local view of the dependency graph? Locally, the rules allow triggering an automatic reaction in all the plans whose contexts are concerned by the error. However, when messages are involved, the peer agents need to be automatically informed through messages that allow them to react to the error too. A transparent connector needs to be integrated in the platform to ensure that an "inform" message is sent from the source agent and interpreted at the destination in case the error signal has to be propagated over a graph edge that passes to another agent. The fact that the "bridge" between agents is automatic means that the rules in Fig. 22 do not need to change.

Note that while the communication is normally performed towards and from an agent rather than a specific plan (agents usually do not have knowledge on the internal workings of other agents, a detail we took into consideration when writing the corresponding rules in Fig. 22), the mechanism described here suffice for propagating the error signal and triggering reparations on the paths of messages.

Once the propagation mechanism is in place, the question is how far should the error signal be propagated? The dependency context in the example above causes all the nodes that are reachable from the detection node in the dependency graph to

react by repairing and eventually reconfiguring in case of unanticipated error. This is quite a radical solution, especially since the dependency handling step is already done in a preventive state of mind: the seriousness of the error is not known, so it is even possible that the informed entities were not affected at all by the error. The current solution can thus result in a *domino effect*, a dreaded phenomenon in fault tolerance [88], causing many components to either fail in cascade or simply all recover (e.g. restart) following a single error. Therefore, policies for propagations and cuts in the graph need to be defined in order to ensure that the domino effect is limited, for example by restraining the propagation distance in the graph. The solution we propose is to give each node the choice of whether to propagate the signal further or not. In other words, the signal is only sent to the components that are situated at a distance of 1 from the detection node in the dependency graph, leaving it to them to decide whether to repair normally or to trigger a new unanticipated error that would propagate the error signal further. The reason is that in this way, the programmer can handle locally the decision which depends on the actions that the plan already executed and may not require the other plans to be informed as well.

As plans can finish and agents can be stopped or cut away from the system, how does this affect the dependency handling step? If a plan is no longer running when the signal is propagated to the node corresponding to it, then we stop the propagation on that graph path. More refined policies can be considered here, for example propagating to the first running plan on each path, but we limit our current approach to this "at most one step" propagation which implies less communication costs and limits the risk of a domino effect.

What happens is a plan is reached by more than one unjustification? This does not pose problems regardless of the chosen propagation policy, as only executing plans are concerned by the unjustification signal and they can only react once to it, so any other attempts are ignored.

Can there be cycles in the graph and if yes, would that be a problem? Yes, as even a simple request-reply exchange creates a cycle in the graph. This does not pose any problem as the plan would not be executing because of the error so no other unjustifications can be created.

CONCLUDING ON THE PROPAGATION So if any of the plan's inputs is unjustified, only the current plan's dependency context is unjustified and it is inside the plan that it is decided whether the unjustification is propagated further to its dependants. The propagation to the dependants is done through the contradiction of the "ok(X)" assumption and is allowed in the following situations:

1. the plan handled an unjustification and the programmer added an "Unexpected" in the *unjustified* branch to propagate the reparation,
2. the plan encountered an unexpected error.

In order to ensure the level by level propagation, we need to change the justifications of the rules in the example above so that instead of the current context, outputs are justified only by the `ok(Pl)` assumption resulting in the SD set in Fig. 23.

FULLY AUTOMATIC In line with our objective of keeping the programmer's involvement minimal, but also to protect the mechanisms from any interference, all

$$
\begin{aligned}
SD = \{ & \\
& \{ok(Pl) \Rightarrow context(Pl, 0)\} && (19)\\
& \cup \\
& \{context(Pl, n) \wedge read(Bel) \Rightarrow context(Pl, n+1)\} && (20)\\
& \cup \\
& \{ok(Pl) \Rightarrow write(Bel)\} && (21)\\
& \cup \\
& \{write(Bel) \Rightarrow read(Bel)\} && (22)\\
& \cup \\
& \{context(Pl, n) \wedge rcv(M, A_{from}) \Rightarrow context(Pl, n+1)\} && (23)\\
& \cup \\
& \{ok(Pl) \Rightarrow send(M, A_{to}, Pl.Agent)\} && (24)\\
& \cup \\
& \{ok(comm(A_{from} - A_{to})) \wedge \\
& \quad send(M, A_{to}, A_{from}) \Rightarrow rcv(M, A_{from})\} && (25)\\
& \cup \\
& \{ok(Pl) \Rightarrow test(Pl) = okay\} && (26)\\
& \}
\end{aligned}
$$

Figure 23: An updated version of the SD in example in Fig. 22 after taking into consideration propagation issues

the techniques proposed for the dependency handling level must be transparent to the software developer. They are to be provided by the platform in the form of specific code included in each module, but separate from the working memory available to the programmer. This is feasible as assumptions and dependency rules can be associated with specific actions in the language (for example sending a message, as in the example above). The reaction to an exception, as well as generating a signal locally or an inform message between agents and reacting to them are also easy to automate.

The separation from agent memory and the automation also mean that the dependency handling mechanism should function even in the presence of agent-level unrecoverable errors, as described in the detection section.

DEPENDENCY HANDLING VS. AUTONOMY FOR ROBUSTNESS What are the advantages of such a mechanism when we already require the programmers to take into consideration the possibility that their peers are autonomous, e.g. they can decide to stop responding? Can taking into account the agent autonomy suffice?

We propose this recovery step because on the one hand it offers gains in:

- speed – the peers are notified as soon as the error is detected, not after a timeout;
- resource consumption – interactions are cancelled, avoiding possibly expensive computations or other interactions;
- robustness – a retraction message is emitted after a possibly erroneous information was sent, information which may otherwise continue to be used obliviously of the detected error, in case no other detection mechanisms are triggered at the receiver.

On the other hand, in case an agent is completely cut away from the others or encounters an error that is so serious that it also impedes the dependency handling mechanism to function, agent autonomy can help improve the system reaction, for example by avoiding blocking situations when lacking a reply.

So while there are certain situations where the two measures overlap, there are also many advantages in using them both.

CONCLUSION Up to this point, we saw how an error can be detected, the architecture allows us to confine it and then dependencies point to the other components that may need to adapt to the event. Let us now see how the dependencies gathered by the dependency handling mechanisms work towards the recovery of the functions of the system.

3.3.2 *Reparation*

REACTION TO DEPENDENCY HANDLING Once an error is detected and a signal is sent to all concerned entities, their execution is no longer justified and will need to stop. Because executing components are involved, there is a risk of system-wide inconsistencies caused by data accesses (e.g. write partial values or lock a database for writing), or actions (e.g. a call for potentially expensive computations or actuator commands that are no longer needed) that were only partially executed. Contrary to the case of the components where the error was detected and that

needed to be abruptly stopped, these other components are still executing normally and should react differently by stopping in an orderly manner, before attempting to reconfigure and continue providing the right functionalities. Indeed, they may attempt to return the system state to a stable state, for example by performing certain memory operations (e.g. delete partial values) and actions (e.g. cancel a call). Given that interactions with the environment may have been performed, an automatic rollback procedure involving the internal system state may be inappropriate. The authors of recovery blocks (presented in Sec. 2.1.5) too note that there may not always be appropriate nor possible to automatically retract the outputs or undo the changes performed by a code segment.

Returning the system to a consistent state is thus something that we need to introduce for our safety net approach. Hence, we propose requiring the system designer (programmer) to provide the necessary reparation steps in specific locations in plans. These would be used in case the current plan needs to be stopped due to a dependency from a component that encountered an unexpected error. Furthermore, these will be the places where it can be specified if the error signal needs to be propagated further or not.

SOLUTION The idea is to include in the development process the definition of reparation procedures that can be triggered by the notification signals from the dependency handling step. A means to require specific programmer-provided reparations is to be included in the language.For this we draw on work on ALMA (described in Sec. 2.4.4) where programmers are demanded, in specific points in the code, to provide specific measures to be taken in case the execution of the concerned code segment is no longer justified. The mechanism is well adapted for unforeseen (as well as foreseen) faults because of its generic approach: the programmer does not have access to the reason that triggered the reparation – he or she only knows that the execution needs to be stopped. Also, the programmer's task is much less tedious if there is only a generic case to consider, rather than a multitude of specific situations.

In case no repair measures are needed, or even to avoid using the mechanism for whatever reason, the programmer could leave it empty to let the reconfiguration apply directly. Another important case is when the programmer decides to use the repair reaction to signal an unanticipated error, leaving the safety net take charge and thus propagating the error signal one step farther. Imagine a lord participating in an event with multiple auctions. In case after a few bids his participation is no longer justified (e.g. he receives a message that his accounts were temporarily blocked), he could:

- simply leave without saying anything, if none of his bids ended in a purchase. This corresponds to the case where no "reparation" is needed.
- go to see the sellers to personally cancel any winning bids and present his excuses. This corresponds to the case where the "reparation" contains specific measures.
- ask his assistant to announce all the auctions where the lord participated that one of their bidders was actually out of money and would not be able to pay. This corresponds to the propagation of the "error signal", which, in this case would cause "much ado about nothing".

A programmer would certainly face the same choice: is there need for a reparation if the execution is forced to stop at this point in the plan? Can a specific reparation be provided? The propagation of the error signal should be the last choice and be avoided, as its overuse could generate the dreaded aforementioned domino effect.

The granularity issue discussed in the confinement section above comes again into play, but with a twist: plans with many interactions would have more complicated reparation code, but longer ones with little to no interactions (e.g. request and display information) may require only limited reparations.

CONCLUSION The reparation step transparently reacts to the signals issued by the dependency handling step from other agents or plans and executes reparation steps in order to bring the system to a safe state before the reconfiguration phase. The reparation code can be provided by the programmer in specific locations in the code.

3.3.3 *Reconfiguration*

PURPOSE With fault tolerance being the objective of our work, we need ensure the system continues performing as specified despite the manifestation of faults. As we are concerned with unforeseen faults, the focus of our solution cannot be in the cause of the problem, but rather on the objectives of the system. While the reparation phase was charged with returning the system to a consistent state, reconfiguration is the process through which the system adapts to ensure its best functioning in order to compensate for errors. Depending on the means put in place and the seriousness of the situation, components (agents, plans etc.) may need to be eliminated or restarted in order to "clean" the effects of the error. However, what is important for the system is to avoid an erratic behaviour by staying within the specifications, in the worst case by performing an orderly shutdown and in the best case by continuing to provide the service it is meant to provide.

FROM FAULT TOLERANCE TO GOAL-DRIVEN AGENTS The recovery blocks fault tolerance approach described in Sec. 2.1.5 requires the programs to be divided into blocks, each with alternative solutions and governed by an acceptance test. Failed block executions can be followed by automatic or programmer specified measures aimed at returning the system to its original state, ready for further block executions. The resulting redundancy of design and the focus on the results constitute valuable properties for the tolerance of unforeseen faults. The definition of agents using goals and plans compares favourably to the recovery block approach: plans correspond to execution blocks and goals with their success conditions correspond to the acceptance tests.

THE GOAL LIFE-CYCLE AND POSSIBLE OUTCOMES Given the expressiveness that can be associated with goals through satisfaction conditions and life-cycles, they offer the possibility to define more refined behaviours than in the recovery blocks approach. An example of a goal life-cycle for which an automaton is used is depicted in Fig. 24. Goals have two possible states: when adopted by the goal plan, they become *desires*, but as long as they are not *intentions*, no plan is searched or executed. The state change can be controlled by various constraints, for example with respect to limited resources. A series of beliefs are used for state changes,

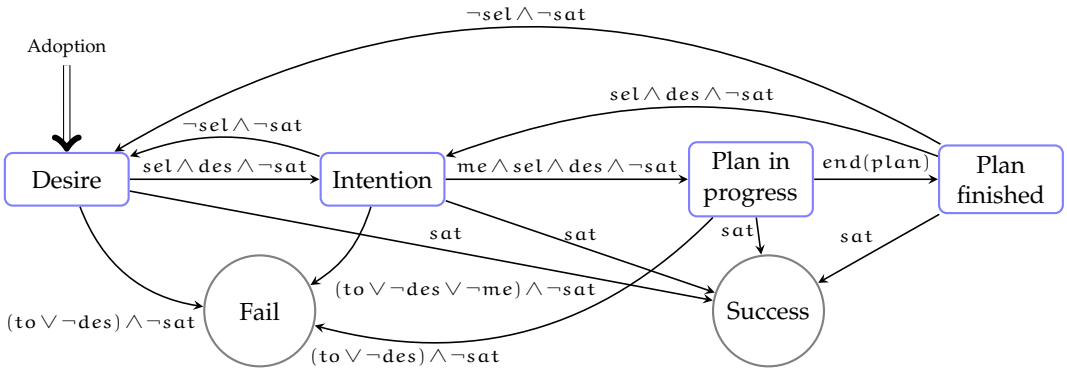


Figure 24: Our generic goal life-cycle with transition conditions on state beliefs (des = desirable, sel = selected, sat = satisfied, me = means, to = timeout)

for example sel (*selected*) indicates the passage in an active Intention state and sat (*satisfaction*) indicates that the goal was achieved. The advantage of goals is that as long as they are not achieved and the situation is still favourable (e.g. resources are still available, the goal timeout has not elapsed), plans can be attempted again and again.

Note that we consider that a goal outcome can only be "Success" or "Fail".

GOAL-DRIVEN RECONFIGURATION The objective-based error detection and confinement properties of the goal-plan paradigm were presented in the previous sections, but here we are interested in the properties it offers for the reconfiguration. When written in a goal-pursuing manner, an agent will adapt its behaviour to retry plans or execute new ones whenever its goals are not achieved. This can happen regardless of the presence of errors during the execution of its plans.

Goal-driven reconfiguration is the process through which an agent reconsiders its behaviour with regard to a concerned goal, resulting in (1) re-attempting the same or another plan, or (2) renouncing at the goal and eventually continuing with the behaviour, if any, corresponding to the failed goal. In both cases, the implications can spread to other components. For example in the first case the agent can re-attempt a plan that failed to obtain a valid radar image but this time uses a different radar and succeeds. Or it uses a completely different plan that uses a different type of sensor to achieve the goal. In the second case, if the goal was adopted in the context of a cooperation, the agent may need to inform its peer or peers that it cannot achieve that specific goal. The idea here is that even if a goal fails, this is a case that can and should be taken into consideration by the programmer as normal program execution and thus the system will contain the suitable measures to take, without this being considered a fault tolerance-related measure.

Therefore, whatever happens, as long as the level at which goals are handled stays functional[7], goals will be retried and pursued as indicated by the agent definition. As already stated in the section on detection, this means that unanticipated errors can be detected and masked by simply the fact that a plan did not execute correctly [26] or more generally unforeseen faults can be successfully tolerated when attempt to achieve a goal failed.

7 As stated before, there can be critical errors when the whole agent needs to be stopped.

REDUNDANCY Furthermore, as discussed above in the section on confinement, the possibility to define multi-level structures as well as the ease of introducing redundancy in the definition of goal-driven agents recommend their use for fault tolerance in general and the tolerance of unforeseen faults in particular. While the simplest way to view redundancy is through duplicating components – e.g. the two inertial systems in the Ariane 501 rocket [73] – or software agents [46], redundancy can also be achieved in a functional manner: providing different means to reach the same goal – e.g. a Mars rover can acquire images using a radar or an optical imaging system (camera). Goal-driven agents can provide specific redundancy in the form of (a) repetition of a plan execution, commitment strategies (weak redundancy, effective for transient errors), (b) plan libraries with multiple plans (medium) and (c) planners that can adapt to the current context to provide well suited solutions (medium-strong redundancy).

PLANS AS A RESOURCE Plans are thus a precious and often limited resource for agents (not many plans are usually provided despite the point above). As certain validations can be performed on their code, we will consider that when an error occurs in a plan, it is not the plan code that is to blame, but rather the plan instance with all its interactions (messages, beliefs), so plans will not be discarded, only stopped. A consequence to this is that temporary errors can be survived (e.g. the code does not check if a sensor is available, so if the sensor is temporarily out, not eliminating the plan upon its crash can allow it to be successful upon a later try). A learning strategy could then be used to detect cases when plan instances of the same prototype (i.e. plan code) often cause problems in order to eliminate that prototype.

RECONFIGURATION CASES In Fig. 25 can be seen the three cases when a component (an agent) reconfigures:

1. when an unanticipated error (exception) is detected, the component is stopped and while the dependencies are handled ("pruned"), reconfiguration is launched;
2. when a component is notified that one of its inputs was produced by a component that was stopped due to an error, the component goes through a reparation phase before reconfiguring;
3. when a component's goal is not achieved following a plan execution, regardless of the reason, the component reconfigures.

ARE TIMEOUTS AND KNOWN ERRORS CONCERNED BY THE RECONFIGURATION? A timeout inside a plan being a specific error case – despite the fact that it covers many unforeseen faults –, its treatment raises the question: what should be the reaction to this event? A first reaction could be to simply generate an unanticipated error and let the safety net mechanism handle the situation from that point. However, as the programmer is supposed to provide this value as a normal event among other, the treatment of this event is more appropriate if provided as a normal continuation of the execution. In the most extreme case, the programmer could decide to throw an unanticipated error (an error that he or she does not intend to catch) corresponding to the timeout, possibly after executing some code with reparation purposes. Even if an unanticipated error is not thrown, the corresponding agent goal could still fail, thus triggering a reconfiguration.

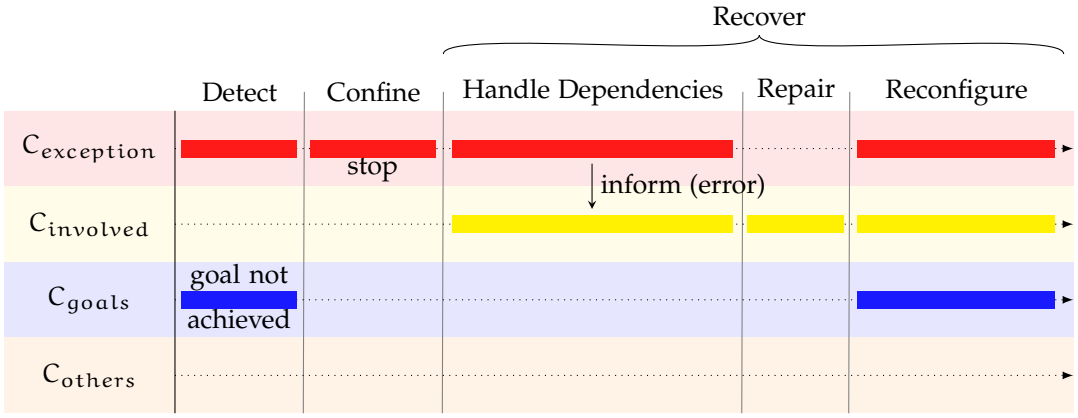


Figure 25: Involvement of components by phase of fault tolerance, with $C_{exception}$: the component where the uncaught error was detected, $C_{involved}$: components that were depending on the first one, C_{goals}: components whose detection occurred through the goal mechanism and C_{others}: other components that are not concerned, but may be eventually required to participate after the reconfiguration of the others, e.g. to compensate for another component that is no longer available.

Furthermore, this treatment of known errors can be employed for other types of error that the programmer can easily detect but does not want to handle – for example due to lack of time. The programmer can thus provide the specific detection mechanism and then leave the handling to the safety net mechanism.

GOALS AND AUTONOMY Goals are central for the autonomy of a system. On one side, the agent contains its own purpose, in the form of goals, which it will pursue until its achievement or until another condition is reached (e.g. it decides that the goal is impossible to reach), in spite of any hardships it may encounter. On the other side, goals are the source of the pro-active behaviour of agents, as they initiate actions and interactions in order to fulfil the agent's role.

An interesting direction of study for reconfiguration is also the idea of involving humans in the process, as long as the level of autonomy of the software system remains high, as for example in [34] where the human operator is given a certain time window to modify the action proposed by the system.

CONCLUSION To sum up the recovery phase, a plan can either encounter an error and be forcefully stopped, or be stopped through a dependency in which case a reparation procedure corresponding to that plan can be applied. In both cases, the goal model requires that the current context is evaluated and other plans are executed, or the control is given to a higher level with the goal failing. There is also the case when reconfiguration is triggered by a goal's success condition following the execution of a plan. Either way, the execution is kept controlled and within the limits imposed by the programmer, regardless of the final outcome – a working system or a graceful degradation. Note that it is thanks to the confinement that the errors are limited to certain components and we can separate between the four types of component in the figure (i.e. there can be components where the error was detected, "involved" components, but also "other" components that are not concerned by the event).

ILLUSTRATIVE EXAMPLE (CONTINUED) Let us go back to the robot subsystem example introduced in Sec. 3.2. Following the advice for the confinement phase, the subsystem was already defined using goals and plans. For the purpose of this example, we assume that the plans $P_{1.1}$ and $P_{2.1}$ function over longer periods of time (thus cumulating dependencies), as opposed to the plans of the S_i agents that are short and are instantiated for each demand from the A_1 and A_2 agents.

We use the same arbitrary state from Fig. 21, with A_1 executing $P_{1.1}$ for A_x and A_y, using S_1 and S_2. Let us consider the moment after the confinement of the crash of $P_{1.1}$. If no dependency handling mechanism is in place, as A_x was programmed using the autonomous agent philosophy, it will not wait indefinitely for an answer and will eventually reconfigure to its second plan $P_{x.2}$. If, however, the complete safety net approach was used, the dependency handling mechanisms would need to send error notifications to the concerned agents. The traced dependencies for plan $P_{1.1}$ indicate that:

- it had inputs (messages) from S_1 and S_2 (the data), and from A_x (a command);
- the outgoing dependencies go towards S_1 and S_2 (requests for data), and to A_y from a previous call processed by the same plan that crashed.

According to the dependency handling policy, all the downstream dependencies of $P_{1.1}$ are notified:

- for S_1 and S_2, as they only had plan instances especially created for the request, no reparation or reconfiguration measures are performed.
- agent A_x is able to reconfigure sooner than when relying only on the timeout, while also applying a reparation procedure, if necessary.
- in the case of A_y, the concerned plan stops, the plan's reparation is applied – e.g. written values are deleted – and the agent reconfigures by attempting a new plan in order to achieve its goal. We can suppose that the reparation procedure of A_y did not require any propagation of the reparation to A_2, for example because the programmer estimated that the message sent to A_2 did not require that kind of treatment. If the only alternative for A_y to continue passes through A_1 again, it will call on that agent and if the error was just a transitory one, it may succeed.

This example illustrated how the safety net approach helped the system to successfully tolerate an unknown (and unforeseen) fault.

THE SAFETY NET In conclusion, as depicted in Fig. 25, the expected course of events in case an unanticipated error is detected is as follows. First of all there need to be implicit means of detection, which can be either based on exceptions or based on objectives (the satisfaction of goals, timeout conditions etc.). Then, given a modular architecture, the impacted elements can be isolated. To achieve this modularity we propose the use of a multi-agent architecture with agents defined using goals and plans. The dependencies between these modules are transparently traced and in case of error, the modules that depend on the primarily affected one can be informed. While the module where the error was detected is directly stopped, the informed modules can automatically repair, given they are endowed with the required procedures. In the end, the whole system, both the stopped and repaired

modules, can reconfigure with the purpose of keeping the system functioning correctly, or experience a graceful degradation. For the reconfiguration part, we propose the use of the goal paradigm, already cited for the confinement, while for the reparation part, specific procedures are to be provided for each plan.

3.4 THE PROGRAMMER'S GUIDE FOR A SAFETY NET

Now that we looked at the unforeseen fault issue from the perspective of the fault tolerance, we can return to the developer's point of view. In the following, we state the 10 principles of the safety net approach, comprising of a safety net-savvy language and an adequate execution platform to be used with specific design requirements.

3.4.1 *Language Requirements*

1. The language is **based on the goal-directed agent paradigm**.
2. The language provides an **exception-based error signalling** system.
3. The language requires the programmers to **regularly specify reparation procedures** that are to be used when a plan loses its justification and needs to be stopped.
4. The language requires a **timeout for every state which implies an agent waiting** for an event.

3.4.2 *Platform Requirements*

5. The execution platform ensures **multi-level confinement** (confinement from the machine and operating system to avoid propagation of errors to them) as well as **horizontal confinement** (between agents).
6. The platform **catches all unanticipated (uncaught) errors/exceptions** which are to be handled by the safety net mechanisms.
7. The platform **performs transparent dependency tracking** that is then leveraged for triggering system-wide reparations in case of unanticipated errors.

3.4.3 *Design Requirements*

8. The programmer uses a **multi-agent architecture** featuring a **significant number of agents** with respect to the application.
9. The programmer uses **goal-driven agents** whose behaviour is split into **multiple goals and plans**.
10. The programmer takes into consideration **redundancy**: allowing goals to retry plans, providing alternate plans or agent designs etc.

3.5 DISCUSSION

In this chapter we presented our solution for the tolerance of unforeseen faults by dividing it into three phases: detection, confinement and recovery. We showed how goal-driven agents play a central role in the approach through their involvement in all three phases. Other detection techniques were presented which, together with the subsequent safety net handling, aim to ensure an increased fault coverage. This is all done while the focus of the programmer is on specifying the normal system behaviour and following the language constraints (e.g. reparation code in specific locations or specifying the maximum waiting time, i.e. timeout, every time the program waits for an event). An automatic dependency tracing mechanism which takes advantage of the goal-plan structure completes the safety net.

The solutions considered for confinement and recovery rely on the granularity used by the programmer when designing the application. If the component that needs to be isolated is very large with respect to the overall system or it has been running for a very long time and has produced a lot of outgoing dependencies, then the safety net approach will not be able to significantly contribute to the fault tolerance of the system. On the other hand, shorter execution segments with less dependencies and smaller components mean that an error would be easier to confine and the recovery would impact a more limited part of the system. The problem with granularity is that it is relative, it depends on each application and is difficult to quantify, but for the safety net approach, it is important to have more rather than less agents, each with more rather than less goals and plans.

Since they are concerned with the plan (and thus agent) outputs, the dependency handling and the reparation steps constitute the means to backtrack the plan effects in an attempt to bring the system back to a safe state, which ideally would be identical to the state before the erroneous execution. There are, however, situations where this is not possible or not covered by our approach. Besides from issues related to badly specified reparation steps, a few other examples include:

- plans that end without signalling an unanticipated error, but whose goals are not achieved and require the application of a new plan, possibly failing;
- actions that cannot be undone (e.g. permanently deleting a piece of information);
- as we discussed when defining the dependency handling propagation strategy, there are situations where the receivers of outputs originating in the "incriminated" plan (e.g. plans using messages sent by the "incriminated" plan) are no longer active, in which case their effects are accepted as they are.

The study of these limitations and how much farther the backtracking of plan outputs can be extended is a subject for future work.

ILLUSTRATIVE EXAMPLE (DISCUSSION) In the robot sub-system example used for illustrating the confinement and recovery phases, one may wonder why the original plan failed. Was it because it was fed corrupted data from the sensors? Even if it were the case, the safe way of programming the plan would have been to test that data and not let the plan crash. Or maybe there had been corrupted data for a while and the crash was just a consequence of their accumulation. If this was the case, than the fact that another agent was transmitted data from that same plan (i.e. A_y) could have propagated that incorrect information, in which case

Table 2: Measures by phase of our safety net approach, split between the offline measures concerning the programmer and the online ones ensured by the platform

	Detection	**Confinement**	**Recovery**		
			Dependency handling	Reparation	Reconfiguration
Offline	Code as required by language	Code with agents, goals and plans	-	Write reparation code as required by the language	Write goals and plans; provide redundancy
Online	Catch unanticipated (uncaught) errors	Protect platform and other components; stop component upon detection	Trace dependencies; inform concerned components upon detection	Receive "inform" signal and react	Handle the goal life-cycle

informing the other agent of the error might have avoided compromising more of the system. Either way, the unforeseen fault was well masked and its exact cause is less important for our work[8]. Did the system suffer from a domino effect? On one hand, there are plans that already finished and which did not produce any propagation of the recovery (e.g. S_2 did not repair and reconfigure needlessly, nor did it propagate the error to A_2 "one goal and plan per request" policy allowed for a good confinement of the possible error). On the other hand, there is the use of specific reparation procedures that provide more insight on the current situation and can permit avoiding a propagation, like in the case of A_y that did not need to inform A_2 of the error.

In conclusion, the safety net approach reacted well in this simple unforeseen fault example, owing to the fact that the required tools were put in place and the system was programmed following our requirements.

PROGRAMMER'S PERSPECTIVE The takeaway from this chapter are the 10 principles of the safety net approach presented in Sec. 3.4. These are divided between language, platform and design requirements. Table 2 takes the programmer's perspective from a slightly different angle: contrasting the development, i.e. "offline", and runtime, "online", aspects of our approach. Note how the "online" behaviour of the system is partly made possible by the programmer-issued design following the language constraints (goals, reparation code), and partly ensured by the platform mechanisms – particularly noticeable in the case of the dependency handling step.

DOES THE SAFETY NET APPROACH IMPACT AGENT AUTONOMY? Platon et al. [82] (discussed in Sec. 2.2.1) argue that exception handling mechanisms need to respect the agent paradigm, in the sense that they need to avoid being intrusive and respect the agent autonomy. The idea of the safety net approach is that agent goals guide the recovery process, thus giving priority to agent autonomy. The dependency handling mechanisms, nevertheless, are made to be transparent to the programmer and the agent reasoning and trigger reparations in the concerned plans regardless of the agent willingness to react. It is only after the reparation that the agent autonomy takes control through the agent goals. We consider that this type

8 As in the recovery blocks work, measures may be taken for logging such events for later diagnosis, but this falls outside the scope of this thesis.

	Conflict set:
f crashes	[a,b,c,d,f]
e crashes	[a,b,e]
Diagnosis:	[a,b]

	Conflict set:
f crashes	[a,b,c,d,e,f,g,h] (all)
e crashes	[a,b,c,d,e,f,g,h] (all)
Diganosis:	[a,b,c,d,e,f,g,h] (all)

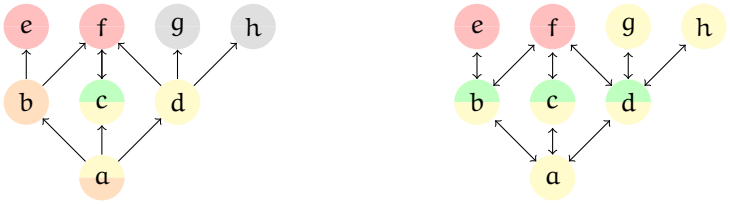


Figure 26: Dependencies propagating between components. The bottom a component sends a request to the upper layer entities via a middle layer. In red components detecting an error, in green outgoing dependencies, in yellow suspected dependencies and in orange components incriminated by a diagnosis strategy that takes advantage of multiple detections to reduce the conflict set.

of reaction is required as we are in the presence of unforeseen faults, which means that we aim to keep the programmer's involvement minimal.

AS MBD IS ALREADY USED, WHY NOT ADD A DIAGNOSIS STEP? Besides the "downstream" perspective described for the dependency handling step (Sec. 3.3.1), given that we are considering unforeseen faults, there is a risk that they manifested well before the detection point. There may be components that contributed to the error propagation and may still do so even after the detection and confinement and recovery measures. There is therefore the possibility that the error originated elsewhere in the system, in which case a diagnosis would help identify other components that were affected before the detection. From the point of view of a software component (agent, plan) where an error was detected, diagnosis would be concerned with the incoming dependencies, "upstream" in the flow of data, in order to attempt to identify the possible source or sources of the error, also pointing to a string of components that were affected from the identified source to the point of detection. This information would then help clean up, repair and reconfigure the system more thoroughly.

Furthermore, we already conceptually use MBD tools, which in theory would facilitate the localisation of errors or even faults. In the example in Fig. 22, MBD logics dictate that at least one of the assumptions used to reach that point is false. These are all the assumptions "gained" through the inputs, plus the assumption linked to the current plan which may itself be the cause of the error. Being a tool for diagnosis, this model helps indicate all the possible sources of the error, which often results in too many "suspects". Furthermore, a diagnosis in our case would also face problems linked to the distribution of the application and the questions related to the reliability of the communication links, represented in the model through the $\mathrm{ok}(\mathtt{comm}(..))$ assumption.

If we consider the example in Fig. 21, we see that the incoming dependencies would not be useful for diagnosing the source of the error (which at this point could be any of the inputs plus the actual plan that crashed) as too many interactions already took place and the actual culprit is no longer distinguishable.

To try to overcome this issue of ambiguity between "suspect" components, we could try to look for more symptoms of the error, which in the case of unforeseen faults would be other error detections. Comparing the sets of suspects for multiple errors would help narrow down the list of suspects and limit the number of components that can be incriminated. To illustrate this issue related to diagnosis, let us consider a scenario where an entity requests a service from a set of peers which in turn use other entities to fulfil the requests (Fig. 26) and some of these communicating components experience unanticipated errors during different moments of their functioning. At the left, once the middle layer entities translate the request, first e and then f crash. Using a diagnosis strategy that considers all the components that participated at the creation of a dependency, we can compare the two initial lists of "suspects" (in yellow) and narrow the possible "incriminated" components (in orange) to a and b. At the right the errors are detected later in the execution, and the diagnosis does not produce any useful conclusion. Note that this example contained two detections in order to help narrow down the list of suspects. Therefore, even this simple example shows the difficulties that the diagnosis would bring to our problem, as the complexity incurred by the diagnosis problem is very important. Furthermore, the detection of unanticipated error shows that there was a vulnerability in the concerned entity, so a first fault is in that instance itself, which was already isolated as part of the confinement phase. However, the fact that the error propagated from somewhere else is a matter of supposition and often requires more than one detection event for filtering the list of suspects. On the other hand, for the outgoing dependencies there is less ambiguity, as the component where the detection took place was certainly affected by the error and is entitled to announce at least its direct dependencies (in green in the example above).

Having a ship that is repaired to a perfect shape is important, but the purpose is to have it perform its mission, and only afterwards repair it, for example in the safety of a port (which would be the equivalent of a maintenance job on the software). Spending time looking for a crack in the hull instead of sealing off the concerned section of the ship may actually endanger the entire vessel, so the speed of the reaction is important. So while diagnosis can be an aspect of fault tolerance, our focus is not on finding the exact source or cause of the error upstream, but on limiting its impact and propagation downstream and ensuring a correct continuation of the functioning of the system.

In this chapter we introduced the main building blocks of the safety net approach. We continue in the next chapter with a proposal of an instantiation of the safety net approach starting from a programming language, together with technical solutions for the platform to use.

4

AN INSTANTIATION OF THE SAFETY NET

The main objective of this thesis is to provide a development framework to assist programmers in building software that is tolerant to unforeseen faults. In the previous chapter we studied the issue of unforeseen faults along the lines of three fault tolerance phases: detection, confinement and recovery. This allowed us to propose 10 principles covering design, programming language and platform requirements.

In this chapter, we propose a language and platform which provide required characteristics to be used for the safety net approach. For this, we aim to cover the corresponding safety net principles (1-4 for the language and 5-7 for the platform):

1. The language is **based on the goal-directed agent paradigm**.
2. The language provides an **exception-based error signalling** system.
3. The language requires the programmers to **regularly specify reparation procedures** that are to be used when a plan loses its justification and needs to be stopped.
4. The language requires a **timeout for every state which implies an agent waiting** for an event.
5. The execution platform ensures **multi-level confinement** (confinement from the machine and operating system to avoid propagation of errors to them) as well as **horizontal confinement** (between agents).
6. The platform **catches all unanticipated (uncaught) errors/exceptions** which are to be handled by the safety net mechanisms.
7. The platform **performs transparent dependency tracking** that is then leveraged for triggering system-wide reparations in case of unanticipated errors.

The language we propose is an extension of ALMA (described in Sec. 2.4). We start from the original language which we discuss with respect to fault tolerance and acceptability by the programmers. We then revisit the three phases of fault tolerance described in the previous chapter: detection, confinement and recovery. These will allow us to define the required modifications for the language and platform in order to comply with the safety net principles.

4.1 THE BASE LANGUAGE

As we conclude in Sec. 2.4, ALMA is an agent programming language designed for working under uncertainty and that comprises elements that allow agents to be fault tolerant. In this section, we discuss the main characteristics of the language and their interest for our work: agents only acting through messages, the specific language structure and the use of rules.

ACTING ONLY THROUGH MESSAGES An agent is defined by a series of sense-reason-act cycles. The actual agent behaviour and intelligence are in its reasoning part, while the sense and act steps can involve hardware components as well as interactions with other elements of the system (usually agents). For the perspective of fault tolerance, it is useful to encapsulate the external hardware components into "artefacts" and therefore limit the sense and act steps to message exchanges at agent level, a characteristic of ALMA. In this way, the agent encapsulates the intelligence and is loosely coupled with the actual hardware devices. These devices can be built with their specific fault tolerance characteristics (at hardware and software levels), with the agent reasoning providing a supplementary level of protection, a safety net.

GRAPH STRUCTURE AND ISOLATED COMPUTATIONS In ALMA, an agent definition is written on two different levels. On the one hand, there is the Directed Acyclic Graph (DAG) structure that provides a scaffolding for the agent definition, containing the most important high level elements defining the agent behaviour: actions, perceptions, reasoning and decisions. On the other hand, there are the computations which are executed in a functional programming manner, i.e. without producing side effects, only returning values. The DAG creates a code that is readable, clear and simple enough to transmit the important details, but sufficiently expressive to allow for the definition of complex agents. A clear and simple code structure makes the programmer's task easier, while in the same time limiting the risk of introducing faults and making such cases more easily detectable during the code review process. The fact that the computation sections can be written in various languages (e.g. Prolog, Java, C etc.) and can contain complicated and error-prone tasks, makes them more prone to introducing faults in the system then the rest of the agent definition. However, executing the computations in a functional programming manner facilitates catching and confining any errors that may appear in these code sections.

With the graph structure sufficiently well written by the programmer and the code sections well confined by the platform, the resulting agent definition should have less faults, while any code errors will be easier to handle. Furthermore, the graph structure of ALMA contains specific branches that are beneficial for the fault tolerance and two of which we will actively use in our error handling mechanisms: the *unjustified* and *timeout*.

THE USE OF REASONING RULES The use of rules for reasoning and belief updating in ALMA is a characteristic that clearly sets it apart from other agent programming languages. A rule in ALMA has the form $premises \Rightarrow belief\_conjunction$, where $premises$ is a conjunction of beliefs and functional code sections that return a boolean value. Code sections can be used to test the values of the beliefs in the left-hand side, but variables can also be used for computing values that can be possibly used for the $conclusions$, e.g. $belief(mass, M) \wedge (WeightForce = M \times 9.81) \Rightarrow belief(weight, WeightForce)$.

Rather than just being simple if-then code constructs, rules are part of the agent reasoning and once added, they are applied every time one of their premises is updated. In conjunction with a Truth Maintenance System – the ATMS in our case – they allow forward chaining as well as the enabling and disabling[1] beliefs. It is this

1 As seen in Sec. 2.4.2, this is a characteristic of non-monotonic reasoning, where varying assumptions can cause a belief to be "believed" or not, depending on the current context or contexts.

"living and executing code outside plans" characteristic that may strike as unusual a programmer used to more procedural languages such as Java.

Let us now list some benefits and drawbacks of the use of rules. We start by briefly restating (1) why they are useful for programming agents. Then we list a couple of reasons (2) why they are useful for the fault tolerance and we see (3) what other advantages they bring for our specific approach. Finally, we list (4) what are the main drawbacks for the use of rules.

(1) RULES FOR PROGRAMMING AGENTS Since it was aimed at programming agents that can handle uncertainty, ALMA was built around a rule-based reasoning engine and endowed with an ATMS. This allows for reasoning in the presence of inconsistent beliefs, for example "we smoke even if we know it is not good for the health". A secondary effect of this ALMA design is the possibility to use the rules in a expert-systems like paradigm, focusing on a large number of rules and their implications instead of the imperative agent definition.

Rules are added to the rule base and are applied indefinitely. This means that they can produce a belief value immediately after being added, or later, whenever a change occurs in the beliefs of their left-hand side. In conjunction with the ATMS, they can produce multiple belief values as long as their input beliefs can have multiple values (e.g. by being supported by different hypotheses). Rules can also be used to specify incoherences, e.g. $\mathtt{flight\_stage} = \mathtt{landing} \wedge \mathtt{door} = \mathtt{open} \Rightarrow \bot$, allowing the underlying system to take measures in case of undesirable situations. Their primary reason of being is thus given by the wide range of reasoning possibilities offered to the *intelligent* agent. This means for example that the agent would be able to go beyond a procedural definition and even combine rules to generate plans.

(2) RULES FOR FAULT TOLERANCE For fault tolerance in general and for the tolerance to unforeseen faults in particular, the use of rules has the following advantages:

- it facilitates the definition of asynchronous behaviours, e.g. rules can be added without worrying about the order in which the data arrives. This results in more flexible and robust agents.
- they are declarative and are more easily understandable by the domain experts, thus leaving less room for faults. Furthermore, they are easier to validate as they are simpler in structure than procedural code (at least the non-code part).

(3) RULES FOR THE SAFETY NET APPROACH On top of these generic rule advantages, in our specific ALMA and safety net use, we also note that:

- the rule representation with isolated "external" code employed in ALMA is similar to the one used in the decision nodes of the DAG code, with the same "external" code confinement properties;
- rules are of interest for the dependency tracking mechanism envisaged for the safety net approach (as discussed in Sec. 3.3.1);
- blocking the application of a rule results in the unjustification of all the beliefs that are based solely on that rule (if a belief value is supported by other

rules, they may remain enabled). In case a rule contains an error, its impact can be limited and its previous applications that produced beliefs can be automatically backtracked. As a consequence, our error handling mechanisms are easily expandable to rules.

(4) DISADVANTAGES OF RULES From the programming perspective, it is more verbose to store values in implications, e.g. $source(Sensor) \Rightarrow temperature(10)$ or even $true \Rightarrow temperature(10)$ or $true \Rightarrow data(temperature = 10)$, rather than by using simple variables such as $temperature = 10$ as in most imperative programming languages. This practice does therefore require writing sensibly more code. Furthermore, due to the increase in complexity, there is also a risk of introducing faults into the written code. These two reasons may possibly impact the acceptability of the language for mainstream programming. While outside the scope of our work, syntactic sugar could be used to improve the programming experience.

Additionally, the fact that rules apply permanently may cause other issues, since errors in the left-hand side code of rules need to be taken into consideration. Rules are evaluated each time their premises change their values. This means that, in theory, a rule may produce an error at any later moment through the code it can contain in its left-hand side. This is independent of the execution of the procedure (RT in the case of ALMA) that added that rule, which may finish its execution successfully with the programmer assuming that there is no error at that location (e.g. no exception was thrown), only for an error to appear a lot later during the execution. Executing the code in the rule only in the parent procedure, as it would be done in a more "classical" style, would allow the evaluation of the error state to be performed immediately and would possibly force the programmer to handle the error cases differently. The re-application of rules and the asynchrony between the rules and their parent procedures are therefore possible risks for the fault tolerance of any written application, when compared to "normal" code. These risks need to be carefully considered when employing rules.

RESTRICTING RULES IN ALMA But what is ALMA if we restrict the use of rules?

If it is the "permanently applying code" of rules that we are concerned about, a solution could be refraining from using rules with a code element and only using a conjunction of beliefs for the left-hand side of rules. In this "ALMA restricted", code elements associated with a rule could be added to a *decision* node just before adding the rule, but its effect would be different from the one in the original ALMA as it would only be applied once, before adding the rule.

A more drastic approach would be an "*ALMA-r*" for "ALMA minus rules", where the use of rules would be completely eliminated from the language. This could for example be done by using only $true \Rightarrow beliefs$ rules, possibly abbreviated to $beliefs$ through syntactic sugar.

Our error handling approach is based on ideas from previous work on ALMA and we need to be careful not to undermine it when restricting the use of rules. So what are the consequences of this restrictions of ALMA on the main concepts present in the language?

- The execution context keeps its reason of being: decisions still cause belief values to be added to the context.

- Unjustifications are still a valid concept, but in ALMA-r they would not be possible for the programmer (hidden mechanisms, e.g. safety net dependency tracking, can still use this).
- Hypotheses can still be added, but in ALMA-r they would be useless, as there is no longer a means to indicate an inconsistency, for example by adding

$$\mathtt{contry}(\mathtt{netherlands}) \wedge \mathtt{landscape}(\mathtt{mountains}) \Rightarrow \bot$$

DISCUSSION As discussed above, the ALMA characteristics including the use of reasoning rules are of interest for the fault tolerance in general and our safety net approach in particular. As shall be seen later on in this chapter, the fault tolerance risks issued from the fact that rules are applied permanently can be handled for the integration with the fault tolerance approach. If, for example for acceptability purposes, the language needs to be restricted with respect to the use or rules, this can be easily done as discussed above. We will now continue with a series of extensions that we propose for the base language in order to support the safety net approach.

4.2 EXTENDING ALMA FOR THE SAFETY NET APPROACH

In order to provide the necessary elements for the safety net approach, the ALMA language needs to be extended, in particular with respect to the definition of declarative goals. We begin, however, with the introduction of a keyword used to explicitly throw an unanticipated error.

4.2.1 *The `unexpected` Keyword*

While aiming our error handling mechanism at unforeseen faults, we also give the programmer the possibility to throw an error to our generic handling system through a keyword: `unexpected`. The `unexpected` keyword triggers the same reactions as the detection of an actual unanticipated error in the program execution, thus allowing the programmer to deliberately "throw the execution into the fault tolerance safety net". We distinguish two use cases:

- the "honest" use: the programmer estimates that a situation should never be reached under normal conditions (e.g. an abnormal *timeout*, an unanticipated *unjustified*);
- the "authorised" use: the programmer identifies the possibility of reaching a certain situation but does not have the time to treat that specific case or does not know how to react more specifically. An advantage is that this can spare him or her from writing code that does not make sense when forced to consider cases which should not occur, thus producing lighter programs.

To these programmer-aimed use cases, we add our own use for injecting unexpected errors in various locations in the code in order to test the post-detection behaviour of the safety net approach, as shall be seen in Chapter 5.

When using the `unexpected` keyword, especially in the "authorised" case, the programmer must be aware of the implications of reaching the call, in particular with respect to the effects of the dependency handling step, which can trigger

Table 3: Goal definition template. *As defined in our GPS work in Part III of this thesis.

Goal Description		
Name	goal name	goal identifier
Satisfaction	rules for testing the goal outcome (success or failure)	
Means-end analysis	plan selection procedure	
Time out	maximum goal waiting time	
Required beliefs	names of input beliefs	types for the input beliefs
Produced beliefs	names of output beliefs	types for the output beliefs
Plans	P_1	plan type – either "action plan" or "goal plan*"
	P_i	

reconfigurations in dependant goals and agents. With the dependency handling policy chosen in this work, this concerns the retraction of the direct outputs of a plan: goals, rules, assumptions and messages. Below, we give an example where the use of the `unexpected` keyword is justified and one where it is better to "silently" end a plan rather than cause many other components to reconfigure:

1. A plan P1 of an agent A sends a request for processing information to an agent B and waits for the result. An unjustification in P1 during this wait can be treated using an `unexpected` as this would retract the request message and cause agent B to stop processing, thus saving resources. The alternative would be for the programmer to manually specify that B needs to be contacted (possibly as part of the protocol between the two agents). The programmer could even ignore the situation or decide that B should not be informed at all of the fact that the reply is no longer needed.

2. A participant in an auction may reach an unjustification after being refused the bid. If the unjustification is treated with an `unexpected`, this can cause the whole auction to be stopped, while in reality this may not be necessary as the participant was refused anyway.

Depending on the specific requirements of each project, the code review phase would allow to identify situations where the `unexpected` is used excessively.

4.2.2 *Goals*

GOAL TYPES As seen in the State of the Art (Sec. 2.3), several goal types have been identified in the literature. In this thesis, we focus on *achievement* goals, which correspond well to our chosen definition of "state of affairs that the agent is attempting to bring about". These are goals whose success is verified by a condition that is not necessarily linked to any plan execution. For our purposes, we consider that the other goal types can be expressed using an achievement goal, albeit with a possibly verbose pattern. A *perform* goal can be represented as an achievement goal whose success condition is linked to the plan execution – a successful plan execution results in an achieved goal. A *maintain* goal is an achievement goal that is adopted in a loop each time a certain condition is breached, while a *query* goal is an achievement goal whose success condition requires the existence of a certain belief or variable value.

ALMA GOAL DEFINITION Goals are defined according to the template seen in Table 3. First, the outcome of the goal needs to be verified and this is done with the help of one or more rules. They allow the current state (beliefs, plan execution state etc.) to be evaluated to decide if the goal was achieved or failed. The next plan to execute is determined using the Means-end analysis (MEA) procedure. In this work, we use programmer-provided plans and the role of the MEA is to sort through these plans and identify the most suitable one to use for the current context. In the simplest form, this procedure simply selects the plans in the order of their appearance. This is also where plan application conditions (pre-conditions) would be implemented. A more complex Means-end analysis alternative would be able to automatically generate plans. There is also a timeout value that ensures that an agent does not block on a goal. Finally, one or more plans can be provided for each goal. Their inputs and outputs need to coincide to the ones indicated for the goal.

In our examples we used two of the three ALMA beliefs: simple beliefs and sets. In the models, for clarity, we distinguish three types:

- *simple*, for any belief containing a Prolog term. This corresponds to the unique assignment belief of ALMA.
- *list*, for when the use of the belief requires it to be in the form of a list. This is actually a simple belief that contains a list.
- *set*, for beliefs that can have multiple values that may change in time. This corresponds to the set beliefs in ALMA.

The implementation of the goal life-cycle introduced in Sec. 3.3.3 is described in Appendix A. The implementation of goals using ALMA RTs means that the RT context is inherited from plans to the goals they adopt and then to the plans executing for those goals. This means that the beliefs can be used to stop the goal execution, as well as the executions of any plans and sub-goals that inherited the original goal's context.

4.2.3 *Plans*

LEVELS OF ABSTRACTION While the original language was constructed using RTs, the introduction of goals changes the level of abstraction of the model. Our introduction of plans in ALMA required an inquiry on their relationship with RTs. Since RTs are meant to constitute the agent behaviour by launching other RTs or new instances themselves, in order to attain a reasonable level of expressiveness, we define a plan as a structure of one or more RTs executing for the achievement of a goal.

ALMA VARIABLES VS BELIEFS In ALMA, while the beliefs and assumptions used by each RT need to be declared in its header (for programming reasons), they are actually global in that any other RT can use them. Prolog terms also exist inside the RTs and can be used to transfer values between different nodes and even transmitted when launching the execution of a new RT, thus circumventing the belief system, which is not a good practice in general. Nevertheless, as in our representation multiple RTs work together for a same goal form a plan, data exchanged inside an RT and between RTs is not a problem as intra-plan data exchanges are not of interest for the dependency handling step of the recovery phase.

LACK OF CYCLES In ALMA, due to the way RTs are constructed, no `for` or `while`-equivalent construct are possible. Dekoker [29] notes that "a succession of states in the graph corresponds to a unique sequence of operations. This makes the agent's behaviour more explicit; in an automaton, to keep track of successive passes through a state, variables are used, while in an RT, this is read graphically. Furthermore, this means of programming leads to the use of shorter RTs.". The problem with this characteristic of the language is that it makes it difficult to write automata-style behaviours and it bears the risk of unforeseen behaviour due to the focus on a single iteration (e.g. infinite loops). The solution we propose is to take advantage of the changing of levels of abstraction from RTs to plans to allow loops in our language, while implementing them using the ALMA workaround – recursive RT calls.

PARALLELISM While the multi-agent architecture has parallelism at its foundation, intra-agent parallelism is also an important feature for programming expressive and thus useful agents. For example, an agent may need to process a message while in the same time wait for a new one. In Java, this can be done by launching a new `Thread`, while in ALMA the equivalent is achieved through the `new_rt` action that starts a new reasoning thread. In a plan, such actions create multiple branches that function in parallel and may even interact with each other. When an error is encountered inside the plan, the repair of these branches is more tedious and may result in stopping multiple branches. Goal adoption, on the other hand, allows for implicit parallelism as goals are natively parallel. Therefore, instead of allowing branch creation inside plans, we require parallelism to be created only through goal adoptions. The consequences of this choice are:

- plans are simpler and easier to read;
- reparations in case of error are simpler as intra-plan interactions are limited;
- more fall-back points – the goals – are present in the agent behaviour thus avoiding massive roll-backs in case of error.

DISCUSSION The introduction of goals therefore also changes the level of abstraction from RTs to goals and plans. These plans are allowed to have cycles, while their parallelism is only only allowed through goal adoptions. The result is plans that are simple enough, yet more complex and expressive than RTs.

4.2.4 *The ALMA+ Model and Language*

THE GRAPHICAL MODEL The enriched model[2] can be seen in Fig. 27.

First, we added an *adoption* node to clearly distinguish goal adoptions from the other agent actions. We used the block arrow symbol to suggest the idea that the goal executes in parallel with its parent plan. Then, there are extended *decision* and *wait* nodes that need to be able to verify or wait for a specific goal outcome. Waiting for a goal outcome is trivial – e.g. an agent adopts a goal to acquire the list of acquaintances and then waits because it needs to use that list for its next action. Being able to verify the status of a goal offers more flexibility in continuing the execution of the goal plan in parallel with the goal execution – e.g. an agent

2 The original nodes are described in Sec. 2.4.4.

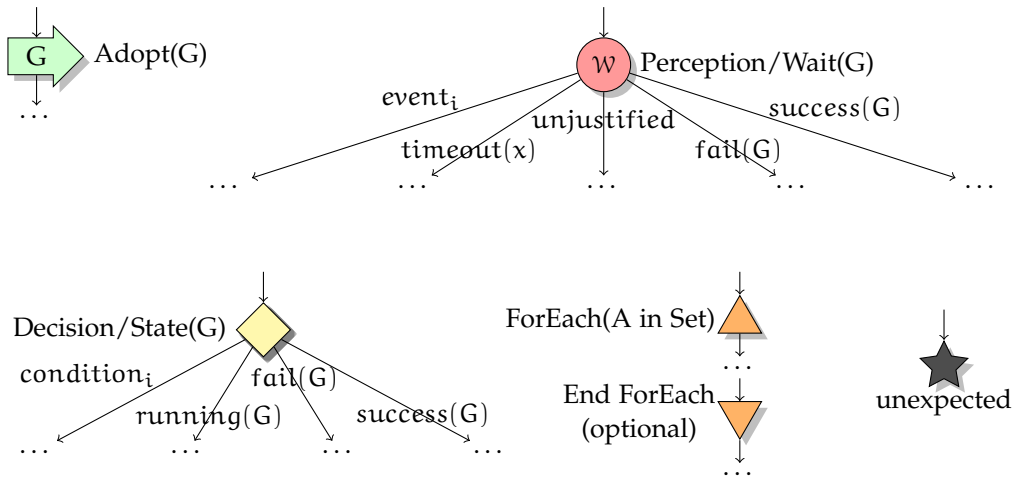


Figure 27: Proposed new nodes: *adopt* (extends action node), *wait* with new condition on goal end, a *decision* node for the outcome of a goal, *for each* nodes and finally an `unexpected` error node symbolising a case deemed impossible by the programmer, or in which the preferred reaction is given by our recovery mechanism. As exemplified in Fig. 28, the *End ForEach* is needed only if the plan does not finish with the *ForEach* block.

adopts a goal to acquire the list of acquaintances and then continues the same plan by adopting other goals after which it tests the status of the first goal in order to decide if it can use its outputs. Therefore, these two nodes allow the goal plans to be expressive enough for most tasks. Note that while we use the terminology from our GPS work, this model can be used without it as well.

We also added the possibility to graphically represent the `for each` command which was already supported by the original ALMA language. There are an opening *ForEeach* node and a closing *End ForEach* one to indicate precisely which steps are performed for each element of the given set or list. As seen in the example in Fig. 28, when the `for each` block is used at the end of a plan, the *End ForEach* node can be omitted.

A black star marks an unanticipated error deliberately triggered by the programmer, which corresponds in ALMA+ to the *unexpected* keyword described above.

INPUTS AND OUTPUTS Plans have input and output beliefs that correspond to the ones required by their goals as specified in the goal description. For each plan, a list of beliefs and their type is thus specified. The types are the same as described in the case of goals – simple, list and set – to which we add their mode: "IN" for input, "OUT" for output and "IN/OUT" for input and output. When a belief is local to the plan and possibly used by its sub-goals, no mode is specified.

A PLAN EXAMPLE In Fig. 29 we give a simple plan example. The plan starts with a *decision* node that verifies that $\texttt{belief}(\texttt{Defense\_approved}, \texttt{yes})$ is justified (thus adding it to the RT context), before dealing with each of the contacts given in the input belief `Jury_list`. For each of these, a goal is adopted to handle the discussion. When the goal is achieved, the list of confirmed jury members is updated. If the RT context is unjustified while waiting for any of the goals, for example because the $\texttt{belief}(\texttt{Defense\_approved}, \texttt{yes})$ is no longer supported, then the safety net mechanisms take control as the `unexpected` keyword was used.

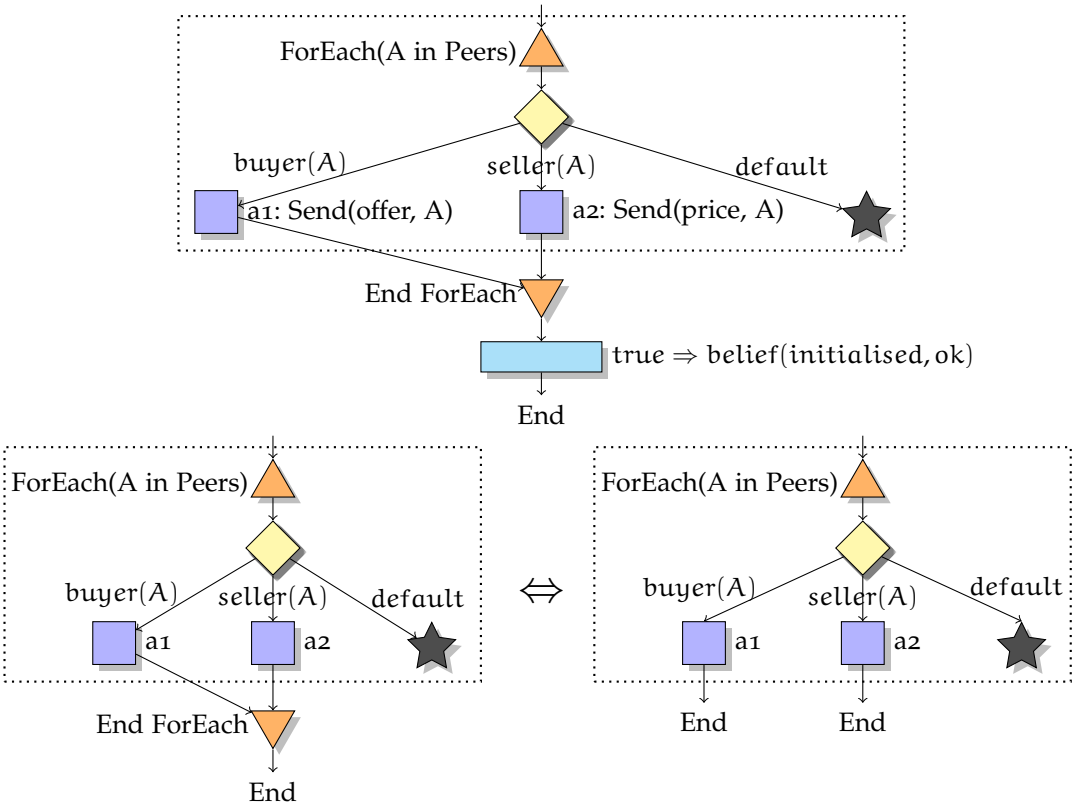
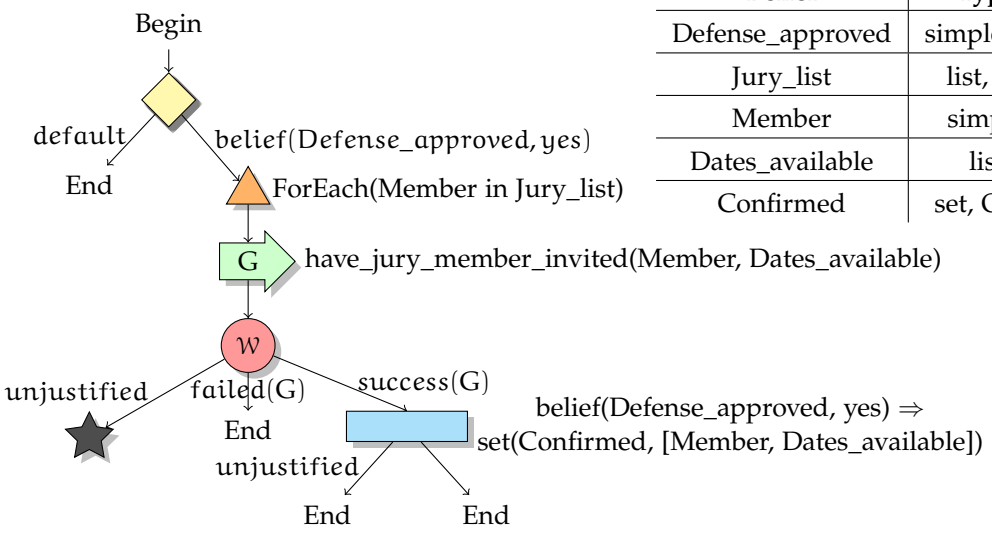


Figure 28: Examples for the use of the *ForEach* node. Note how at the end of a plan the *End ForEach* node is optional. Also, note that the *unexpected* node is a terminal node (just as an *End* node). The plan sections in the dotted rectangles are identical between them.



Belief	Type
Defense_approved	simple, IN
Jury_list	list, IN
Member	simple
Dates_available	list
Confirmed	set, OUT

Figure 29: A plan example for a goal to organise the PhD defense. Goal G: have_jury_member_invited uses two beliefs: `Member` of type "simple, IN" and `Dates_available` of type "list, OUT". Goal G is achieved only when the contacted jury member confirms his or her availability, and fails otherwise.

In ALMA, this plan would involve an RT testing the belief and then adopting a new RT for each contact. The second RT would contain a goal adoption and the subsequent *wait* and *reasoning* nodes.

GOAL RELATIONSHIPS The model defined here takes advantage of the expressive power of the ALMA+ model for integrating goal adoptions into complex behaviours which use conditions, parallelism etc. However, the model does not directly allow the representation of more subtle goal relationships like inhibitions (when a goal is achieved, another ceases to exist) as these are not present as such in the underlying language. These kinds of relationship can be represented, nevertheless, using reasoning mechanisms that are available to the programmer through the ATMS and rules. For example, the relationship "if goal G1 is achieved, goal G2 is no longer desirable" could be introduced using a rule "$achieved(G1) \wedge desirable(G2) \Rightarrow \bot$" which causes the assumption "$desirable(G2)$" to be disabled once goal G1 is achieved.

4.3 THE THREE FAULT TOLERANCE PHASES IN ALMA+

Having chosen a programming language, we can now revisit the three phases of fault tolerance in the more specific setting of ALMA+.

4.3.1 *Detection*

The ALMA+ language structure now allows us to pinpoint more precisely the cases in which an unanticipated error can appear. In Table 4 we list the 9 error classes we identified, together with an example for each. In the same table, we also indicate four levels for the error detection (represented in the fourth column):

i. C for code level;

ii. P for plan level;

iii. RB for rule base level;

iv. G for goal level.

Let us now introduce each of the error classes.

CLASS 1 The first class corresponds to an unhandled code exception. Due to the fact that computation "external" code in ALMA is executed only in specific locations – in decisions and in the left-hand side of rules – these locations constitute excellent places for exceptions to be caught by the platform.

CLASS 2 A particular error that is related to the first class is when the memory allocated to the code execution is exceeded. These first two classes of error must be caught at platform level (more on this in Sec. 4.4).

CLASS 3 While a *wait* node or goal timeout are part of the normal behaviour, a code section that takes too long to execute can block an entire agent due to the way parallelism is implemented in ALMA. Safeguards are therefore needed at this level. While not present in ALMA and currently not introduced in ALMA+ either, lower level timeouts could be required for each "external" code section. The risk is that such specification may add an important burden on the programmer, without bringing considerable robustness benefits. A solution could be to set default large values for these as upper limit, possibly allowing the programmer to specify a different value if appropriate.

CLASS 4 Similarly to the previous class, plans may also take too long to execute. This too is not a case explicitly covered in ALMA or AMLA+. However, since goals already have a timeout condition that can cause a goal to abort after a specified amount of time, our intuition is that with code timeout to avoid the agent completely blocking, a plan should not require another safeguard.

CLASS 5 The next class concerns the use of the `unexpected` keyword introduced in Sec. 4.2.1 for explicitly using the safety net.

CLASS 6 Moving on to the use of rules, a first error class is when the agent memory is exceeded, i.e. the knowledge base comprising agent rules and beliefs is full. This is another mechanism that is not present in the original language and that we did not yet study for ALMA+ but which constitutes a possible error class for the safety net approach.

CLASS 7 Another error is the *global inconsistency*, which is reached when a contradiction is supported by true facts (contradictions are added explicitly, e.g. R1: $\mathtt{position} = \mathtt{MontBlanc} \wedge \mathtt{altitude} < 4809\mathtt{m} \Rightarrow \bot$, or automatically as is the case of single assignment variables in ALMA), in other words when $\mathtt{true} \Rightarrow \mathtt{false}$ can be deduced from the existing rules. In this case, there are no assumptions that can be disabled to bring the agent knowledge base to a consistent state (as is the case in the next error class) and rules cannot be removed. This means that the agent needs to be stopped. Adding, for example, R2: $\mathtt{true} \Rightarrow \mathtt{altitude} = 1000\mathtt{m}$ and R3:

$true \Rightarrow position = MontBlanc$ together with R1 above leads to an implication that is equivalent to $true \Rightarrow \bot$.

CLASS 8 A less serious situation, is an *inconsistency*, which is reached when a contradiction is supported by the current assumptions. While this can be part of the normal agent reasoning, it can also indicate an error, for example rule R1 above can help detect error in the auto-pilot function of a plane. Through its normal functioning, the ATMS ensures that the context or contexts are kept consistent by removing the necessary assumptions, as long as this is possible, otherwise, the agent finds itself in the previous error class. Disabling assumptions may cause one or more plans to become unjustified, which is a situation already covered by the current implementation of ALMA.

CLASS 9 The last class corresponds to the non-achievement of a goal following an apparently successful plan execution and, as discussed in the previous chapter, is handled normally as defined in the goal life-cycle.

With respect to the two error families discussed in Sec. 3.1, classes 1, 2, 5, 6, 7 correspond to exception-based detections, while classes 8 and 9 correspond to objective-based detections. As shall be seen from the handling strategies we present below, we use classes 3 and 4 as exception-based rather than objective-based detections.

4.3.2 *Confinement*

For the confinement phase, we start by listing the ALMA+ properties that contribute to the confinement of errors:

- the execution of computations as "external" code in specific locations and without "external" effects, i.e. writing to the agent memory, sending messages or adopting goals;
- support for the goal-plan paradigm;
- the platform creating one Prolog instance per agent.

WHERE In order to identify the entity directly impacted by the error, we need to study where the considered errors can occur, with another advantage of the language structure being that the errors are localised in the four cases cited below. In Table 5 we revisit the error classes described in the previous subsection with the corresponding confinement phase, while in Table 6 we turn perspective on the confinement cases:

I. **inside a plan** – corresponding to a level C detection (a code crash, memory or time condition violation in the code of a *decision* node, classes 1, 2 and 3 respectively), or at level P (from plan-level timeout – class 4 – or an `unexpected` – error class 5);

II. **inside a rule** – level C detection (a crash or violation of a memory or time condition in the code of the rule, error classes 1, 2 and 3 respectively);

Table 4: Faults and errors

No.	**Error** How we detect	**Fault** Cause	**Level** For detection	**Example** What it can look like
1	code crash	computation fault (in decision or rule)	C (code)	division by zero; corrupted message (even malicious attack)
2	code level memory safety mechanism	out of memory crash (in decision or rule)		an infinite loop that consumes memory too fast (before timeout) in code
3	code level timeout	infinite loop/ too much waiting (in decision, rule or plan)		infinite loop in code
4	plan level timeout		P (plan)	infinite loop in plan structure; wrongly timed wait (e.g. timeout set to very large number, local timeout not acceptable at plan level); cumulative large waiting times
5	`unexpected`	situation that should not occur (in plan)		a *default* branch that should never be reached; an error the programmer does not want to handle
6	plan level memory safety mechanism	rule or belief base full (in plan)	RB (rule base manager - IE/ATMS)	logical error causes too many rules to be added
7	global inconsistency	bad rules		$\mathtt{true} \Rightarrow ... \Rightarrow \bot$
8	inconsistency	bad use of rules or bad rule inputs		(a) $\mathtt{altitude} = 1000\mathrm{m} \wedge \mathtt{position} = \mathtt{MontBlanc}$ and $\mathtt{position} = \mathtt{MontBlanc} \wedge \mathtt{altitude} < 4809\mathrm{m} \Rightarrow \bot$; (b) $\mathtt{altitude} = 1000\mathrm{m}$ and $\mathtt{altitude} = 300\mathrm{m}$ in the same context
9	goal not achieved	undetected error during plan execution	G (goal satisfaction condition)	hidden/unobserved variable (e.g. the boiling time is correct but the goal to cook food is not achieved – see example in the quote from Sec. 1.4)

Table 5: Faults and errors with the reactions: what is stopped in each case, with D=decision and R=rule

No.	**Error** How we detect	**Location** Of detection	**Stops** By confinement	**Confin.** Case
1	code crash	in the decision D that crashes	plan containing D	I
		in the rule R that crashes	rule R	II
			goal whose satisfaction test is R	III
2	code level memory safety mechanism	code (D or R)	I for D, II for R or III for goal whose satisfaction test is R	
3	code level timeout			
4	plan level timeout	plan	plan	I
5	`unexpected`			
6	plan level memory safety mechanism	rule base manager (IE/ATMS)	agent	IV
7	global inconsistency			
8	inconsistency		$\emptyset$ (handled by the *unjustified*)	V
9	goal not achieved	satisfaction condition	$\emptyset$ (handled as part of the goal life-cycle)	V

Table 6: Confinement cases

Confinement		Detection	
Case	Action	Level	Covered error classes
I	stop plan	P (plan)	4, 5
II	stop rule	C (code)	1, 2, 3 for error in decision
III	abort goal		1, 2, 3 for error in rule
IV	stop agent		1, 2, 3 for error in goal satisfaction rule
V	$\emptyset$	RB (rule base)	6, 7
			8 (normal *unjustified*)
		G (goal)	9 (normal goal life-cycle)

III. **inside a goal verification** – a special case of rule code failure (classes 1, 2 and 3);

IV. at the level of **the agent memory** (error classes 6 and 7).

A fifth case presented in the tables corresponds to the error classes 8 and 9, where an active confinement phase is not needed. In class 8, it is the language structure that, through its handling of contexts and unjustifications, already ensures a proper response to the situation, i.e. the inconsistency. In class 9, it is the goal that guides the reaction to an undetected error in its plan. This fifth confinement case will not be discussed further as there is no confinement reaction, nor dependency handling to it.

In what follows, we consider the MEA procedures associated to each goal as simple platform-provided patterns which are not concerned by errors. However, if programmers were allowed to write the code of these procedures, the possible faults introduced would be handled similarly to the case III above.

WHAT The active part of the confinement requires stopping the entity, which, for each of the four cases, means:

I. for a **plan**, the execution is abruptly stopped: the fact that an error occurred during the plan execution implies that there is no corresponding *unjustified* branch to use. However, note that the plan's parent goal will handle the event just as a failed[3] plan execution.

II. for a **rule**, stopping means ensuring that the rule is never executed again;

III. for a **goal**, stopping means making it no longer desirable (some works use the term "drop" or "abort" [111]);

IV. when the memory of an agent is compromised, the **agent** needs to stop its entire behaviour.

HOW The confinement needs to be ensured by mechanisms included in the platform.

All executions of the "external" code are to be performed with a code "sandbox" to help confine any error. As we discuss in Sec. 4.4, timeouts and generic error catching mechanisms can be associated with these executions.

While an "external" code section can be easily stopped as there are no side effects involved, ideally, a plan would be stopped by unjustifying it, thus taking advantage of its own repair mechanism. The problem is that when a plan timeout is exceeded, such a mechanism may not be reachable (e.g. no appropriate nodes are present in the currently executing section). In this case, a two level system may be useful: first attempt to unjustify the plan, and after a fixed amount of time, kill the plan completely and act as in the case of a code crash.

TECHNICAL ASPECTS In ALMA, as described in Sec. 2.4.4, each agent executes in a single Prolog thread, which is thus managed by the Prolog platform which handles its confinement with respect to the host operating system.

3 If we consider that plans can indicate a successful execution, similar to a boolean return value.

Table 7: The direct dependency table. Reading by line: goals can cause their plans to be unjustified, plans should unjustify the goals they adopt, messages they send, rules they add and assumptions they make etc. Reading by column: goals depend of the plans that adopted them etc.

	Goal	Plan	Message	Rule	Belief	Assumption
Goal		x				
Plan	x		x	x		x
Message		x				
Rule					x	
Belief/Assumption		x		x		

4.3.3 *Recovery*

As presented in Sec. 3.3, the recovery phase is comprised of 3 steps: dependency handling, reparation and reconfiguration. One of the reasons for choosing ALMA was that there is already a dependency-reparation mechanism in place in the form of the RT context and the *unjustified* branch. The use of beliefs in a *decision* node causes them to be added to the context, so that in case any of them is no longer justified, the RT can be stopped and placed into reparation mode, as defined in the *unjustified* branch corresponding to the current execution. These will be extended for the use as part of the safety net approach. The use of goals for reconfiguration completes the recovery.

4.3.3.1 *Dependency handling*

DEPENDENCIES For the dependency handling step, the idea is to support the clean-up process after an error by triggering reparations in the components that are downstream from the error in the dependency graph. This means issuing an error signal on all the paths that originate in the current component. As seen in Sec. 4.3.2 on confinement, in ALMA+ there are more plan outputs and dependency types than the ones discussed in Chapter 3, so the dependency handling model will have to be adapted accordingly. In Table 7 we can see the types of dependency possible in our model. To ensure these dependencies are correctly propagated, we need to enrich the platform behind ALMA+ with the appropriate dependency mechanisms. However, these dependency mechanisms and any rules and beliefs they include must not interfere in any way with the ones that are actually used by the programmer. It is also important that the programmer does not tamper with the recovery mechanisms.

WHAT In Table 8 we present the dependencies and reactions following each of the confinement cases discussed in Sec. 4.3.2. After the confinement phase stops the entities concerned by the error, the dependency handling step needs to ensure the correct propagation of the error signal to the entities concerned. While several types of component are concerned by the dependency handling, only the plans react through a reparation. The others – goals, rules and agents – are either stopped or blocked:

I. for a plan, the retractions are:

Table 8: Dependencies to "prune" for each confinement case

Case	Confinement action	Dependency handling	
		Dependency	Reaction
I	stop plan	assumptions	retract
		rules	stop (as II below)
		goals	abort (as III below)
		messages	remotely unjustify
II	stop rule	beliefs	retract (automatically)
III	abort goal	executing plan (if any)	unjustify
IV	stop agent	messages for all executing plans	remotely unjustify
V	$\emptyset$	n/a	n/a

a) any assumptions made are disabled (no longer believed, but not contradicted)

b) the application of any added rules is blocked (see II. below)

c) any goals adopted by the plan are aborted (see III. below)

d) messages are no longer supported, for each one that the plan sent, another message that states that the original is no longer valid is dispatched (this implies that the receiver agent can interpret such message)

II. for a rule, the confinement phase permanently blocks the application of that rule thus disabling automatically any beliefs that are supported only by it, the "retraction" part is implicit, possibly causing the unjustification of certain plans (or RTs) as described in Sec. 4.3.3.2

III. for a goal, its abortion implies stopping through an unjustification the plan executing for that goal, if any. The goal outcome will be "failed".

IV. when an agent needs to be stopped, it is the outside actions that are concerned, as the internal retractions are no longer useful. Therefore, for all executing plans, the retraction concerns the sent messages (as in the case of point I.d above).

APPROACH As introduced in Sec. 3.3.1, the idea is to build a rule-based structure reflecting the observed behaviour of the agents with their interdependencies. In each agent, a view of the dependencies concerning that agent will be locally managed. For example the rules will be saying: "if X can be trusted and X sends a message to Y, then Y can trust that message" and "if Y can trust a message and its own reasoning, then the results of that reasoning on the message are correct". In this way, for example if we learn that X should not be trusted, we can stop Y from reasoning with the message from X.

This *dependency context* acts just as the RT context in ALMA+: it accumulates dependencies and if any of these dependencies is no longer justified during the execution, the components that based their execution on that dependency are no longer justified either. For a plan, this implies entering the unjustified state. For a rule, this implies not applying the rule any more. It is important to note that a

belief needs to be justified by both the RT context and the dependency context to be enabled.

MBD DEPENDENCIES FOR ALMA+ In order to identify the required MBD rules for creating the dependency context, we studied the interactions that generate dependencies. For ALMA+, these interactions concern: goals, plans, rules, beliefs, assumptions and messages. We aim to be able to control the programming elements corresponding to the four elements that can be stopped during the confinement phase: (I) plans, (II) rules, (III) goals and (IV) agents. Each of these will have a corresponding ok(...) assumption that will be disabled to propagate the error signal to the element's outputs.

Figure 30 shows the three MBD sets, with SD split into SD.NORMAL corresponding to the normal functioning of the system listed in Fig. 31 and SD.ERROR for reacting to an error detection and retracting the corresponding ok(X) assumption in Fig. 32.

We place in COMPS the components we that the model can cause to stop and for which we will create ok(X) assumptions to indicate their correct functioning:

$$\text{COMPS} = \{\text{Plans}, \text{Rules}, \text{Goals}, \text{Agents}\}$$

We consider three types of "observation", as seen in the OBS set:

1. code executions that return `true`[4], corresponding to the successful application of a rule or transition in a *decision* node. As the reads are already verified as part of the MBD rules, only the output of the code execution is required. If the required belief values are not available, their corresponding MBD rule is not concerned. If the code output is other than *true* – i.e. *false* or *error* – the MBD rule will not be applied either. The interest of this observation is for distinguishing between tests that are attempted but do not influence the dependency context of the plan, and the ones that actually do and will allow the continuation of the plan or the writing of a belief. The distinction is important when a code section encounters an error, as for example in the case of the *decision* nodes, all branches are concerned with the test MBD rule but only the chosen one is concerned by the context changing MBD rule.

2. message injustifications received from another agent in case it encountered an unanticipated error and it does not support the sent message any more.

3. error detections which, as described in the previous sections, can appear at 4 levels: code (level C), plan (level P), rule base (level RB) and goal (level G). While the last level corresponds to the unsuccessful execution of a plan for a goal which is already taken into account by the goal life-cycle, the first three correspond to detection events that we take into consideration for our confinement and dependency handling phases. They are thus included in the OBS set. When an unanticipated error is detected in the corresponding component, the observed value for *text* is "error".

Let us now describe the SD rules. The idea is to keep track of all inputs of a plan instance in what we call its *dependency context*. As these create a logical structure, il

4 Note that a code execution can return `false` as part of its normal functioning, without this being an error.

$$\begin{aligned} \text{COMPS} &= \{\text{Plans}, \text{Rules}, \text{Goals}, \text{Agents}\} && (27)\\ \text{OBS} &= \{\text{Code} = \text{true}\} \cup \\ &\quad \{\text{supported}(\text{Message}) = \text{false}\} \cup \\ &\quad \{\text{test}(X) = \text{error} | X \in \{\text{Code}, \text{Plan}, \text{RuleBase}\}\} && (28)\\ \text{SD} &= \text{SD.NORMAL} \cup \text{SD.ERROR} && (29) \end{aligned}$$

Figure 30: The three sets from the MBD model for the dependency context tailored to ALMA+

will be possible to control the execution by contradicting hypotheses on the normal functioning of components.

The execution of an agent starts with a first implicit goal and its plan. We thus need to add an MBD rule for justifying the initial goal that is assigned to the agent at its creation, adding on this occasion an assumption that will allow the whole agent to be stopped if needed, "ok(Agent)":

$$\text{ok}(\text{Agent}) \wedge \text{initial}(\text{Goal}) \Rightarrow \text{execute}(\text{Pl})$$

When the execution of a plan is started, the plan inherits from its goal the context in which it was started. We also need to add the assumption that the plan instance is functioning correctly – *ok(Pl)*. The successive evolutions of the dependency context are then represented through the $\text{context}(\text{Pl}_i)$ predicate. The dependencies of a plan instance get richer only with its inputs: outputs have no effect on the local context.

$$\text{execute}(\text{Pl}) \wedge \text{ok}(\text{Pl}) \Rightarrow \text{context}(\text{Pl}_0)$$

Receiving a message adds the dependencies of that message to the current dependency context:

$$\text{context}(\text{Pl}_n) \wedge \text{rcv}(\text{Message}, \text{A}_{\text{from}}) \Rightarrow \text{context}(\text{Pl}_{n+1})$$

A message is linked to the current plan's assumption:

$$\text{ok}(\text{Pl}) \Rightarrow \text{send}(\text{Message}, \text{A}_{\text{to}}, \text{Pl})$$

The *send* predicate is based on the assumption of the plan that created the message, but this needs to be transmitted through other mechanisms that are not represented at this level. $\text{justified}(\text{Message})$ is an assumption that is created automatically when an agent receives and uses a message. In case the sender agent encounters an unanticipated error, it automatically sends another message stating that it does no longer support the original message, which is transparently handled by the receiver agent. In this MBD model, this is translated into an observation that invalidates the "justified" assumption causing the plans that used that message to react.

$$\text{send}(\text{Message}, \text{A}_{\text{to}}, \text{Pl}_{\text{from}}) \wedge \text{justified}(\text{Message}) \Rightarrow \text{rcv}(\text{Message}, \text{A}_{\text{from}})$$

A single case of belief read can occur outside decisions and ALMA+ rules: in *wait* nodes.

$$\text{context}(\text{Pl}_n) \wedge \text{read}(\text{B}_{\text{from}}) \Rightarrow \text{context}(\text{Pl}_{n+1})$$

Adding an ALMA+ rule does not affect the context and is only linked to the "ok(Pl)" assumption. Tests on the memory are made in parallel and the error case is discussed later in this section.

$$ok(Pl) \Rightarrow add\_rule(Rule)$$

When the condition is observed to be true and the plan execution continues, the dependency context needs to be enriched by the values of the beliefs used by the condition. Note that we are considering here the test condition rather than the entire ALMA+ *decision* node.

$$Pl.Dec = (\bigwedge_k B_k \wedge Code_{Pl.Dec}) \wedge \bigwedge_k read(B_k) \wedge Code_{Pl.Dec} = true \quad \begin{array}{l} context(Pl_n) \wedge \\ \Rightarrow context(Pl_{n+1}) \end{array}$$

Similarly, the execution of reasoning rules is addressed by a MBD rule corresponding to its successful applications. The main differences from the similar rule for the *decision* nodes are the fact that no plan context is involved, that the ALMA+ rules result in written beliefs which can be read by other ALMA+ rules or plans and that there is an "ok(Rule)" assumption that can be used to block the application of the rule.

$$Rule = (\bigwedge_k B_k \wedge Code_{Rule} \rightarrow B) \wedge \bigwedge_k read(B_k) \wedge Code_{Rule} = true \quad \begin{array}{l} add\_rule(Rule) \wedge ok(Rule) \wedge \\ \Rightarrow read(B) \end{array}$$

An assumption added by a plan is only linked to that plan (remember that an assumption is just a normal belief but which is enabled without being justified by a rule):

$$ok(Pl) \Rightarrow read(Assumption)$$

Any branching in a plan, possible through a `new_rt` in ALMA+, causes the current context to be copied to that branch as well:

$$context(Pl_n) \Rightarrow context(Pl_n^{NewBranch})$$

The goal adoption is similar to the other outputs of the plan – it does not change the context – but it does inherit the plan's current context.

$$context(Pl_n) \Rightarrow adopt(NewGoal)$$

A goal's context is based on the context inherited from the parent plan to which the assumption that the goal is executing correctly is added.

$$adopt(Goal) \wedge ok(Goal) \Rightarrow context(Goal)$$

The goal context is then linked to the goal verifications which are rules as well:

$$context(Goal) \Rightarrow add\_rule(Goal.Rule)$$

While the dependencies are handled just as in the case of normal rules, as shall be seen in Fig. 32, an error detected in a goal satisfaction rule causes the whole goal to be unjustified.

An executing plan inherits its goal's dependency context:

$$context(Goal) \Rightarrow execute(Pl)$$

Plan execution finishes when either the last action finished or an error occurs or the plan is stopped. Rules, however, remain in the rule base and are applied indefinitely, as long as they are still justified, i.e. no error occurs in their code.

In Fig. 32 we introduce the rules corresponding to the observation of errors and the active confinement reactions, together with the rule corresponding to the unjustification of messages. These rules have on their left side assumptions of correct functioning of the chosen components – plans, rules, goals and agent – or the assumption that a message is justified, and on their right, the expected observation – that the corresponding component executed correctly – or that the message is supported by the sender agent. In case any of these observations is contradicted, regardless of the cause, the left side assumption is disabled and the component's outputs are no longer justified. Each of these rules are added once the corresponding assumption is first used, so at the creation of each agent, plan, goal and rule, as well as the use of a message by a plan[5].

CONCLUSION We showed in this section that more than one component concerned by the dependency handling step. The rules, together with the other underlying mechanisms that ensure they are added at the right moment as well as the inter-agent propagations all have to be transparently ensured by the platform. After this automatic propagation, in the next step plans are given the possibility to apply reparation steps tailored for the specific situation and provided by the programmer. Note that while the error signal always propagates all the way to the next plans, regardless of the type of dependency – through one or more rules, assumptions, adoptions or messages.

4.3.3.2 *Reparation*

THE IDEA Once the dependency handling signals plans that at least one of their inputs is no longer supported, the plans need to enter a reparation mode, which is the next step in recovery. As stated before, of the components concerned by the dependency handling, only plans contain reparation elements. In ALMA+, the role of reparation branches is carried by the *unjustified* which are compulsory for the *wait* and *add_rules* nodes. These are triggered when the current RT context is unjustified, in other words when at least one of the beliefs that were used by the current RT or its ancestors is unjustified, leaving the execution in an inconsistent state. The idea here is to extend the use of the *unjustified* branches in ALMA+ from the RT context to include the *dependency context* as well. In this way, the semantic of the unjustification is extended to include inputs that were no longer justified due to an error detection in their source.

THE PROGRAMMER'S STATE OF MIND The state of mind of the programmers writing *unjustified* branches remains the same as in the original ALMA language, as the question they ask themselves is: what if during the time this plan is waiting, its existence is no longer justified? Why it would be the case? It can be that:

1. one of the beliefs that led the program here is no longer supported,

5 A message is received in the "mailbox" (incoming messages buffer) of an agent and can be used (read) by zero or more plans. A dependency is created only when a message is used.

$$
\begin{aligned}
\text{SD.NORMAL} = \{ & \\
\{ok(Agent) \wedge initial(Goal) &\Rightarrow execute(Pl)\} && (30)\\
\cup & \\
\{execute(Pl) \wedge ok(pl) &\Rightarrow context(Pl_0)\} && (31)\\
\cup & \\
\{context(Pl_n) \wedge rcv(Message, A_{from}) &\Rightarrow context(Pl_{n+1})\} && (32)\\
\cup & \\
\{ok(Pl) &\Rightarrow send(Message, A_{to}, Pl)\} && (33)\\
\cup & \\
\{send(Message, A_{to}, Pl_{from}) \wedge & \\
justified(Message) &\Rightarrow rcv(Message, A_{from})\} && (34)\\
\cup & \\
\{context(Pl_n) \wedge read(B_{from}) &\Rightarrow context(Pl_{n+1})\} && (35)\\
\cup & \\
\{ok(Pl) &\Rightarrow add\_rule(Rule)\} && (36)\\
\cup & \\
\{context(Pl_n) \wedge & \\
Pl.Dec = (\bigwedge_k B_k \wedge Code_{Pl.Dec}) \wedge & \\
\bigwedge_k read(B_k) \wedge Code_{Pl.Dec} = true &\Rightarrow context(Pl_{n+1})\} && (37)\\
\cup & \\
\{add\_rule(Rule) \wedge ok(Rule) \wedge & \\
Rule = (\bigwedge_k B_k \wedge Code_{Rule} \rightarrow B) \wedge & \\
\bigwedge_k read(B_k) \wedge Code_{Rule} = true &\Rightarrow read(B)\} && (38)\\
\cup & \\
\{ok(Pl) &\Rightarrow read(Assumption)\} && (39)\\
\cup & \\
\{context(Pl_n) &\Rightarrow context(Pl_n^{NewBranch})\} && (40)\\
\cup & \\
\{context(Pl_n) &\Rightarrow adopt(NewGoal)\} && (41)\\
\cup & \\
\{adopt(Goal) \wedge ok(Goal) &\Rightarrow context(Goal)\} && (42)\\
\cup & \\
\{context(Goal) &\Rightarrow add\_rule(Goal.Rule)\} && (43)\\
\cup & \\
\{context(Goal) &\Rightarrow execute(Pl)\} && (44)\\
\} &
\end{aligned}
$$

Figure 31: The SD.NORMAL set for the dependency context tailored to ALMA+

$$
\begin{aligned}
\text{SD.ERROR} = \{ & \\
& \{\text{ok}(\text{Pl}) \Rightarrow \text{test}(\text{Pl}) = \text{okay}\} && (45)\\
& \cup \\
& \{\text{ok}(\text{Ag}) \Rightarrow \text{test}(\text{RuleBase}) = \text{okay}\} && (46)\\
& \cup \\
& \{\text{ok}(\text{Pl}) \Rightarrow \text{test}(\text{Code}_{\text{Pl.Dec}}) = \text{okay}\} && (47)\\
& \cup \\
& \{\text{ok}(\text{Goal}) \Rightarrow \text{test}(\text{Code}_{\text{Goal.Rule}}) = \text{okay}\} && (48)\\
& \cup \\
& \{\text{ok}(\text{Rule}) \Rightarrow \text{test}(\text{Code}_{\text{Rule}}) = \text{okay}\} && (49)\\
& \cup \\
& \{\text{justified}(\text{Message}) \Rightarrow \text{supported}(\text{Message}) = \text{true}\} && (50)\\
& \}
\end{aligned}
$$

Figure 32: The SD.ERROR set for the dependency context tailored to ALMA+

2. or one of the messages that allowed the program to get to the current point was retracted,
3. or one of the components that this plan is working with is no longer considered reliable.

Whatever the case, a programmer is the best placed to give the local reparation that needs to be performed.

THE REPARATIONS As previously noted, reparations can range from doing nothing (an empty branch, for example at the beginning of a plan where no outputs were generated yet), to automatic retractions via an `unexpected` (if the programmer considers that the unjustification should not the safety net retractions are appropriate for the given situation), to a specific reparation (that can eventually end with an `unexpected`, but the risk is that it may undo some of the reparation steps if they count as outputs).

Entering the reparation mode, i.e. *unjustified*, does not guarantee plan stops, for example when the branch leads to more nodes and branches, or simply in the case of long reparations. The result is that for example if a goal requests the cancellation of its plan, it may need to wait for a long time for the plan to stop so that the goal can go on living. The solution can be to simply consider that the unjustification signal is enough, or to include a timeout for the unjustification itself and to kill the plan afterwards. For now we use the first.

FROM RTS TO PLANS As we note in Sec. 4.2, our approach operates a change in the level of abstraction, from RTs to plans. A plan can therefore be comprised of multiple RTs, possibly executing simultaneously. This means that in case of unjustification, there may possibly be more than one RTs that perform reparations for the given plan and due to the way ALMA is conceived, these will be performed sequentially.

UNJUSTIFIED AND UNEXPECTED While essentially different tools, `unexpected` and *unjustified* both mark unusual executions and they both imply the end of a plan. The *unjustified* is the ALMA compulsory reparation branch used in the *wait* and *add rules* nodes to indicate the desired reparation behaviour in case the plan is no longer required to execute in the current context. The *unjustified* branch needs to be written with the idea of undoing any plan action and then ending the plan. The *unjustified* branch is reached when an error occurs in a related plan, as well as when the actual program logics caused the execution context to no longer be justified. The `unexpected`, on the other hand, is an imperative keyword that stops the plan immediately and triggers the corresponding recovery measures. The possibility of placing `unexpected` in the *unjustified* gives the programmer the ability to retract the outputs of a specific plan as a reaction to an unjustification, possibly another unexpected error.

4.3.3.3 *Reconfiguration*

The addition of goals to ALMA, as described in Sec. 4.2 provides the necessary mechanisms for the agent reconfiguration. After a plan ends its execution, regardless if it was a normal end, an unjustification or an unexpected, the execution will be conditioned by the evaluation of that goal's satisfaction condition.

CONCLUSION This concludes the presentation of the three fault tolerance phases in ALMA+. We shall now continue with the description of the impact these have on the execution platform.

4.4 EXTENDING THE PLATFORM

There are two directions for extending the platform: (1) the language modifications as well as (2) the safety net properties need to be supported by the platform.

4.4.1 *Language Extension Support*

The goal adoption and automaton were implemented using RTs (described in more technical details in Appendix A). As a consequence, the ALMA RT context is automatically transferred from plans to goals and then to other plans, maintaining its base function.

While when modelling agents we discuss plans, they are based on RTs at ALMA level so the language properties are the same.

4.4.2 *Safety Net Support*

The study of the ALMA+ language for the application of the safety net approach raised a series of issues that need to be covered by the platform executing the code.

For the detection and confinement phases, the error classes described in Sec. 4.3.1 need to be covered using specific mechanisms:

- For the error classes 1 and 2 concerning code "crashes" and memory overruns, the "external" code executions need to be guarded by a generic exception catching mechanism that ensures that the error is confined to the precise code that was executing.

- For the error class 3, which we did not yet implement, the "external" code execution needs to be guarded by a timeout, which in Prolog could be easily achieved with a `wait`. As stated before, the issue here is defining the timeout, which cannot be generic as an arbitrary value would be impossible to cover all cases. Such code timeouts are also too low grain for them to be a feasible programming solution, which leaves the question open for further investigation.
- In the same situation, the error class 4 on plan timeouts needs to be controlled when launching a plan, probably in the goal automaton.
- For the error class 5, the `unexpected` keyword, the corresponding safety net reaction needs to be ensured by the platform by unjustifying the "ok(Pl)" assumption.
- For the error class 6 – agent memory full – the agent memory handling mechanisms need to be adapted to avoid the whole agent blocking in case this error occurs. This case is also subject to future work.
- For the error class 7 concerning a global inconsistency, the only reaction being to stop the agent, the "ok(Agent)" assumption needs to be retracted.
- For the error classes 8 – "normal" inconsistencies – and 9 – goal not achieved –, there is nothing to add as they are already taken care of by the current implementation.

For the recovery phase, in particular the handling step, tables 9 and 10 summarise the requirements for each execution event (Table 9) and error event (Table 10) that the platform needs to ensure, trough a mix of rules and mechanisms in the ALMA+ interpreter. In Sec. 5.3 we propose an implementation for these requirements.

4.4.3 *Agent Architecture*

The resulting ALMA+ agent architecture is presented in Fig. 33. The programmer is required to provide the goal and plan definitions. The goal automaton and Means-end analysis[6] are already available as part of the ALMA+ extension, while the reasoning and communications facilities were already part of ALMA, as presented in Fig. 19 (Sec. 2.4.4). The safety net mechanisms are represented in red and correspond to the error catching at goal, plan and knowledge base level, sending and receiving error signals and writing rules to control the execution.

4.5 DISCUSSION

In the previous chapter we presented our safety net approach for which we established a set of 10 principles. In this chapter we introduced a programming language and execution platform that comply with the requirements of the safety net approach. These choices do not modify the design requirements presented in the previous chapter.

The programming language that we propose for this instantiation of the safety net approach has, as discussed in Sec. 4.1, good fault tolerance properties to start

6 As stated before, the MEA can also be written by the programmer, but we did not consider this case for the moment.

Table 9: "Safety net" specifications for normal execution

Event	**Specifications**
New goal	Link the goal "execution" to its "parent" plan
	Create the support for aborting the goal and unjustifying its plans and verification rule in case of error in the rule
New plan	Link the plan execution to its "parent" goal
	Create the support for stopping the plan and unjustifying its outputs
Received (used) message	The message is used under the assumption that it was sent in correct conditions (no error); the agent can be informed through a message from the original sender if the message is unjustified
Sent message	Be prepared to inform the receiver of the message in case of error-based unjustification (a normal unjustification does not cause this message to be sent)
New rule "Beliefs_in $\wedge$ Code $\Rightarrow$ Beliefs_out"	Ensure the rule is no longer applied in case of error in the plan
	Ensure that the rule does not even re-attempt to execute its code in case of error in the rule code
New assumption	Ensure it is retracted if the plan encounters an error (just as for rules)
Goal ends	Nothing to do
Plan ends	Stop waiting for local unjustifications for sent and message ones received messages

Table 10: "Safety net" specifications for error events

Event	**Specifications**
`unexpected`	Stop plan and trigger recovery
Decision code crash	Confine the error to the concerned code, stop plan and trigger recovery
Rule code crash	Confine the error to the concerned code, block the application of the rule (avoid reattempting the rule)
	In a goal satisfaction test rule: confine the error to the concerned code, stop goal and declare it *failed*
Agent memory compromised	All running plans become "not safe"

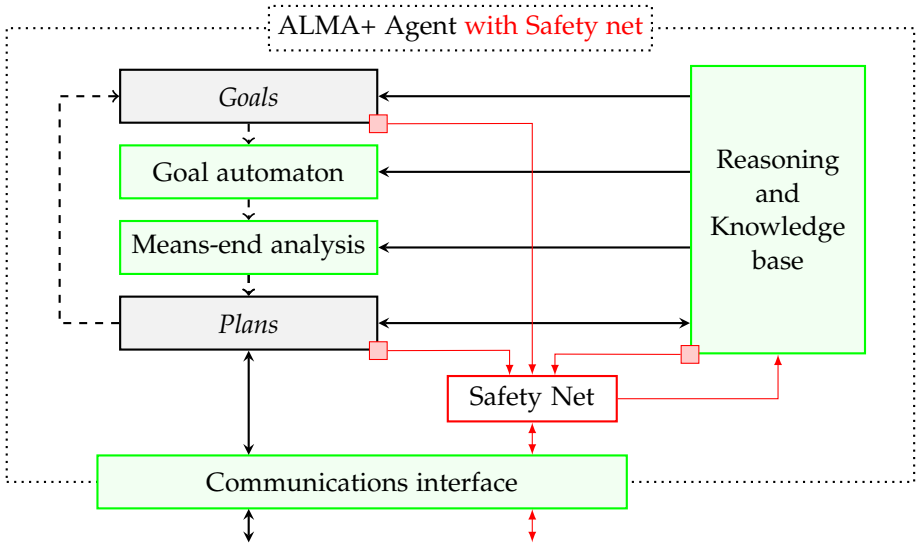


Figure 33: The architecture of an ALMA+ agent with safety net. Black rectangles represent the components provided by the programmer (goals and plans), the safety net mechanisms are represented in red and the rest (green) are provided by the platform. Dashed lines represent the execution flow, while full lines are for information exchanges (messages, belief reads, rule writes).

with, both at language and platform level. On top of those, we use the language's reasoning capabilities to integrate the mechanisms required for the safety net approach, in particular those concerning the dependency handling step at platform level. Therefore, ALMA+ and its modified platform comply with the safety net principles:

- Language requirements:
  - ✓1. ALMA+ does support the goal-directed agent paradigm.
  - ✓2. ALMA+ does have an exception-based system for handling errors.
  - ✓3. ALMA+ does contain regular reparation procedures in the form of the *unjustified* branches.
  - ✓4. ALMA+ does require systematic timeouts for its *wait* nodes.
- Platform requirements:
  - ✓5. The ALMA+ safety net platform does ensure confinement between the agent code and the operating system, as provided by Prolog. The fact that each agent executes in its own Prolog thread ensures the horizontal confinement – i.e. between agents.
  - ✓6. The ALMA+ safety net platform does catch all unanticipated errors, as studied in this chapter.
  - ✓7. The ALMA+ safety net platform does perform transparent dependency tracking that is used to trigger reparations and reconfigurations in case of unanticipated error.

ERROR EXAMPLES Let us now discuss a few error examples, some of which already mentioned in this thesis.

First, let us see how the safety net approach handles the examples given when we defined the concept of unforeseen fault in Sec. 1.4:

1. Residual code error ("bug"), uncaught exceptions: "segmentation fault", division by zero etc.: both in the generic safety net approach, as presented in Chapter 3, and in the ALMA+ instantiation of the approach, these errors are caught and handled at plan level, triggering reparations in any plans that depend on the "incriminated" one, including in other agents. Examples of such errors will be seen in Chapter 5.

2. System error: an error code interpreted as data, as in the failure of the Ariane 501 rocket [73] where the two inertial sensors failed in the same time due to a common software bug. This caused the rocket to abruptly steer in order to correct what it thought was a completely wrong direction. After the initial forces started tearing apart the fuselage, the automatic self-destruct mechanism was automatically triggered. Bad specifications in a system can mean that an error can exist and manifest without any meaningful reaction. Such system-level faults can be avoided through good design philosophies. A similar error situation could cause a safety net ALMA+ design to use a *default* branch when receiving the unkwnown error code, followed by a recovery that would most likely still lead to a foreseen self-destruction of the rocket.

3. Hidden variables: when Darwin's men were unable to cook potatoes as they were not aware of the influence of the altitude on the boiling point of water: a goal-driven approach would have them find other solutions. For example the goal to cook potatoes could lead them to cook them without using water (e.g. placing them on the hot coals), and even moving a level higher in the goal hierarchy and trying to cook something else.

4. Unconsidered situation: an important computer for the system in question stops (for example the power cable is disconnected): timeout conditions would avoid the other agents blocking, and a sufficiently-redundant design would allow the agents to find alternative agents on other machines to achieve their goals.

In the robot example from Sections 3.2 and 3.3, one of the ideas to note is the importance of providing a form of redundancy, e.g. functional redundancy – certain tasks can be performed either using the camera or the sonar. As we discussed in the corresponding section, the agent architecture that we used was beneficial for the fault tolerance of the resulting system.

In case an anticipated error is treated inappropriately by an existing – usually programmer-provided – handler, the system may end up reacting well either because the handling did eventually cause another error event (a "crash"), or that a goal is not achieved despite de handling. Once again, the safety net would provide the property that we are looking for: tolerance to unforeseen faults.

A dreaded fault in computer systems comes from radiations that can randomly change bits of data, a particular problem in very large computers [43] as well as in space missions where the Earth magnetic field and atmosphere no longer protect the computer chips. Such random bit changes may well fall in the domain of unforeseen faults and our safety net may bring a solution for the tolerance to such faults.

WHY WE DO NOT EXPLICITLY CONSIDER COMMUNICATION ERRORS Given that the communication errors are very common in distributed systems and multi-agent systems, they are a constant discussion point in works on fault tolerance in

these fields. This also means that they are quite well catered for, with many solutions at various levels, from checksums at low level to Potiron's work at behaviour level [86, Chapter 5]. These errors include lost, late and modified messages. When designing agents, a lost message, together with a refuse to reply, should be seen as a possibility and not an unforeseen fault. Late messages can be coped with to some extent thanks to our reasoning rules. Modified messages are a complex problem and our mechanisms will react only if they are bad enough to create a code crash. All these, however, can be treated with our error handling approach if the programmer uses the *unexpected* keyword with a communication specific detection mechanism.

This concludes the description of our safety net approach. In the next chapter we present an experimentation of the approach on a CNP-based scenario.

5

EXPERIMENTING

In the previous chapters we defined the characteristics of the safety net approach. In this chapter we will present a scenario and its implementation following the safety net principles. For this, we will use ALMA+ and its adapted execution platform described in the previous chapter, which we showed that are compliant with the safety net requirements. We will then study the system behaviour by testing various error situations.

This experimentation serves as a concept proof as well as illustration for the safety net approach.

5.1 THE CNP+ SCENARIO

In order to illustrate our safety net approach, we need a scenario that is simple enough to be understood, implemented and controlled, and yet that offers enough details and complexity for the approach to be evaluated in a realistic context. The scenario needs to allow the definition of multiple communicating agents, each with a sufficient internal complexity so that the use of goals and plans is justified. For these reasons, we will use a scenario based on the Contract Net Protocol (CNP) [106] which we call *CNP+*.

SCENARIO DESCRIPTION The scenario starts with an initiator agent *I* broadcasting *CFP* (Call For Proposals) messages towards possible main contractors MC_i, which then initiate negotiations in order to form teams of workers W_j for the job. It is thus a CNP on two levels, with two calls (Fig. 34): a first call for proposals CFP seeking one single subcontractor and a call for workers CFW_i for each MC_i looking for a team. In our example, we limit the number of winners for each level to one. Note that we use different indices (i and j) to mark the difference in numbers (there is no correlation between the two) between the MC_i and W_j agents. Once an MC_i has a possible team to work with, it replies to the chosen worker(s) with a $Preliminary\ Accept_j$ message and rejects all the others, then sends the proposal to the initiator. If its bid is selected, the MC_i confirms the deal to its workers and waits for the results of their work that it can then process and deliver to the initiator. Otherwise, it informs the workers that the deal is cancelled. The object of the CFP is not relevant to our illustration and will therefore be ignored.

5.2 MODELLING THE AGENTS

The system was designed with the safety net design requirements – principles 8-10 – in mind:

8. The programmer uses a **multi-agent architecture** featuring a **significant number of agents** with respect to the application.

9. The programmer uses **goal-driven agents** whose behaviour is split into **multiple goals and plans**.

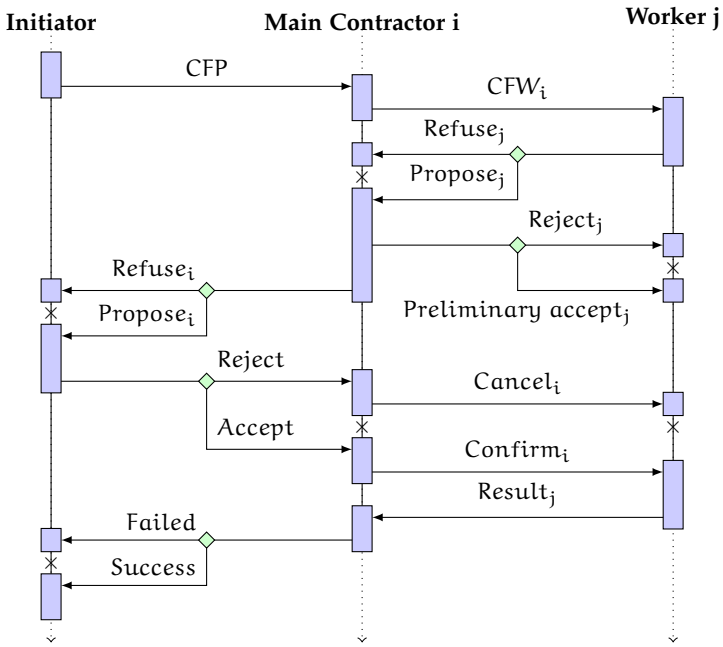


Figure 34: Two-level CNP+ protocol description

10. The programmer takes into consideration **redundancy**: allowing goals to retry plans, providing alternate plans or agent designs etc.

To achieve the desired level of granularity, we aimed at designing agents with multiple goals and plans, using plans that are short and simple. Each agent has a main "goal plan" which contains only goal adoptions and other nodes such as *decision* and *wait* nodes, but no actions. This allows an easy description of the high level agent behaviour and also marks the checkpoints in the behaviour in the form of goals. We will detail this approach in Part III of this thesis.

In the following we briefly describe the agents using the ALMA representation enriched with goals, ALMA+, as discussed in Sec. 4.2. The complete agent models (including the modifications discussed in Sec. 5.2.4) can be found in Appendix B.

5.2.1 *The Initiator Agent*

The Initiator agent has a need (e.g. a computational task) that it will fulfil through a Call for Proposals. The call will be sent to agents which are initially unknown to the Initiator, so the first step is to acquire a list of eligible contractor agents (goal G1). It then obtains solution proposals from these agents through goal G2, which it can sort (goal G3) in order to find the winner agent. All the others are notified of their failure to obtain the job (goal G5), while the winner is asked to provide the results (goal G4). The goal plan in Fig. 35 clearly reflects this behaviour. In the plan, the ALMA+ constraints for the *wait* nodes force an interrogation from the programmer for situations which may or may not occur: *timeout*, *unjustified*, besides from the more obvious achievement and sometimes failure of a goal. In this version of the scenario model, we only adopt the main goals and let the plan end if anything goes wrong (e.g. goal G1 fails). The goal paradigm does allow us to consider reparations, for example in case the sorting takes too long or the CFP is no longer justified, all agents that sent proposals are informed that they lost the bid (goal G6) – this is a

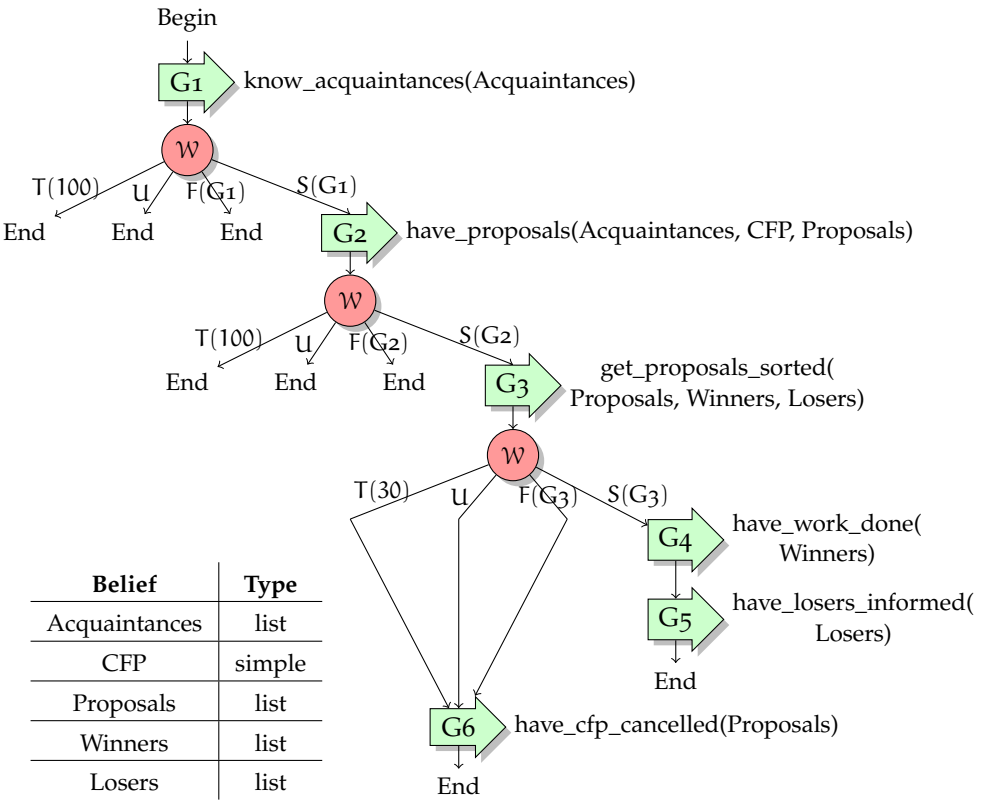


Figure 35: The Initiator agent's main goal plan P_{I0-1}

specific treatment for a clear case, a foreseen fault. Note that for clarity, the main beliefs used in the model are listed together with their types.

Goal G1 – "know_acquaintances" (Table 11) is successful if when its plan finished executing, the list of acquaintances contains at least two agents. Its plan (Fig. 36) simply sends a request to a DirectoryReference (in our case a Yellow Pages agent) and waits for a reply. A timeout ensures that the plan finishes in a finite time.

Goal G2 – "have proposals" (Table 12) has a plan P2-1 (Fig. 37 top) that adopts for each acquaintance a goal G2-1 that deals with that acquaintance and then waits for a certain period for the calls to expire, regardless of the outcomes of the adopted sub-goals: there is no blocking condition on the replies from the other agents and the success of G2 is only conditioned by the receiving of at least one proposal. After the *wait* node in P2-1, the list of proposals can be considered final and is

Table 11: Goal G1 – "know_acquaintances" of agent I

Goal Description		
Name	know_acquaintances	1
Satisfaction	$plan\ done \wedge length(Acquaintances) > 1$	
Means-end analysis	Ordered list	
Time out	150s	
Required beliefs	$\emptyset$	
Produced beliefs	Acquaintances	list
Plans	P_{I1-1}	Action plan

Belief	Type
DirectoryReference	simple
Acquaintances	list, OUT

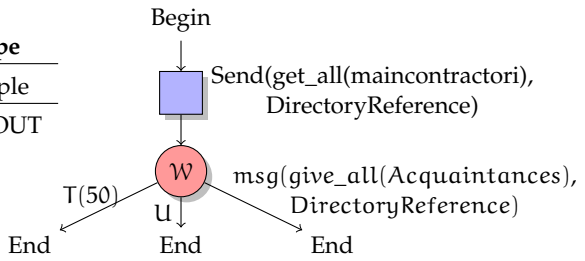


Figure 36: Plan P_{I1-1} – "know acquaintances" – for enquiring *DirectoryReference* (e.g. another agent, a web service etc.) for a list of all agents of type *maincontractori*.

Table 12: Goal G2 – "have proposals" of agent I

Goal Description		
Name	have_proposals	2
Satisfaction	plan done $\wedge \neg$ empty(Proposals)	
Means-end analysis	Ordered list	
Time out	150s	
Required beliefs	Acquaintances, CFP	list, simple
Produced beliefs	Proposals	list
Plans	P_{I2-1}	Goal plan

therefore written in a single assignment belief, which allows the rest of the process to continue under the implicit assumption that the proposals do not change. The plan P2-1-1 sends a CFP message and then waits for a reply, being very similar to P2-1, but we marked explicitly the fact that it writes the received information to a set rather than simple belief.

Goal G3 – "get proposals sorted" is charged with finding a list of winners – containing one element in our case – and will be successful only if it can produce that list.

Goals G4 – "have work done" and G5 – "have losers informed" are adopted in parallel, the former using a plan containing a request – reply exchange with the winner agent, while the latter's plan is `foreach` message sending. Goal G6 is very similar with G5, but uses the list of proposals as senders list. Note how one of the situations in which G6 is adopted is as a reparation measure following an unjustfication during G3.

This multiple level goal-plan structure for the case of a successful call with received results can be seen in the goal-plan hierarchy in Fig. 38. This is similar to a Goal-Plan Tree (GPT)[1], but here, we only use a simplified version where we do not indicate the sequence or parallelism between goals, focusing only on the goal-plan parent-child relations. Consequently, we also omitted the AND-OR logical relations. This is also because in our representation, the goal satisfaction tests and the fact that not all sub-goal outcomes are tested in the goal plans means that the relationships are more complicated than in the original GPT. Here, all child nodes of a plan are sub-goals adopted by that plan.

1 Formalism described in Sec. 2.3.4.

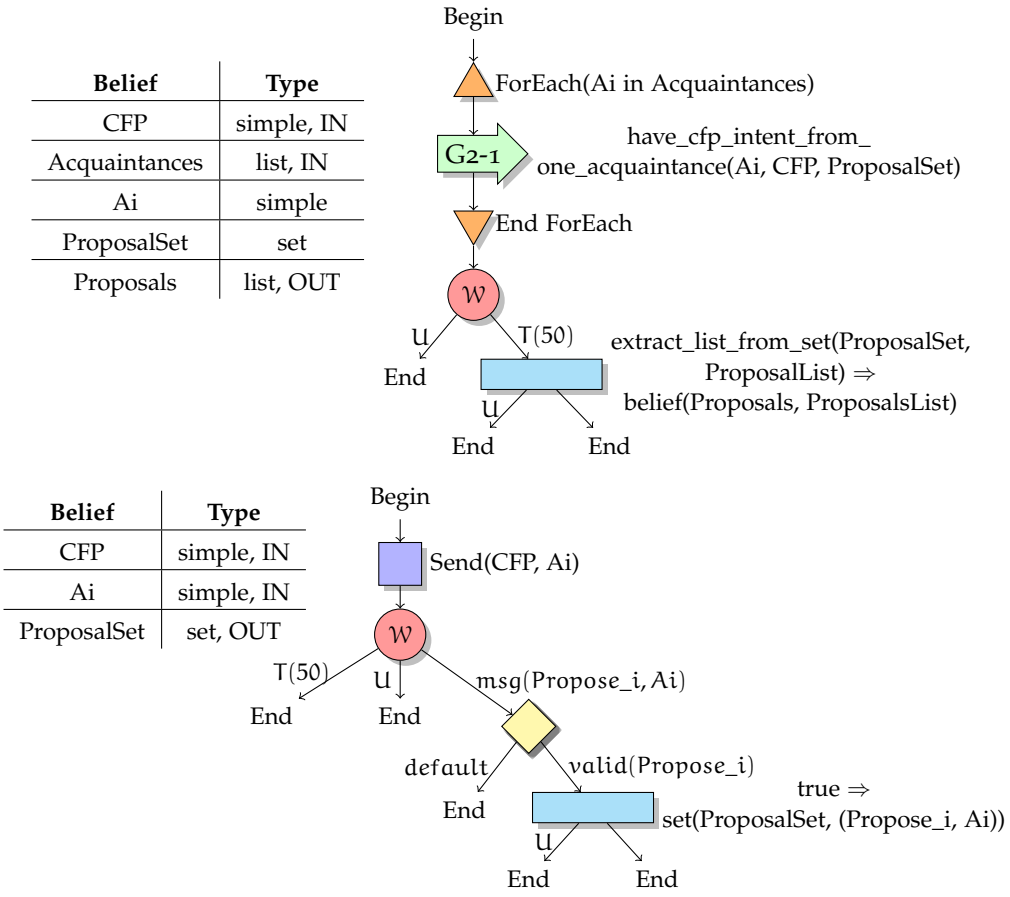


Figure 37: Plans P_{I2-1} "have proposals" and P_{I2-1-1} "have cfp intent from one acquaintance" of the Initiator agent.

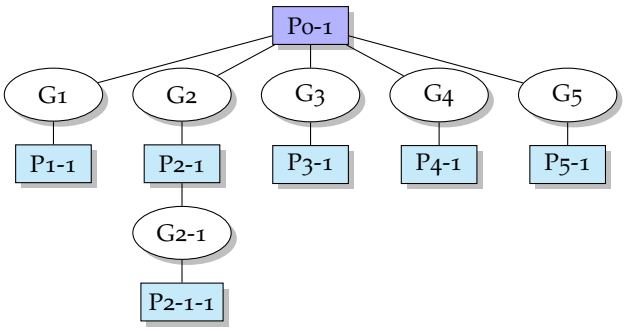
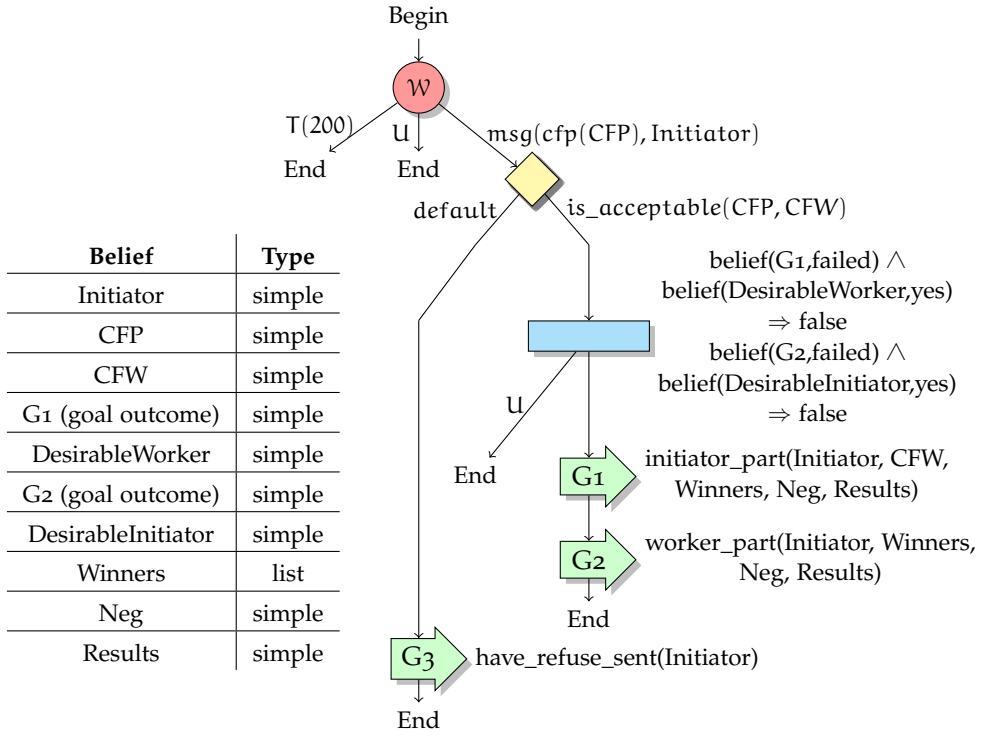


Figure 38: The goal-plan hierarchy for the initiator agent for a successful call (results are received)

5.2.2 *The Main Contractor Agent*



Belief	Type
Initiator	simple
CFP	simple
CFW	simple
G1 (goal outcome)	simple
DesirableWorker	simple
G2 (goal outcome)	simple
DesirableInitiator	simple
Winners	list
Neg	simple
Results	simple

Figure 39: The Main Contractor agent's main goal plan (P_{MCi0-1})

As it is both at the receiving and the initiating end of calls, the Main Contractor agent performs two functions: it has a worker function with respect to the Initiator agent and an initiator function with respect to the Worker agents. To express this duality, the two functions are represented by two goals that execute in parallel, G1 for the initiator function and G2 for the worker one. As can be seen in Fig. 39, the two goals are strongly connected as the failure of one renders the other useless – this is expressed through the two rules added just before their adoption. $belief(G1, failed) \wedge belief(DesireableWorker, yes) \Rightarrow false$ translates into saying "if G1 fails then G2 is no longer desirable", as otherwise a contradiction would be reached. This representation for the Main Contractor agent also allows us to exemplify the dependency of two plans that share beliefs, because each of the plans P1-1 and P2-1, the plans corresponding to the goals G1 and G2, use beliefs written by each the other (beliefs Winners allows P2-1 to know that P1-2 has successfully produced a list of workers, while the belief Neg signals P2-1 that the Initiator accepted this agent's bid and the work can start). `Is_acceptable(CFP, CFW)` is a predicate that returns `true` and instantiates the CFW if local conditions are met for the MCi to try and reply positively to the CFP (e.g. demand parameters are within acceptable limits, resources to spare etc.). Plan P1-1 compares favourably with the main goal plan of the Initiator agent so only the different section is showed in Fig. 40. Note how G1-7 is used as a reparation in case the plan is unjustified while waiting for the outcome of the negotiation, but also in case of timeout.

Plan P2-1 (Fig. 41) waits for a list of Winners before adopting goal G2-1 – "have proposal sent". This goal is successful only if the Initiator accepts this agent's bid, which allows P2-1 to set the Neg belief that causes the P1-2 plan to request the results from the Worker agents. When these results are acquired, the "worker part"

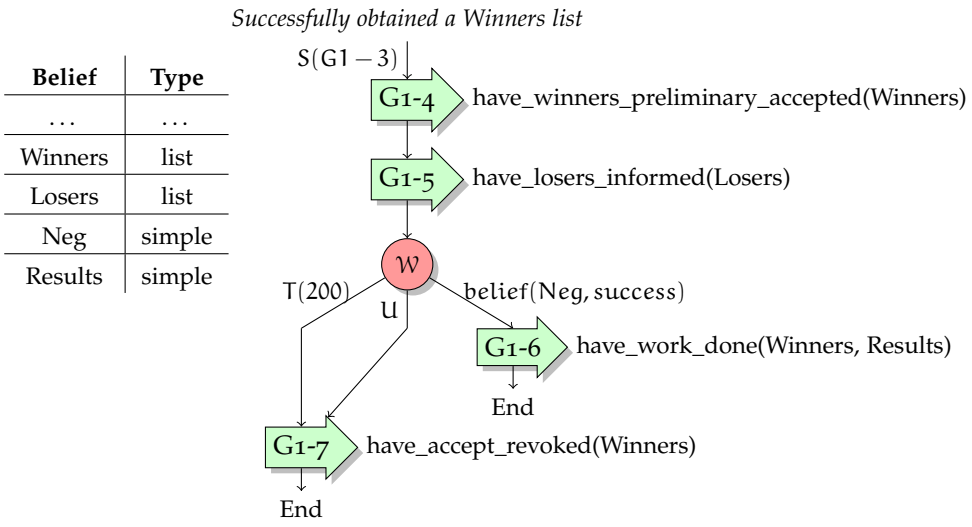


Figure 40: Extract from plan P_{MCi1-1} "initiator part". This goal plan is very similar to the one in the Initiator agent, but with supplementary elements linked to its interaction to the worker part: the agent needs to wait for the negotiation with the Initiator agent to be successful before giving the green light to the Worker agents.

plan can adopt the goal G2-2 to have them sent to the Initiator. In P2-1, the adoption of G3 to refuse the CFP is a reparation, used in case of unjustified or timeout while waiting for a winner to be chosen among the Worker agents.

The goal-plan hierarchy corresponding to a successful bid can be seen in Fig. 42.

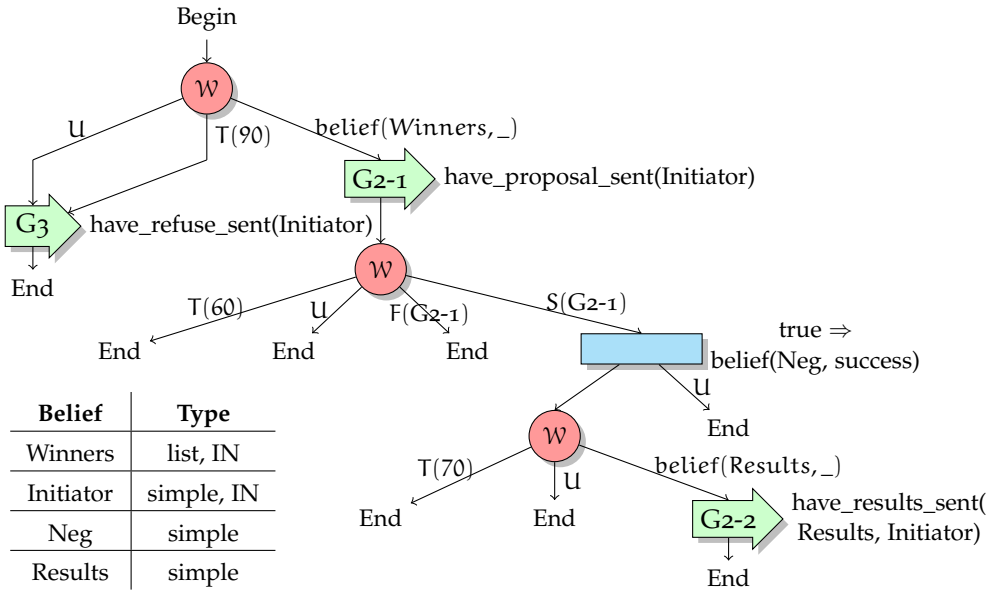


Figure 41: Plan P_{MCi2-1} "worker part". Setting the "Neg" belief to `true` signals to the initiator part that it can continue its CFW, while adding a contradiction to the desirable belief causes the initiator part to stop. This plan is very similar to the Worker agent's main goal plan.

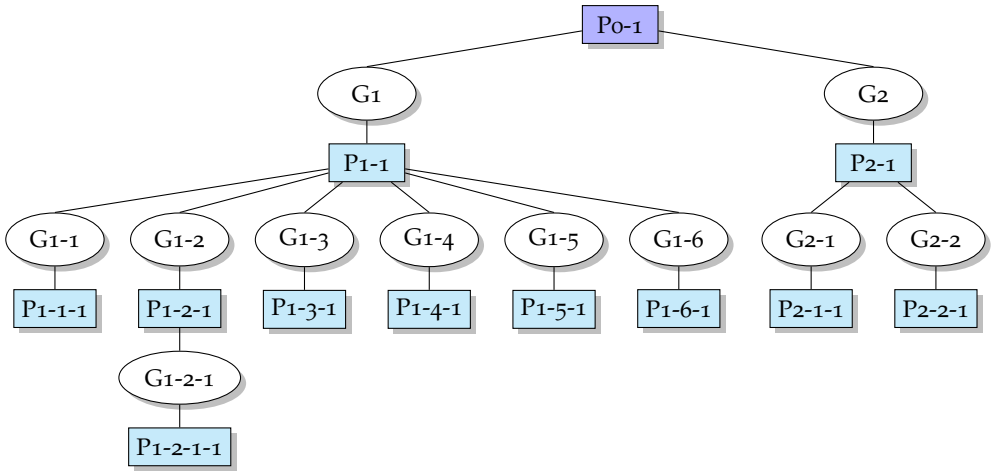


Figure 42: The goal-plan hierarchy for the main contractor agent in case it wins the bid

5.2.3 *The Worker Agent*

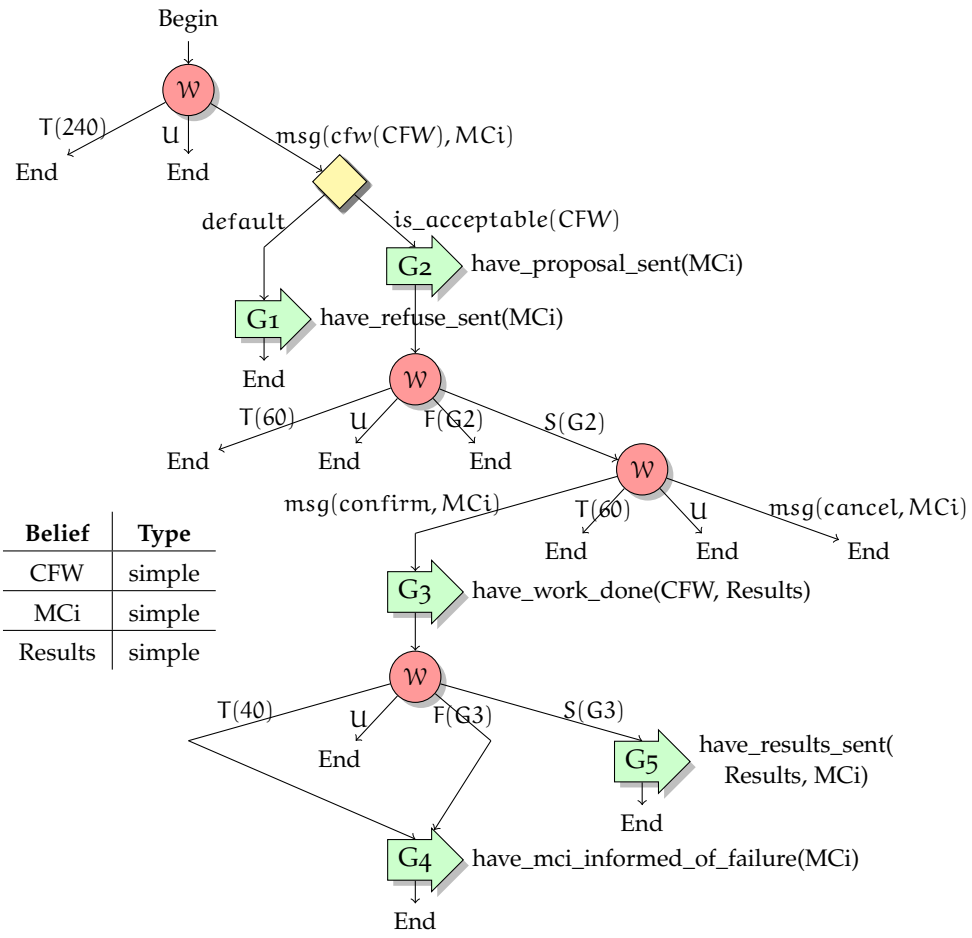


Figure 43: The Worker agent's main goal plan P_{Wj0-1}

The Worker agents are programmed to wait for a single CFW (Call for Workers) message and then evaluate it to see if it corresponds to its work criteria (e.g. if it can be done with the worker's resources). As can be seen in Fig. 43, if the answer is positive, goal G2 – "have proposal sent" will attempt to obtain the job from the MCi agent. The goal is achieved only if a "Preliminary Accept" message is received, in which case the Worker continues waiting for a second confirmation that would trigger the adoption of goal G3 – "have work done". In our case, this goal executes a computation predicate, whose result can then be transmitted back to the main contractor agent using goal G5 – "have results sent".

The goal-plan hierarchy corresponding to the three cases – (1) worker refuses the call; the bid is accepted and the agent (2) fails or (3) succeeds to perform the task – are presented in Fig. 44.

5.2.4 *Giving Unanticipated Errors a Thought*

As the subject of this work are the unforeseen faults and the first version of the scenario was already designed following the design requirements of the safety net approach, the only change that can be performed – but is not compulsory! – when modifying the platform is to add the `unexpected` keyword where appropriate.

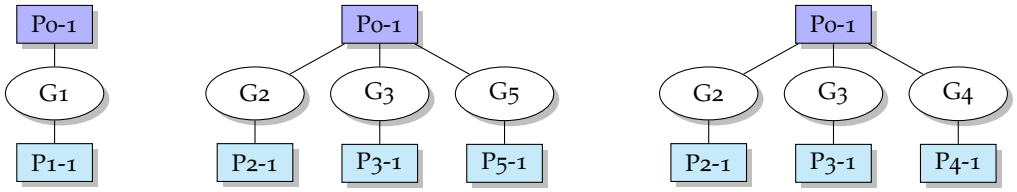


Figure 44: Goal-plan hierarchies for the worker agent in case it refuses the bid (left), it completes the task successfully (centre) or it fails to complete the task (right)

THE `unexpected` KEYWORD The CNP+ models were reconsidered in order to estimate where there was more appropriate to have an `unexpected` rather than just a normal *end* node.

There are cases that the programmer is confident the execution cannot reach (e.g. a timeout that is covered by a higher level "contract" and should never be reached, but it is required nevertheless by ALMA+). There are cases in which it is beneficial to leave the reparation to the safety net mechanism to clean up after the plan – for example by retracting messages or stopping goals. There are also cases in which the use of the keyword would not produce other results than simply using an *end* node, for example due to the fact that the plan did not produce any outputs to be retracted.

5.3 ADDING THE SAFETY NET MECHANISMS

THE PLATFORM From the perspective of ALMA+ integration and implementation, we have three aspects to consider:

1. the error detection and triggering the fault tolerance mechanism;
2. the connections that represent a dependency model of the actual system, including a "bridge" system for automatically unjustifying messages;
3. the integration with the language context through decisions in order to trigger the same unjustifications as the "normal" ALMA RT context.

Ideally, a second hidden agent memory (ATMS) would be used, but for the purposes of this demonstration we use the existing ATMS and ALMA+ mechanisms but with careful definition of the rules in order to keep the interferences to a minimum. This allows us for this prototype to minimise the number of added mechanisms by taking advantage of the RT context, for example when creating goals, starting the execution of plans, branching the plan execution and in the normal functioning of *decision* nodes. We therefore built the dependency context on top rather than in parallel with the RT context. In this way, the rules in Fig. 31 (from Sec. 4.3.3.1) are reduced to the ones in Fig. 45, because:

- rules 40, 41, 44 and partially rule 30 (only the part concerned with the context transfer, but not the one concerning the "ok(Agent)" assumption) are covered by the fact that goals and branching are all represented as RTs, which by default inherit each other's contexts;
- rules 35 and 37 are covered through the normal context enrichment in ALMA+.

$$
\begin{aligned}
\text{SD.NORMAL} = \{ & \\
\{ok(Agent) \wedge initial(Goal) \Rightarrow execute(Pl)\} & \quad (51)\\
\cup & \\
\{execute(Pl) \wedge ok(pl) \Rightarrow context(Pl_0)\} & \quad (52)\\
\cup & \\
\{context(Pl_n) \wedge rcv(Message, A_{from}) \Rightarrow context(Pl_{n+1})\} & \quad (53)\\
\cup & \\
\{ok(Pl) \Rightarrow send(Message, A_{to}, Pl)\} & \quad (54)\\
\cup & \\
\{send(Message, A_{to}, Pl_{from}) \wedge & \\
justified(Message) \Rightarrow rcv(Message, A_{from})\} & \quad (55)\\
\cup & \\
\{ok(Pl) \Rightarrow add\_rule(Rule)\} & \quad (56)\\
\cup & \\
\{add\_rule(Rule) \wedge ok(Rule) \wedge & \\
Rule = (\bigwedge_k B_k \wedge Code_{Rule} \rightarrow B) \wedge & \\
\bigwedge_k read(B_k) \wedge Code_{Rule} = true \Rightarrow read(B)\} & \quad (57)\\
\cup & \\
\{ok(Pl) \Rightarrow read(Assumption)\} & \quad (58)\\
\cup & \\
\{adopt(Goal) \wedge ok(Goal) \Rightarrow context(Goal)\} & \quad (59)\\
\cup & \\
\{context(Goal) \Rightarrow add\_rule(Goal.Rule)\} & \quad (60)\\
\} &
\end{aligned}
$$

Figure 45: The SD.NORMAL set for the dependency context added on top of the RT context

The *ad hoc* solutions for the specifications described in Sec. 4.4 are presented in Tables 13 and 14. Note that in this experimentation, we did not take into consideration agent level failures. The specifications were defined for and used with Sicstus Prolog 4.2.0.

5.4 THE SAFETY NET AT WORK

To illustrate the behaviour of the safety net mechanisms, we study error cases from two perspectives. First we discuss errors by confinement type, as presented in the previous chapter. Then, we study the location where these errors can occur and the variation of the system reaction. For each error we show the behaviour with and without the dependency handling mechanisms. We also discuss the gain obtained through the decision confinement reaction – stopping an entire plan in case of error in a decision code.

GRAPHICAL REPRESENTATION In order to facilitate the study, we represent the system using an adapted sequence diagram. In Fig. 46 two samples from a sequence diagram corresponding to the CNP+ implementation are shown. We use this representation not only to depict inter-agent message exchanges (in red in the diagrams), but also intra-agent dependencies such as goal adoptions and plan execution starts – black continuous arrows in the diagram –, plan and goal end events – black dotted arrows – and belief sharing between parallel not directly hierarchically connected plans – green arrows. Each agent type has a main goal plan which and multiple levels of goals with their corresponding plans. To better represent the dual function of the main contractor agent and the fact that its two goals G_1 and G_2 are executed in parallel, we draw them and their plans and sub-goals each on one side of the main goal plan.

Figure 47 is the complete diagram of a successful CNP execution. As the GPTs given when describing the agents (and with which they share the same colours), the sequence diagrams represent execution situations. Also, note how the representation by level helps distinguish the corresponding GPTs easily from a sequence diagram.

ERROR INJECTION As discussed before, we take advantage of the the existence of the two levels of abstraction in the ALMA language – the DAG structure and the "external" (Prolog) code. Due to the way the error catching and confinement mechanism is implemented, any type of Prolog-generated error can be caught and used to trigger a safety net reaction. This covers a large range of errors, from "bugs" in the code, an errors the input data, infinite loops and memory leaks. As the nature of the "unforeseen faults" can be varied, our focus here is not on listing or testing "exotic" faults, but to show credible cases and the reaction of the safety net to their manifestations.

The introduction of the "`unexpected`" keyword allowed us to easily "jump in the safety net" in various places in the plan for testing the varying reactions of the system.

For the errors involving an unreachable agent, simply killing the Prolog process was enough.

Table 13: "Safety net" specifications and the implementation for normal execution. Goal-plan and plan-goal links were not included as in ALMA+ they are ensured implicitly through context transmission.

Event	Specifications	Proposed ALMA+ implementation
New goal	Create the support for aborting the goal and unjustifying its plans and verification rule in case of error in the rule	At the beginning of the automaton (adoption RT): a decision executes `assert(is_safe(Adoption_RT_id))`, the rule `goal(is_safe(Adoption_RT_id)) => belief(Adoption_RT_id,ok)` is added and the belief is tested in another decision. Verification rules are then added as: `goal(is_safe(Adoption_RT_id)) ^ Beliefs_in ^ catch(Code, _, (retract(is_safe(Adoption_RT_id), fail))) => Beliefs_out`.
New plan	Create the support for stopping the plan and unjustifying its outputs	Plans are wrapped so that: (1) before their actual execution, a decision calls `assert(is_safe(Plan_id))`, the rule `goal(is_safe(Plan_id)) => belief(Plan_id,ok)` is added and the belief is tested in another decision; (2) `Plan_id` (the id of the wrapper RT) is transferred to all other RTs of the plan.
Received (used) message	The message is used under the assumption that it was sent in correct conditions (no error); the agent can be informed through a message from the original sender if the message is unjustified	Add `new_assumption(belief(Message,ok))` and then add it to the execution context (decision) just after the message use; an *action* node is then added to create an RT containing a `wait msg(unjustified(Message))` that can contradict the previous assumption on the message. The wait finishes when the current plan ends (one of the branches is `rt_end(Plan_id)`), as in this case unjustifications are not propagated further; message_id is sent with message (see below).
Sent message	Be prepared to inform the receiver of the message in case of error-based unjustification (a normal unjustification does not cause this message to be sent)	Add to plan wrapper (see new plan above) the creation of an RT that uses a set of all of the messages sent by the plan to send unjustifications for all sent messages in case the plan needs to be stopped from an unanticipated error. This RT's context contains only the parent context plus `belief(Plan_id,ok)`, and in case of unjustification it tests `goal(is_safe(Plan_id))` to confirm that it is indeed an error that caused this unjustification. The wait finishes when the current plan ends (as above). For each sent message, its unique identifier (Message_id=agent_name+plan_id+static counter) and recipient(s) are stored.
New rule "`Beliefs_in ^ Code => Beliefs_out`"	Ensure the rule is no longer applied in case of error in the plan	`goal(is_safe(Plan_id)) ^ Beliefs_in ^ Code => Beliefs_out` (but this does not cover errors in the rule - see next row)
	Ensure that the rule does not even reattempt to execute its code in case of error in the rule code	Execute `assert(is_safe_rule(Rule_id))` in a decision before adding the rule; `goal((is_safe_rule(Rule_id), is_safe(Plan_id))) ^ Beliefs_in ^ catch(Code, _, (retract(is_safe_rule(Rule_id), fail))) => Beliefs_out`; Rule_id=plan_id+static counter.
New assumption	Ensure it is retracted if the plan encounters an error (just as for rules)	`new_assumption(belief(Support_assumption, V)), goal(is_safe(Plan_id)) ^ belief(Support_assumption, V) => belief(Assumption, V)`
Goal ends	Nothing to do	-
Plan ends	Stop waiting for local unjustifications for sent and message ones received messages	Already covered above.

Table 14: "Safety net" specifications and the implementation for error events

Event	Specifications	Proposed ALMA+ implementation
`Unexpected`	Stop plan and trigger recovery	Use the keyword as the name of a *decision* node that executes `retract(is_safe(Plan_id))` and then ends (see New plan in Table 13)
Decision code crash	Confine the error to the concerned code, stop plan and trigger recovery	Execute code in `catch(:ProtectedGoal, _, (retract(is_safe(Plan_id)),fail))`. In order to stop the plan right after the error, add `goal(is_safe(Plan_id))^` to all decision branches and add a final branch with `goal(not(is_safe(Plan_id)))` that leads to `Unexpected` just before `default`.
Rule code crash	Confine the error to the concerned code, block the application of the rule (avoid reattempting the rule)	Everything already taken care of at the creation of the rule (see New rule in Table 13)
	In a goal satisfaction test rule: confine the error to the concerned code, stop goal and declare it *failed*	Everything already taken care of at the creation of the goal (see New goal in Table 13); an *unjustified* in the automaton leads to a *failed* goal.
Agent memory compromised	All running plans become "not safe"	Case not covered here

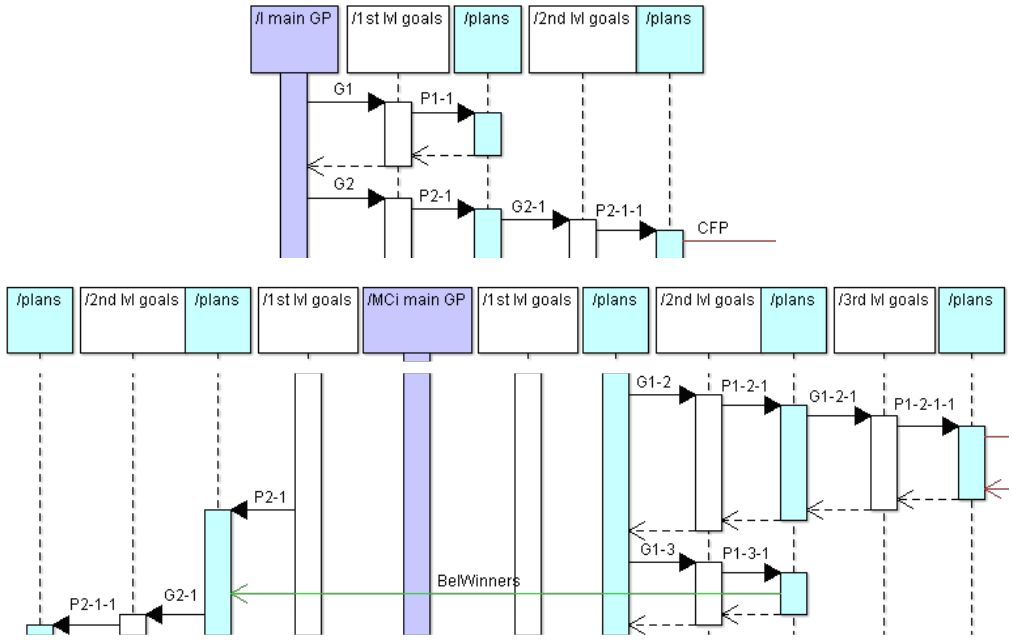


Figure 46: Sequence diagram samples. Top: a cut from the initiator (I) agent, bottom: a cut from the main contractor agent (MCi).

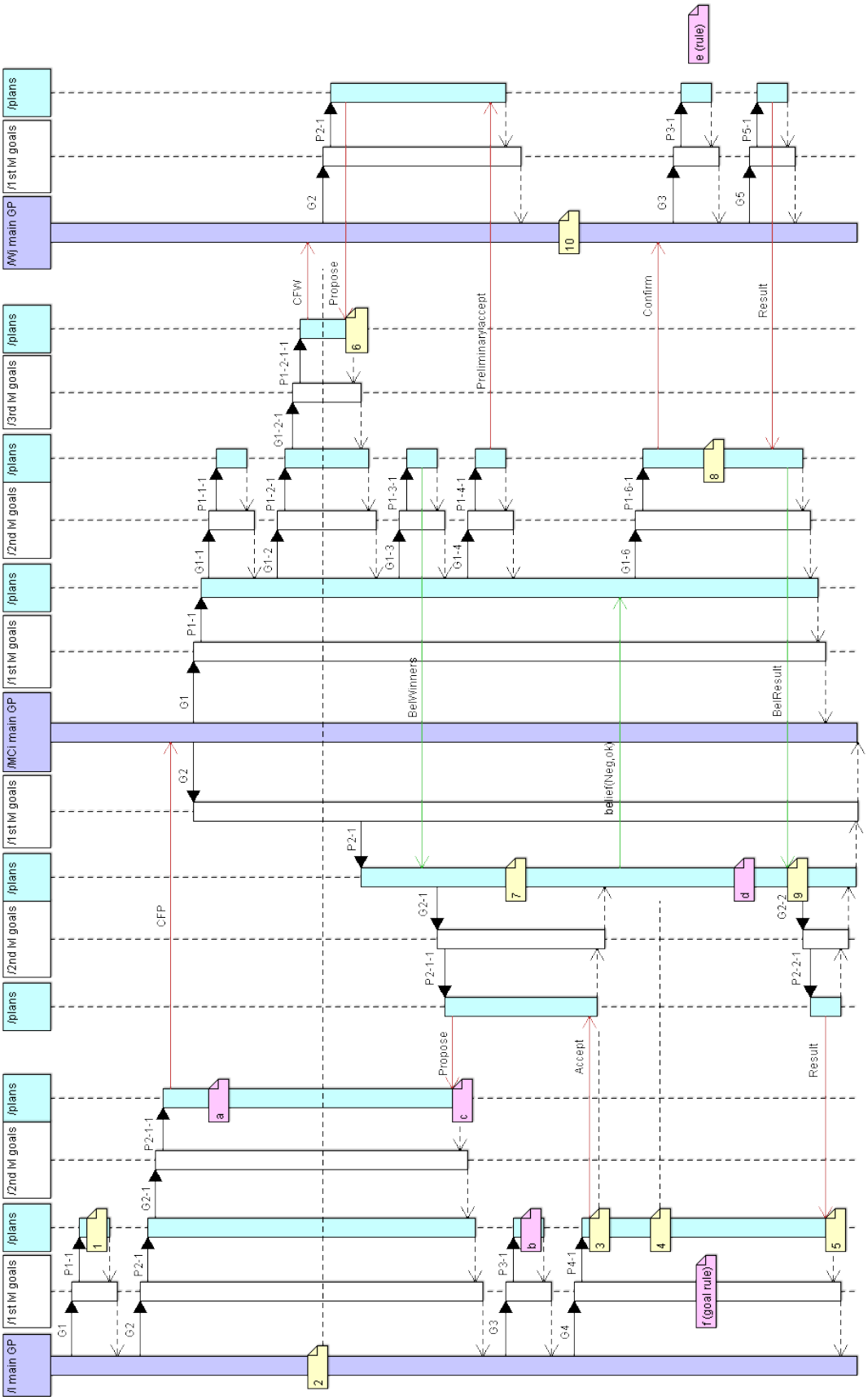


Figure 47: Sequence diagram for a successful CNP+ negotiation (the Initiator agent ends up with a result). Pink tags with letters correspond to the error cases from Sec. 5.4.1 and the yellow tags with numbers to the ones in Sec. 5.4.2.

Table 15: Error examples

	Error	Behaviour of safety net	
		without dependency handling	**complete safety net** (+ from left)
a	I: **decision crash** in P2-1-1	G2-1 may retry and eventually fail (MCi will cancel its deal with Wj only when its wait will timeout); G2 can still be achieved with the other instances of G2-1	The corresponding MCi is informed of the error which causes the corresponding MCi to cancel its bidding process with the Wj agents
b	I: **decision crash/error** in P3-1	G3 retries and if the error was caused by the input data and the decision code, it will reappear and the goal will eventually fail	No dependencies to retract
c	I: **`unexpected` in plan** P2-1-1 after adding the rule	As in case *a* above: G2-1 fails	The corresponding MCi is informed of the error and the rule is also retracted
d	MCi: **`unexpected` in plan** P2-1 while waiting for results	G2 will probably fail and cause G1 to stop as well, stopping G1-6 in the process	The rule corresponding to the Neg belief is retracted causing P1-1 to unjustify and stop G1-6, but the consequences are the same as in the normal case
e	Wj: **rule crash** for the rule of P3-1	Rule application is blocked to prevent error manifestation	Not concerned by the dependency handling step
f	I: **goal rule crash** in the test for G4	Goal is stopped (with its plan unjustified) and fails	Not concerned by the dependency handling step

5.4.1 *Study by Type of Confinement*

For this part of the study, the error examples are given in Table 15, with their identified with pink tags in Fig. 47. The examples correspond to three of the confinement types introduced in Sec. 4.3.2: at plan, rule and goals level. The agent level error and confinement case was not included in this experimentation.

PLAN LEVEL Cases *a* and *b* correspond to errors in the *decision* nodes, being taken care of by the plan-level confinement:

- the message validation code of plan P2-1-1 of the Initiator agent in case a, for example due to a bug in the code, an error in the message, or even a malicious interference with message content.
- the proposal sorting procedure in P3-1 of the Initiator agent in case b, for example because of a bug in the code, an infinite loop, or a memory leak.

In the base ALMA implementation, a code error would cause a whole agent to stop. This, while well confined from the other agents who continue functioning correctly[2], is a rather radical and dangerous outcome as other executing plans could still continue functioning, or at least go through a reparation phase before stopping. The next step is to confine the error to the affected code section and only produce a `false` outcome for the decision condition. In our two examples, the result would be the application of the *default* condition in the corresponding *decision* node followed by a plan end, which would not change much. However, the risk here is

2 Without even blocking if they were interacting with the affected agent, as all their *wait* nodes also contain a timeout event.

that an important decision is taken in this manner rather than through the actual code application, for example "`if (estimation(food resources)<10 days) turn back; else continue`" could have catastrophic consequences in case of error in the evaluation function. We therefore support the "let it crash" approach, as stated in the previous chapters, stopping the concerned plan and rely on the recovery mechanisms for the continuation, in particular the goals.

Cases *c* and *d* also concern plan-level confinement, but this time the error was signalled by the programmer through the "`unexpected`" keyword.

Between these four cases we can see various instances of application for the dependency handling, from no dependencies concerned, to local rules blocked to triggering recovery in other agents.

RULE AND GOAL LEVEL In case *e*, the error occurs in the code of a rule, while case *f* concerns an error in the goal verification rule: the *ok(Result)* test causes a crash. In both situations, the dependency handling is not involved, with only the confinement acting to stop the rule or the goal and its concerned rule from executing.

The same discussion as above on the confinement applies here as well: in case of error in the rule code, the goal is declared "failed" and the rule is retracted in order to avoid incorrect behaviours. Otherwise, as the rules do not apply correctly, they may or may not eventually produce results, and these may not be reliable (e.g. if a rule contained a code error or the input data that was causing the crashes was incorrect). In the case of the goal rule, the goal will attempt multiple plans until timeout or another condition, but the rule may also eventually succeed resulting in an achieved goal. However, our policy is to conclude that the goal is no longer reliable and to stop it.

DISCUSSION This first comparison also shows that the first implementation already exhibits good properties due to the compliance with the safety net design requirements and use of ALMA+. The addition of the dependency handling mechanisms helps the system recover sooner than the timeouts of the involved plans.

5.4.2 *Study by Location of Error Occurrence in the Agent Code*

Let us now study how the system reaction varies depending on the location of the error occurrence. The accent is on plan-level errors as they are the ones that take advantage the most of the safety net mechanisms (the error in a goal only triggers ALMA-like unjustifications, the error in a rule only blocks that rule). The retractions are caused by plan-level errors – decision crashes and the use of the `unexpected` keyword. Note that the cases discussed do not necessarily correspond to decisions or `unexpected` in the current version of our models, but were chosen due to their representativeness for the system reaction. The error cases are represented using yellow tags in Fig. 47, with the corresponding explanations in Table 16. These and other cases are listed in Appendix C.

HANDLING DEPENDENCIES The objective of the dependency handling mechanisms is to help trigger reparations in plans that depend on the one where the error was detected, but also to retract any outputs (usually rules, e.g. case 10) of these plans that were not yet used by any plans. The examples show different situations, the simpler of which involving a message that is retracted to trigger recovery in

the receiver agent (cases 3 and 6). In case 8, we also see that these recovery measures may involve stopping plans in the concerned agent. Rules are also subject to retractions (e.g. in cases 6 and 9) and even when this does not trigger reparations, it is still a good means to clean up the agent memory. The propagation is best seen in the error case 2 where the recovery is triggered on many levels in a step by step manner through `unexpected` calls, causing many agents to stop the bidding process when the Initiator agent encounters a problem. In case 7 we see that the error signal also propagates from level to level and to another agent, but the latter does not recover as the concerned plan is no longer running.

CASES WITH LIMITED EFFECT There are cases where the effect of dependency handling does seem limited. This is a result of the chosen goal-plan structure as well as the propagation policy. For example, in cases 4, 5 and at the end of 7, there is no longer a plan executing at the receiver side in order to react to the error signal. While in case 5, there is nothing that can and needs to be done as the agent already finished its job, the other two cases beg the question: "what if the other plans reacted to the signal, despite the fact that the original plan that used the messages was over"? This change of policy should be studied further, but looking at case 7, a different implementation or dependency handling policy could have caused the CFP to stop, causing a domino effect.

Also, note that in case 10, the plan does not have any direct[3] dependencies so there is no need for dependency handling.

OPEN SYSTEM CHARACTERISTIC For case number 1, we consider that the DirectoryService (DS) was not programmed using our approach and will not react in any way. This shows that there is compatibility with agents that are not programmed using the dependency handling mechanisms of the safety net, as long as the messages do not interfere with the normal protocols and do not fill up the inbox of the concerned agents.

COMPARISON In many cases, we can see that the dependency handling mechanism helps trigger reactions quicker than would be the case when relying only on timeouts. Timeouts are good, but they are "hardcoded" values and depend a lot on the hardware running the application, so while they are useful last resort measures, they should not be relied upon too much.

We note that the used CNP protocol was not at all modified in order to take into account failures, e.g. through the addition of retries.

5.4.3 *Other Error Situations*

Here are some other examples of possible error cases:

- a machine crash causing agents to disappear, an agent isolated due to communication failure, or simply one or more messages that do not arrive: these are all handled by the other agents through their timeout conditions.
- a message that is corrupted or false can cause a code crash, in which case it is handled as above, or may also fail a validation condition in which case the *default* branch gives the continuation (in the examples above, this simply means

3 One could argue that retracting the *Propose* sent through a sub-goal of the plan would be beneficial for the system recovery, but this is again a discussion of dependency propagation policy.

Table 16: Error examples by location of occurrence (DS stands for Directory Service)

	Error occurrence	Normal goal-driven reaction	Safety net (+ from normal)
1	I: During P1-1	G1 may retry and if it fails, P0-1 will stop gracefully (foreseen)	if the error occurred after the message to the DS was sent, retract message
2	I: Unjustified (or timeout) in P0-1, during G2, in time for at least one P1-2-1-1 of an MCi to be active	G2 is unjustified too because it inherited the RT context from P0-1 (goal continues in the case of timeout). No CFP is cancelled so MCi and Wj agents continue pointlessly.	`unexpected` causes all goal hierarchy to unjustify; the corresponding unexpected in any still running P2-1-1 propagates the error signal to the MCi agents unjustifying all their plans (as this concerns their P0-1) and for any active P1-2-1-1, the error signal is propagated to the corresponding Wj agent
3	I: During P4-1 (after *Accept*, but before P2-1-1 of MCi ends)	G4 may retry, but receiving multiple *Accept* messages is not included in the current model (however, in the absence of a strict message identification, the new *Accept* message would be ignored and the MCi agent would receive the *Result* which comes as a reply to the first *Accept*).	The *Accept* message is retracted, P2-1-1 of MCi is unjustified, then its goal, G2-1, will probably fail.
4	I: During P4-1 (after *Accept*, but AFTER P2-1-1 of MCi ends)	As above	The *Accept* message is retracted but with no consequences for MCi
5	I: During P4-1 (after receiving *Result*)	Any retries for G4 will end in timeouts in the current model as MCi already finished.	*Accept* is retracted, but MCi is already done.
6	MCi: During P1-2-1-1 (after writing Propose)	G1-2-1 may retry but the corresponding Wj is not informed to stop waiting for a confirmation; G1-2 can still be achieved with the other instances of G1-2-1; the written *Propose* remains enabled in the agent memory.	*CFW* is retracted for the Wj corresponding to the plan, triggering recovery in Wj; the written *Propose* is retracted too
7	MCi: unjustified (or timeout) in P2-1 while waiting for G2-1	G2-1 is unjustified too because it inherited the RT context from P2-1 (goal continues in the case of timeout). Furthermore, the I agent is not informed and may wrongly choose this agent as winner.	`unexpected` causes goal G2-1 to be unjustified, which in turn unjustifies P2-1-1 which because of its `unexpected` will retract the *Propose*; no reaction in agent I as P2-1-1 is over
8	MCi: During P1-6-1, after confirming and before the *Result*	G1-6 may retry but the protocol is broken and the outcome is not guaranteed.	The *Confirm* is retracted causing P0-1 of Wj to unjustify, also aborting any active sub-goals (e.g. G3 or G5).
9	MCi: During P2-1, just after receiving *Result*	Retries for G2 would result in a broken protocol.	`belief(Neg,ok)` is retracted, but no reparation is triggered as P1-1 finished.
10	Wj: During P0-1, while waiting for *Confirm*	the MCi agent waits for *Result* until its wait deadline	no direct dependencies of the plan, so no retractions

ignoring the message). A third possibility is that any of the corresponding goals fails and the agent reconfigures without going through an actual error state. If none of these apply, then the system continues with the error undetected.

- an infinite loop, as discussed in the previous chapter: in the plan it can reach the goal timeout or be detected and handled through the other agents; in the code it is handled by other agents (because the current agent is blocked due to the way ALMA+ handles parallelism).

5.5 DISCUSSION

In this chapter we presented a CNP-based scenario and its modelling and implementation using the safety net approach. Using this example, we studied the behaviour of the system in various error cases to show the benefits of the approach.

MODELLING AND DEVELOPMENT The modelling and development were done following the safety net principles using the ALMA+ language and notation executed on the adapted ALMA+ execution platform. The design requirements of the safety net approach were covered as follows:

✓8. The design of the CNP+ scenario does use a multi-agent architecture with several agents corresponding to the actors involved in the scenario.

✓9. The design of the CNP+ scenario does use goal-driven agents defined with a multi-level structure of goals and plans.

✓10. The design of the CNP+ scenario does exhibit time redundancy through the possibility to retry goals and space redundancy through the number of participating agents in the case of MCi and Wj agents.

To facilitate this process and gain in readability, we chose not to place goal adoptions and actions in the same plan, thus complying with the GPS approach that we present in Part III of this thesis.

For the autonomy aspect, as required by the ALMA+ model, timeouts were specified for all *wait* nodes. Furthermore, our extension also added timeouts with the goal definitions. These values allow the system to avoid blocking behaviours, both when it comes to interactions with other agents and when it comes to behaviours that do not produce the result expected by the goal, regardless of the reason, thus covering various unexpected faults.

We note that because different plans handle different parts of the protocol, retries, which are normal in a goal-plan architecture, can easily cause the protocol to be broken as long as there are no specific solutions in the protocol design. Our focus was on showing the way the system benefits from the safety net approach and in particular how the dependencies are handled. A solution for better protocol designs is using goal-oriented interaction protocols [13].

ENRICHING THE PLATFORM The introduction of the platform level safety net features was facilitated by the ALMA+ language structure and underlying platform. For the purposes of this experiment, we added dependency elements to the already existing RT context. However, a separate memory for the dependency context would provide a more robust mechanism and would be safer with respect to programmer interference.

ERROR DETECTION The structure of the language made it possible to reduce the error detection to a limited number of situations, with the most important being code errors in decisions and rules, followed by timeouts guarding various aspects such as waiting for events and goal achievement.

The `unexpected` keyword also proved a useful means to trigger the safety net reaction. The question that the programmer needs to ask when considering adding an `unexpected` node is "would the program be able to reach this point under normal circumstances?". Furthermore, if he or she was aware of the existence of the safety net including the chosen handling policy, he or she would also ask "would jumping into the safety net be beneficial in this case?", in other words "would the program benefit from the safety net in this case, for example through the retraction of this plan's outputs?".

CONFINEMENT The first aspect of the confinement was the system modularity. Agents were split into goals and plans corresponding to their local tasks. This provided a good confinement in case of errors, facilitating the reparation and helping limit the propagation of the error signal to avoid domino effect. A drawback was the fact that the protocol was not designed to cope with retries, which meant that even if the goal structure was robust enough to survive an error, the successful continuation was unlikely.

The active reaction to stop a plan when an error is detected was shown to be beneficial for the system design.

RECOVERY As stated before, the recovery with its three steps is a very important phase of the safety net approach.

The repair step with was visible through the *unjustified* branches which had varying outcomes, depending on the local needs of each plan. Several of them simply led to the plan finishing, possibly leading to a reconfiguration through the goal of those plans. Others were used to trigger the safety net mechanisms through an `unexpected`, often also helping propagate the dependency further when the unjustification was already the result of dependency handling. The *unjustified* branches were also used to adopt goals with reparation effect, for example G6 of the Initiator agent, used to inform the Main Contractor agents that the CFP was cancelled. Note that these *unjustified* branches are used both when the RT context is no longer valid and when a dependency has encountered an unanticipated error.

The usefulness of the dependency handling mechanism was demonstrated in the comparative study where it could be seen how it helps trigger recovery sooner, but also helps retracting useless rules.

Reconfiguration is always present through the goal-plan structure to eventually guide the agent behaviour, regardless of the presence or not of errors in the system.

CONCLUSION In conclusion, the safety net approach provides a development framework for designing systems with minimal overhead, yet the results are beneficial for the fault tolerance of the system, especially with respect to faults for which no handling was not included by the programmer.

This concludes the part of the thesis dedicated to the safety net approach. The third part of the thesis presents an approach for designing goal-driven agents.

Part III

CONTRIBUTION TO GOAL PROGRAMMING

The Goal-Plan Separation Approach

6 THE GOAL-PLAN SEPARATION

In this part of the thesis we provide an approach for programming goal-driven agents that brings more clarity to agent code. As we argue in Sec. 1.3, the fact that agent plans can in the same time adopt goals and perform actions creates a mix of between the reasoning and the acting parts of the agent. To counter this, we propose the Goal-Plan Separation (GPS) approach and we show how it benefits agent development.

In this chapter we present the original approach of this part of our work which is illustrated using two examples. Chapter 7 discusses implementation issues and gives examples of GPS-compliant plans. In Chapter 8 we present two examples of applications designed following the GPS approach: one in the domain of maritime patrol and the other for deploying Ambient Intelligence applications on a distributed infrastructure.

We continue by introducing a representation model from the literature which we use to illustrate our proposition through a first generic example. This allows us to discuss the consequence of the Goal-Plan Separation, followed by the more refined example of a Mars rover.

6.1 GOAL-PLAN TREES TO GOAL-PLAN SEPARATION

To illustrate the Goal-Plan Separation approach, we show in Fig. 48 two representations of the same agent side by side: a GPT[1] (a) and a possible GPS version (b). We chose to use the GPT representation because even if it is used more as an analysis than a development tool, it shows well the issues we are addressing, in particular how the goal and plan levels alternate. The plans that are the most important in the example at hand are *P1*, *P3* and *P4* as they are the ones that can contain both actions on the environment and goal adoptions. The new representation, which decomposes goals into sub-goals is an AND-OR tree (very similar to the one used in [76]) with only the leaf nodes having plans containing actions, but no goal adoptions. To save space, we consider that the default operator for the *AND* nodes is the sequence operator, unless stated otherwise, e.g. in the case of *SG23*. To preserve the original structure, goals are also allowed to be *OR* nodes, in order to depict cases where a goal or sub-goal can be achieved in more than one way. Similarly, goals that have more than one plan are *OR* nodes. While the original goals were preserved, the plans that were not leaves were replaced by sub-goals, e.g. *SG11*. To compensate, plan names of the form *P'* were used to indicate a variation of an original *P* plan which at least removes the goal adoptions. Note, however, that this exact transformation is not unique for the given example as it depends on the plan's specific features[2]. More examples can be seen in Sec. 6.3. *SG12* was introduced to avoid the existence of siblings of different types. This example shows that transforming an existing agent is possible. Nevertheless, as is the case with many

1 Formalism described in Sec. 2.3.4.

2 E.g. a plan that *turns on a sensor*, adopts a goal to *retrieve data* and then *saves that data*. Such a plan would rather transform into a main goal with three sequential sub-goals, the first corresponding to the beginning of the original plan, and the last corresponding to its final part.

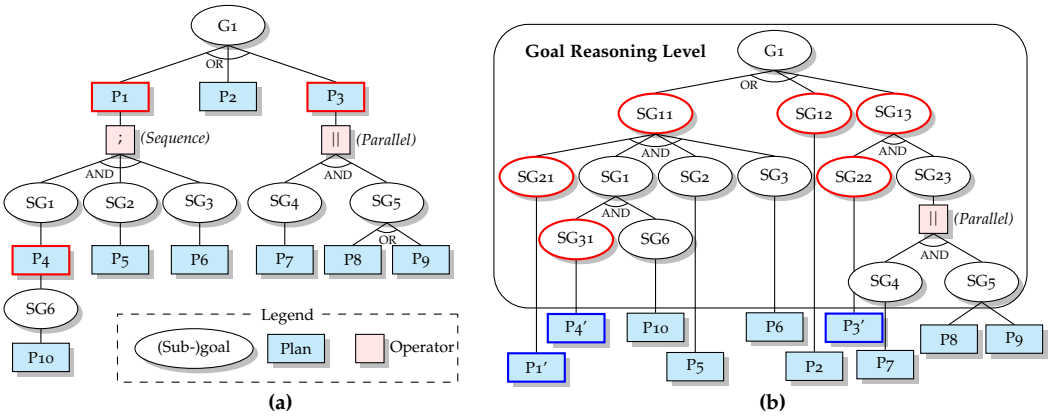


Figure 48: An example of Goal-Plan Tree (same as in Fig. 10) (a) and a Goal-Plan Separation of the same example (b)

such translations and as we discovered during the experimentation we describe in Sec. 8, a complete redesign of the agent produces a more appropriate result.

6.2 THE *goal reasoning level*

As can be seen in Fig. 48 (b), a direct consequence of the separation of goal adoptions from the actions on the environment is the appearance of two levels in the definition of the agent: a *goal reasoning level* and an *action level*.

The *goal reasoning level* is the part of the agent concerned with goal adoption, control, dependencies and interactions. Here we are concerned mostly with the specification (by a programmer or designer) of dependencies between goals and issues related to the adoption and life-cycle control. For the purpose of the Goal-Plan Separation, no actions on the environment are present at this level. However, as will be discussed further on, other mechanisms can appear at this level, e.g. for handling perceptions, events or various types of goal dependency.

6.3 MARS ROVER SCENARIO

To further illustrate the GPS, let us consider a Mars rover example from [108]. Figure 49 (a) represents a Goal-Plan Tree for a Mars rover's goal to analyse soil samples. The depth of the tree varies between *P7: ExpSoilByDelegationPlan* that is at a depth of one and *P6: TransmitTo(Lander)Plan*, at a depth of 5. While all leaf nodes are plans, there are also intermediary plans which adopt goals and can contain actions: *P1: ExpSoilBySelfPlan* and *P4: RecordResultsPlan*. If these two plans had no actions on the environment, the representation would be GPS-compliant as no unwanted action-goal adoption mix would be present. In this case, an alternative representation can also be obtained in the same manner as in the example in Sec. 6. As depicted in Fig. 49 (b), *P1* changes into a sub-goal and *P4* disappears completely as there is already *SG3* to regroup the corresponding sub-tree. For *P7*, a parent sub-goal *SG12* is created to avoid having two siblings of the *G1* node of different types, i.e. a goal and a plan. *SG12* also carries the precondition originally contained by *P7*.

Another approach would be to rewrite the Mars rover's behaviour in a format similar to the goal diagram from Tropos [44], as in Fig. 49 (c). The representation can also be seen as a type of plan. It starts with a decision node that corresponds to

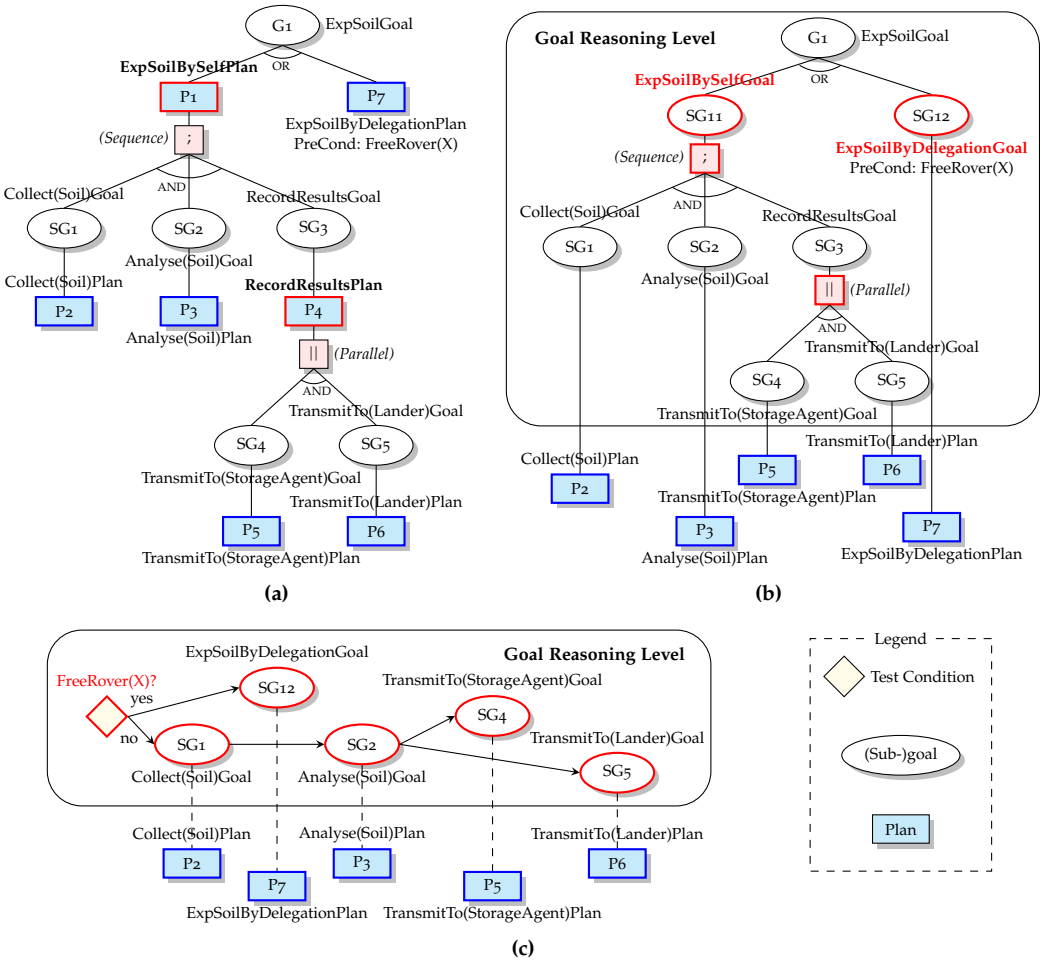


Figure 49: (a) the Goal-Plan Tree of a Mars rover from [108], (b) a translation of the Mars rover scenario in the form of a GPS-compliant AND-OR goal decomposition and (c) a modified representation of the scenario with a clear goal-plan separation

P7's precondition from the original scenario. The sequence operator is represented through the arrows that depict the dependencies between goals, while the parallelism is implied through the fact that two arrows start from the same entity, in this case *SG2*.

If, however, *P1* and *P4* also contained actions on the environment, the transformation would become more complicated. Figure 50 shows only the sub-tree starting from *SG3* with three simple examples of possible cases: (1) actions in parallel with, (2) before or (3) after the goal adoptions. This shows the hidden complexity associated with the action-goal mix.

The examples in this section obey the GPS principle since in each case, the two levels, the goal reasoning level and the plan level, can be clearly distinguished. This shows the applicability of the Goal-Plan Separation is not restricted to a specific goal reasoning formalism.

Now that we introduced our approach, let us present a means of implementing it when designing or programming goal-driven agents.

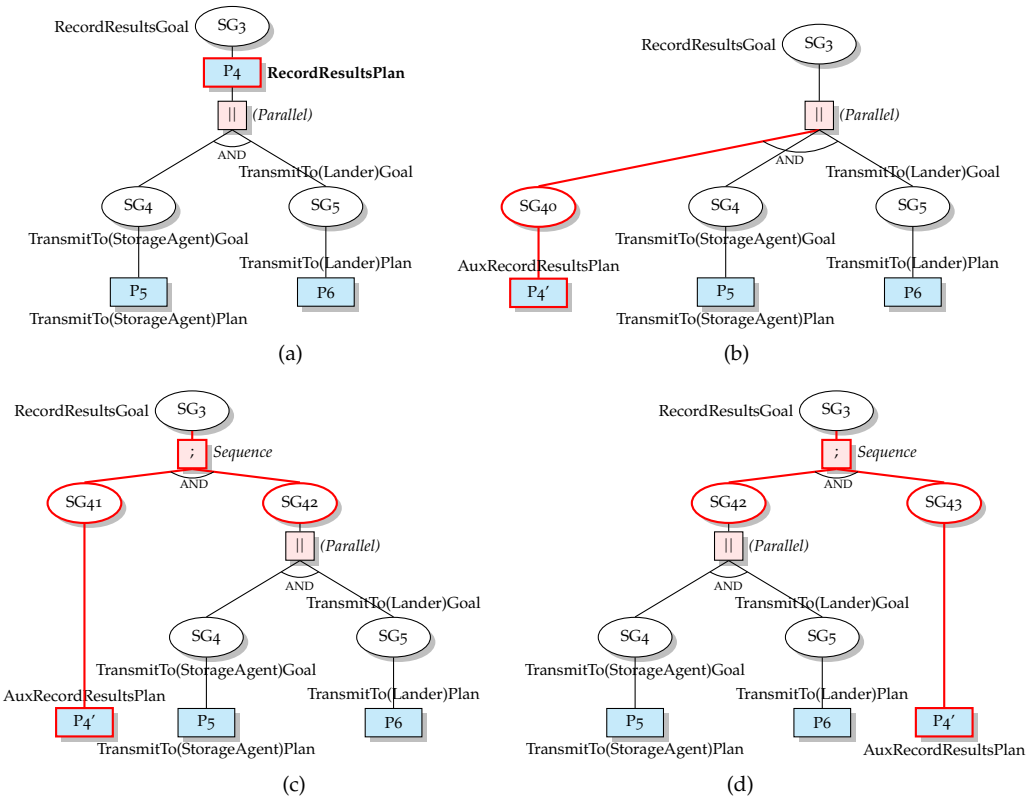


Figure 50: Transformation of the SG3 sub-tree (a) from the Mars rover scenario (Fig. 49) into a GPS-compliant form, in some of the non trivial cases: P4 contains actions on the environment that happen in parallel with the goal adoption (b), P4 contains actions on the environment that happen before (c) or after (d) the goal adoption

7 GPS METHOD IMPLEMENTATION

Throughout the evolution of programming, languages and development tools often advanced by limiting the programmer's freedom to access lower level elements such as registers and pointers to data, and offering in exchange higher level tools and constructs such as variables and dynamically created references to data. These evolutions allowed for the creation of increasingly complex systems while decreasing the possibilities for coding errors. Similarly, we do not refrain from restraining the freedoms of the programmers and designers in the interest of clarity and reliability.

To achieve the Goal-Plan Separation, rather than adopting sub-goals, at execution time an agent's action level (usually composed of action plans) would accomplish the necessary actions and then relinquish control to the higher level where the reasoning and possibly a following goal is adopted. This creates, as illustrated in the examples above, a distinct goal reasoning level where an agent's goals are chosen and their execution is managed.

As shall be discussed in this and the following chapters, the representation on multiple levels, either by using sub-goals or through other mechanisms, is important for the scalability and intelligibility of the resulting agents and therefore constitutes an important characteristic of the models that should be at least taken into consideration for the goal reasoning level.

In [110], GPTs are used as support for a study on plan coverage and overlap, with the hypothesis that the plan libraries discussed have no cycles. This is important to note as in the general case adopting goals inside plans may produce cycles, sometimes even with unwanted consequences similar to the infinite loops in classic programming. We, on the other hand, do not restrict cycles, as will be seen in the scenario in Sec. 8. However, the Goal-Plan Separation does not allow cycles created through plans that also have actions on the environment.

As the Goal-Plan Separation approach in its simplest form is the requirement to keep a clear distinction between the two abstraction levels, it is general enough so that it can be applied using any of the BDI frameworks that allow goal adoptions in plans. The important condition, however, is to make sure no goals are adopted in plans that act on the environment. Examples of representations that can be used are given next, followed by a more detailed description of a model based on what we call *goal plans* and that we use in Sec. 8.

7.1 EXAMPLES OF POSSIBLE MODELS FOR THE GOAL REASONING LEVEL

7.1.1 *Reasoning through Rules.*

Using goal trigger rules, an almost "reactive" agent can be created. The goal relationships are implicit but a dependency tree similar to the one seen in Fig. 49 (c) above can be constructed at runtime for tracing purposes. This reasoning model can be implemented in Jadex by simply specifying trigger conditions for each goal but without creating explicit connections between these goals. The advantage of this approach is that the representation can handle more complex systems that act

$$
\begin{aligned}
P &= < N, E > \ // \text{ action plan} \\
N &= A \cup O \cup T \ // \text{ nodes} \\
A &= \{action \mid action \neq goalAdoption\} \\
O &= \{o \mid o \in \{startNode, finishNode, \\
&\quad AND, \ \|, wait(duration)\}\} \\
T &= \{test(stateCond) \mid stateCond \in \\
&\quad \{Beliefs, Events\}\} \ // \text{ conditions} \\
E &= \{n_1 \rightarrow n_2 \mid n_1, n_2 \in N\} \ // \text{ edges}
\end{aligned}
$$

(a)

$$
\begin{aligned}
GP &= < N_g, E > \ // \text{ goal plan} \\
N_g &= A_g \cup O \cup T \ // \text{ nodes} \\
A_g &= \{adopt(G) \mid G \in Goals\} \\
O &= \{o \mid o \in \{startNode, finishNode, \\
&\quad AND, \ \|, wait(duration)\}\} \\
T &= \{test(stateCond) \mid stateCond \in \\
&\quad \{Beliefs, Events\}\} \ // \text{ conditions} \\
E &= \{n_1 \rightarrow n_2 \mid n_1, n_2 \in N\} \ // \text{ edges}
\end{aligned}
$$

(b)

Figure 51: Action plan (a) compared to goal plan (b). Only the action nodes differ.

in highly dynamic environments, with new goals added effortlessly. However, this model lacks *look-ahead* capabilities.

7.1.2 *Reasoning Using a Planner.*

Rather than having goals simply triggered by rules, a planner can be used to select among available goals, as for example in CANPLAN [99]. The difference then from the reasoning model described above is that this time the reasoning allows the choice of goals to be prepared in advance starting form the current context. Another difference is that a planner would render the agent pro-active, as it would not have to wait for events in order to act. The job of the planner would be to select, order and parallelise goals according to the current needs, and for this it could use certain operators [21]. The example in Sec. 8 does not correspond to this method as no planner is used and its *goal plan* (see below) is defined at design time. Our intuition is also that the GPS approach benefits this model as planning should be easier to perform only on goals, without the interference of details from actions.

7.2 REASONING THROUGH A *goal plan*

Between the reactivity of the first model above, and the planning capabilities of the second, we propose here a middle solution that allows for a certain level of *look-ahead* owing to the use of pre-written goal dependencies, just as plan libraries can be used with BDI systems. As required by the GPS method, the goal reasoning level should be kept separate from the plans that handle action composition. Considering that relations between goals can be similar to those between actions, we can envisage using a modified plan language to represent the relations between goal adoptions. We call the resulting plans that handle goal composition *goal plans* and we oppose them to *action plans*.

As defined in Fig. 51, a *goal plan* is an oriented graph with three types of node:

- A_g, the *goal adoption* nodes, as the unique action allowed in the goal plan. Each node represents the invocation of an automaton associated with the goal. Note that this is the only distinction from the action plans which have $A = \{action \mid action \neq goalAdoption\}$.
- O, the *operator* nodes, with operations including a unique *start* node and at least one *finish* node. Different *finish* nodes can be used to indicate final states

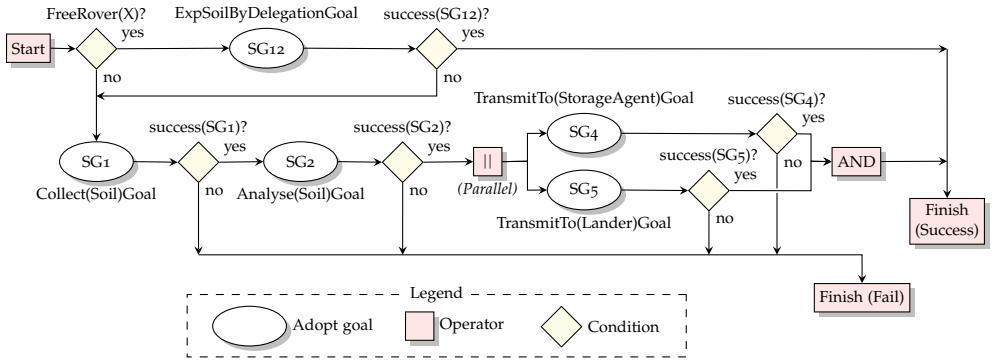


Figure 52: A goal plan for the Mars rover scenario from Fig. 49

for a plan, e.g. "successful completion" or "partial failure". There is also an operator for branching parallel threads and one for the logical condition *AND* that can be used to synchronise threads or to indicate the obligation of two or more conditions to be all true, for example to require several goals to be achieved in order for the execution to continue.

- T, the *condition test* nodes that can handle state conditions for belief values and events such belief change and message arrival. They can either be used to test for a momentarily condition, or to wait for a condition to become true or for a message to arrive.

Edges indicate the succession of nodes in the goal plan and, as stated before, cycles are possible, for example to indicate a recurrent goal adoption.

The Mars rover scenario in Fig. 49 (c) with its inline goal dependencies can easily be transformed into a goal plan, as seen in Fig. 52. There are two possible *finish* nodes, with one for a successful mission where either *SG12* or both *SG4* and *SG5* were achieved, and one to indicate all other cases as failures.

While implicit relations between entities (such as the rule-triggered goals above) may be enticing due to their ease of definition and generality, they are also difficult to follow and may hide unwanted interactions. The goal plans, however, favour the use of *explicit* specifications of dependencies between goals. If for example a Mars rover needs to perform an experiment at a location X and it has two goals for achieving this, one being G_1="*move to X*" and the other G_2="*drill*", then it is clearer to link the adoption of G_2 to the successful achievement of G_1 rather than for example the belief that the rover is at location X.

In a framework like Jadex, this model can be implemented using a plan that is triggered at agent's birth. The plan would specify the dependencies between sub-goals and adopt them *without any other actions*.

In practice, this model can become difficult to manage as the agent grows in complexity. A solution to this problem is to group together parts of the goal plan and abstract them into *sub-goal plans*, that are to be expanded only when needed. In this way, the representation can still be conceptually on one level, while having the advantages, in particular the scalability, of a hierarchical representation.

This kind of reasoning is suitable for agent systems where the behaviour can be thoroughly specified at design time so that all dependencies can be accurately included. Adding new goals and other modifications, however, are difficult to apply. The first implementation described in Sec. 8.1.1 corresponds to this approach.

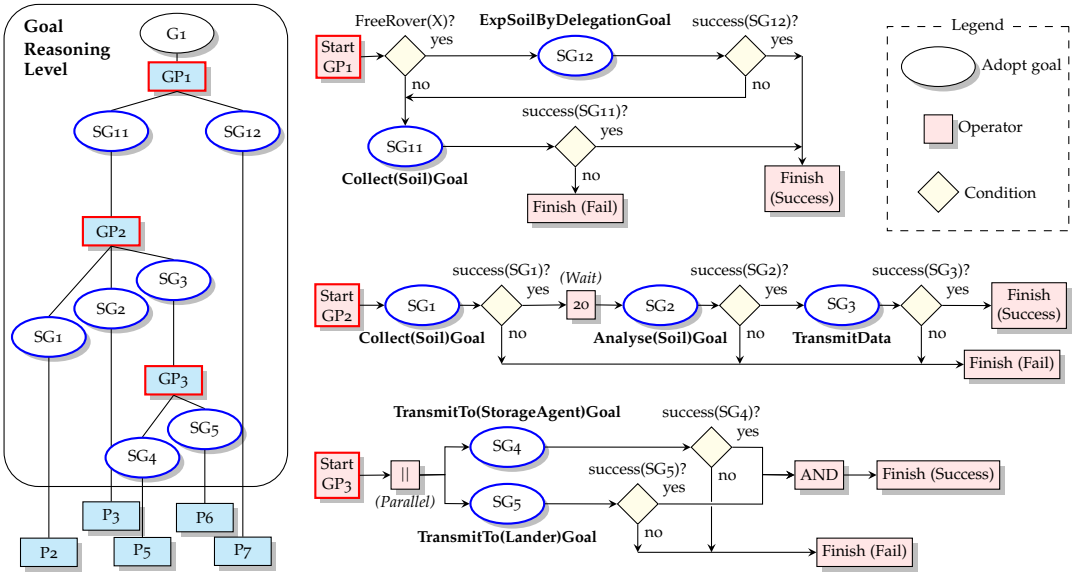


Figure 53: A multiple level goal plan for the Mars rover scenario from Fig. 49, with (a) the resulting goal-plan hierarchy (as used in Sec. 5.2 and similar to a GPT) representation and (b) the corresponding goal plans. Note the separation in (a) between the action plans, i.e. P2, P3, P5, P6 and P7, and the goal reasoning level comprising the goals and the three goal plans, i.e. GP1, GP2 and GP3.

7.3 REASONING THROUGH MULTIPLE GOAL PLANS

The method above has the advantage of providing a "big picture" of the agent's behaviour but, as stated before, does not scale well to complex agents. Designing the behaviour of an agent that can run for hours can for example create a large goal plan that is difficult to follow and which risks being too rigid in case of unforeseen events. The solution is then to decouple the sub-goal plans from their "parent" goal plans by using goals to manage the expansion, in other words, by allowing any goal not only to have action plans, but also goal plans. This means using the "classic" BDI mechanisms – i.e. goals, plans and automata – with just the subtle difference in the construction of plans: no goal adoption will be in the same plan as an action on the environment. Note, however, that in this case the states indicated through *finish* nodes do not necessarily reflect the achievement or failure of the parent goal, as the goal would normally have its own conditions for success and failure. Figure 53 (b) shows the Mars rover's behaviour represented with this model. The resulting model can be represented through a structure that is similar to the GPT as can be seen in the Fig. 53 (a), but this tree contains fewer details as more logic is included in the goal plans, while in the same time complying with the GPS approach.

There are many advantages of this multiple goal plan model. First of all, splitting the behaviour into more levels of goals and sub-goals with the corresponding plans improves flexibility and fault tolerance – in case a plan fails, the BDI logics can require a retry using the same or a different plan, provided that such plan is available. Then, splitting the behaviour into more manageable chunks leaves less room for hidden faults. The use of goal plans for managing goal dependencies allows for a more refined specification than what was available through the *AND*, *OR* and the operators in the GPT. For example, in Fig. 53, the suite of goal adoptions in

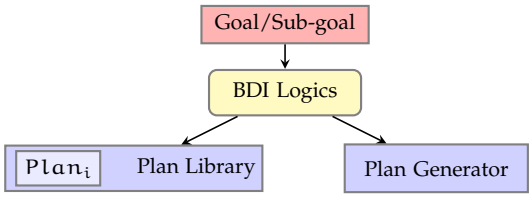


Figure 54: BDI logics: handler of the goal-plan relation at runtime

GP2 does represent the sequence that was originally in the GPT, but other operators – such as the *delay* in the example – can be added through this specification, and precise goal failures can be handled accordingly (while not present in the given example, one could add other goals to account for these specific sub-goal failures). This model is therefore preferred to the simple goal plans presented above, and is illustrated in the second implementation in Sec. 8.1.1.

7.4 EXECUTION

While not explicitly presented in the GPT, as stated before and seen in Fig. 54, between the goal and plan levels there are the BDI logics or more commonly a goal automaton handling the goal life-cycle (as presented in the state of the art in Sec. 2.3 and also in Sec. 3.3.3 on the safety net contribution). This life-cycle usually starts with the adoption of the goal and includes the choice and execution of plans.

For the GPS approach the automaton is a black box that is given a goal to adopt and possible plans to execute and this is why we represent only goals and plans in our modelling examples. The execution can cause side effects such as belief changes that can lead the reasoning level to take actions with respect to current goal or even the adoption or execution of other goals. For example, this can cause the goal to be aborted in case it is estimated to take the agent in an unsafe state, or it can cause the adoption of a reparation or compensation goal to counter certain unwanted effects. Note that several automata can function at a given moment as parallelism is allowed in our method. While conflicts are normally treated at goal reasoning level and can even be explicitly handled in the goal plans, conflict management is not within the scope of this thesis.

7.5 KEY LITERATURE ASPECTS

While we discuss the goal reasoning level in the need to better organise the levels "below", i.e. the plans, Morandini et al. [76] approach the same level from a different perspective: the need to fill in the gap between goal based engineering and goal implementations. They propose a tool for transforming an agent designed using the Tropos methodology [44] into Jadex code, for which they introduce a formalism based on rules for the life-cycle of non-leaf goals in a goal hierarchy. This segregation between leaf and non-leaf goals creates a goal level that corresponds to our goal reasoning level and thus their work is consistent with the GPS approach. This further confirms our statement with respect to the utility of a goal-plan separation for the implementation of goal-based methodologies. Furthermore, our proposition of using goal plans on multiple levels means that even goals that are internal to the goal reasoning level will have the same life-cycles as goals that use action plans. A specific life-cycle, as proposed by Morandini et al. is therefore no longer

needed, deeming the development process easier, as there are less types of goal to consider. One of the interesting aspects is that Morandini et al. take into account the fact that even if the sub-goals are achieved, the parent goal may still fail due to its own achievement condition, which is often not taken into consideration when discussing the Goal-Plan Trees. While this formalism is rich and GPS-compliant, as our application example shows, our approach aims to provide a model that allows for a more refined representation, with more diverse goal relations, event-based goal reasoning and time constraints.

There are many parallels that can be drawn between our approach and the one employed by the Prometheus agent development methodology [114] in the detailed design phase. This is where functionalities identified in the previous phases of the methodology – system specification and architectural design – are used as a starting point for designing capabilities. A capability is a module within the agent that can contain further capabilities, and at the bottom level plans, events and data, e.g. capability C_1 uses data D or plan P_1 sends message to plan P_2. Internal messages are used to connect between different design artefacts, such as plans and capabilities. This functionality is assured by either beliefs or direct goal dependencies in our work. This nested structure of capabilities is similar to the sub-goal plans (Sec. 7.2) in its pursuit of *"understandable complexity at each level"*, and while semantically different, it does provide a very similar functionality to our goal reasoning level. Furthermore, the use of internal messages to indicate dependencies between internal artefacts (mostly capabilities and plans) creates a very similar structure to our goal plans where we explicit dependencies between goals, often guided by tests on beliefs and messages. In Prometheus, BDI goals at agent level can be represented through a specific type of event, because events can trigger plans. As events, i.e. goal events, but also messages, percepts and internal messages, can be produced in plans as well as in outside the agent, a clearly defined goal reasoning level in the GPS sense cannot be delimited in the current form of the methodology. The Goal-Plan Separation approach would, however, benefit from the integration with the first two phases of the Prometheus methodology: the system specification and the architectural design. Due to the fact that these two phases correspond to a top-down design approach, and also, as we showed above, the fact that there are already similarities in the current form of Prometheus, we feel that such an integration would be possible, resulting in a methodology tailored for goal-directed GPS agents.

The *task expansion tree* described in [77] represents the decomposition of a task (a concept similar to goals in our work) into subtasks. The particularity is the introduction of special *composite tasks* that are used to compose other tasks in a functional manner. These include, besides the sequence and parallel operators present in the GPT model described in here, other tasks that allow other types of branching and tests. The use of these operators in a tree structure situates their model between classic goal hierarchies and our goal plan.

Clement et al. [25] champion the advantages of abstraction for solving various problems such as large scale planning and scheduling. They argue that by abstracting the less critical details in a large problem, the overall solution is easier to find, and can then be expanded to the actual detailed solution. This applies well to our Goal-Plan Separation approach, as well as to their approach on planning in a hierarchical way. They extend HTNs to take time into consideration and use summary information at higher levels in the HTN to identify possible interactions between plans while working with abstract actions (which are similar to the BDI concept of

goal). HTNs are quite similar to goal hierarchies in that they too offer a gradual refinement for the behaviour of an agent from the more abstract to the actual actions. The advantage of using goals instead of "abstract plans" is given by the flexibility and resilience offered through the goal life-cycles where a goal's achievement can be attempted through various plans, with different constraints etc. Nevertheless, our work does not exclude the possibility of using HTNs for plan selection, for example in a similar fashion with CANPLAN [99].

Having proposed a model for using the Goal-Plan Separation approach, in the next chapter we present two applications where the approach was successfully used.

8

EXPERIMENTING WITH GPS

Now that we introduced the Goal-Plan Separation approach, we will present two occasions in which we experimented with it. In Sec. 8.1 we describe our experience with the implementation of goal-driven agents in the context of a Thales application for maritime surveillance. While the actual implementation was done in ALMA, the modelling part was done using Petri nets. In Sec. 8.2 we present the modelling of a software for the deployment of Ambient Intelligence (AmI) applications on a distributed infrastructure, which we represented using a restricted version of the ALMA notation. This second application was part of work that was accepted at AAMAS 2016 [81] and EMAS 2016 [80] (both in May).

8.1 AN APPLICATION FOR MARITIME SURVEILLANCE

The GPS approach has been experimented in an industrial context at TSA on an application designed for experimenting on AI in general and more precisely on Interval Constraints propagation and MAS. The purpose of this application, Interloc, is the localisation of boats from a maritime patrol aircraft. It is implemented as a MAS and can contain dozens of agents implemented as Prolog processes.

Interloc was initially designed as a set of non goal-directed autonomous agents. This means that the agents had only one purpose that was achieved through a set of associated plans. Subsequently, it was redesigned in order to improve the level of autonomy of the agents by endowing them with goals. The pursuit of intelligibility brought along the idea of having a clear separation between the levels of abstraction of goals and plans.

A first implementation in the spirit of GPS used a goal plan formalism as the one described in Sect. 7.2. This meant designing a plan where the only possible action was goal adoption. For the ease of use, sub-goal plans – which anticipate the hierarchical approach later implemented – were also used, adding their activation to the goal adoption as the only possible "actions" in the goal plan. The intention of the designer (prior to the GPS methodology presented here) was to exhibit an abstract (goal) level describing the main features of the behaviour of agents so that one would find it sufficient to only read the goal level description in order to understand the salient behaviour of the agents. Agents were then implemented following the idea described in Sect. 7.3 as the flexibility and robustness of goals seemed preferable to the simple invocation of sub-goal plans.

In the pursuit of a more formal representation, we abstracted the goal plans into Time Petri Nets (TPNs) [9], seen in Figs. 55, 57 (b) and 58 (a-d). We chose the TPNs because they present many advantages through their graphical and intuitive representation, as well as their expressive power (parallelism, sequence, synchronisation etc.). This extension over classic Petri nets gives the possibility of assigning firing time intervals to the transitions, which we used for representing waiting in the agent behaviour. Furthermore, the TPNs allowed us to structurally verify the goal plans and ensure their correctness. We also used a type of Petri net that resemble the Recursive Petri Nets (already used for representing agent plans [35]) where we distinguished two types of transition: the elementary transitions to be fired accord-

ing to the standard semantics of Petri nets and the abstract ones corresponding to the action of adopting a goal. However, the expansion of this action, the goal adoption, is not handled in this network, and its transition corresponds to a call to the associated automaton, e.g. the one in Fig. 24 in Sec. 3.3.3.

We first present the application itself, then the particular case of one of the main agents, the aircraft, in the two goal plan-based implementations mentioned above. This section concludes with a discussion on the advantages of the GPS approach in the specific case of the Interloc application.

THE APPLICATION The main goal of the application is the localisation of boats using a *goniometer*[1] on-board a maritime patrol aircraft. The sole use of a goniometer allows for a stealth detection, i.e. detect without being detected, of boats which is important for some missions such as gas-freeing prevention[2]. If the boats were steady, the problem would be simple. The fact that they move necessitates a reliance on non-linear regression methods, as is the case of existing commissioned implementations, or interval constraint propagation, in Interloc. Most of the agents, i.e. boats, the goniometer and the data visualisation agent, were designed for the purpose of simulation. The main agent, the aircraft, must (1) follow all the boats visible from its location, (2) compute in real-time their position by accumulating bearings and interacting with computation agents (more precisely *artifacts*[3]) operating interval propagation, (3) adapt its trajectory to observations and contingencies and (4) transmit results to the visualisation agent. For the patrol aircraft, boats may appear or vanish at any time. Several aircraft might be present at the same time, but so far they do not communicate with each other. Typically 20 to 30 agents or artifacts are active in the system at a given time.

8.1.1 *In the Lead Role: The Aircraft Agent*

Boats and aircraft have been designed following the GPS method. We present here the aircraft, which is the most complex agent type and hence the most interesting for illustrating the methodology.

Five goals corresponding to five main activities of the agent were identified:

- *Init.* (for "initialisation") of the system: get data related to the aircraft trajectory (pre-defined, planned or human-guided) and various parameters characterising the simulation;
- *Move*: execute one *step* forward;
- *Measure*: initiate measurement of the bearing of all the visible boats;
- *Treat.* (for "treatment"): process a received measurement;
- *Visu.* (for "visualisation"): process a single request from the visualisation agent.

The sole knowledge of the various goals present in the system is not sufficient to understand (and define) its behaviour. One must also describe the way in which these goals are adopted and what happens when they are achieved, for example

1 Tool which displays the direction towards the source of a signal, in this case a boat and its radar.
2 Deterring tankers from polluting the environment by cleaning their fuel tanks at sea.
3 Concept introduced by Ricci et al. [93] and briefly defined here in Sec. 1.4.

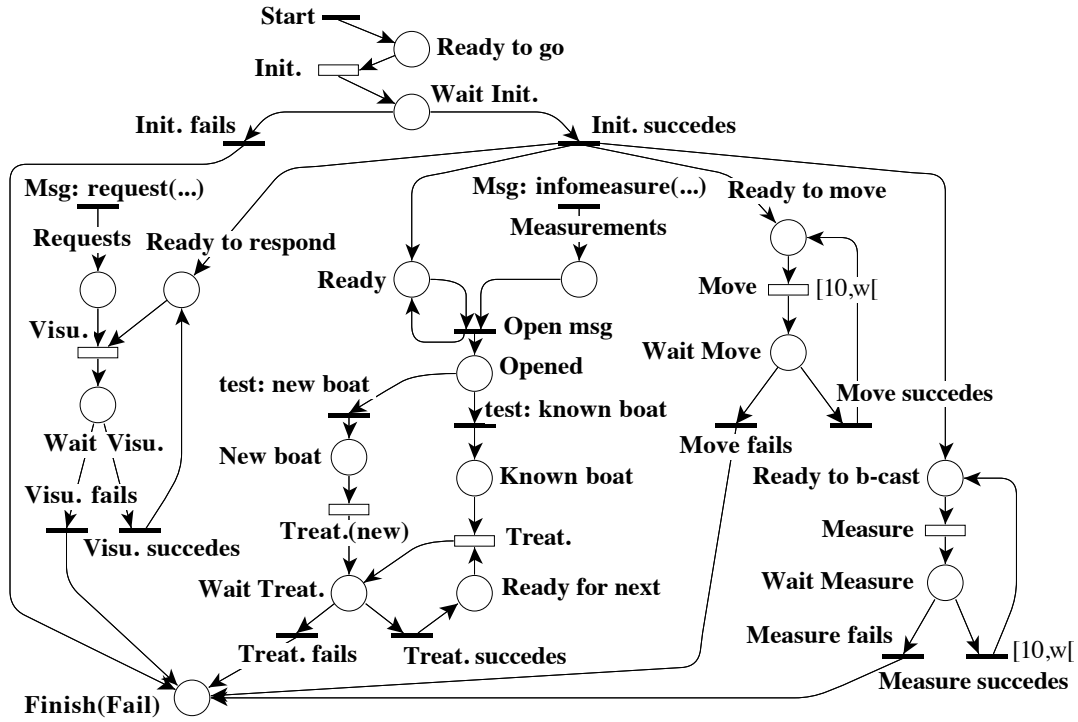


Figure 55: Petri net representation of the goal plan for the aircraft agent with goal adoptions represented as hollow transitions. The *Treat.* (for "treatment") goal is adopted in two different contexts in order to insure that messages from each boat are treated sequentially, but in parallel with the other boats. Multiple instances of the goal with different beliefs are thus created.

by specifying their chronology, conditions for becoming a desire, conditions for becoming an intention. This knowledge may be provided in different forms, corresponding to the different ways of applying the GPS approach.

8.1.2 *GPS for Modelling the Aircraft Agent*

USING A SINGLE GOAL PLAN. For the first implementation we present here, the aircraft agent in Interloc was designed using a goal plan with four sub-plans to indicate the dependencies of the goals above. These dependencies correspond to the goal reasoning level in the GPS approach.

Informally, the goal plan is the following (a more formal description of this plan is given in Fig. 55 as a Petri net): the achievement goal Init. is adopted. If the goal is not achieved, the system is halted. Otherwise, four sub-branches implemented as sub-goal plans are activated in parallel: *main_move*, *main_measure*, *main_visualisation* and *main_analyse*.

The *main_move* sub-plan:

- Wait for a *move_time_step* delay;
- Adopt the Move goal, whose associated plans will compute and execute the next time step;
- Wait for the Move goal achievement;
- Loop.

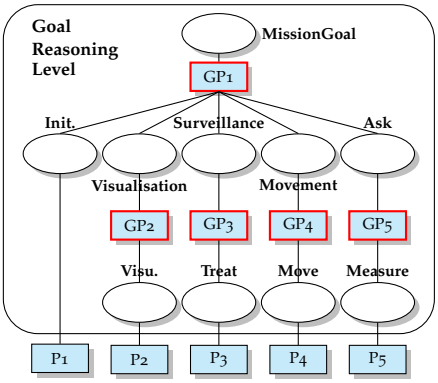


Figure 56: The goal-plan structure of the aircraft agent

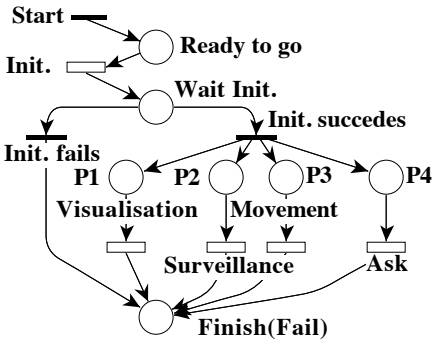


Figure 57: Petri net representation of the *GP1* goal plan. Goal adoptions are represented as hollow transitions.

The *main_measure* sub-plan:

- Adopt the Measure goal, where the associated plans will measure the bearings of all the visible boats through interactions with the measurement artifact and the (simulated) boat agents;
- Once achieved, the goal will be re-adopted after a given time delay.

The *main_analyse* sub-plan:

- Wait for a measurement, in the form of a message that arrives randomly after a measurement request message is issued;
- Record the newly present boats;
- Adopt the Treat. goal, whose associated plan will generate a constraint to be added to the previously received measurements and send it to an interval constraint propagation artifact which will compute a more and more precise boat location;
- Loop, in order to process waiting measurements.

The *main_visualisation* sub-plan:

- Wait for a request from the visualisation agent;
- Adopt the Visu. goal in order to process the request;
- Wait for the achievement;
- Loop to process pending requests.

USING A MULTIPLE LEVELS OF GOAL PLANS. When the pursuit for flexibility and robustness pushed us further and we separated the goal plans and their sub-goal plans through new goals, we obtained the tree structure seen in Fig. 56. *GP1*, in Fig. 57, guides the adoption of four intermediary goals that are internal to the goal reasoning level, i.e. they do not have action plans. *GP2*-*GP5* in Fig. 58 correspond roughly to the sub-goal plans described above and can easily be matched with the corresponding branches in the initial one-level goal plan (Fig. 55).

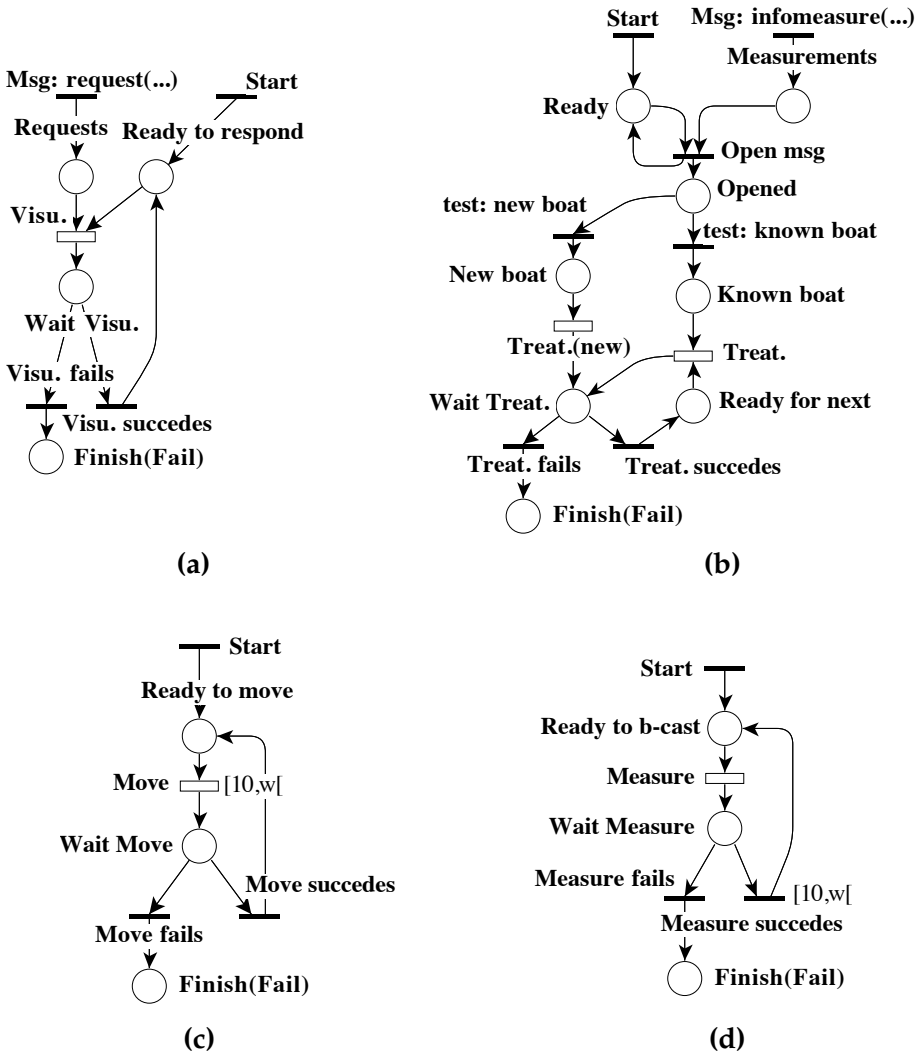


Figure 58: Petri net representations of the *GP2* (a), *GP3* (b), *GP4* (c) and *GP5* (d) goal plans. Goal adoptions are represented as hollow transitions.

8.1.3 *Discussion*

With GPS, iterative and timed behaviours appear at goal level: in the pre-GPS version of the application, the natural tendency was to incorporate dynamic aspects into the plans, making them fairly complex. For instance, the Move goal was not conceived as a single step as presented above, instead, it was charged with the complete management of the aircraft's trajectory, including the loop sequencing individual steps. This rather straightforward design would close the loop inside the plans and after the actions on the environment – e.g. the movement or broadcast of measure request messages – were performed. The *move time-step*, which is important for the global understanding of the behaviour of the aircraft, was also "buried" in the plan pursuing the goal. In the GPS-compliant versions, deciding to rewrite the plan and change the scope of the goal to the achievement of a single movement step, created the need for the definition of the time-step and the iterative behaviour at the goal reasoning level, leading to a clearer design. The fact that such details are at an upper level of abstraction emphasises their importance and improves the understanding of the agent behaviour.

DISCUSSION ON FAULT TOLERANCE WITH GOAL REASONING In real life applications agents tend to have more refined representations than the ones discussed in Sec. 6. In particular, when it comes to handling errors, as we discuss in Part II of this thesis, the specification easily grows in complexity as specific cases have to be taken into consideration. As we stated before, goals give agents a level of abstraction that is beneficial for a system's robustness as errors, exceptions, anomalies etc. usually occur during plan execution which, in a robust[4] system, only cause the plan to fail and the goal automaton to react normally and reattempt to achieve the goal. While there are studies that treat the more general case of partial goal satisfaction [95] (described in Sec. 2.3), if we only consider a binary goal definition, a goal's adoption has only two possible outcomes at reasoning level: the goal is either achieved or not. Requiring the programmer to specify not only the actions to take after the achievement of a goal, but also the actions to take in case the goal fails enhances the reliability of the agent without dramatically increasing its complexity. This is in the same idea as the *default* branches for the *decision* nodes in ALMA (Sec. 2.4.4) whose role is to avoid unforeseen situations.

In the Mars rover scenario represented in Fig. 52, the failure to delegate the task to another agent, i.e. the failure of *SG12*, causes the rover to attempt to accomplish the mission by itself through the adoption of *SG1*. Similarly, in the aircraft specification of the Interloc application (Sec. 8.1), both the successful achievement and the failure of goals are represented in the Petri net and also in the implementation. However, for simplicity reasons, in our example, no special actions are taken and the only result of a goal failure is to ensure the agent does not reach unforeseen states. Also, the current format implies an infinite life for the agent, which is not necessarily desirable in a real application.

IMPLEMENTATION For Interloc, we used ALMA to implement the goal plans. All the required primitives were available, since a goal plan is a type of plan. Nonetheless, it appears that specific primitives could be introduced to facilitate the programming of the goal level. These concern mainly iterative and time-controlled behaviours.

4 In this case, we understand by *robust* an agent system in which an error or exception in a plan is confined to that plan, so it is caught and only causes that plan to fail, while the rest of the agent continues to function normally, i.e. does not cause the whole agent to fail.

8.2 THE DEPLOYMENT OF AMBIENT INTELLIGENCE APPLICATIONS

Another application of the GPS approach was for modelling a MAS for deploying AmI applications on a distributed infrastructure. This work was a continuation of previous contribution by our co-authors Piette et al. [79] on the centralised deployment on AmI applications using graph-based representations of the infrastructure and application requirements together with a graph-matching algorithm. This time, the objective was building a distributed software that can assist the context-dependant deployment of applications (e.g. an intelligent video doorkeeper) on a diverse and distributed infrastructure (e.g. camera, various displays situated in different environments to which the user has more or less access rights). As users evolve in a varying environment including infrastructure with different owners, resource privacy was an important aspect of this work.

8.2.1 *Scenario*

The scenario we used in this work highlights both the dynamic deployment of distributed applications and the privacy management encapsulated in both agents and agent organisations. Mr Snow uses a *video doorkeeper* for dependant persons (e.g. visually impaired) application in his home. When someone rings at the door, the image of the entrance camera is displayed on a screen near Mr Snow, making sure he can properly see the person. He can then discuss with the person and decide whether or not to remotely open the door.

It is Saturday morning and Mr Snow is waiting for a parcel that will be delivered to his home at any time. While he is grooming himself in the bathroom, his neighbour, Mr Den, rings the door. The smart house, aware that Mr Snow is in his bathroom, selects the connected mirror of the bathroom, instead of any of the other display screens of the house, as a support to display the image stream of the entrance camera. Mr Snow, not being able to receive his guest, informs him, thanks to the microphone in the mirror, that he will meet him in an hour. After getting ready, Mr Snow goes to his neighbour. In the middle of their conversation, he is notified that an unknown man rings at his door again. He tries to recognise with his neighbour by displaying the image on Mr Den's television. By default, Mr Snow does not have the right to use any devices that he does not own, but the latter has authorised him to access the television when he is at home. The doorkeeper application is redeployed dynamically to use the requested hardware entities. Neither Mr Snow nor his neighbour know the visitor. Mr Snow decides to activate the microphone of the camera which allows him to learn that the unknown person is the expected transporter, which he can now go and see in person.

The important point in this scenario is not the video doorkeeper application, but the way it is deployed dynamically in the environment, considering the user's context. The scenario shows two deployment situations:

1. the application was deployed for use in the user's own home infrastructure, but in a less usual place: the bathroom;

2. the application was deployed on the infrastructure of another user, as the necessary access rights had been granted.

8.2.2 *Multi-agent Modelling*

Our scenario highlights several necessary specificities of the deployment software. This software has to dynamically deploy and undeploy distributed AmI applications in an environment that is also dynamic: when a visitor rings the doorbell, the deployment of the video doorkeeper should start, considering the available hardware entities and the location of the user, in order to choose the most relevant screen for displaying the image from the camera. This scenario helps emphasise resource and information privacy: Mr Snow is the owner of the hardware entities in his house and he does not want that unauthorised persons use or even know of the existence of these resources. Furthermore, autonomy and robustness of the system are very important specificities: if my neighbour's system fail, mine should continue functioning normally and should not be impacted.

As the required software demanded distribution, privacy, context management, autonomy and robustness, MAS were identified as a suitable solution. Through its modularity, this paradigm facilitates a local processing of the data and guarantees the autonomy of the different parts of the hardware infrastructure, thus handling aspects of privacy and robustness.

To solve the dynamic deployment problem, we use the work of Piette and al. [79] in which the available hardware infrastructure is described by a graph. Nodes represent hardware entities or relations between these entities and properties can be attached to each node. The requirements of the deployable applications are also described using such graphs. A graph matching algorithm can then be used on the available infrastructure graph to find the entities that can support the running of the application.

Next, we present the modelling of agents and the agent organisation for our deployment solution, while focusing on the encapsulation of resource privacy.

8.2.2.1 *Agents and Artifacts*

The deployment application involves the user deploying applications on an infrastructure. Three types of agent were therefore defined to represent and clearly separate each of the parties in handling the deployment: *User Agent*, *Application Agent* and *Infrastructure Agent*. A fourth type of agent was introduced for enhancing resource privacy: the *Infrastructure Super Agent*. For each type of agent we identified a few main functions that, as we present in Sec. 8.2.3, will be their main goals:

- An *Infrastructure Agent* deals with a part of the global hardware infrastructure. It uses the graph representation of this available infrastructure [79] (hardware entities, relations and properties). This graph representation is never shared with other agents. The *Infrastructure Agent* reasons on it in order to propose partial solutions for the deployment of applications, thanks to a graph-matching algorithm. This class of agent has several functions, as it has to:
  1. keep the infrastructure graph up to date;
  2. propose solutions for the deployment of applications, considering the available hardware infrastructure, but also the sharing and privacy policy;
  3. deploy or undeploy functionalities of an application.

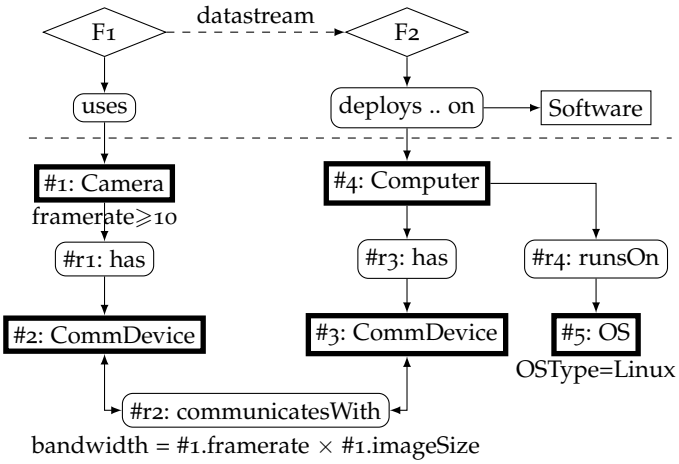


Figure 59: Example of a basic application graph

- An *Infrastructure Super Agent* is a representative of a set of *Infrastructure Agents* which are related to it forming a *group*. It acts as a proxy between the agents inside and outside of the group.

- An *Application Agent* manages an entire application during its runtime. It has a graph-based description of the application [79]. An example of such graph is represented in Fig. 59: the upper part represents the functionalities of the application and the bottom part shows their hardware requirements. The objectives of this class of agent are to:

  1. guarantee the consistency of the application and
  2. deploy or undeploy functionalities of the application if necessary.

  The *Application Agent* has to interact with several *Infrastructure Agents* in order to deploy the functionalities of the application over the infrastructure.

- At last, the *User Agent* is the interface between the user and the other agents of the deployment software. Through this agent, a user can request the (1) deployment or (2) undeployment of applications.

In addition to these four classes of agent, we also propose two classes of artifact[5] which are resources and tools that can be instantiated and/or used by agents in order to interact with the environment:

- *Deployment artifacts* can be used by the *Infrastructure Agents* in order to effectively deploy some parts of an application, or configure hardware entities so that they can be used by the application.

- The second class of artifact are the *functionalities* of the applications themselves. Some of them can provide useful contextual information to the deployment software (location of a user, available bandwidth etc.), to help the agents keep their application or infrastructure graph up to date.

In the video doorkeeper scenario, there are three *Infrastructure Agents*. The first one manages the hardware entities located in the living room of Mr Snow, e.g. the television. The second one manages the entities of the bathroom, such as the connected mirror. The last *Infrastructure Agent* manages Mr Den's house. We also find

5 Concept introduced by Ricci et al. [93] and briefly defined in this thesis in Sec. 1.4.

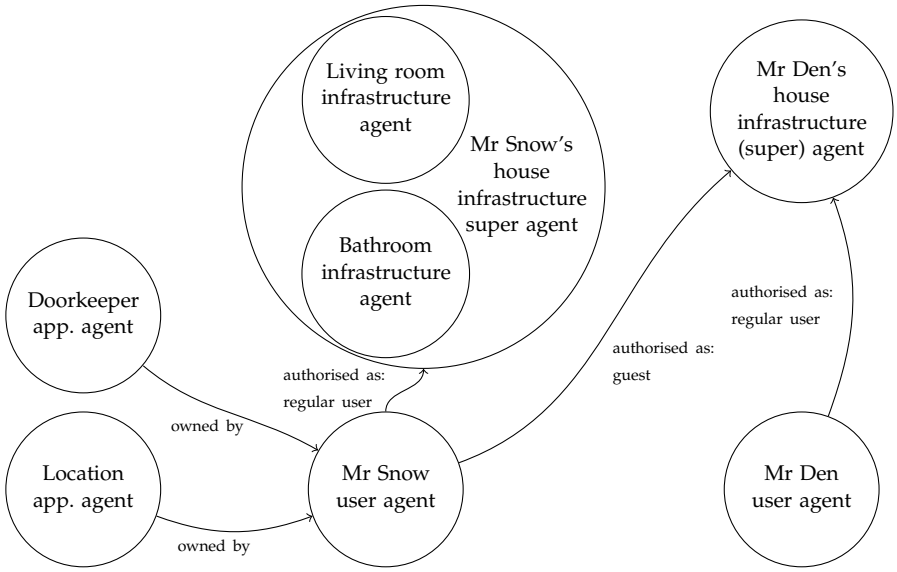


Figure 60: Agent organisation

two *Application Agents*. The first one manages the video doorkeeper application: when a visitor rings the doorbell, this *Application Agent* triggers the deployment of the video interaction functionality. The second one manages the application which provides the location of the Mr Snow inside his own house to his own *Infrastructure Agents*. The contextual location information is useful for deploying other applications. Indeed, the display screen of the video doorkeeper application has to be chosen near the user. Then, we have two *User Agents*. The first one is the interface between the deployment software and Mr Snow, and the second one is owned by Mr Snow's neighbour. At last, we have a certain number of deployment artifacts that can configure the display screens, the cameras, or deploy software on devices (TV box, connected mirror etc.).

The agent decomposition encapsulates a part of the privacy mechanism. Indeed, the graph representation of the available hardware infrastructure managed by an *Infrastructure Agent* is only known by this agent and is never shared with others. Moreover, the architecture used helps keep a clear separation between the applicative part, managed by the *Application Agents*, and the hardware part, monitored by the *Infrastructure Agents*. As agents only have a local view of the system, the privacy is enhanced.

8.2.2.2 *Organisation and Interactions*

The interactions between the agents presented above are regulated through their organisation and a privacy policy. *Infrastructure Agents* are grouped behind an *Infrastructure Super Agent* which, as stated before, acts as a proxy for the agents of the group. From an outside view, this *Infrastructure Super Agent* is seen as a normal *Infrastructure Agent*.

In our scenario, the living room and the bathroom *Infrastructure Agents* of Mr Snow are grouped behind an *Infrastructure Super Agent* representing the house of Mr Snow. Similarly, the *Infrastructure Agent* managing the house of Mr Snow's neighbour is a super agent, regrouping several *Infrastructure Agents* (or other sub-super agents). The advantage of this organisation is that it is easy to abstract groups of agents and make them invisible from the outside, resulting in a multi-scale or-

ganisation that helps improve privacy. Indeed, Mr Snow knows about his own *Infrastructure Agents* (bathroom and living room), but he does not have to know anything about the details of Mr Den's infrastructure organisation. If he wants to interact with his neighbour's house, he has to interact with Mr Den's *Infrastructure Super Agent* with the required access rights granted (as described below). The upper part of Fig. 60 shows the organisation of the *Infrastructure Agent* from Mr Snow's point of view.

The hierarchical organisation of *Infrastructure Agents* ensures privacy by hiding information about the structure of its sub-organisations. However, to improve privacy by controlling the use of resources, we also propose sharing policies. *User Agents* can be authorised, by the owner of some hardware infrastructure, to use some parts of its infrastructure, and cooperate with the associated *Infrastructure Agents* or *Super Agents*, to deploy applications. If a *User Agent* is not authorised by the *Infrastructure (Super) Agent*, it cannot use the hardware resources proposed by this agent. Otherwise, it can have different authorisation levels. For example:

1. Administrator level: the agent (and implicitly its user) has full access to the resources proposed by the *Infrastructure Agent*, can reconfigure the *Super Agent* organisation and manage the authorisation levels;
2. Regular user: the agent has access to the resources of the *Infrastructure (Super) Agent* but it cannot reconfigure authorisation levels or agent organisation and
3. Guest: the agent has a restricted access to the resources. Only the resources considered as non critical by an administrator are allowed to be shared.

These authorisation levels are not limited to three and can be modified by the administrator of the *Super Agent*. In the video doorkeeper scenario, Mr Snow's *User Agent* is a Regular user for his home *Infrastructure Super Agent*, but it is just a Guest to his neighbour's home *Infrastructure Super Agent*. As such, it has only access to the television of Mr Snow's neighbour. This allows to ensure privacy of the other resources of Mr Den. The *Application Agents* have the same authorisation level as the *User Agent* that creates them. They can interact with the authorised *Infrastructure Agents* in order to effectively deploy their application. Figure 60 shows the agent structure of the doorkeeper scenario; the agent organisation, the authorisation level, and the *Application Agents* that are bounded to their *User Agent* creator.

In this section, we have shown how privacy is preserved through encapsulation in our MAS. *Infrastructure Agents* keep the information about the hardware infrastructure secret. The *Infrastructure Agent* hierarchy keeps the details of the agent organisation hidden. Privacy policies can allow or prevent the sharing of resources to *User Agents*.

8.2.3 *Design and Implementation*

Agents are goal-directed, hence the description focuses on the goal specification which suffices for understanding the agent behaviour. Goals are specified by describing their associated plans following the GPS approach: higher level *goal plans* describing relationships between goals and lower level *action plans* for concrete actions. This approach helped handle agent complexity through the multi-level description, from top level abstract behaviours with goals to concrete action plans. Using goal-plans also has the advantage of specifying the relationships between goals in a plan format.

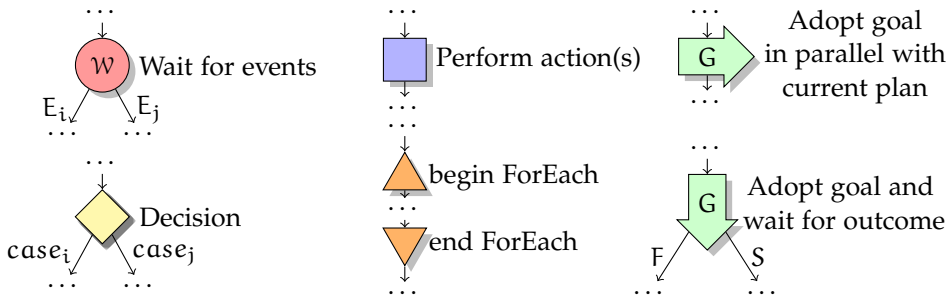


Figure 61: Flowchart nodes for efficiently describing the plans of goal-driven agents

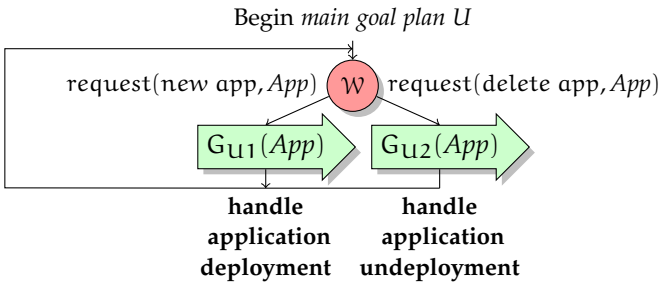


Figure 62: Main goal plan for the *User Agent*

For this application, we used flowchart notation based on ALMA (Fig. 61). For convenience, we introduced the possibility to add cycles in the flowchart (which was an directed acyclic graph in ALMA), as well as a *synchronous goal adoption* node which can be obtained using a normal *goal adoption* followed by a *wait* node. The *reasoning* (*add_rules*) node was not necessary and thus was omitted. For this application, we considered a simple goal model where a goal is successful ("S") when the plan executing for it ends with "End ok".

We continue by describing in detail the agents of the system. As the *Infrastructure Super Agent* is only a proxy between the agents of the group it represents and the other agents outside this group, its implementation is not detailed here. In what follows, P_{Xi-j} are the plans for a goal G_{Xi}.

8.2.3.1 *User Agent*

The *User Agent* acts as an interface between the user and the deployment MAS. The goal plan of the *User Agent* (Fig. 62) waits for user input and, depending on the received request, adopts the corresponding goal. The plans of G_{U2} and G_{U1} are similar: they create an *Application Agent* (Fig. 63) or request an application to be undeployed, wait for a confirmation and display the information to the user. The *User Agent* also allows the changing of the privacy policies, but this was not represented here.

8.2.3.2 *Application Agent*

The *Application Agent* is created by a *User Agent*. It tries to deploy a precise application by cooperating with one or more known *Infrastructure (Super) Agents*, from which it does not need to have any infrastructure details.

Upon its creation, an *Application Agent* adopts two goals (Fig. 64): G_{A1} for deploying an initial functionality (Fig. 65) and G_{A2} that waits for internal events

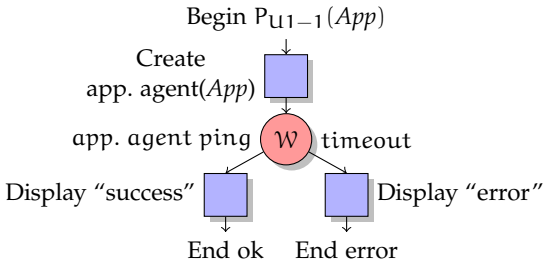


Figure 63: Plan for G_{U1}: "handle application deployment"

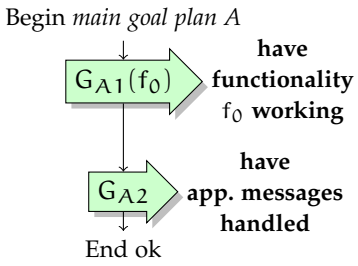


Figure 64: Main goal plan for the *Application Agent*

for new deployments or undeployments (Fig. 66). The deployment is done in two steps: first the agent obtains a deployment solution from *Infrastructure Agents* via G_{A3} and then it requests the deployment according to this solution through G_{A4}. The *Application Agent* sends a list of the requirements described in the application graph to the *Infrastructure Agent* and the solution it receives contains the list of requirements that could be fulfilled. Note that the reply does not contain any actual infrastructure details, which is important for the privacy of the infrastructure. It can be seen (Fig. 67) that the agent may need to call multiple *Infrastructure Agents* in order to obtain a complete deployment solution. Indeed an *Infrastructure Agent* tries to find in its own infrastructure the hardware entities that match the requirements of the application. However, if these requirements only partially match, the *Infrastructure Agent* will return a partial solution to the *Application Agent*. In this case, the latter will call another *Infrastructure Agent* that will continue to match the requirements of the application. Once a solution has been found, the *Application Agent* interacts again with the concerned *Infrastructure Agents* to effectively deploy the functionalities of the application: plan P_{A4-1} in Fig. 69 simply sends a message and waits for a confirmation.

After a functionality was deployed, the agent monitors it through G_{A5} (with its plan in Fig. 68) in order to adapt the deployment to the current context: infrastructure inconsistency (e.g. changing infrastructure availability, changing user location) and messages from the application itself (e.g. new guest at the door). An application message can result in multiple requests for deployments and undeployments. Internal events are used to control the execution of different plans. Deploy and undeploy events originate in the plan for G_{A5} and trigger the adoption of G_{A1} for the deployment of other functionalities or redeployment of the current one, and G_{A6} (Fig. 70) for the undeployment of the functionality. As each functionality is monitored by an instance of G_{A5}, in case of an undeployment, the plan of G_{A6} signals the corresponding G_{A5} to stop through a *kill* event (besides sending a request message to the corresponding *Infrastructure Agent*).

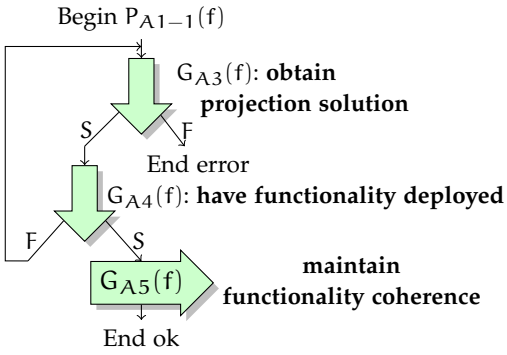


Figure 65: Goal plan for G_{A1}: "have functionality f working"

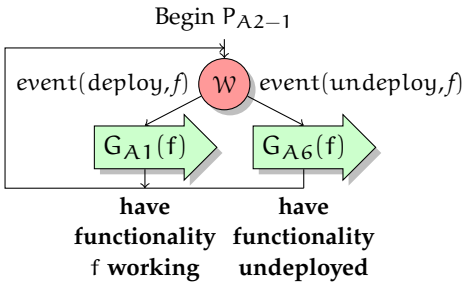


Figure 66: Goal plan for G_{A2}: "have app. messages handled"

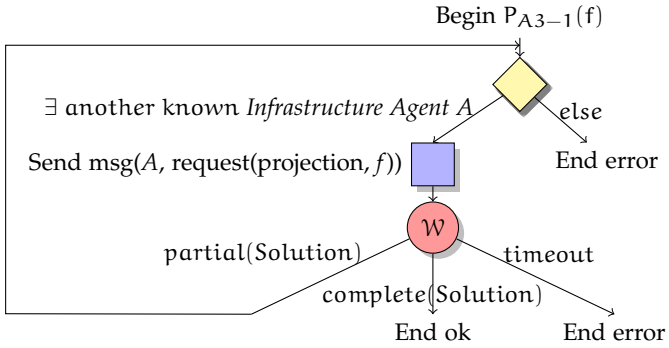


Figure 67: Plan for G_{A3}: "obtain projection solution"

Note here that the *Application Agents* only handle the application deployment. The application itself is in charge of its own actions, data and privacy.

8.2.3.3 *Infrastructure Agent*

An *Infrastructure Agent* receives requests from *Application Agents* that it tries to satisfy (Fig. 71). Only requests originating from known *User Agents* are treated, in other words only applications from agents that were granted one of the levels of authorisation are accepted.

When it receives a request for a deployment solution, the *Infrastructure Agent* uses the graph matching algorithm to determine if it can fulfil the requirements of

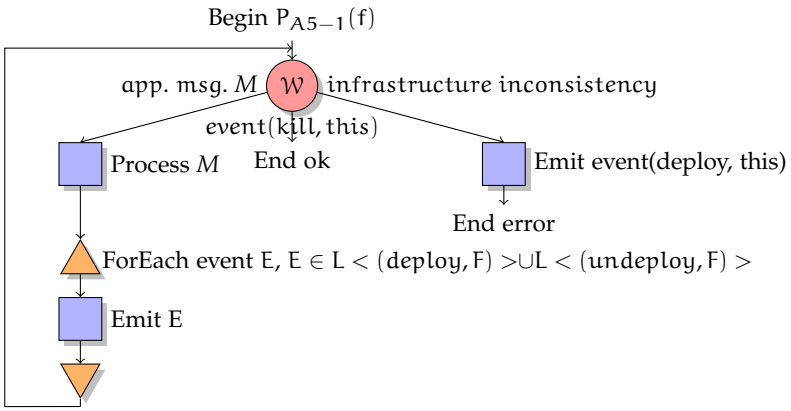


Figure 68: Plan for G_{A5}: "maintain functionality coherence"

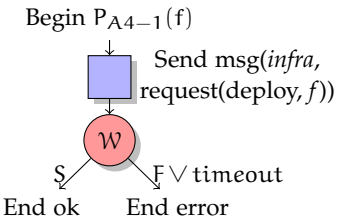


Figure 69: Plan for G_{A4}: "have functionality deployed"

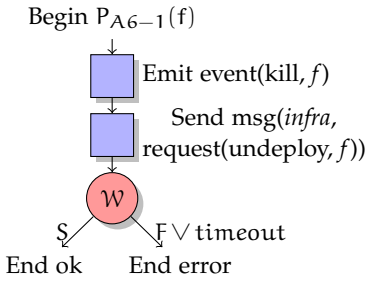


Figure 70: Plan for G_{A6}: "have functionality undeployed"

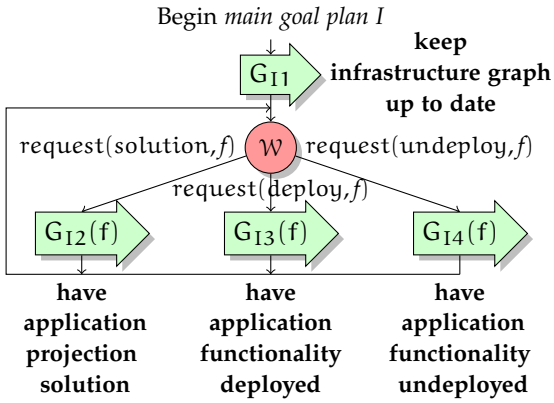


Figure 71: Main goal plan for the *Infrastructure Agent*

the request (Fig. 74) using the devices it manages. The algorithm takes into consideration the levels of authorisation of the involved *User Agents*. If it cannot produce a complete solution, the *Infrastructure Agent* requests the help of other agents in its group, but without informing the *Application Agents*. In this way, the components of the infrastructure remain private. If a complete solution is eventually produced and the *Infrastructure Agent* is given the order to deploy the application, it will do so by adopting G_{I3} (Fig. 75) and its sub-goals (Fig. 76 and 77) to dispatch the deployment tasks to its own deployment artifacts as well as to any other *Infrastructure Agents* that were included in the final solution. In case any of these requests fails (e.g. an artifact malfunctions), the whole application is undeployed using G_{I4} (Fig. 78) and the *Application Agent* is informed (Fig. 79), which will cause it to restart the deployment procedure.

In parallel with the request handling, the agent also adopts G_{I1} (Fig. 72) which listens for agent and artifact information in order to manage the graph the devices corresponding to the *Infrastructure Agent*. In case of an inconsistency (e.g. Mr Snow leaves Mr Den's home, so any display he used there are no longer relevant for the application), the agent informs the *Application Agents* via G_{I11} shown in Fig. 73 that it will need to redeploy the concerned parts of their applications.

8.2.3.4 *Implementation*

A demonstration model of the deployment software has been developed in a laboratory apartment replica. This home replica implements various scenarios applied to home care for dependent persons, including the presented scenario. These scenarios are using commercial connected devices tweaked to be horizontally connected,

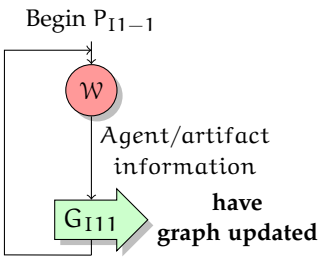


Figure 72: Plan for G_{I1}: "keep infrastructure graph up to date"

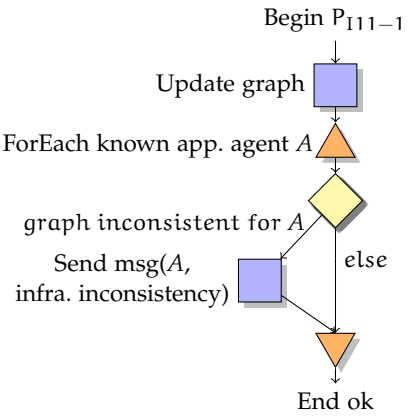


Figure 73: Plan for G_{I11}: "have graph updated"

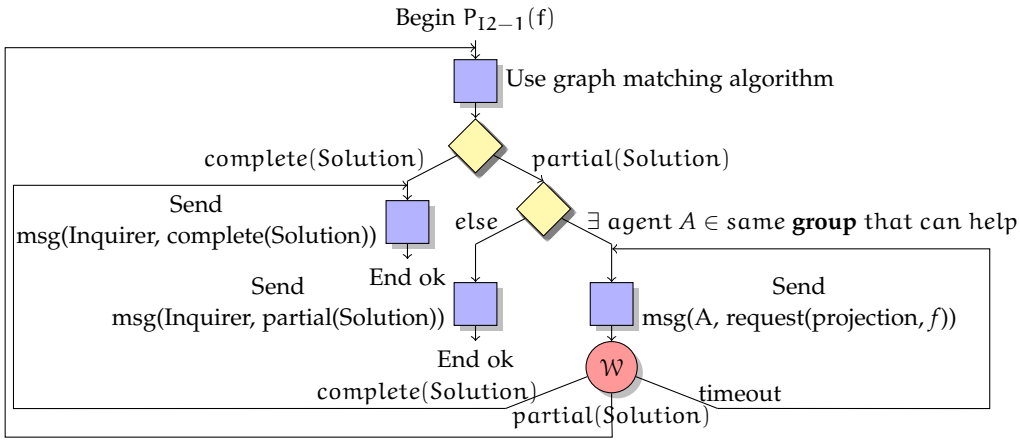


Figure 74: Plan for G_{I2}: "have application projection solution". "Inquirer" can be an *Application Agent* or another *Infrastructure Agent*.

thanks to the deployment software. This realisation was used to evaluate the difficulties of handling the heterogeneity of hardware entities.

8.2.4 *Discussion*

THE APPROACH This work allowed us to test the GPS approach as well as the graphical notation with other researchers and on a second application, thus giving a better understanding of the approach. The model proved its in top-down design for defining the agent behaviour in terms of goals and their relationships, before moving to more concrete levels to describe the action or even goal plans for these goals.

THE ALMA-DERIVED MODEL This modelling work on the AmI application proved that the ALMA can be adapted for general purpose goal-driven agent modelling. The ALMA-derived notation proved useful and more appropriate than the Petri net based notation used in the Interloc scenario. Its closeness to actual code means that a code generation tool could be envisaged to aid the design process. The inclusion of "external" code – represented an an action in the plan – represents a good abstraction for the required level – the agent behaviour. This modelling approach is plan centred and lacks organisational view of the MAS.

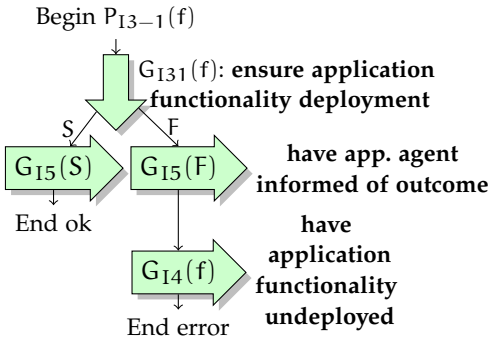


Figure 75: Plan for G_{I3}: "have application functionality deployed"

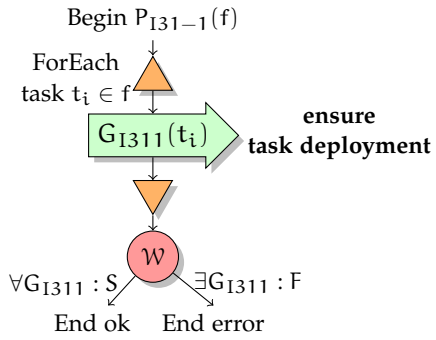


Figure 76: Plan for G_{I31}: "ensure application functionality deployment"

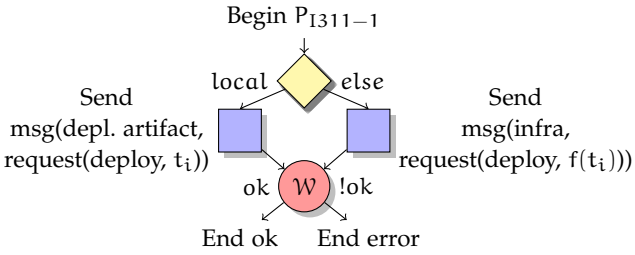


Figure 77: Plan for G_{I311}: "ensure task deployment" (similar to G_{I41}: "ensure task undeployment")

8.3 OVERVIEW

PERCEPTION HANDLING **With GPS, relevant perceptions of the environment are required at the goal reasoning level:** it is the case of messages coming from the visualisation or the measurement agents in Interloc, and various deployment requests in the AmI application. This comes from the fact that certain perceptions can be essential for the global understanding of the agent behaviour. In both applications, messages trigger the adoption of a goal whose achievement is more or less secondary since other measurements or requests can arrive rapidly. That is the reason why it seems to be a good approach to handle these inputs at the upper level of abstraction. A perception filtering strategy, to avoid unnecessary inputs or even overloading the agent, can also appear in this goal plan, possibly through the adoption of a specific goal prior to the adoption of the concerned goal itself.

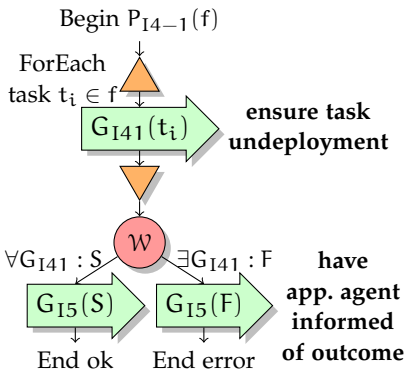


Figure 78: Plan for G_{I4}: "have application functionality undeployed"

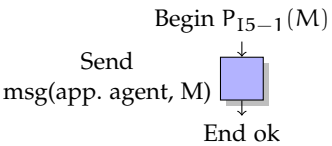


Figure 79: Plan for G_{I5}: "have app. agent informed of outcome"

ERROR HANDLING **With GPS, handling errors is easier to take into account:** this is because errors, whatever their cause, often manifest through the failure of goals. This provides an adequate range of exception mechanisms in the language in which plans are written. Hence, the programmer's effort with regard to fault tolerance is mainly to take into account the processing of non-achieved goals. Of course, this does not concern the goal plan itself, which has to be designed traditionally by explicitly introducing fault tolerance actions. However the amount of code regarding the "classic" action plans is far greater than the amount of the goal plan code. Furthermore, as we show in Part II of this thesis, using multiple levels of goals and plans in an agent help improve its fault tolerance through confinement and the reconfiguration that is built into the goal paradigm.

In the Interloc application, no specific fault tolerance effort has been carried out but a clean processing of non-achieved goals in order to stop the system rather than have it crash. As a consequence, application debugging was greatly facilitated. For the same reasons, the GPS approach proved to facilitate the evolution of the multi-agent system. Thus, the aircraft agent was easily changed into an Unmanned Aerial Vehicle (UAV), with a larger autonomy in the trajectory choice. Here again, the abstraction obtained by separating goals and plans seems to be the reason.

ON THE USE OF GPT Note that, while we use the GPT representation to justify our approach, the GPS is concerned with more general agent models. Also, we do not argue against the GPT formalism, neither do we dispute the plethora of works that use it as a model, but rather we discuss the more general issue of specifying agents with interleaved goal and action levels. Our research complements the works on goal interactions cited in Sec. 2.3.3 (e.g. [103, 109]) as it concerns the agent specification rather than the runtime mechanisms that aim to improve the efficiency, pro-activity, reactivity etc. of the agents. These, as well as other works that use GPTs, such as [102] on intention conflicts, can be used with GPS, and our intuition is that by separating the goal reasoning level, goal interactions can be managed more easily.

Part IV

CONCLUSIONS

9 CONCLUSIONS

"*Monsieur Jourdain* : Par ma foi ! Il y a plus de quarante ans que je dis de la prose sans que j'en susse rien, et je vous suis le plus obligé du monde de m'avoir appris cela."[a]

a English: "*Monsieur Jourdain*: By my faith! For more than forty years I have been speaking prose without knowing anything about it, and now I am grateful to you for having taught this to me."

— Molière, *Le Bourgeois gentilhomme*, 1670

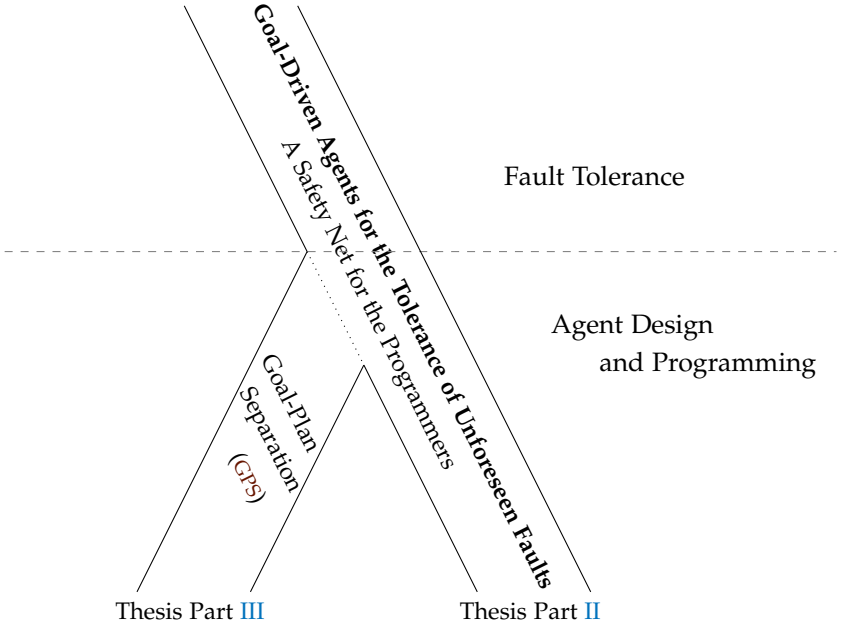


Figure 80: Thesis contributions: the safety net and the Goal-Plan Separation (GPS), with the mention that the Fault Tolerance and Agent Design and Programming domains are by no means disjoint

In this thesis, we started from the questions "What happens if we overlook a fault case? How can we improve the behaviour of the system in such situations?" for which we set out to propose a "safety net" approach for developing software that is tolerant to unforeseen faults. These faults can be introduced accidentally or as accepted risks during the development process. The safety net approach is centred on using the multi-agent paradigm with goal-driven agents, which, together with a series of programming language and platform requirements, provides the desired fault tolerance properties. This work also allowed us to propose the GPS approach for designing and programming agents. This approach requires the reasoning and acting levels to be clearly separated in each agent.

We shall now conclude the thesis by summarising each of the two main contributions and the perspectives they present.

9.1 THE SAFETY NET APPROACH

Another way of describing the idea of the safety net is that we aim to help programmers write software that is tolerant to faults, without them being aware of this, just as Monsieur Jourdain in the quote at the beginning of this chapter was "speaking prose without knowing anything about it".

Due to the nature of the considered faults, we focused on means of runtime detection and handling rather than static (offline) methods and validations, which remain usable as complementary to our approach. Furthermore, for the purpose of the correct continuation of the functioning of the system, we aim at limiting the impact of errors and propagation from the point of detection rather than diagnosing and describing the fault that is to blame.

The safety net approach is comprised of 10 principles, split between programming language, platform and design requirements. In order to define them, we changed the perspective and studied the issue of unforeseen faults following three phases borrowed from the "classic" fault tolerance approaches: detection, confinement and recovery. First we examined what and how can be detected, focusing on means that exist or can be integrated in the programming language and produce exceptions – e.g. single assignment variables and data typing –, as well as objective-based techniques – such as goal verifications – which trigger reactions differently – by calling on the goal life-cycle in the case of goals. Timeout conditions for wait states are also an important mechanism. We then proposed means to confine the system into modules in order to be able to limit the propagation of errors and have a good base for reacting to them in the last phase of the error handling: the recovery. For confinement, the system is designed using a considerable number of agents, each with multiple goals and plans. The last phase, i.e. the recovery, is performed in three steps: (1) dependency handling propagates the error signal to the concerned neighbouring entities, which then (2) repair and (3) reconfigure. The dependency handling is performed transparently by the platform by tracing component interactions and using those traces to inform the concerned components in case of error. For the reparation step, the programmer is required by the language to regularly provide specific procedures in the code, by considering what measures need to be taken in case the current plan needs to be stopped. Reconfiguration is ensured by the agent goals. Goal-driven agents are therefore the central point of our approach. The resulting safety net approach allows the system to react to and recover from errors at agent level while also triggering recovery in agents which were possibly impacted by the detected error. Based on the safety net principles, we proposed the ALMA+ programming language and its platform, whose functioning we illustrated on an application based on a well known multi-agent protocol, the Contract Net Protocol (CNP).

There is an utopian aspect to the target of this thesis: the tolerance to unforeseen faults. Indeed, all failures – notorious or not – can be traced to unforeseen faults, but can we actually prevent all of them from producing disastrous consequences? Our approach aims at extending the fault coverage of programs but cannot guarantee a complete fault coverage.

Both the objective/goal-based and the more classical exception-based detections are dependant on the precision and correctness of their definitions, as for example an objective that is set too loose – a very large timeout value – can compromise the detection mechanism. The example in the introduction of the thesis suffers from a specifications problem: while objective driven, HAL lacks a good definition of

these objectives and ends up endangering the objectives of the other elements of the mission, the crew. As a problem of bad specifications with unforeseen emergent behaviour, this falls outside the scope of the current proposal of safety net.

On the issue of granularity, our experiences showed that an architecture comprised of relatively few large agents is not adapted for illustrating our approach as these agents are often critical for the overall system. Even if the detection is successful, the confinement is too coarse-grained with respect to the system size and the recovery steps do not have much margin to provide the desired fault tolerance, other than orderly stopping the system or restarting processes – in the wide sense – which in the case of protocols may mean that many resources are wasted. A finer segmentation of the system into agents as well as of agents into goals and plans is thus beneficial for tolerating faults with the safety net approach.

APPROACH ACCEPTABILITY FOR PROGRAMMERS The safety net approach, as presented in Chapter 3, is applicable to potentially any programming language and platform. As the idea is to provide a safety net with minimal intrusion, i.e. minimal programmer involvement, the acceptability of the approach comes down to two main components: the design requirements and the language used. To these, the costs in terms of memory, computational and communications overhead are added. The central point of the design requirements is the use of agents with multiple goals and plans. As we note in our work on GPS, one of the benefits of the use of goal-driven agents is that the human concepts are used.

For the safety net approach, we propose ALMA+ as a solution for designing and implementing fault tolerant software. As discussed in Chapter 4, certain aspects of the (ALMA and) ALMA+ language(s), in particular the importance given to the use of rules, may be seen as "exotic" by software developers. However, ALMA+ presents a series of advantages for the tolerance of unforeseen faults, such as the two level code – with the DAG and the "external" code – and the fact that it facilitates the introduction of the dependency handling mechanisms. The graphical notation of the DAG proved to be intuitive enough when introduced to other engineers for our experimentations. In particular, the ALMA-based model used in the CNP+ scenario proved to be more appropriate for modelling and representation than the more analysis-focused Petri nets used in the Interloc example from the GPS contribution. For the "external" code sections, other languages than Prolog can be used, for programmer comfort or other language specific properties, as long as the code confinement is ensured and the code does not produce side effects (e.g. the agent writing memory) that are difficult to trace. As we discuss below, the overhead issue needs more investigation.

BENEFITS The safety net approach offers benefits for easier and safer prototyping, helping lower development costs and the time to market. In the long term, critical applications could also benefit from the increased fault tolerance brought by our approach, due to our complementarity to classic approaches.

The goal-driven multi-agent architecture proposed as a central point of the safety net approach provides good properties for handling system complexity and easily designing distributed applications, thus making the approach suitable for a large array of applications.

As discussed in Chapter 5, agents that do not have the dependency handling mechanisms can be included alongside safety net agents, in other words, the safety net can be used in open systems. The only condition is that the error signal mes-

sages are ignored, which should normally be the case if there is no semantic overlap with any of the used protocols. This is one of the benefits of communication by messages (and the agent architecture). Note, however, that while a safety net agent would be able to function even if no other similar agents were present in the MAS, the multi-agent component brought by the dependency handling would be lost with the reparations and reconfigurations being performed only locally in the safety net agents.

PERSPECTIVES The work covered by the safety net approach is vast and opens many directions for future developments.

Firstly, the ideas put forward in Chapter 4 pave the way for a full integration of the safety net approach with ALMA+ and its platform. This includes covering all error cases discussed in the same chapter, after, in certain cases such as the timeout conditions for the "external" code, studying their pertinence and feasibility. Putting these tools to test with software developers would then allow the approach to be refined and moved closer to an operational software development solution. This would also involve enhancing the ALMA+ prototype (the platform and language features), including providing a, possibly graphic-based, integrated development environment (IDE). These would also allow for a maturation of the ideas for a generic safety net approach, as presented in Chapter 3.

This direction of work would also provide an opportunity to study the way the safety net approach scales. While the use of goal-driven agents and the multi-agent architecture are both recommended for the use for complex and distributed applications, there are other components of the approach that may need a closer inspection with respect to their scalability. In particular, the dependency handling mechanisms may require garbage collecting mechanisms in order to avoid slowing down the system or taking up too much of its memory. Issues related to the domino effect may also be considered for large and long-running systems.

On the subject of dependency handling, the limitations of the proposed solutions should be investigated. In particular, alternative propagation policies could be studied and compared to see if other solutions are better with respect to the fault tolerance – costs trade-off. As mentioned before, care should be taken when choosing the policies so that the agent paradigm be respected, e.g. respecting the agent autonomy [82]. The issue of broken propagation links when agents are terminated could be solved using "ghost" agents that exist only with the purpose of propagating any late error signals.

For the case of global inconsistency errors (i.e. "$\mathtt{true} \Rightarrow \cdots \Rightarrow \mathtt{false}$"), a possible solution would be considering the introduction of elements of default reasoning [60] to replace the total certainty of "$\mathtt{true} \Rightarrow \ldots$" rules and allow for more flexibility in the agent reasoning.

As we discuss in the examples of Ariane 501 and HAL, there are certain limitations to our approach related to the specification issues. A solution to consider for future work is introducing the requirement for organisational norms [24] (mentioned in Sec. 2.2.6 of this thesis), as well as other goal-oriented approaches to design such as goal-oriented interaction protocols [13] as part of the safety net.

In our current work, the focus is on a situation where a single detection occurs. If multiple components detect errors consecutively, the safety net approach ensures that the concerned components reconfigure as they should. However, the system does not take advantage of the information provided by the multiple detections, e.g. indicating a common failing component. As discussed before, the diagnosis

part of the recovery could take advantage of such events, possibly correlated with a reputation mechanism (e.g. as the one used by TibFit, described in Sec. 2.1.3) for filtering the components that are more likely to have caused the error or errors.

As we stated before, the safety net approach is complementary to the "classic" fault tolerance approaches. Extending the approach requires studying which mechanisms can be easily integrated into the approach without this interfering with the main purpose of the safety net – keeping the programmer's involvement minimal. These extensions could help improve the error coverage by adding other existing mechanisms aiming both agent and multi-agent level errors. A direction of work could be extending the detection to other agents, for example using a mechanism similar to the Socially Attentive Monitoring described in Sec. 2.2.4 or trust and reputation from Sec. 2.2.6. These can be used for remotely detecting errors and identifying which entities should be "thrown" into the safety net – to trigger the steps of our approach.

In the same idea of fault tolerance with minimal involvement of the programmer, it could also be investigated to what extent legacy systems can benefit from the propositions in this work [18]. While the tools for dependency tracking are transparent and can be integrated in other platforms, there will be design aspects that may not have been considered when the original system was developed. For example, modularity is one of the central points and so is the goal-plan definition. Furthermore, the reparation specifications that are omnipresent in our plans are not commonly used and would have to be provided through default policies or automatic mechanisms. There are therefore certain requirements for such a retrofit to be possible, but a goal-based agent system may to some extent benefit from a safety net approach.

9.2 THE GOAL-PLAN SEPARATION APPROACH

In the second part of the thesis, we argued that the separation of reasoning and acting is important for the specification and construction of goal driven (e.g. BDI) agents. It was shown that the possibility to mix actions on the environment with goal adoptions in various agent models and languages can have negative effects on the resulting representation and can hinder the development process. A series of examples illustrated what an agent would look like when complying with the *Goal-Plan Separation* approach, with emphasis on the two resulting levels: a *goal reasoning level* and an *action level*. As a possible representation for the former, *goal plans* were introduced. These, while written using the exact same constructs, are the opposite of *action plans* which are allowed to contain actions, but no goal adoptions. The GPS therefore imposes a constraint on agent design that does go against the reflex of adopting a goal in any place it is needed but produces a better-structured result. The GPS also results in agents that "step back and look at the overall picture" rather than react "rashly" to their current situation, making it suitable for "strategic", proactive and complex behaviours, without necessarily neglecting the reactive ones, e.g. *GP2* in Fig. 58. We experimented with the GPS approach in two applications: a maritime patrol application and a software for deploying AmI applications on a distributed infrastructure. The importance of tidy agent representation lies with the ease of development, which can, in turn, facilitate the wide-scale adoption of the development model. Furthermore, a clean representation that helps diminish the number of design and development faults and also improves maintainability helps bring the overall project costs down.

The main downsides of this approach spring from the fact that it is a supplementary constraint that is placed on the programmer. He or she can thus be tempted to circumvent it, for example by creating a specific goal for each action that would otherwise be at the same level as other goals. This is not the purpose of the approach and such implementations should be looked for during the code verifications. In the same time, as any constraint, it can affect the appeal for programming goal-driven agents. However, as it results into more structured and clearer code, we consider this is not the case.

As presented in Chapter 8, we have already began the empirical evaluation of the approach and its advantages on agent design through two applications. Among the characteristics of the approach that appear promising, we note the fact that the goal-based GPS should scale well as it promotes a multi-level model that allows for the definition to manage complex and large agents. Furthermore, the use of plans and goal plans based on the same language simplifies the development process, while allowing both complex "strategic" and simple "reactive" behaviours (GP3 vs. GP2 in Fig. 58 from Sec. 7.3). Increasing the number of goals and plans also requires taking into consideration goal and plan interferences.

GPS PERSPECTIVES On the side of BDI agent modelling there are many studies on goal representations and goal life-cycles. However, the higher level that is placed above these automata is less examined in the literature. Our work contributes to this discussion by clearly separating the goal reasoning level, opening research opportunities into the formalisms for specifying this level, the goal plans presented here being only a possible direction. Among other primitives, the handling of temporal constraints is important for agent systems and should be taken into consideration.

If we take into account the MAS level, another perspective opened by the GPS is for allowing agents to exchange data on their behaviours for cooperation or coordination purposes. Having a clearly-defined goal reasoning level allows them to exchange only this level, leaving unnecessary and specific details – the actions and possibly the lower level goals – out of the discussion. Going multi-agent from the GPS level can be tackled in two manners:

- via a top-down approach, where we keep the designer/methodology state of mind and we ask ourselves how to build a MAS with GPS agents. This corresponds to a Prometheus-GPS approach.
- by considering the agent's point of view with respect to interactions with other agents, in various cases that involve cooperation, fault tolerance and delegation.

9.3 PUTTING IT ALL BACK TOGETHER

As we argue in our work on the Goal-Plan Separation, the part of the agent that is in charge of reasoning on goals should be clearly separated from the actions. Apart from the gains in code clarity and traceability, this separation offers another important advantage: as acting is present in plans and separated from the goal reasoning level, if proper confinement is assured at plan level, any error would be limited to the running plan and the corresponding actuators and beliefs, leaving the goal reasoning level functional. While the GPS approach is independent from the safety net approach, the Goal-Plan Separation remains a good development practice for

designing and programming goal-driven agents. In the long run, the goal is to integrate the two, together with a system-level development methodology such as Prometheus [114] in order to propose a complete development methodology for reliable applications.

Autonomy is both an input and a by-product of our work. On the one hand, we promoted autonomous agents and considering the autonomy of others during the development process for loose coupling and we used goals which are important for agent autonomy. On the other hand, having a system that tolerates unforeseen faults makes it less susceptible to requiring user or programmer assistance to achieve its objectives, thus being more autonomous.

For today's world where distribution is everywhere and systems become more and more complex, the Multi-Agent Systems with goal-directed agents are an excellent paradigm through their fault tolerance properties, high level of abstraction and closeness to the human reasoning. Furthermore, in the right setup – when the safety net principles are applied – the paradigm can help programmers write fault tolerant software without even knowing it.

Part V

APPENDIX

A

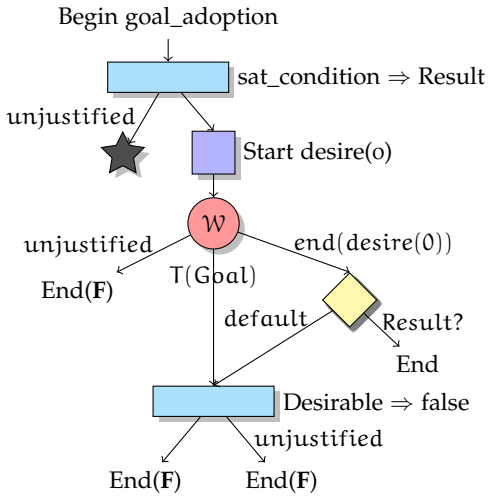
CONTROLLING GOAL EXECUTION

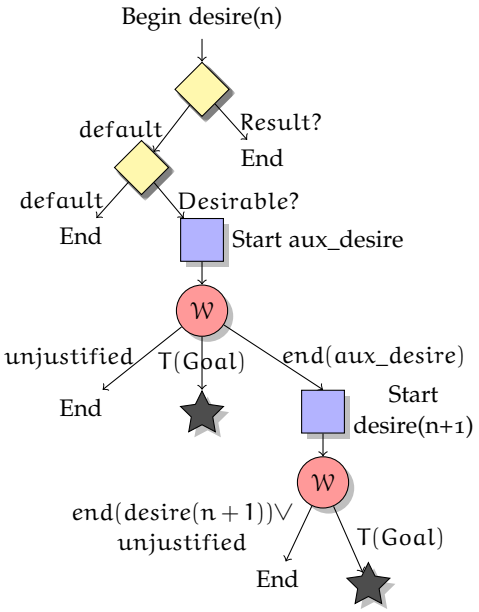
In Fig. 81 we present the model corresponding to the goal automaton implementation in ALMA for ALMA+. It contains four Reasoning Threads (RTs):

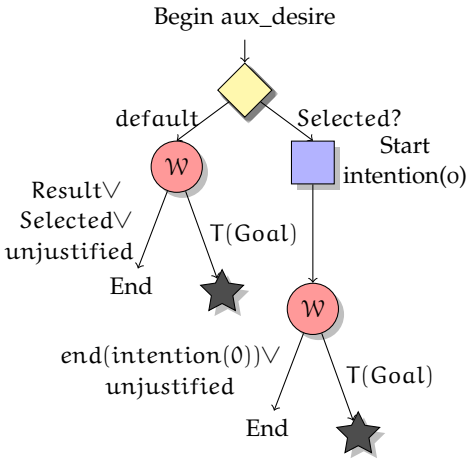
- *goal adoption* – for adding the satisfaction condition and handling the goal timeout and unjustification, making sure the outcome is "failed" in these cases;
- *desire* and *aux_desire* – for allowing the goal to be in a desire state without being an intention (`Selected` is not believed, but `Desirable` is);
- *intention* – for cycling through the plans until the goal result is set, there are no more plans or the goal is stopped or set back into a desire state.

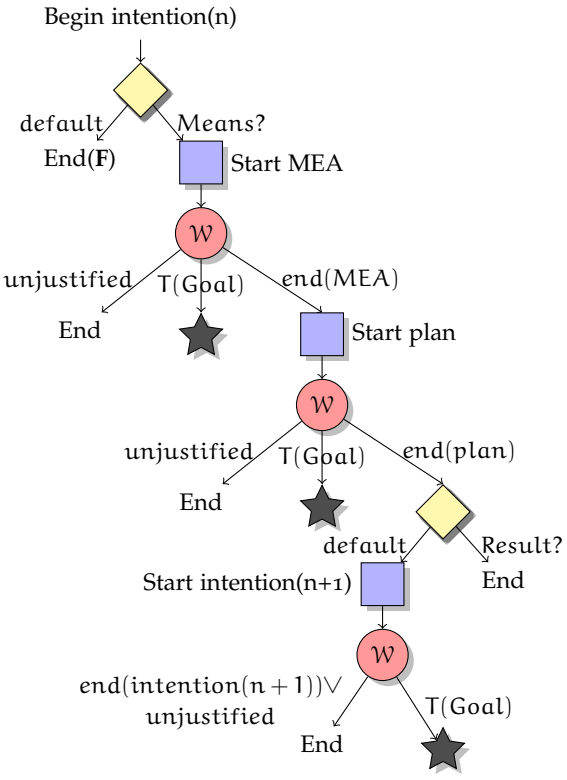
The `unexpected` nodes at this level (the node is normally defined for ALMA+, not ALMA) correspond to abnormal executions which are to be treated just as unanticipated errors in the goal rules: the goal execution is stopped and the goal result is set to "failed".

In order to be able to keep a global goal timeout, all intermediary threads are synchronised (i.e. the parent thread cannot finish as long as its children are still executing).









Belief	Type
Desirable (for all 4 RTs)	simple, IN/OUT
Selected (for all 4 RTs)	simple, IN
Result (for all 4 RTs)	simple, OUT
Plan outcome (for all 4 RTs and plan)	set

Figure 81: The goal life-cycle (automaton) for ALMA+. Top left: *goal adoption*, top right: *desire* thread, bottom left: *aux desire* thread, bottom right: *intention* thread. T(x) = Timeout(*x* seconds). End(**F**) ends the thread only after marking the goal as "failed", thus hiding a *reasoning* node.

B

MODELS OF THE CNP+ AGENTS

B.1 THE INITIATOR AGENT

B.1.1 *Agent Goals*

Goal Description		
Name	main_goal	0
Satisfaction	$plan\ outcome = success$	
Means-end analysis	Ordered list	
Time out	300s	
Required beliefs	$\emptyset$	
Produced beliefs	$\emptyset$	
Plans	P_{I0-1}	Goal plan

Goal Description		
Name	know_acquaintances	1
Satisfaction	$plan\ done \wedge length(Acquaintances) > 1$	
Means-end analysis	Ordered list	
Time out	150s	
Required beliefs	$\emptyset$	
Produced beliefs	Acquaintances	list
Plans	P_{I1-1}	Action plan

Goal Description		
Name	have_proposals	2
Satisfaction	$plan\ done \wedge \neg empty(Proposals)$	
Means-end analysis	Ordered list	
Time out	150s	
Required beliefs	Acquaintances, CFP	list, simple
Produced beliefs	Proposals	list
Plans	P_{I2-1}	Goal plan

Goal Description		
Name	have_one_acquaintance_dealt_with	2-1
Satisfaction	$plan\ outcome = success$	
Means-end analysis	Ordered list	
Time out	100s	
Required beliefs	Acquaintance, CFP	simple (both)
Produced beliefs	ProposalSet	set
Plans	P_{I2-1-1}	Action plan

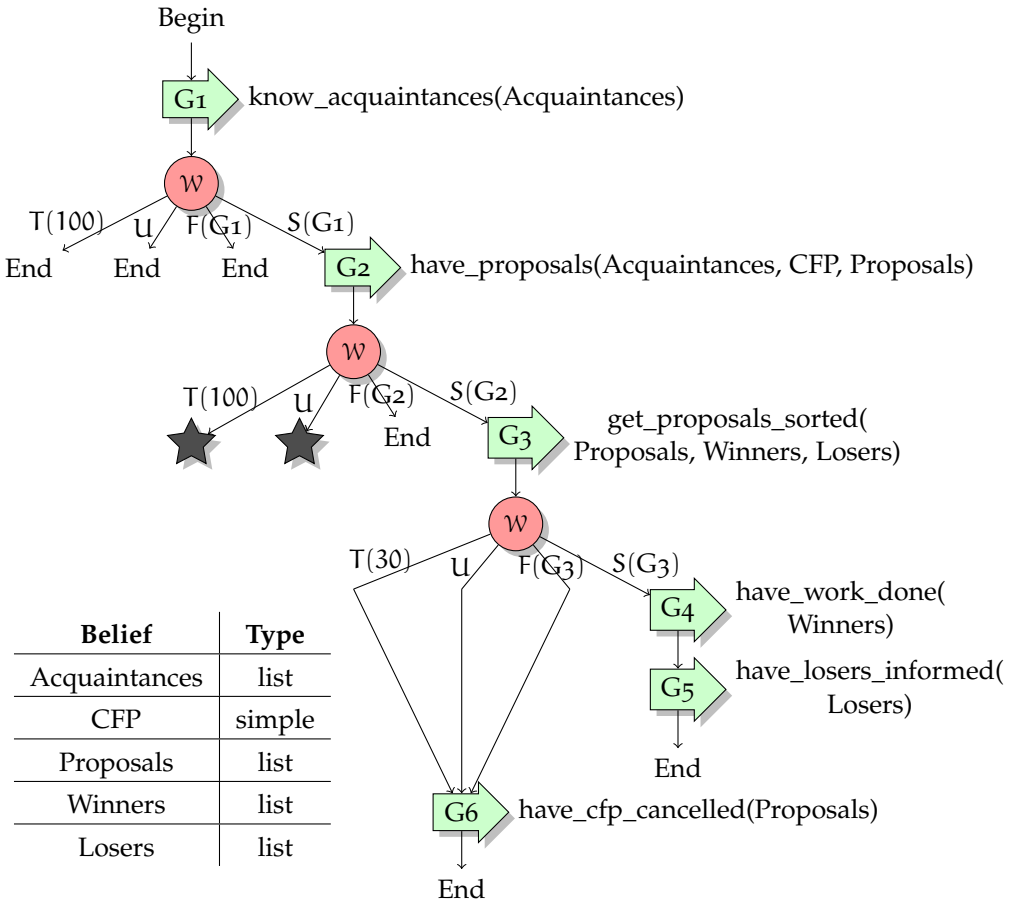
Goal Description		
Name	get_proposals_sorted	3
Satisfaction	$plan\ done \wedge \neg empty(Winners)$	
Means-end analysis	Ordered list	
Time out	30s	
Required beliefs	Proposals	list
Produced beliefs	Winners, Losers	list, list
Plans	P_{I3-1}	Plan (no goals, no actions)

Goal Description		
Name	have_work_done	4
Satisfaction	$plan\ done \wedge ok(Results)$	
Means-end analysis	Ordered list	
Time out	100s	
Required beliefs	Winners	list
Produced beliefs	Results	simple
Plans	P_{I4-1}	Action plan

Goal Description		
Name	have_losers_informed	5
Satisfaction	$plan\ done$	
Means-end analysis	Ordered list	
Time out	40s	
Required beliefs	Losers	list
Produced beliefs	$\emptyset$	
Plans	P_{I5-1}	Action plan

Goal Description		
Name	have_cfp_cancelled	6
Satisfaction	$plan\ done$	
Means-end analysis	Ordered list	
Time out	40s	
Required beliefs	Proposals	list
Produced beliefs	$\emptyset$	
Plans	P_{I6-1}	Action plan

B.1.2 *Agent Plans*



Belief	Type
Acquaintances	list
CFP	simple
Proposals	list
Winners	list
Losers	list

Figure 82: Plan P_{I0-1} - main goal plan

Belief	Type
DirectoryReference	simple
Acquaintances	list, OUT

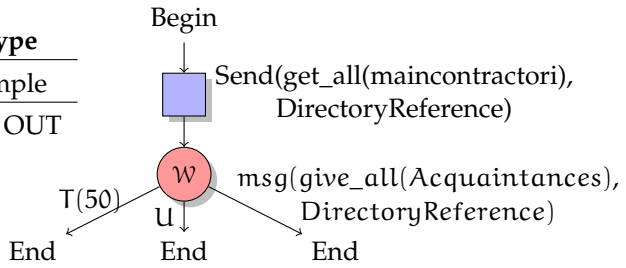
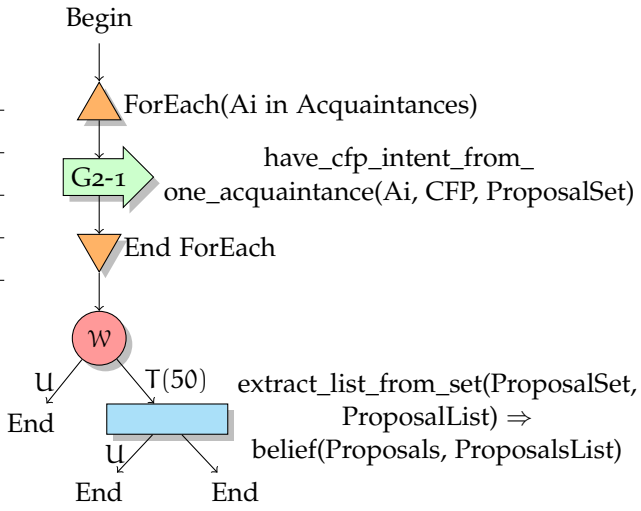


Figure 83: Plan P_{I1-1} - "know acquaintances" - for enquiring *DirectoryReference* (e.g. another agent, a web service etc.) for a list of all agents of type *maincontractori*

Belief	Type
CFP	simple, IN
Acquaintances	list, IN
Ai	simple
ProposalSet	set
Proposals	list, OUT



Belief	Type
CFP	simple, IN
Ai	simple, IN
ProposalSet	set, OUT

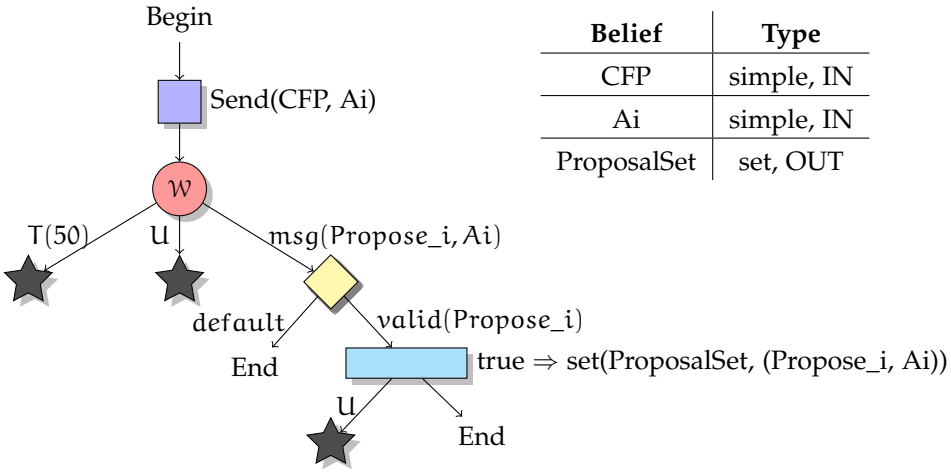
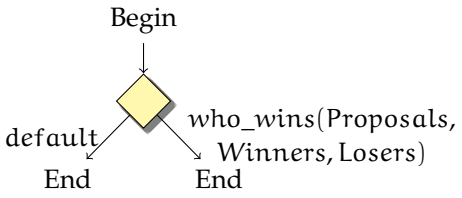


Figure 84: Plans P_{I2-1} "have proposals" and P_{I2-1-1} "have cfp intent from one acquaintance"

Belief	Type
Proposals	list, IN
Winners	list, OUT
Losers	list, OUT



Belief	Type
Winners	list, IN
Result	simple, OUT

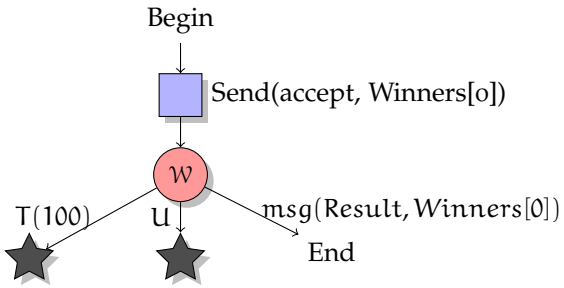
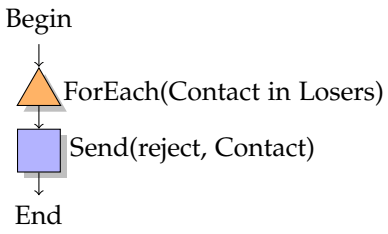


Figure 85: Plans P_{I3-1} - "get proposals sorted" and P_{I4-1} - "have work done". Plan P_{I3-1} simply uses an external method to sort according so some criteria and produce the two lists: Winners and Losers. The goal will fail if the Winner list is empty. Plan P_{I4-1} sends "accept" message to the **single** winner and waits for the reply.

Belief	Type
Losers	list, IN



Belief	Type
Proposals	list, IN

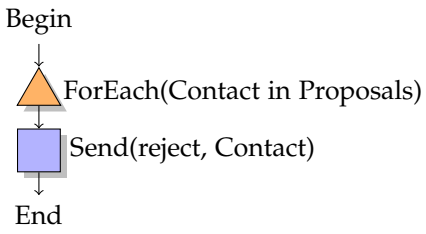


Figure 86: Plans P_{I5-1} "have losers informed" (left) and P_{I6-1} "have cfp cancelled" (right) to send "reject" message to all the refused agents

B.2 THE MAIN CONTRACTOR AGENT

B.2.1 *Agent Goals*

Goal Description		
Name	main_goal	0
Satisfaction	$plan\ outcome = success$	
Means-end analysis	Ordered list	
Time out	300s	
Required beliefs	$\emptyset$	
Produced beliefs	$\emptyset$	
Plans	P_{MC_i0-1}	Goal plan

Goal Description		
Name	initiator_part	1
Satisfaction	$plan\ otucome = success$	
Means-end analysis	Ordered list	
Time out	200s	
Required beliefs	Initiator, CFW, Desirable-Worker, Winners, Neg, Results	simple (Winners is list)
Produced beliefs	$\emptyset$	
Plans	P_{MC_i1-1}	Goal plan

Goal Description		
Name	worker_part	2
Satisfaction	$plan\ outcome = success$	
Means-end analysis	Ordered list	
Time out	200s	
Required beliefs	Initiator, DesirableInitiator, Winners, Neg, Results	simple (Winners is list)
Produced beliefs	$\emptyset$	
Plans	P_{MC_i2-1}	Goal plan

Goal Description		
Name	have_refuse_sent	3
Satisfaction	$plan\ done$	
Means-end analysis	Ordered list	
Time out	40s	
Required beliefs	Initiator	simple
Produced beliefs	$\emptyset$	
Plans	P_{MC_i3-1}	Action plan

Goal Description		
Name	know_acquaintances	1-1
Satisfaction	$plan\ done \wedge length(Acquaintances) > 1$	
Means-end analysis	Ordered list	
Time out	150s	
Required beliefs	$\emptyset$	
Produced beliefs	Acquaintances	list
Plans	$P_{MC_i 1-1-1}$	Action plan

Goal Description		
Name	have_proposals	1-2
Satisfaction	$plan\ done \wedge \neg empty(Proposals)$	
Means-end analysis	Ordered list	
Time out	150s	
Required beliefs	Acquaintances, CFW	list, simple
Produced beliefs	Proposals	list
Plans	$P_{MC_i 1-2-1}$	Goal plan

Goal Description		
Name	have_one_acquaintance_dealt_with	1-2-1
Satisfaction	$plan\ outcome = success$	
Means-end analysis	Ordered list	
Time out	100s	
Required beliefs	Acquaintance, CFW	simple (both)
Produced beliefs	ProposalSet	set
Plans	$P_{MC_i 1-2-1-1}$	Action plan

Goal Description		
Name	get_proposals_sorted	1-3
Satisfaction	$plan\ done \wedge \neg empty(Winners)$	
Means-end analysis	Ordered list	
Time out	30s	
Required beliefs	Proposals	list
Produced beliefs	Winners, Losers	list, list
Plans	$P_{MC_i 1-3-1}$	Plan (no goals, no actions)

Goal Description		
Name	have_winners_preliminary_accepted	1-4
Satisfaction	$plan\ outcome = success$	
Means-end analysis	Ordered list	
Time out	60s	
Required beliefs	Winners	list
Produced beliefs	$\emptyset$	
Plans	$P_{MC_i 1-4-1}$	Action plan

Goal Description		
Name	have_losers_informed	1-5
Satisfaction	$plan\ done$	
Means-end analysis	Ordered list	
Time out	40s	
Required beliefs	Losers	list
Produced beliefs	$\emptyset$	
Plans	$P_{MC_i 1-5-1}$	Action plan

Goal Description		
Name	have_work_done	1-6
Satisfaction	$plan\ done \wedge belief(Results, \_)$	
Means-end analysis	Ordered list	
Time out	100s	
Required beliefs	Winners	simple
Produced beliefs	Results	simple
Plans	$P_{MC_i 1-6-1}$	Action plan

Goal Description		
Name	have_accept_revoked	1-7
Satisfaction	$plan\ done$	
Means-end analysis	Ordered list	
Time out	40s	
Required beliefs	Winners	list
Produced beliefs	$\emptyset$	
Plans	$P_{MC_i 1-7-1}$	Action plan

Goal Description		
Name	have_cfw_cancelled	1-8
Satisfaction	$plan\ done$	
Means-end analysis	Ordered list	
Time out	40s	
Required beliefs	Proposals	list
Produced beliefs	$\emptyset$	
Plans	$P_{MC_i 1-8-1}$	Action plan

Goal Description		
Name	have_proposal_sent	2-1
Satisfaction	$plan\ outcome = success$	
Means-end analysis	Ordered list	
Time out	60s	
Required beliefs	Initiator	simple
Produced beliefs	$\emptyset$	
Plans	$P_{MC_i 2-1-1}$	Action plan

Goal Description		
Name	have_results_sent	2-2
Satisfaction	$plan\ outcome = success$	
Means-end analysis	Ordered list	
Time out	40s	
Required beliefs	Initiator, Results	simple, simple
Produced beliefs	$\emptyset$	
Plans	$P_{MC_i 2-2-1}$	Action plan

B.2.2 *Agent Plans*

Belief	Type
Initiator	simple
CFP	simple
CFW	simple
G1 (goal outcome)	simple
DesirableWorker	simple
G2 (goal outcome)	simple
DesirableInitiator	simple
Winners	list
Neg	simple
Results	simple

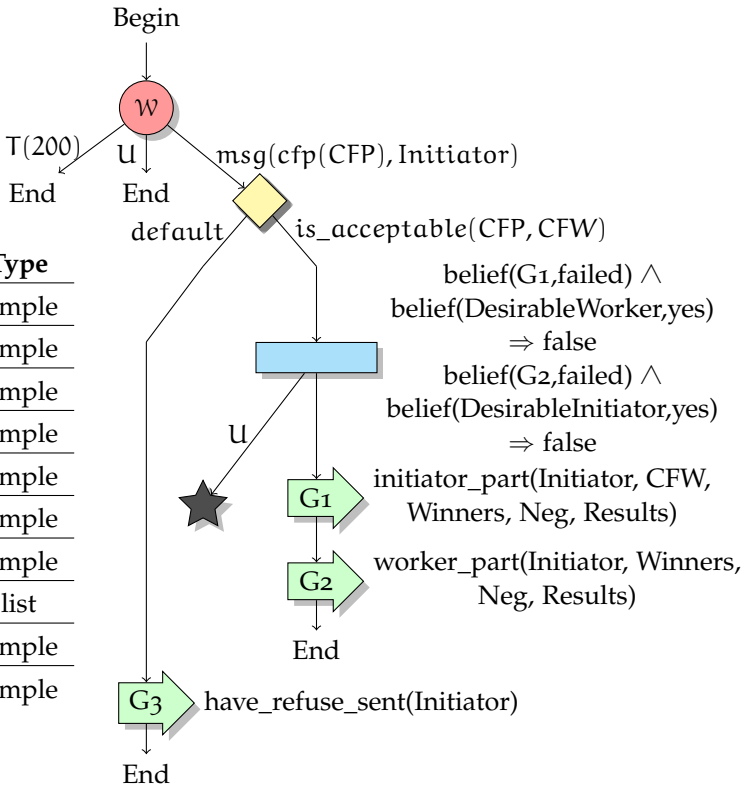


Figure 87: Main goal plan for the MCi agents (P_{MCi0-1}) - it adopts two complementary goals: one for dealing with the worker agents to whom it acts as an initiator (G1) and one for managing the initial CFP, in which case it acts as a worker (G2). The two goals G1 and G2 have strong links so the failure of one causes the other to stop and this is ensured by the two rules added before their adoption. "Is_acceptable(CFP, CFW)" is a predicate that returns `true` and instantiates CFW if local conditions are met for the MCi to try and reply positively to the CFP (e.g. demand parameters are within acceptable limits, resources to spare etc.).

Belief	Type
Initiator	simple, IN

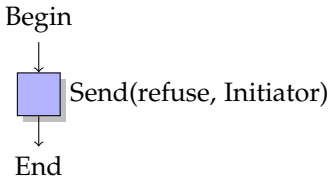
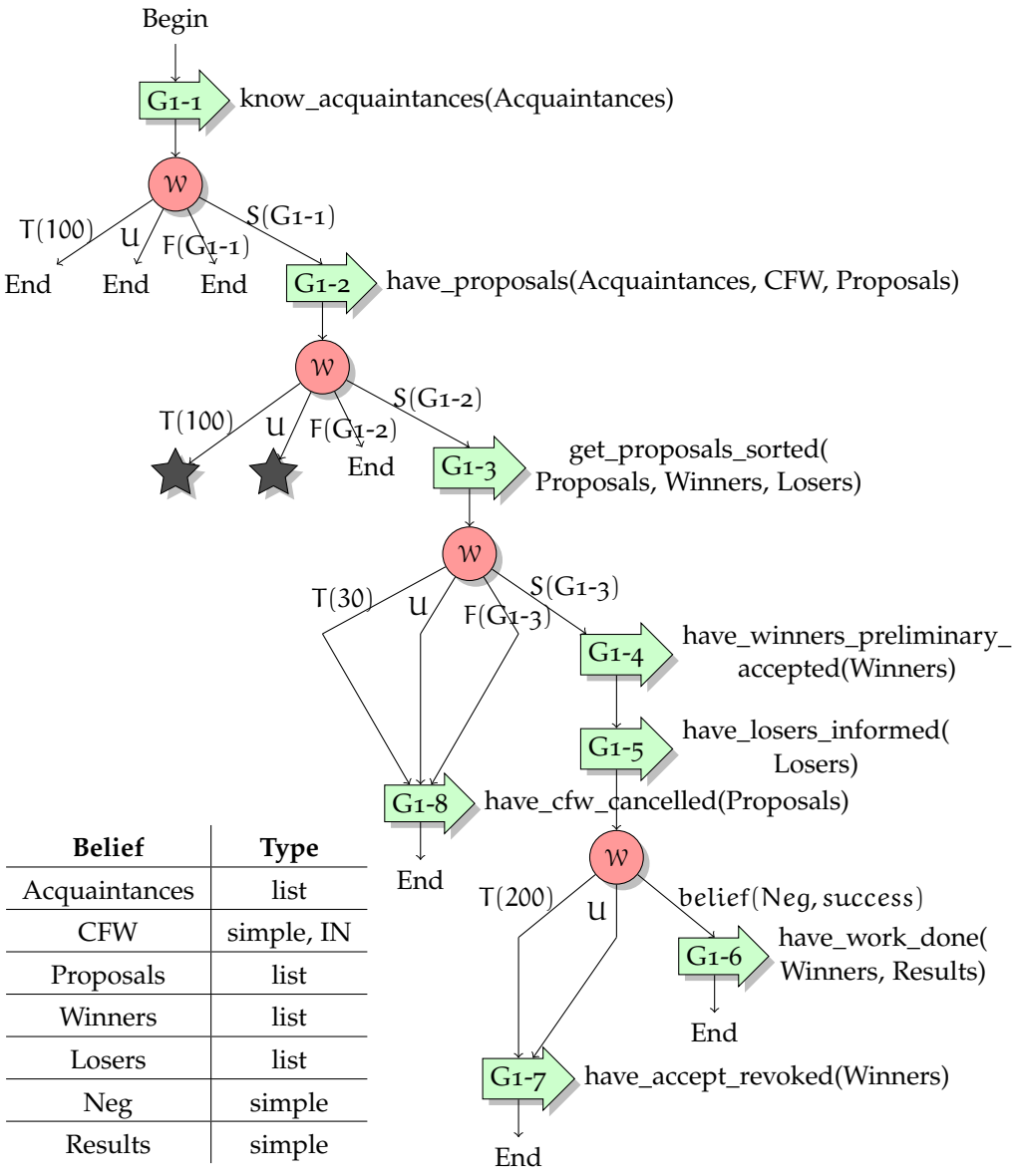


Figure 88: Plan P_{MCi3-1} "have refuse sent" that refuses the CFP with a message to the Initiator



Belief	Type
Acquaintances	list
CFW	simple, IN
Proposals	list
Winners	list
Losers	list
Neg	simple
Results	simple

Figure 89: Plan P_{MCi1-1} "initiator part". This goal plan is very similar to the one in the Initiator agent, but with supplementary elements linked to its interaction to the worker part: the agent needs to wait for the negotiation with the Initiator agent to be successful before giving the green light to the Worker agents.

Belief	Type
DirectoryReference	simple
Acquaintances	list, OUT

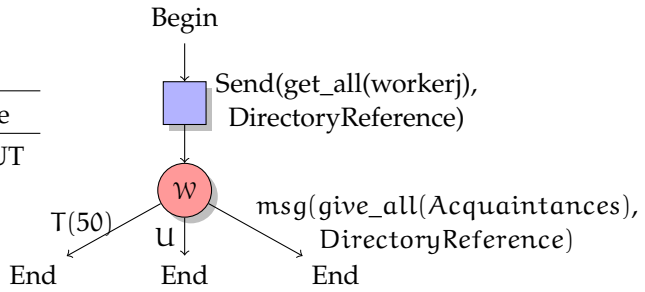
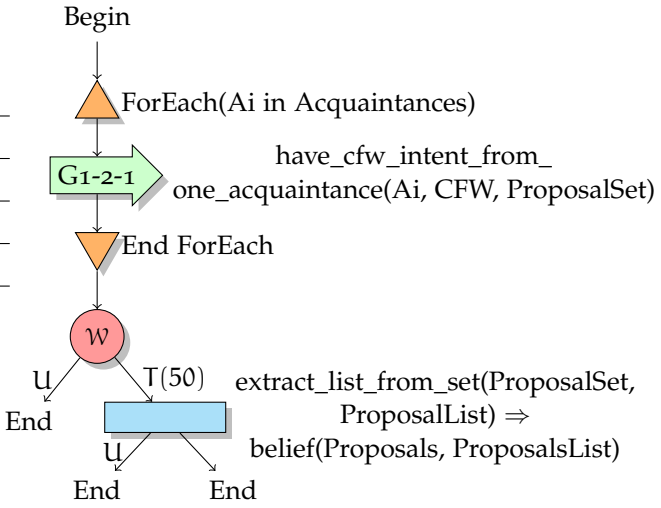
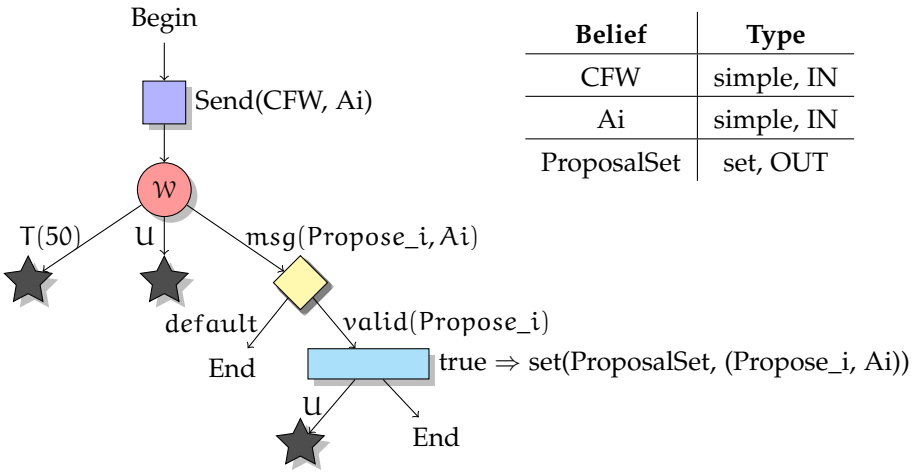


Figure 90: Plan $P_{MCi1-1-1}$ - "know acquaintances" - for enquiring *DirectoryReference* (e.g. another agent, a web service etc.) for a list of all agents of type *workerj*

Belief	Type
CFW	simple, IN
Acquaintances	list, IN
Ai	simple
ProposalSet	set
Proposals	list, OUT





Belief	Type
CFW	simple, IN
Ai	simple, IN
ProposalSet	set, OUT

Figure 91: Plans $P_{MCi1-2-1}$ "have proposal set" and $P_{MCi1-2-1-1}$ "have cfw intent from one acquaintance"

Belief	Type
Proposals	list, IN
Winners	list, OUT
Losers	list, OUT

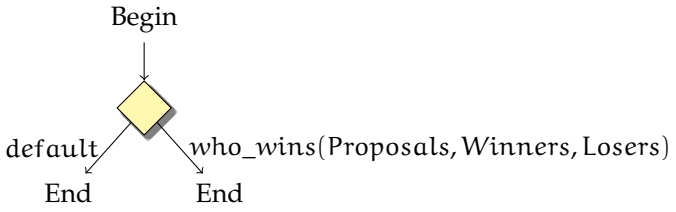
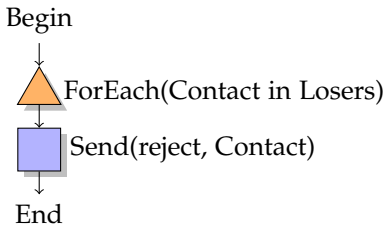


Figure 92: Plan $P_{MCi1-3-1}$ - "get proposals sorted" simply uses an external method to sort according so some criteria and produce the two lists: Winners and Losers. The goal will fail if no Winners list is produced.

Belief	Type
Losers	list, IN



Belief	Type
Proposals	list, IN

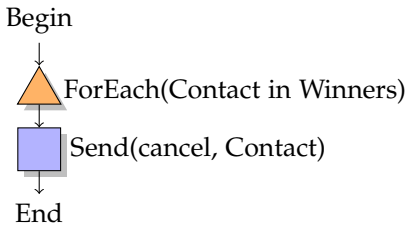
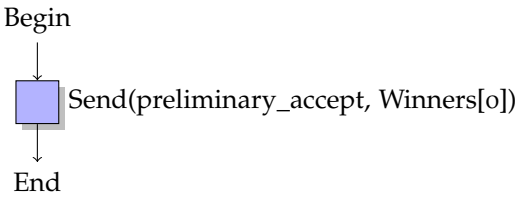


Figure 93: Plans $P_{MCi1-5-1}$ "have losers informed" (left) to send "reject" message to all the refused agents ($P_{MCi1-8-1}$ "have cfw cancelled" is identical but uses Proposals as list of receivers) and $P_{MCi1-7-1}$ "have accept revoked" (right) to send "cancel" message to all previously accepted worker agents (for example in case of failed negotiations with the Initiator).

Belief	Type
Winners	list, IN



Belief	Type
Winners	list, IN
Result	simple, OUT

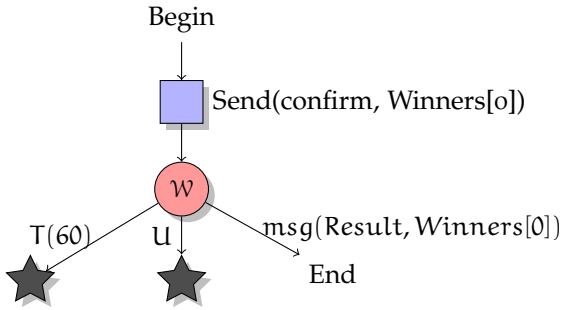


Figure 94: Plans $P_{MCi1-4-1}$ "preliminary accept" and $P_{MCi1-6-1}$ "have work done" that confirms the first message to the **single** winner and waits for the reply

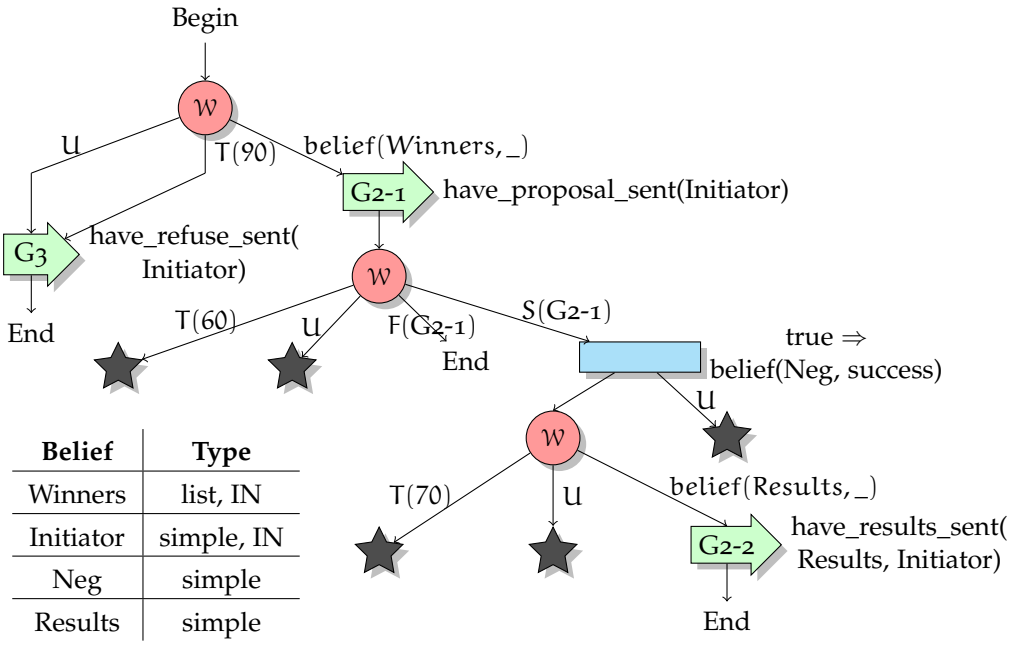


Figure 95: Plan P_{MCi2-1} "worker part". Setting the "Neg" belief to `true` signals to the initiator part that it can continue its CFW, while adding a contradiction to the desirable belief causes the initiator part to stop.

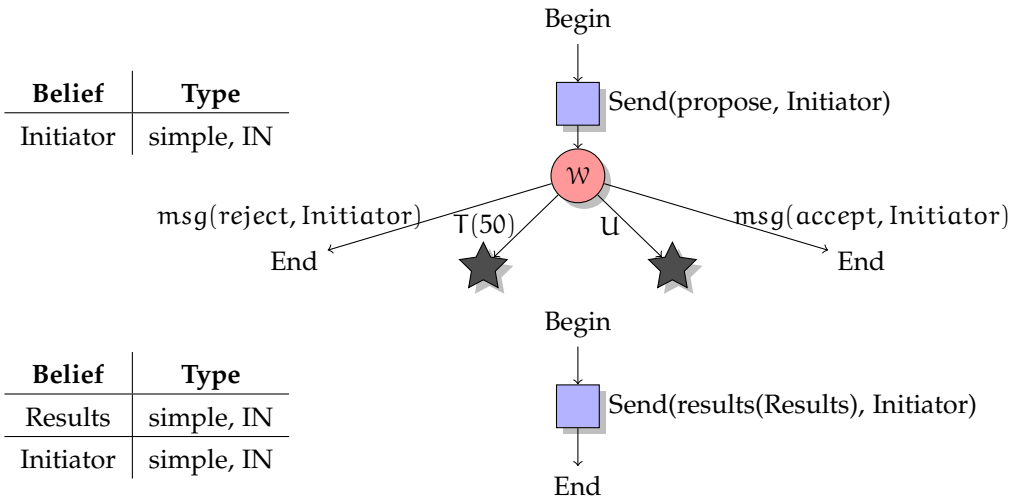


Figure 96: Left: $P_{MCi2-1-1}$ "have proposal sent" that sends a "propose" message to the Initiator and waits for the reply. The goal is achieved if the reply is "accept". Right: $P_{MCi2-2-1}$ "have results sent" sends the Results to the Initiator.

B.3 THE WORKER AGENT

B.3.1 *Agent Goals*

Goal Description		
Name	main_goal	0
Satisfaction	plan outcome = success	
Means-end analysis	Ordered list	
Time out	300s	
Required beliefs	$\emptyset$	
Produced beliefs	$\emptyset$	
Plans	$P_{W_j 0-1}$	Goal plan

Goal Description		
Name	have_refuse_sent	1
Satisfaction	plan done	
Means-end analysis	Ordered list	
Time out	20s	
Required beliefs	MCi	simple
Produced beliefs	$\emptyset$	
Plans	$P_{W_j 1-1}$	Action plan

Goal Description		
Name	have_proposal_sent	2
Satisfaction	plan outcome = success	
Means-end analysis	Ordered list	
Time out	60s	
Required beliefs	MCi	simple
Produced beliefs	$\emptyset$	
Plans	$P_{W_j 2-1}$	Action plan

Goal Description		
Name	have_work_done	3
Satisfaction	belief(Results, _)	
Means-end analysis	Ordered list	
Time out	30s	
Required beliefs	CFW	simple
Produced beliefs	Results	simple
Plans	$P_{W_j 3-1}$	Action plan

Goal Description		
Name	have_mci_informed_of_failure	4
Satisfaction	*plan done*	
Means-end analysis	Ordered list	
Time out	20s	
Required beliefs	MCi	simple
Produced beliefs	$\emptyset$	
Plans	$P_{W_j 4-1}$	Action plan

Goal Description		
Name	have_results_sent	5
Satisfaction	*plan outcome* = *success*	
Means-end analysis	Ordered list	
Time out	40s	
Required beliefs	MCi, Results	simple, simple
Produced beliefs	$\emptyset$	
Plans	$P_{W_j 5-1}$	Action plan

B.3.2 *Agent Plans*

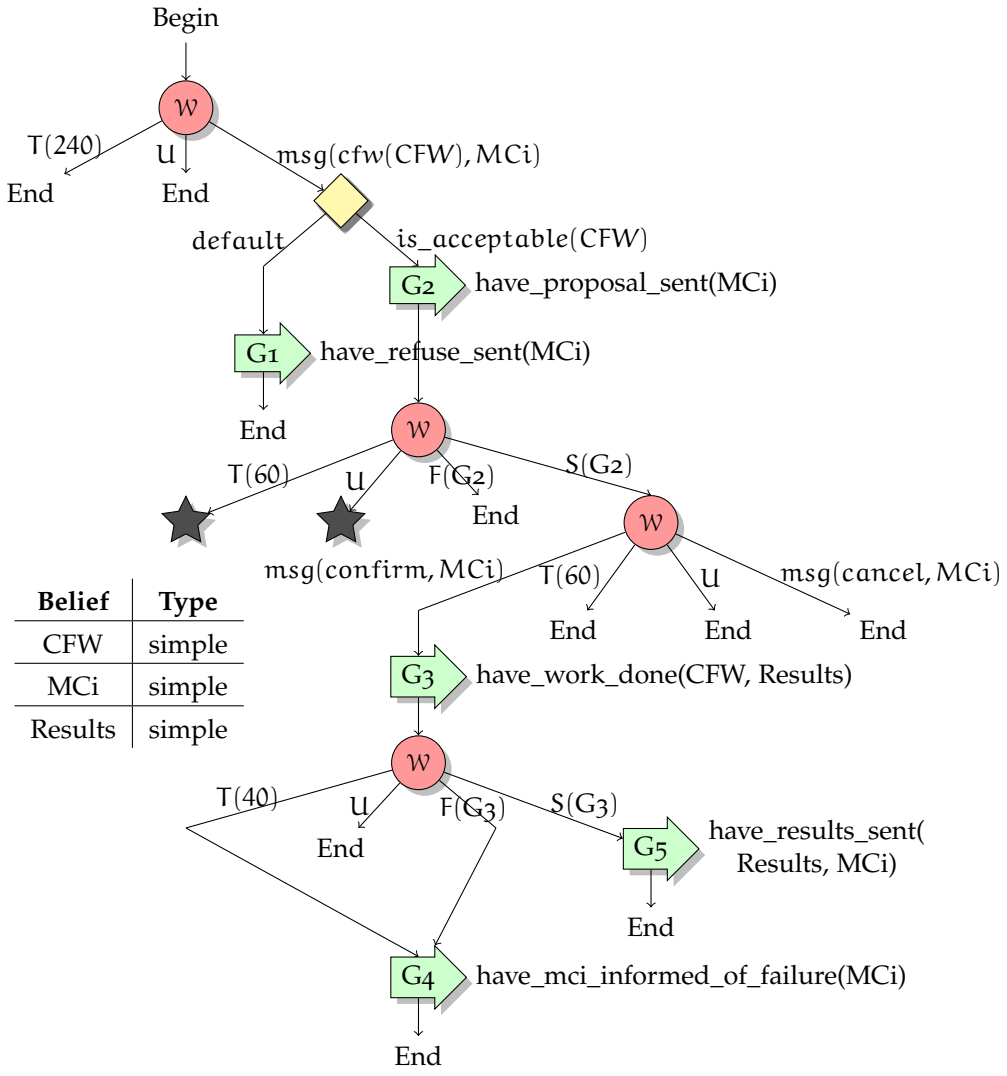


Figure 97: Main goal plan for the Wj agents P_{Wj0-1}

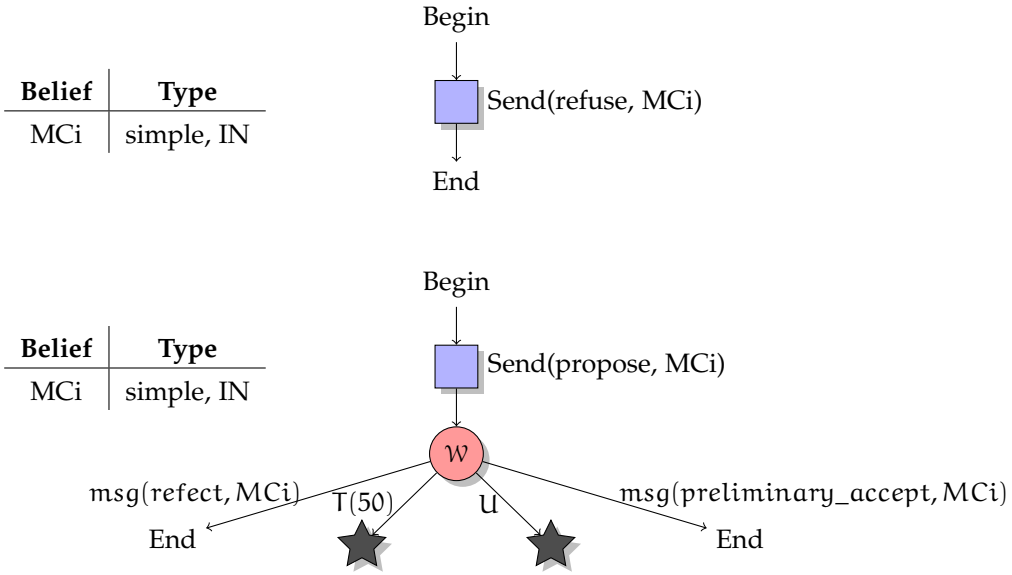


Figure 98: Plans P_{Wj1-1} "have refuse sent" and P_{Wj2-1} "have proposal sent", whose goal is achieved if the reply is "preliminary accept"

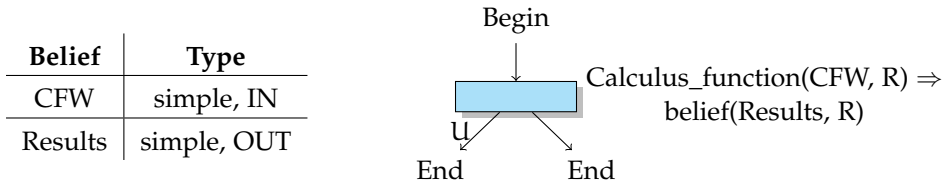


Figure 99: Plan P_{Wj3-1} "have work done"

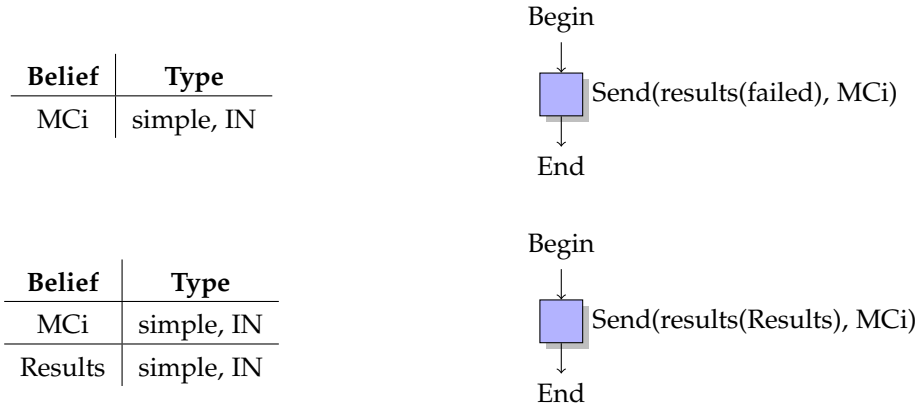


Figure 100: Plans P_{Wj4-1} "have mci informed of failure" and P_{Wj5-1} "have results send"

C

ERROR RESPONSE BY LOCATION OF OCCURRENCE IN CNP+

In this appendix, we give a more complete list of errors by location for the CNP+ example in Chapter 5, as an extension of the example in Sec. 5.4.2.

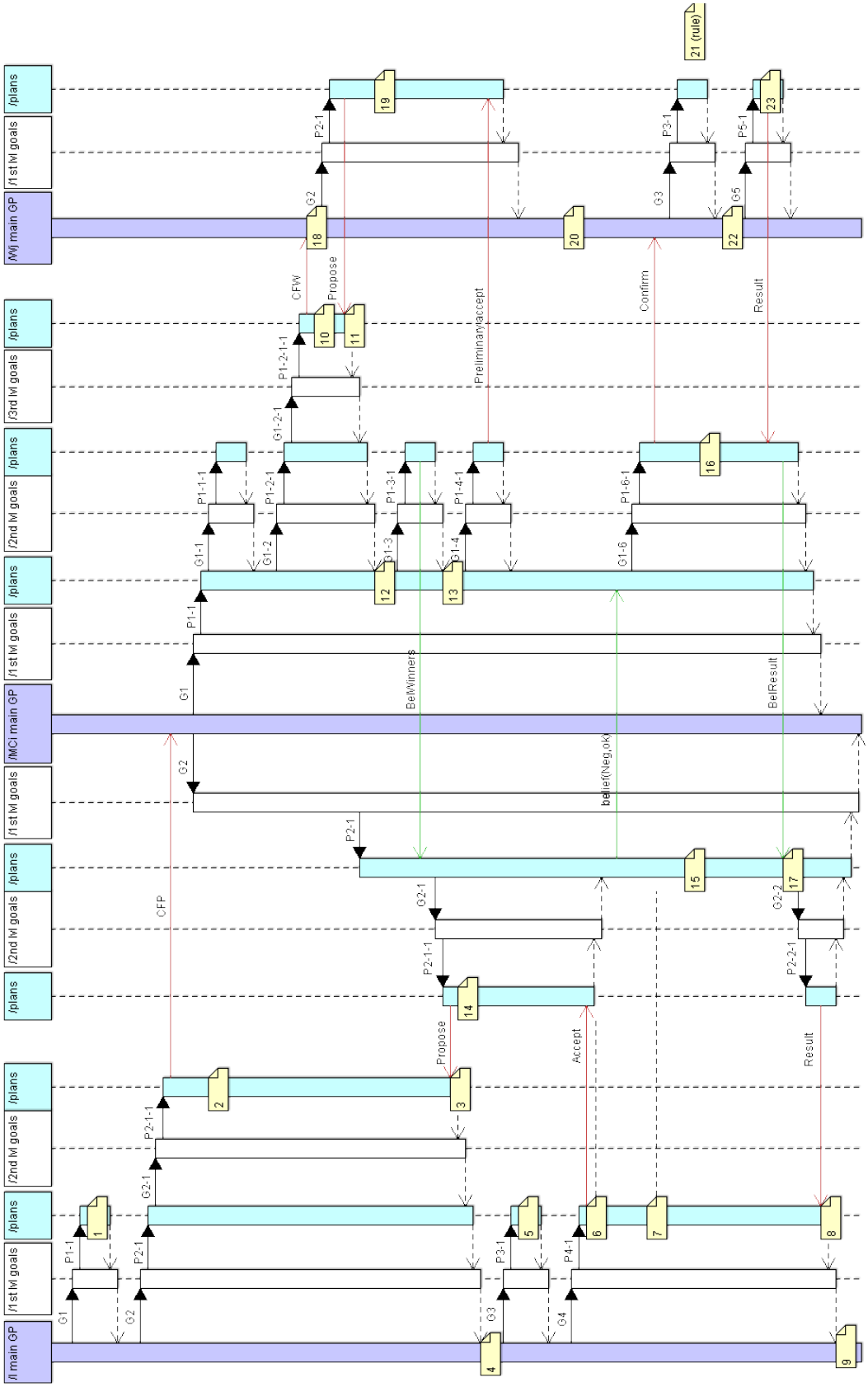


Figure 101: Sequence diagram for a successful CNP+ negotiation (the Initiator agent ends up with a result). Yellow tags correspond to the error cases from Tables 17-19.

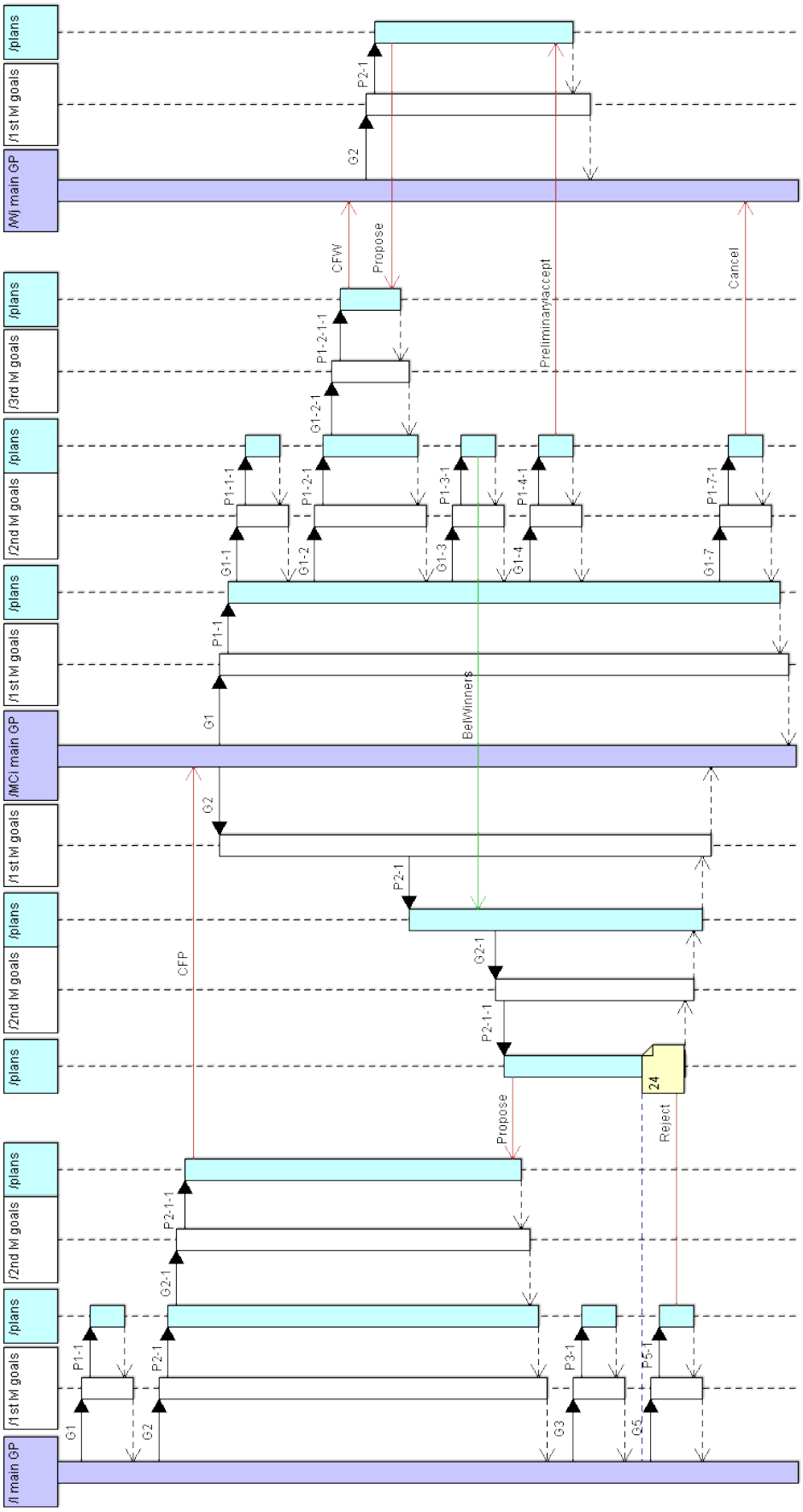


Figure 102: Sequence diagram for a successful CNP+ negotiation (the Initiator agent ends up with a result). The yellow tag corresponds to an error case from Table 20.

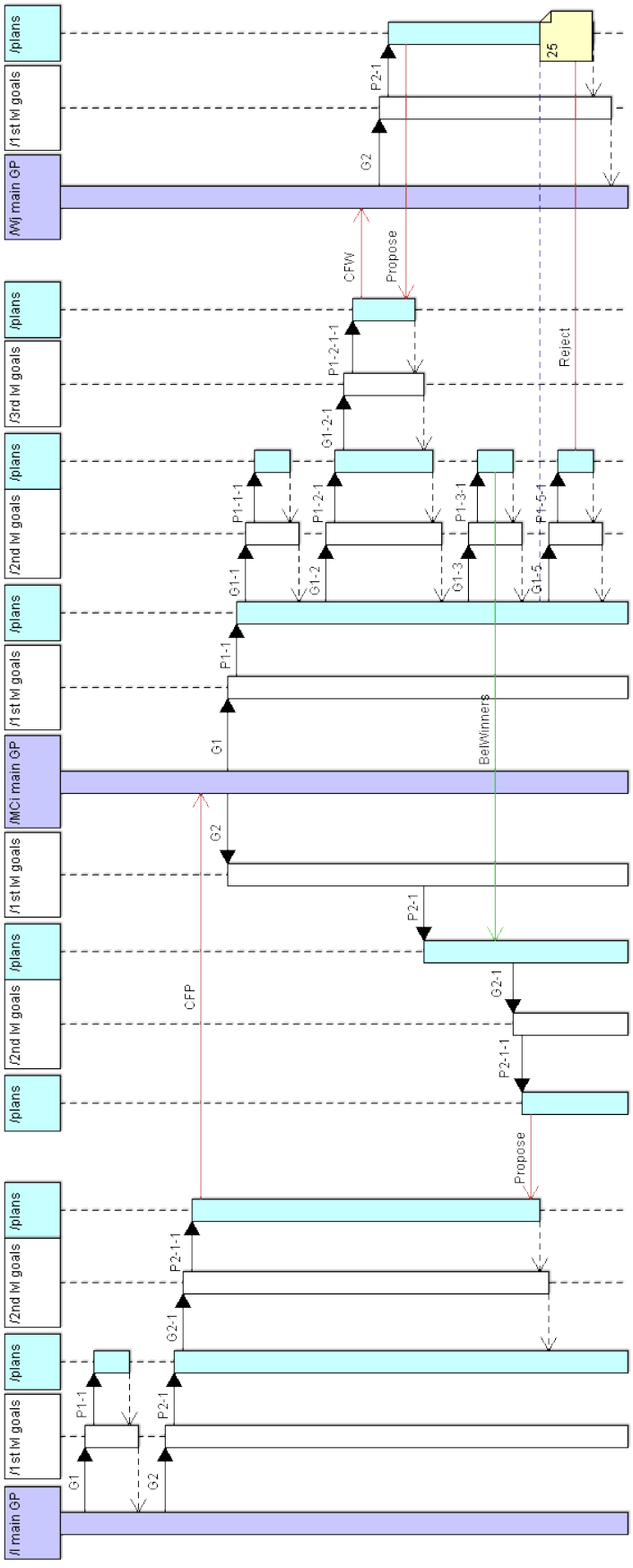


Figure 103: Sequence diagram for a successful CNP+ negotiation (the Initiator agent ends up with a result). The yellow tag corresponds to an error case from Table 20.

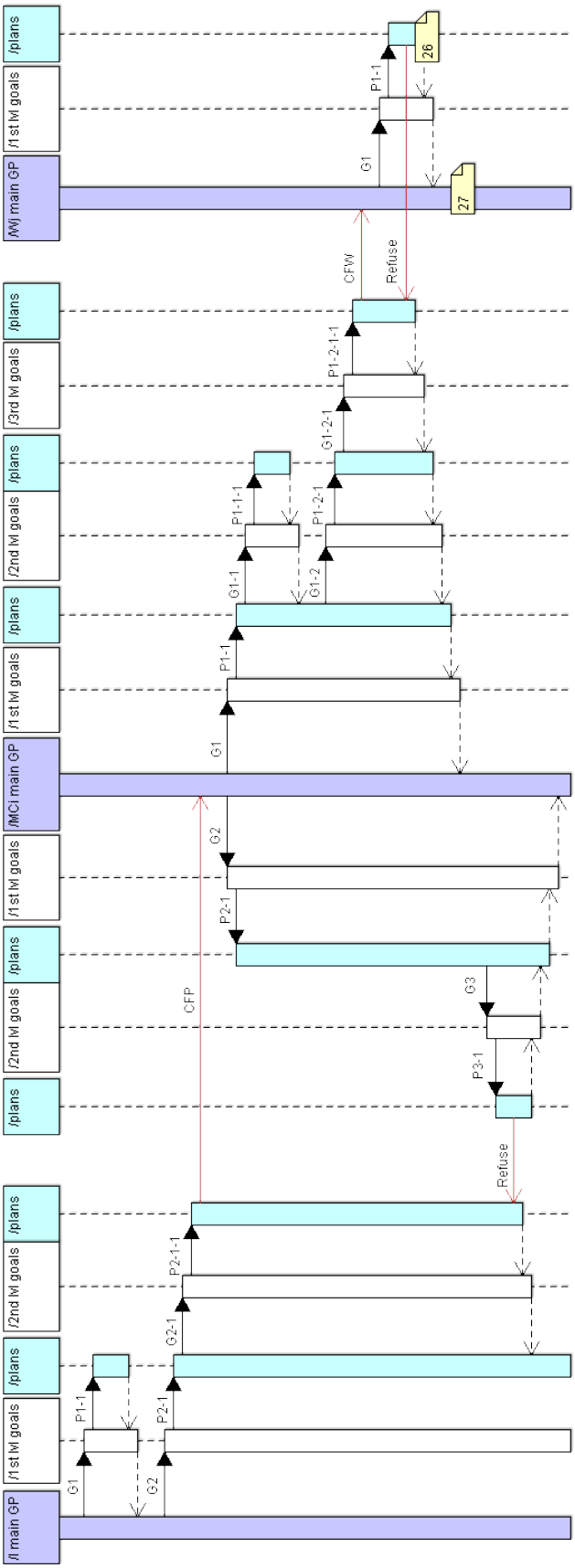


Figure 104: Sequence diagram for a successful CNP+ negotiation (the Initiator agent ends up with a result). Yellow tags correspond to error cases from Table 20.

Table 17: Error cases for the Initiator agent (DS stands for DirectoryService)

	Error occurrence	**Normal goal-driven reaction**	**Safety net (+ from normal)**
1	During P1-1	G1 may retry; if it fails, P0-1 will stop gracefully (foreseen)	if the error occurred after the message to the DS was sent, retract message
2	During P2-1-1 (after sending *CFP*)	G2-1 may retry but the corresponding MCi is not informed and it may contact Wj agents in order to prepare a *Propose*; G2 can still be achieved with the other instances of G2-1	*CFP* is retracted which causes unjustifications in MCi in all sub-goals adopted by P0-1 after receiving the message (eventually *CFW* should be called off if already sent)
3	During P2-1-1 (after writing Propose)	As above, and the written *Propose* remains enabled in the agent memory.	As above, but the written *Propose* is retracted too
4	During P0-1, just after G2	MCi agents wait for replies until their deadlines	no direct dependencies of the plan, so no retractions
5	During P3-1	G3 may retry and fail. All foreseen	-
6	During P4-1 (after *Accept*, but before P2-1-1 of MCi ends)	G4 may retry, but receiving multiple *Accept* messages is not included in the current model (however, in the absence of a strict message identification, the new *Accept* message would be ignored and the MCi agent will wait for *Result* which comes as a reply to the first *Accept*).	The *Accept* message is retracted, P2-1-1 of MCi is unjustified, then its goal, G2-1, will probably fail.
7	During P4-1 (after *Accept*, but AFTER P2-1-1 of MCi ends)	As above	The *Accept* message is retracted but with no consequences for MCi
8	During P4-1 (after receiving *Result*)	Any retries for G4 will end in timeouts in the current model as MCi already finished.	*Accept* is retracted, but MCi is already done.
9	During P0-1 after G4	-	Nothing to retract - all goals are over

Table 18: Error cases for the Main Contractor$_i$ agent (DS stands for DirectoryService)

	Error occurrence	Normal goal-driven reaction	Safety net (+ from normal)
10	During P1-2-1-1 (after sending *CFW*)	G1-2-1 may retry but the corresponding Wj is not informed to stop preparing a *Propose*; G1-2 can still be achieved with the other instances of G1-2-1	*CFW* is retracted for the Wj corresponding to the plan, triggering recovery in Wj
11	During P1-2-1-1 (after writing `Propose`)	G1-2-1 may retry but the corresponding Wj is not informed to stop waiting for a confirmation; G1-2 can still be achieved with the other instances of G1-2-1; the written *Propose* remains enabled in the agent memory.	As above, but the written *Propose* is retracted too
12	During P1-1, just after G1-2	Wj agents wait for replies until their deadlines; G1 may still be achieved leading to a positive outcome of the *CFP*, but the Wj agents may not be able to reply to a new *CFW* while waiting for a confirmation for the first	-
13	During P1-1, just after G1-3	Wj agents wait for replies until their deadlines, but P2-1 now negotiates using a list of winners that is no longer valid; if G1 fails, G2 will be unjustified, but if a new plan goes well for G1, the two plans will work with inconsistent beliefs	-
14	During P2-1-1, after sending *Propose*	G2-1 may succeed with another plan, but the protocol with I is broken. The I agent is not informed and may wrongly choose this agent as winner.	The *Propose* message is retracted but I will react only if P2-1-1 is still running.
15	During P2-1, after setting `belief(Neg,ok)`, before *Result*	G2 will probably fail and cause G1 to stop as well, stopping G1-6 in the process.	`belief(Neg,ok)` is retracted, causing P1-1 to unjustify and stop G1-6, but the consequences are the same as in the normal case.
16	During P1-6-1, after confirming and before the *Result*	G1-6 may retry but the protocol is broken and the outcome is not guaranteed.	The *Confirm* message is retracted causing the Wj agent to stop.
17	During P2-1, just after receiving *Result*	Retries for G2 would result in a broken protocol.	`belief(Neg,ok)` is retracted, but no reparation is triggered as P1-1 finished.

Table 19: Error cases for the Worker_j agent

	Error occurrence	Normal goal-driven reaction	Safety net (+ from normal)
18	During P0-1, when evaluating *CFW*	MCi is not informed and its P1-2-1-1 will timeout. Protocol is broken.	-
19	During P2-1, after having sent the *Propose*	G2 may retry or just fail, but the protocol is broken and MCi may even wrongly choose this Wj as winner	*Propose* is retracted, but if P1-2-1-1 of MCi was done, no reparation is triggered
20	During P0-1, while waiting for *Confirm*	-	-
21	The *rule* added by P3-1 crashes while G3 still executes	The goal fails (its outcome is based on the rule output)	The rule is retracted
22	During P0-1, just before G5	MCi will wait for reply until its timeout	-
23	During P5-1, when attempting to send	As above	-

Table 20: Error cases in the not-perfect case.

	Error occurrence	Normal goal-driven reaction	Safety net (+ from normal)
24	P2-1-1 of MCi after the I agent decided the MCi agent is not among the winners (before or after receiving the *Reject* message)	No negative impact for the negotiation, all well.	*Propose* message retracted, but no plan reacts.
25	P2-1 of Wj after the MCi agent decided the Wj agent is not among the winners (before or after receiving the *Reject* message)	No negative impact for the negotiation, all well.	*Propose* message retracted, but no plan reacts.
26	P1-1 of Wj after the *Refuse* message was sent	No negative impact for the negotiation, all well.	*Refuse* is retracted, but no plan unjustifies.
27	P0-1 of Wj after G1 was achieved (the *Refuse* message was sent)	No negative impact for the negotiation, all well.	-

BIBLIOGRAPHY

[1] Agent Oriented Software Pty. Ltd. *JACK Intelligent Agents® - Agent Manual*. Release 5.3. June 2005 (cit. on pp. 31, 44).

[2] Joe Armstrong. "A History of Erlang." In: *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*. HOPL III. New York, NY, USA: ACM, 2007, pp. 6–1–6–26 (cit. on pp. 4, 25, 26).

[3] Joe Armstrong and Robert Virding. "Erlang - An Experimental Telephony Programming Language." In: *Switching Symposium, 1990. XIII International*. Vol. 3. May 1990, pp. 43–48 (cit. on p. 25).

[4] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. "Basic Concepts and Taxonomy of Dependable and Secure Computing." In: *IEEE Transactions on Dependable and Secure Computing* 1.1 (2004), pp. 11–33 (cit. on pp. 12, 13, 15).

[5] J. M. Ayache, P. Azema, and M. Diaz. "Observer, a Concept for on Line Detection for Control Errors in Concurrent Systems." In: Madison, June 1979 (cit. on p. 17).

[6] Mihai Barbuceanu and Mark S Fox. "COOL: A Language for Describing Coordination in Multi Agent Systems." In: *ICMAS*. 1995, pp. 17–24 (cit. on p. 28).

[7] John Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003 (cit. on p. 25).

[8] Steve S. Benfield, Jim Hendrickson, and Daniel Galanti. "Making a Strong Business Case for Multiagent Technology." In: *Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems*. AAMAS '06. New York, NY, USA: ACM, 2006, pp. 10–15 (cit. on p. 5).

[9] Bernard Berthomieu and Michel Diaz. "Modeling and Verification of Time Dependent Systems Using Time Petri Nets." In: *IEEE Transactions on Software Engineering* 17.3 (1991), pp. 259–273 (cit. on p. 153).

[10] Staffan Blau, Jan Rooth, Jörgen Axell, Fiffi Hellstrand, Magnus Buhrgard, Tommy Westin, and Göran Wicklund. "AXD 301: A New Generation ATM Switching System." In: *Comput. Netw.* 31.6 (Mar. 1999), pp. 559–582 (cit. on p. 25).

[11] Rafael Bordini, Jomi Hübner, and Renata Vieira. "Jason and the Golden Fleece of Agent-Oriented Programming." In: *Multi-Agent Programming*. Ed. by Rafael Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah Seghrouchni. Vol. 15. Multiagent Systems, Artificial Societies, and Simulated Organizations. Springer US, 2005, pp. 3–37 (cit. on pp. 10, 31).

[12] Radja Boukharrou, Ahmed-Chawki Chaouche, Amal El Fallah Seghrouchni, Jean-Michel Ilié, and Djamel Eddine Saïdouni. "Dealing with Temporal Failure in Ambient Systems: a Dynamic Revision of Plans." In: *Journal of Ambient Intelligence and Humanized Computing* 6.3 (2015), pp. 325–336 (cit. on p. 32).

[13] Lars Braubach and Alexander Pokahr. "Goal-Oriented Interaction Protocols." In: *Proceedings of the 5th German Conference on Multiagent System Technologies*. MATES '07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 85–97 (cit. on pp. 31, 136, 176).

[14] Lars Braubach, Alexander Pokahr, and Winfried Lamersdorf. "Jadex: A Short Overview." In: *Net.ObjectDays 2004: AgentExpo*. 2004 (cit. on pp. 10, 13, 31).

[15] Lars Braubach, Alexander Pokahr, Daniel Moldt, and Winfried Lamersdorf. "Goal Representation for BDI Agent Systems." In: *Programming Multi-Agent Systems*. Ed. by Rafael Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah Seghrouchni. Vol. 3346. LNCS. Springer Berlin Heidelberg, 2005, pp. 44–65 (cit. on pp. 32, 33).

[16] Bruno Cabral and Paulo Marques. "Exception Handling: A Field Study in Java and .NET." In: *ECOOP 2007 – Object-Oriented Programming*. Ed. by Erik Ernst. Vol. 4609. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2007, pp. 151–175 (cit. on pp. 5, 23).

[17] Bruno Cabral and Paulo Marques. "A Case for Automatic Exception Handling." In: *Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on*. Sept. 2008, pp. 403–406 (cit. on p. 23).

[18] Radu Calinescu. "Towards a Generic Autonomic Architecture for Legacy Resource Management." In: *Innovations and Advanced Techniques in Systems, Computing Sciences and Software Engineering*. Ed. by Khaled Elleithy. Dordrecht: Springer Netherlands, 2008, pp. 410–415 (cit. on p. 177).

[19] Costin Caval, Amal El Fallah Seghrouchni, and Patrick Taillibert. "Keeping a Clear Separation between Goals and Plans." In: *Engineering Multi-Agent Systems*. Ed. by Fabiano Dalpiaz, Jürgen Dix, and M. Birna van Riemsdijk. Vol. 8758. Lecture Notes in Computer Science. Springer International Publishing, 2014, pp. 15–39 (cit. on p. 11).

[20] Varun Chandola, Arindam Banerjee, and Vipin Kumar. "Anomaly Detection: A Survey." In: *ACM Comput. Surv.* 41.3 (2009), 15:1–15:58 (cit. on pp. 18–20).

[21] Ahmed-Chawki Chaouche, Amal El Fallah Seghrouchni, Jean-Michel Ilié, and Djamel Eddine Saïdouni. "A Higher-order Agent Model for Ambient Systems." In: *Procedia Computer Science* 21.0 (2013). The 4th International Conference on Emerging Ubiquitous Systems and Pervasive Networks and the 3rd International Conference on Current and Future Trends of Information and Communication Technologies in Healthcare, pp. 156 –163 (cit. on pp. 32, 146).

[22] Christopher Cheong and Michael Winikoff. "Hermes: Designing Goal-oriented Agent Interactions." In: *Proceedings of the 6th International Conference on Agent-Oriented Software Engineering*. AOSE'05. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 16–27 (cit. on p. 31).

[23] Caroline Chopinaud. "Contrôle de l'émergence de comportements dans les systèmes d'agents cognitifs autonomes." French. Title in English: "Controlling the Emergence of Behaviours in Autonomous Cognitive Agent Systems". PhD thesis. Paris: Université Pierre et Marie Curie, 2007 (cit. on p. 14).

[24] Caroline Chopinaud, Amal El Fallah Seghrouchni, and Patrick Taillibert. "Prevention of Harmful Behaviors Within Cognitive and Autonomous Agents." In: *Proceedings of ECAI 2006:* Amsterdam, The Netherlands, The Netherlands: IOS Press, 2006, pp. 205–209 (cit. on pp. 14, 30, 176).

[25] Bradley J. Clement, Edmund H. Durfee, and Anthony C. Barrett. "Abstract Reasoning for Planning and Coordination." In: *Journal of Artificial Intelligence Research* 28.1 (2007), pp. 453–515 (cit. on p. 150).

[26] Fabiano Dalpiaz, Paolo Giorgini, and John Mylopoulos. "Adaptive Socio-Technical Systems: a Requirements-Based Approach." In: *Requirements Engineering* 18.1 (2013), pp. 1–24 (cit. on pp. 31, 77).

[27] Mehdi Dastani, M. Birna van Riemsdijk, Frank Dignum, and John-Jules Meyer. "A Programming Language for Cognitive Agents: Goal Directed 3APL." In: *Programming multiagent systems, first international workshop (ProMAS'03)*. Vol. 3067. LNAI. Berlin: Springer, 2004, pp. 111–130 (cit. on p. 11).

[28] Sylvain Dekoker. "Detection of Unjustified Plans for Cognitive Agents." In: *COGnitive systems with Interactive Sensors (COGIS'09)*. Paris, France, 2009 (cit. on pp. 14, 47).

[29] Sylvain Dekoker. "Alma : un langage de programmation d'agents cognitifs." French. Title in English: "ALMA: A Programming Language for Cognitive Agents". PhD thesis. Paris: Université Pierre et Marie Curie, May 2012 (cit. on pp. 6, 13, 14, 37, 38, 94).

[30] Benjamin Devèze, Caroline Chopinaud, and Patrick Taillibert. "ALBA: A Generic Library for Programming Mobile Agents with Prolog." In: *Programming Multi-Agent Systems*. Ed. by RafaelH. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah Seghrouchni. Vol. 4411. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2007, pp. 129–148 (cit. on p. 48).

[31] Michel Diaz, Guy Juanole, and Jean-Pierre Courtiat. "Observer-A Concept for Formal On-Line Validation of Distributed Systems." In: *IEEE Trans. Softw. Eng.* 20.12 (1994), pp. 900–913 (cit. on p. 17).

[32] Edsger W. Dijkstra. "Structured Programming." In: ed. by O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. London, UK: Academic Press Ltd., 1972, pp. 1–82 (cit. on p. 4).

[33] Nicola Dragoni and Mauro Gaspari. "Crash Failure Detection in Asynchronous Agent Communication Languages." In: *Autonomous Agents and Multi-Agent Systems* 13.3 (Nov. 2006), pp. 355–390 (cit. on p. 28).

[34] Pierre-Yves Dumas, El Fallah Seghrouchni, Amal, and Patrick Taillibert. "Aerial: A Framework to Support Human Decision Making in a Constrained Environment." In: *Tools with Artificial Intelligence (ICTAI), 2012 IEEE 24th International Conference on*. Vol. 1. Nov. 2012, pp. 626–633 (cit. on p. 79).

[35] Amal El Fallah Seghrouchni and Serge Haddad. "A Recursive Model for Distributed Planning." In: *Proceedings of the 2nd International Conference on Multi-Agent Systems*. Kyoto, Japan: AAAI Press, 1996, pp. 307–314 (cit. on p. 153).

[36] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. "A Survey of Rollback-Recovery Protocols in Message-Passing Systems." In: *ACM Comput. Surv.* 34.3 (Sept. 2002), pp. 375–408 (cit. on p. 16).

[37] European Space Agency. *Rosetta - Living with a Comet*. Brochure. 2015 (cit. on p. 4).

[38] Alan Fedoruk and Ralph Deters. "Improving Fault-Tolerance by Replicating Agents." In: *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems: Part 2*. AAMAS '02. New York, NY, USA: ACM, 2002, pp. 737–744 (cit. on p. 28).

[39] Jacques Ferber. *Les systèmes multi-agents : vers une intelligence collective*. Title in English: "Multi-Agent Systems: Towards a Collective Intelligence". InterEditions Paris, 1995 (cit. on p. 26).

[40] Kenneth D. Forbus and Johan de Kleer. *Building Problem Solvers*. MIT Press, 1993 (cit. on pp. 38–40).

[41] Norbert E. Fuchs. "Specifications Are (Preferably) Executable." In: *Softw. Eng. J.* 7.5 (Sept. 1992), pp. 323–334 (cit. on p. 25).

[42] Ryohei Fujimaki, Takehisa Yairi, and Kazuo Machida. "An Approach to Spacecraft Anomaly Detection Problem Using Kernel Feature Space." In: *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*. KDD '05. New York, NY, USA: ACM, 2005, pp. 401–410 (cit. on p. 19).

[43] Al Geist. "Supercomputing's Monster in the Closet." In: *IEEE Spectrum* (Mar. 2016). Title of the online version: "How To Kill A Supercomputer: Dirty Power, Cosmic Rays, and Bad Solder" (cit. on p. 115).

[44] Fausto Giunchiglia, John Mylopoulos, and Anna Perini. "The Tropos Software Development Methodology: Processes, Models and Diagrams." In: *Agent-Oriented Software Engineering III*. Ed. by Fausto Giunchiglia, James Odell, and Gerhard Weiß. Vol. 2585. LNCS. Springer Berlin Heidelberg, 2003, pp. 162–173 (cit. on pp. 11, 31, 64, 142, 149).

[45] John B. Goodenough. "Exception Handling: Issues and a Proposed Notation." In: *Commun. ACM* 18.12 (Dec. 1975), pp. 683–696 (cit. on p. 12).

[46] Zahia Guessoum, Nora Faci, and Jean-Pierre Briot. "Adaptive Replication of Large-Scale Multi-Agent Systems: towards a Fault-Tolerant Multi-Agent Platform." In: *SIGSOFT Softw. Eng. Notes* 30.4 (2005), pp. 1–6 (cit. on pp. 28, 78).

[47] Felix C. Gärtner. "Fundamentals of Fault-tolerant Distributed Computing in Asynchronous Environments." In: *ACM Comput. Surv.* 31.1 (Mar. 1999), pp. 1–26 (cit. on p. 16).

[48] Staffan Haegg. "A Sentinel Approach to Fault handling in Multi-Agent Systems." In: *Multi-Agent Systems Methodologies and Applications*. Ed. by Chengqi Zhang and Dickson Lukose. Vol. 1286. LNCS. Springer Berlin / Heidelberg, 1997, pp. 181–195 (cit. on pp. 27, 29).

[49] James Harland, David N. Morley, John Thangarajah, and Neil Yorke-Smith. "An Operational Semantics for the Goal Life-Cycle in BDI Agents." In: *Autonomous Agents and Multi-Agent Systems* 28.4 (2014), pp. 682–719 (cit. on pp. 10, 33–36).

[50] Nick Hawes. "A Survey of Motivation Frameworks for Intelligent Systems." In: *Artificial Intelligence* 175.5–6 (2011). Special Review Issue, pp. 1020 –1036 (cit. on p. 32).

[51] Fergus Henderson, Zoltan Somogyi, and Thomas Conway. "Determinism Analysis in the Mercury Compiler." In: *Australian Computer Science Communications* 18 (1996), pp. 337–346 (cit. on p. 26).

[52] Pascal Van Hentenryck. "A Gentle Introduction to NUMERICA." In: *Artificial Intelligence* 103.1-2 (1998), pp. 209–235 (cit. on p. 59).

[53] Koen V. Hindriks, Frank S. de Boer, Wiebe van der Hoek, and John-Jules Meyer. "Agent Programming with Declarative Goals." In: *Intelligent Agents VII Agent Theories Architectures and Languages*. Ed. by Cristiano Castelfranchi and Yves Lespérance. Vol. 1986. LNCS. Springer Berlin Heidelberg, 2001, pp. 228–243 (cit. on p. 11).

[54] James J. Horning, Hugh C. Lauer, Peter M. Melliar-Smith, and Brian Randell. "A Program Structure for Error Detection and Recovery." In: *Operating Systems*. Vol. 16. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1974, pp. 171–187 (cit. on p. 22).

[55] Kennie H. Jones. "Engineering Antifragile Systems: A Change In Design Philosophy." In: *Procedia Computer Science* 32 (2014). The 5th International Conference on Ambient Systems, Networks and Technologies (ANT-2014), the 4th International Conference on Sustainable Energy Information Technology (SEIT-2014), pp. 870 –875 (cit. on p. 17).

[56] Zbigniew T. Kalbarczyk, Ravishankar K. Iyer, Saurabh Bagchi, and Keith Whisnant. "Chameleon: A Software Infrastructure for Adaptive Fault Tolerance." In: *IEEE Trans. Parallel Distrib. Syst.* 10.6 (1999), pp. 560–579 (cit. on p. 21).

[57] Gal A. Kaminka and Milind Tambe. "What is Wrong with Us? Improving Robustness Through Social Diagnosis." In: *Proceedings of the Fifteenth National/Tenth Conference on Artificial Intelligence/Innovative Applications of Artificial Intelligence*. AAAI '98/IAAI '98. Menlo Park, CA, USA: American Association for Artificial Intelligence, 1998, pp. 97–104 (cit. on p. 28).

[58] J.O. Kephart and D.M. Chess. "The Vision of Autonomic Computing." In: *Computer* 36.1 (Jan. 2003), pp. 41–50 (cit. on p. 16).

[59] Johan de Kleer. "An Assumption-Based TMS." In: *Artif. Intell.* 28.2 (1986), pp. 127–162 (cit. on p. 40).

[60] Johan de Kleer. "Extending the ATMS." In: *Artificial Intelligence* 28.2 (1986), pp. 163 –196 (cit. on p. 176).

[61] Johan de Kleer. "Problem Solving with the ATMS." In: *Artificial Intelligence* 28.2 (1986), pp. 197 –224 (cit. on p. 40).

[62] Johan de Kleer and Brian. C. Williams. "Diagnosing Multiple Faults." In: *Artificial intelligence* 32.1 (1987). (corrected version of 2008), pp. 97–130 (cit. on pp. 6, 41, 43, 68).

[63] Gerwin Klein et al. "seL4: Formal Verification of an OS Kernel." In: *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*. SOSP '09. New York, NY, USA: ACM, 2009, pp. 207–220 (cit. on p. 4).

[64] Mark Klein and Chrysanthos Dellarocas. "Exception Handling in Agent Systems." In: *Proceedings of the Third Annual Conference on Autonomous Agents*. AGENTS '99. New York, NY, USA: ACM, 1999, pp. 62–68 (cit. on p. 29).

[65] Mark Klein, Juan-Antonio Rodriguez-Aguilar, and Chrysanthos Dellarocas. "Using Domain-Independent Exception Handling Services to Enable Robust Open Multi-Agent Systems: The Case of Agent Death." In: *Autonomous Agents and Multi-Agent Systems* 7.1 (2003), pp. 179–189 (cit. on p. 29).

[66] Matthew Klenk, Matt Molineaux, and David W. Aha. "Goal-Driven Autonomy for Responding to Unexpected Events in Strategy Simulations." In: *Computational Intelligence* 29.2 (2013), pp. 187–206 (cit. on p. 35).

[67] Mark Krasniewski and Bryan Rabeler. "TIBFIT: Trust Index Based Fault Tolerance for Arbitrary Data Faults in Sensor Networks." In: *Proceedings of the 2005 International Conference on Dependable Systems and Networks*. DSN '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 672–681 (cit. on p. 20).

[68] Christopher Kruegel and Giovanni Vigna. "Anomaly Detection of Web-Based Attacks." In: *Proceedings of the 10th ACM conference on Computer and communications security*. CCS '03. New York, NY, USA: ACM, 2003, pp. 251–261 (cit. on p. 20).

[69] Kazuhiro Kuwabara. "Meta-level control of coordination protocols." In: *Proceedings of the third international conference on multi-agent systems*. 1996, pp. 104–111 (cit. on p. 28).

[70] Guy Lamarche and Patrick Taillibert. "Utilisation des réseaux de Petri pour le test des programmes temps réel." French. In: *Technique et Science Informatiques* 4.1 (1985). Title in English: "Using Petri Nets for Testing Real-Time Software", pp. 83 –87 (cit. on p. 17).

[71] Axel van Lamsweerde. "Goal-Oriented Requirements Enginering: a Roundtrip from Research to Practice [Enginering Read Engineering]." In: *Requirements Engineering Conference, 2004. Proceedings. 12th IEEE International*. Sept. 2004, pp. 4–7 (cit. on pp. 5, 32).

[72] Jean-claude Laprie. "From Dependability to Resilience." In: *In 38th IEEE/IFIP Int. Conf. On Dependable Systems and Networks*. 2008 (cit. on p. 17).

[73] Jaques-Louis Lions. *Ariane 5 Flight 501 Failure*. Report by the Inquiry Board. Paris, France: CNES/ESA, 1996 (cit. on pp. 3, 4, 12, 78, 115).

[74] Bertrand Meyer. "Eiffel*: A language and environment for software engineering." In: *Journal of Systems and Software* 8.3 (1988), pp. 199–246 (cit. on p. 24).

[75] Bertrand Meyer. "Applying Design by Contract." In: *IEEE Computer* 25 (1992), pp. 40–51 (cit. on p. 24).

[76] Mirko Morandini, Loris Penserini, and Anna Perini. "Operational Semantics of Goal Models in Adaptive Agents." In: *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems*. Vol. 1. Budapest, Hungary: International Foundation for Autonomous Agents and Multiagent Systems, 2009, pp. 129–136 (cit. on pp. 37, 141, 149).

[77] David N. Morley, Karen L. Myers, and Neil Yorke-Smith. "Continuous Refinement of Agent Resource Estimates." In: *Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems*. Hakodate, Japan: ACM, 2006, pp. 858–865 (cit. on p. 150).

[78] Yannick Moy, Emmanuel Ledinot, Herve Delseny, Virginie Wiels, and Benjamin Monate. "Testing or Formal Verification: DO-178C Alternatives and Industrial Experience." In: *IEEE Softw.* 30.3 (May 2013), pp. 50–57 (cit. on p. 4).

[79] Ferdinand Piette, Cédric Dinont, Amal El Fallah Seghrouchni, and Patrick Taillibert. "Deployment and Configuration of Applications for Ambient Systems." In: *Procedia Computer Science* 52 (2015). The 6th International Conference on Ambient Systems, Networks and Technologies (ANT-2015), the 5th International Conference on Sustainable Energy Information Technology (SEIT-2015), pp. 373 –380 (cit. on pp. 159–161).

[80] Ferdinand Piette, Costin Caval, Cédric Dinont, Amal El Fallah Seghrouchni, and Patrick Taillibert. "A Multi-Agent Solution for the Deployment of Distributed Applications in Ambient Systems." In: *Engineering Multi-Agent Systems Workshop 2016 (EMAS 2016)*. Singapore, May 2016 (cit. on pp. 11, 153).

[81] Ferdinand Piette, Costin Caval, Amal El Fallah Seghrouchni, Cédric Dinont, and Patrick Taillibert. "A Multi-Agent System for Resource Privacy: Deployment of Ambient Applications in Smart Environments." In: *Proceedings of the 2016 International Conference on Autonomous Agents and Multiagent Systems*. to appear. Singapore, May 2016 (cit. on pp. 11, 153).

[82] Eric Platon, Nicolas Sabouret, and Shinichi Honiden. "Challenges for Exception Handling in Multi-Agent Systems." In: *Software Engineering for Multi-Agent Systems V*. Vol. 4408. Lecture Notes in Computer Science. 10.1007/978-3-540-73131-3_3. Springer Berlin / Heidelberg, 2007, pp. 41–56 (cit. on pp. 27, 28, 83, 176).

[83] Eric Platon, Nicolas Sabouret, and Shinichi Honiden. "An Architecture for Exception Management in Multiagent Systems." In: *Int. J. Agent-Oriented Softw. Eng.* 2.3 (2008), pp. 267–289 (cit. on p. 27).

[84] Alexander Pokahr, Lars Braubach, and Winfried Lamersdorf. "A Goal Deliberation Strategy for BDI Agent Systems." In: *Multiagent System Technologies*. Ed. by Torsten Eymann, Franziska Klügl, Winfried Lamersdorf, Matthias Klusch, and Michael Huhns. Vol. 3550. LNCS. Springer Berlin / Heidelberg, 2005, pp. 82–93 (cit. on p. 35).

[85] Katia Potiron. "Systèmes Multi-Agents et tolérance aux fautes : conséquences de l'autonomie des agents." French. Title in English: "Multi-Agent Systems and Fault Tolerance: Consequences of the Agent Autonomy". PhD thesis. Paris: Université Pierre et Marie Curie, 2010 (cit. on p. 14).

[86] Katia Potiron, Amal El Fallah Seghrouchni, and Patrick Taillibert. *From Fault Classification to Fault Tolerance for Multi-Agent Systems*. Briefs in Computer Science. Springer, 2013 (cit. on pp. 5, 13, 14, 30, 60, 63, 116).

[87] Harald Psaier and Schahram Dustdar. "A Survey on Self-Healing Systems: Approaches and Systems." In: *Computing* 91.1 (2011). 10.1007/s00607-010-0107-y, pp. 43–73 (cit. on p. 16).

[88] Brian Randell. "System Structure for Software Fault Tolerance." In: *SIGPLAN Not.* 10.6 (Apr. 1975), pp. 437–449 (cit. on pp. 22, 72).

[89] Anand S. Rao and Michael P. Georgeff. "Modeling Rational Agents Within a BDI-Architecture." In: *Principles of Knowledge Representation and Reasoning. Proceedings of the second International Conference*. San Mateo: Morgan Kaufmann, 1991, pp. 473–484 (cit. on p. 34).

[90] Anand S. Rao and Michael P. Georgeff. "BDI-Agents: from Theory to Practice." In: *Proceedings of the First International Conference on Multiagent Systems*. San Francisco, USA: AAAI Press, 1995, pp. 312–319 (cit. on pp. 31, 33).

[91] Robert D. Rasmussen. "Goal-Based Fault Tolerance for Space Systems Using the Mission Data System." In: *Aerospace Conference, 2001, IEEE Proceedings.* Vol. 5. 2001, pp. 2401–2410 (cit. on pp. 5, 21).

[92] Donald J. Reifer. "Software Failure Modes and Effects Analysis." In: *IEEE Transactions on Reliability* R-28.3 (Aug. 1979), pp. 247–249 (cit. on p. 15).

[93] Alessandro Ricci. "Agents and Coordination Artifacts for Feature Engineering." In: *Objects, Agents, and Features, International Seminar, Dagstuhl Castle, Germany, February 2003, Revised and Invited Papers*. Ed. by Mark Dermot Ryan, John-Jules Ch Meyer, and Hans-Dieter Ehrich. Vol. 2975. Lecture Notes in Computer Science. Springer, 2003, pp. 209–226 (cit. on pp. 13, 154, 161).

[94] M. Birna van Riemsdijk, Mehdi Dastani, and Michael Winikoff. "Goals in Agent Systems: A Unifying Framework." In: *Proceedings of the seventh international joint conference on autonomous agents and multiagent systems (AAMAS'08)*. Estoril: IFAAMAS, 2008, pp. 713–720 (cit. on p. 32).

[95] M. Birna van Riemsdijk and Neil Yorke-Smith. "Towards Reasoning with Partial Goal Satisfaction in Intelligent Agents." In: *Programming Multiagent Systems, 8th International Workshop (ProMAS'10)*. Vol. 6599. LNAI. Springer, 2012, pp. 41–59 (cit. on pp. 32, 158).

[96] M. Birna van Riemsdijk, Mehdi Dastani, Frank Dignum, and John-JulesCh. Meyer. "Dynamics of Declarative Goals in Agent Programming." In: *Declarative Agent Languages and Technologies II*. Ed. by João Leite, Andrea Omicini, Paolo Torroni, and pInar Yolum. Vol. 3476. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2005, pp. 1–18 (cit. on p. 34).

[97] Jordi Sabater and Carles Sierra. "Review on Computational Trust and Reputation Models." In: *Artificial Intelligence Review* 24.1 (2005), pp. 33–60 (cit. on p. 30).

[98] Hesam Samimi, Ei Darli Aung, and Todd Millstein. "Falling Back on Executable Specifications." In: *Proceedings of the 24th European Conference on Object-oriented Programming*. ECOOP'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 552–576 (cit. on p. 25).

[99] Sebastian Sardina and Lin Padgham. "A BDI Agent Programming Language with Failure Handling, Declarative Goals, and Planning." In: *Autonomous Agents and Multi-Agent Systems* 23.1 (2011), pp. 18–70 (cit. on pp. 10, 11, 32, 34, 35, 37, 146, 151).

[100] Nazaraf Shah, Kuo-Ming Chao, Nick Godwin, and Anne James. "Exception Diagnosis in Open Multi-Agent Systems." In: *Proceedings of the IEEE/WIC/ACM International Conference on Intelligent Agent Technology*. IAT '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 483–486 (cit. on p. 29).

[101] Nazaraf Shah, Kuo-Ming Chao, Nick Godwin, Anne James, and C-F Tasi. "An Empirical Evaluation of a Sentinel Based Approach to Exception Diagnosis in Multi-Agent Systems." In: *Proceedings of the 20th International Conference on Advanced Information Networking and Applications - Volume 01*. AINA '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 379–386 (cit. on p. 29).

[102] Steven Shapiro, Sebastian Sardina, John Thangarajah, Lawrence Cavedon, and Lin Padgham. "Revising Conflicting Intention Sets in BDI Agents." In: *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems*. Vol. 2. Valencia, Spain: International Foundation for Autonomous Agents and Multiagent Systems, 2012, pp. 1081–1088 (cit. on p. 170).

[103] Patricia Shaw and Rafael Bordini. "Towards Alternative Approaches to Reasoning About Goals." In: *Declarative Agent Languages and Technologies V*. Ed. by Matteo Baldoni, TranCao Son, M. Birna van Riemsdijk, and Michael Winikoff. Vol. 4897. LNCS. Springer Berlin Heidelberg, 2008, pp. 104–121 (cit. on pp. 36, 170).

[104] Patricia Shaw and Rafael Bordini. "An Alternative Approach for Reasoning about the Goal-Plan Tree Problem." In: *Languages, Methodologies, and Development Tools for Multi-Agent Systems*. Ed. by Mehdi Dastani, Amal El Fallah Seghrouchni, Jomi Hübner, and João Leite. Vol. 6822. LNCS. Springer Berlin Heidelberg, 2011, pp. 115–135 (cit. on p. 36).

[105] Dhirendra Singh, Sebastian Sardina, Lin Padgham, and Geoff James. "Integrating Learning into a BDI Agent for Environments with Changing Dynamics." In: *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence*. Vol. 3. Barcelona, Spain: AAAI Press, 2011, pp. 2525–2530 (cit. on p. 36).

[106] Reid G. Smith. "The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver." In: *IEEE Transactions on Computers* C-29.12 (1980), pp. 1104–1113 (cit. on p. 117).

[107] Dean H. Stamatis. *Failure Mode and Effect Analysis: FMEA from Theory to Execution*. Asq Press, 2003 (cit. on pp. 4, 15).

[108] John Thangarajah. "Managing the Concurrent Execution of Goals in Intelligent Agents." PhD thesis. Melbourne, Australia: RMIT University, 2005 (cit. on pp. 34, 36, 142, 143).

[109] John Thangarajah and Lin Padgham. "Computationally Effective Reasoning About Goal Interactions." In: *Journal of Automated Reasoning* 47.1 (2011), pp. 17–56 (cit. on pp. 10, 34, 36, 170).

[110] John Thangarajah, Sebastian Sardina, and Lin Padgham. "Measuring Plan Coverage and Overlap for Agent Reasoning." In: *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems*. Vol. 2. Valencia, Spain: International Foundation for Autonomous Agents and Multiagent Systems, 2012, pp. 1049–1056 (cit. on p. 145).

[111] John Thangarajah, James Harland, David Morley, and Neil Yorke-Smith. "Operational Behaviour for Executing, Suspending, and Aborting Goals in BDI Agent Systems." In: *Proceedings of the 8th international conference on Declarative agent languages and technologies VIII*. DALT'10. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 1–21 (cit. on pp. 33, 34, 102).

[112] Wilfredo Torres-Pomales. *Software Fault Tolerance: A tutorial*. Tech. rep. NASA Langley Research Center, USA, 2000 (cit. on pp. 15, 16).

[113] David A. Watt. *Programming Language Design Concepts*. pp. 216-218. John Wiley & Sons, 2004 (cit. on p. 5).

[114] Michael Winikoff and Lin Padgham. *Developing Intelligent Agent Systems: A Practical Guide*. Wiley Series in Agent Technology. John Wiley and Sons, 2004 (cit. on pp. 11, 31, 150, 179).

[115] Michael Winikoff, Lin Padgham, James Harland, and John Thangarajah. "Declarative and Procedural Goals in Intelligent Agent Systems." In: *Proceedings of the 8th International Conference on Principles of Knowledge Representation and Reasoning*. Toulouse, France: Morgan Kaufman, 2002, pp. 470–481 (cit. on pp. 11, 32, 34).

[116] Michael J. Wooldridge. *Introduction to Multiagent Systems*. New York, NY, USA: John Wiley & Sons, Inc., 2001 (cit. on p. 27).

[117] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. "Finding and Understanding Bugs in C Compilers." In: *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '11. New York, NY, USA: ACM, 2011, pp. 283–294 (cit. on p. 4).

[118] Y.C. Yeh. "Triple-Triple Redundant 777 Primary Flight Computer." In: *Aerospace Applications Conference, 1996. Proceedings., 1996 IEEE*. Vol. 1. Feb. 1996, pp. 293–307 (cit. on p. 4).