Execution trace management to support dynamic V&V for executable DSMLs

Erwan Bousse

ANNÉE 2015

UNIVERSITÉ DE RENNES 1

THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Européenne de Bretagne

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Informatique

École doctorale Matisse

présentée par

Erwan Bousse

préparée à l'unité de recherche IRISA (UMR 6074)
Institut de Recherche en Informatique et Systèmes Aléatoires
Istic — UFR en Informatique et Électronique

Execution Trace Management to Support Dynamic V&V for Executable DSMLs

Thèse soutenue à Rennes le 3/12/2015

devant le jury composé de :

Gerti Kappel
Professor, Technische Universität Wien / *rapporteur*

Franck Barbier
Professeur, Université de Pau et des Pays de l'Adour / *rapporteur*

Julien DeAntoni
Maître de conférences, Université Nice Sophia Antipolis / *examinateur*

François Taïani
Professeur, Université de Rennes 1 / *examinateur*

Benoit Baudry
Chargé de recherche, Inria Rennes / *directeur de thèse*

Benoit Combemale
Chargé de recherche, Inria Rennes – Université de Rennes 1 / *examinateur*

Acknowledgments

First of all I would like to thank all the members of my jury for accepting to review my work and for attending my defense. In particular, I thank Professor Gerti Kappel and Prof. Franck Barbier for thoroughly reviewing this report. You all gave me a lot of interesting and inspiring feedback that I will keep in mind in the future.

These three years of doctorate were a very intense and enriching experience that would not have been possible without the incentive, input, and support of a wide range of people. I apologize in advance for forgetting anyone!

To begin with, I would like to thank the people abroad I had the chance to collaborate with. In particular, I thank Tanja Mayerhofer from TU Wien, and both Jonathan Corley and Professor Jeff Gray from the University of Alabama. Working with you was a real pleasure, and our results really pushed me forward during my thesis.

Next, I would like to thank my advisors, Benoit Combemale and Benoit Baudry, for their great tutoring and extreme patience. You convinced me to follow you in the former Triskell team (now DiverSE), and you teached me a tremendous amount of things about research and science. Most importantly, I enjoyed a great deal working with you!

Continuing, I would like to extend my thanks to all the DiverSE research team, which includes both former and actual members. Some of you really played a very important role and contributed to put me back on my feet during difficult times. I really enjoyed these years in this friendly corridor of ours, and I wish the best for all of you.

Merci à tous mes amis de Rennes, de l'INSA, ou d'ailleurs de m'avoir autant encouragé et aidé ces dernières années. Vous êtes trop nombreux pour être listés, mais vous vous reconnaîtrez sans doute. J'espère ne pas vous avoir trop agacé avec mes histoires de doctorant (quoique beaucoup géraient leurs propres thèses). Merci d'avoir été là !

Je remercie bien évidemment mes parents Soizik et Marc qui ont énormément fait, mais aussi mon frère Alexandre et ainsi que ma sœur Morgane qui étaient déjà passés par leurs thèses respectives et qui ont su continuellement me soutenir. Mes grands parents ont également toujours été là et m'ont énormement encouragé, et je les en remercie énormément en retour. Et encore au delà, car je les ai bien trop embêté avec cela, je remercie aussi tous mes cousins pour tous leurs encouragements. Merci à vous tous !

Pour finir, j'aimerais remercier infiniment mon épouse Clémentine. Après ces années de soutien mutuel pour arriver au bout de nos thèses — avec un mariage au beau milieu ! — nous allons pouvoir poursuivre nos aventures ailleurs. Pour sûr, je n'aurais vraiment jamais tenu bon sans toi. Tu es géniale, merci tellement ! ☺

Contents

Introduction en français **xi**
Contexte . . . xi
Énoncé du problème . . . xii
Contributions . . . xiii
Applications . . . xiv
Contexte de cette thèse . . . xvi

1 Introduction **1**
1.1 Context . . . 1
1.2 Problem Statement . . . 2
1.3 Contributions . . . 3
1.4 Applications . . . 4
1.5 Context of this Thesis . . . 5
1.6 Outline . . . 6

I State of the Art **9**

2 State of the Art **11**
2.1 Model-Driven Engineering . . . 11
2.1.1 Metamodel . . . 12
2.1.2 Model . . . 13
2.1.3 Model Transformations . . . 14
2.2 Object and Model Duplication . . . 18
2.2.1 Object Duplication . . . 18
2.2.2 Model Duplication . . . 20
2.3 Executable Metamodeling . . . 22
2.3.1 Operational vs. Translational Semantics . . . 22
2.3.2 Definition . . . 24

2.3.3 Execution State Definition . . . 25
2.3.4 Initialization Transformation . . . 27
2.3.5 Execution Transformation . . . 28
2.3.6 Interacting with the Environment . . . 29
2.4 Model Execution Tracing . . . 31
2.4.1 Execution Traces . . . 31
2.4.2 Execution Trace Management . . . 33
2.4.3 A Look at Execution Trace Data Structures . . . 35
2.4.4 Navigation Paths . . . 40
2.5 Interactive Debugging of Executable Models . . . 40
2.5.1 Enabling Interactive Debugging . . . 41
2.5.2 Model Interactive Debugging . . . 43
2.6 Model Omniscient Debugging . . . 45
2.6.1 Omniscient Debugging Definition . . . 46
2.6.2 Omniscient Debugging Methods . . . 47
2.6.3 Omniscient Debugging for xDSMLs . . . 49

II Contributions 51

3 Foreword to the Contributions 53
3.1 Observations . . . 53
3.1.1 Generic vs. Domain-Specific Trace Manipulations . . . 53
3.1.2 Limitations of Existing Execution Trace Data Structures . . . 54
3.2 Overview of the Contributions . . . 55

4 Scalable Armies of Model Clones through Data Sharing 57
4.1 Introduction . . . 57
4.2 Cloning Requirements and Proposal . . . 59
4.2.1 Requirements for Cloning . . . 59
4.2.2 Existing Cloning Approaches and Intuition . . . 59
4.3 On Model Cloning . . . 61
4.3.1 Mutable Subset of a Metamodel . . . 61
4.3.2 Implementation of Metamodels and Models . . . 61
4.3.3 Cloning . . . 63
4.4 Memory Efficient Cloning Operators . . . 64
4.4.1 Data Sharing Strategies . . . 64
4.4.2 Generic Cloning Algorithm . . . 65
4.4.3 Family of Cloning Operators . . . 68
4.4.4 EMF-Based Implementation . . . 69
4.5 Evaluation and Results . . . 69
4.5.1 Dataset . . . 69
4.5.2 Measures . . . 70
4.5.3 Metrics . . . 70

4.5.4 Results . . . 70
4.5.5 Threats to Validity . . . 73
4.6 Conclusion . . . 73

5 Multidimensional Domain-Specific Trace Metamodels 75
5.1 Introduction . . . 75
5.2 Motivation and Proposal . . . 76
5.3 From Executable Metamodeling to Execution Traces . . . 77
5.4 Generating Multidimensional Domain-Specific Trace Metamodels . . . 79
5.4.1 Observations and Technical Challenges . . . 79
5.4.2 Trace Metamodel Generation . . . 79
5.4.3 Resulting Benefits . . . 86
5.4.4 EMF Implementation . . . 87
5.5 Related Work . . . 87
5.5.1 Domain-Specific Trace Data Structures . . . 87
5.5.2 Multidimensional Trace Data Structures . . . 88
5.5.3 Self-defining Trace Formats . . . 88
5.6 Conclusion . . . 89

III Applications and Tooling 91

6 Foreword to the Applications to Dynamic V&V 93
6.1 Objectives . . . 93
6.2 Overview . . . 94
6.3 Case Study: fUML . . . 94
6.3.1 Abstract syntax . . . 95
6.3.2 Operational semantics . . . 96
6.3.3 Example of Model . . . 98
6.3.4 Considered Dataset . . . 98

7 Efficient Semantic Model Differencing 99
7.1 Introduction . . . 99
7.2 Semantic Model Differencing . . . 100
7.2.1 Existing Approach . . . 100
7.2.2 Application to fUML . . . 102
7.2.3 Observations . . . 106
7.3 Efficient Semantic Model Differencing . . . 106
7.3.1 Enhancement of the Existing Approach . . . 106
7.3.2 Application to fUML . . . 107
7.4 Evaluation . . . 110
7.4.1 Set-up . . . 110
7.4.2 Complexity Reduction (RQ#7.2). . . . 110
7.4.3 Performance Improvement (RQ#7.1). . . . 111

7.5 Conclusion . . . 112

8 Efficient and Advanced Omniscient Debugging 113
8.1 Introduction . . . 113
8.2 Multidimensional Omniscient Debugging . . . 115
8.2.1 Comparison of Interactive Debugging Approaches . . . 115
8.2.2 Multidimensional Omniscient Debugging Services . . . 116
8.2.3 Example Debugging Scenario . . . 117
8.3 Efficient and Advanced Omniscient Debugging for xDSMLs . . . 118
8.3.1 Overview of the Approach . . . 118
8.3.2 Execution Engine . . . 120
8.3.3 Multidimensional Domain-Specific Trace Metamodel . . . 121
8.3.4 Trace Constructor . . . 121
8.3.5 Generic Trace Metamodel . . . 122
8.3.6 State Manager . . . 123
8.3.7 Domain-Specific Trace Manager . . . 123
8.3.8 Generic Multidimensional Omniscient Debugger . . . 124
8.3.9 Implementation in the GEMOC Studio . . . 127
8.4 Evaluation . . . 127
8.4.1 Efficiency of the Approach . . . 127
8.4.2 Benefits of Multidimensional Facilities . . . 130
8.5 Related Work . . . 130
8.5.1 Generic Omniscient Debugging in MDE . . . 130
8.5.2 Trace Visualization and Debugging in MDE . . . 130
8.6 Conclusion . . . 131

9 Tool Support in the Context of GEMOC 133
9.1 Implementation of Scalable Model Cloning for EMF . . . 133
9.1.1 From Theory to Practice . . . 134
9.1.2 Extending EMF Librairies . . . 134
9.1.3 Design-time: Analysis and Code Generation . . . 135
9.1.4 Runtime: Cloning . . . 137
9.1.5 Resulting Plugins and Usage . . . 137
9.2 Step Management Facilities for Kermeta . . . 139
9.2.1 Kermeta: an Extension of Xtend . . . 139
9.2.2 Adding Execution Steps Declaration and Management . . . 140
9.3 Enhanced GEMOC Studio Execution Framework . . . 141
9.3.1 Presentation of the GEMOC Studio . . . 141
9.3.2 From One to Multiple Execution Engines . . . 143
9.4 Execution Trace Manager Generator in the GEMOC Studio . . . 144
9.4.1 Extracting Data from the Operational Semantics . . . 145
9.4.2 Generic Generation of the Trace Metamodel and Manager . . . 146
9.4.3 Integration in the GEMOC Studio . . . 146
9.5 Omniscient Model Debugging in the GEMOC Studio . . . 146

9.5.1 Execution Trace Manager Addon Generator 147
9.5.2 Generic Omniscient Debugging Addons 147

IV Conclusion and Perspectives 151

10 Conclusion and Perspectives 153
10.1 Conclusion 153
10.2 Perspectives 154
10.2.1 Model Cloning 154
10.2.2 Execution Trace Metamodel Generation 155
10.2.3 Model Omniscient Debugging 157

List of Figures 159

List of Tables 163

Author's publications 167

Bibliography 169

Introduction en français

Contexte

Un important défi dans l'ingénierie des logiciels et des systèmes est le développement et la maintenance de systèmes *complexes*, tels que les systèmes cyber-physiques ou l'Internet des objets. La conception de tels systèmes nécessite des experts de domaines divers et hétérogènes. Cette complexité compromet à la fois leur bon développement et leur bon fonctionnement, ce qui implique un véritable besoin de méthodes, méthodologies et outils appropriés [26].

L'ingénierie dirigée par les modèles (IDM) est un paradigme de développement qui vise à faire face avec la complexité des systèmes par la séparation des préoccupations à l'aide de *modèles*. Un modèle est une représentation d'un aspect particulier d'un système, et est défini en utilisant des abstractions spécifiques fournies par un langage de modélisation dédié (LMD) [137]. Au cœur de l'IDM se trouve l'idée de passer de modèles *descriptifs* représentant des systèmes existants, à des modèles *productifs* qui peuvent être utilisés pour construire le système cible [152]. Ces dernières années, des études ont mis en évidence les nombreux avantages d'IDM pour le développement de systèmes complexes, tels que des améliorations de la productivité des développeurs ou de la qualité des systèmes produits [119, 86]. Un facteur explicatif est l'utilisation de modèles afin d'effectuer *la vérification et la validation au plus tôt* (V&V) des systèmes (*e.g.,* [24]). En effet, la plupart des erreurs logicielles se produisent dans les premières phases du développement (*i.e.,* exigences et conception), et sont plus coûteuses à retirer dans les étapes ultérieures [17, 16].

Alors que de nombreux modèles ne représentent que les aspects structurels de systèmes, une grande quantité exprime des aspects comportementaux de ces mêmes systèmes. Dans ce cas, pour assurer qu'un modèle est correct vis à vis de son comportement prévu, des techniques *dynamiques* de V&V au plus tôt sont nécessaires, tels que le débogage omniscient [38], la différenciation sémantique [99] ou la vérification d'exécution [102]. Ces techniques nécessitent que les modèles soient *exécutables*, ce qui est possible en définissant la sémantique d'exécution des LMDs utilisés pour les décrire.

Bien que, techniquement, seuls les modèles qui y sont conformes soient dits « exécutables », ces langages sont appelés *LMDs exécutables* (LMDx). En plus de permettre la V&V dynamique au plus tôt, l'exécutabilité au niveau modèle donne également la possibilité de directement déployer un modèle exécutable sur un système de production.

Énoncé du problème

Alors qu'un modèle exécutable exprime intrinsèquement un comportement en *intention*, les techniques dynamiques de V&V nécessitent une représentation en *extension* d'un comportement au fil du temps. Une représentation courante du comportement d'un modèle est la *trace d'exécution*, qui est une séquence contenant toutes les informations pertinentes à propos d'une exécution au fil du temps. Ces informations peuvent inclure les *états* atteint lors de l'exécution, les *pas* responsables de ces changements d'état, et les *stimuli* provenant de l'environnement d'exécution du système.

Toutes les approches de V&V mentionnées précédemment reposent sur des traces d'exécution : le débogage omniscient repose sur une trace d'exécution afin de revisiter un état précédent de l'exécution; la différenciation sémantique consiste à comparer les traces d'exécution de deux modèles afin de comprendre les variations sémantiques entre eux; la vérification d'exécution consiste à vérifier si une trace d'exécution est conforme à une propriété temporelle. En outre, les traces d'exécution sont au cœur de la *vérification d'équivalence comportementale* de LMDxs, comme la bisimulation [118], et peuvent être utilisées comme indices [51] partagés entre différentes approches de V&V combinées [19].

Par conséquent, il apparaît que *fournir des dispositifs pour gérer des traces d'exécution* est essentiel pour rendre possible la V&V dynamique pour les LMDxs. Cela inclut l'*acquisition*, le *traitement* et la *visualisation* des traces d'exécution provenant à la fois du test et du déploiement de modèles exécutables. Cependant, une condition préalable importante doit être satisfaite : la définition d'une *structure de données* qui définit le contenu et l'agencement des traces d'exécution de modèles conformes à un LMDx. Cette entreprise est pas triviale pour au moins deux raisons. Premièrement, la sémantique opérationnelle d'un LMDx peut être arbitrairement complexe, à la fois en ce qui concerne la définition de l'état d'un modèle exécuté, et en ce qui concerne la définition de la transformation de modèle qui change cet état. Structurer et adapter ces informations pour définir une structure de données de trace d'exécution est donc difficile. Deuxièmement, une trace d'exécution a tendance à être un grand artefact : une courte exécution d'un programme Java simple de 20 classes et de 3 000 lignes de code peut mener à 150 000 appels de méthode à stocker dans une trace d'exécution [39]. Par conséquent, la structure de données utilisée doit être adaptée à une représentation et un traitement efficace de grandes traces.

En résumé, fournir des dispositifs pour gérer des traces d'exécution revient à faire face à trois principaux challenges étroitement liés [21] :

Ch#1: La *facilité d'utilisation* d'une structure de données de trace d'exécution doit être assurée pour faire face à la complexité des données. Plus précisément, il est nécessaire

de prendre en compte à la fois les *manipulations génériques* (*e.g.,* comparer le nombre de différents états ou le nombre de pas) et les *manipulations dédiées* (*e.g.,* déterminer le nombre de jetons qui ont traversé une place dans un réseau de Petri) de traces d'exécution.

Ch#2: Étant donné que l'exécution d'un modèle ou programme simple peut conduire à de très grandes traces d'exécution, le *passage à l'échelle en mémoire* doit être pris en compte. En effet, même si les solutions reposant sur une base de données pour stocker les modèles[1] (*e.g.,* des traces d'exécution) sont de plus en plus efficaces, charger des modèles directement en mémoire reste plus efficace pour les grands modèles et les manipulations lourdes [11].

Ch#3: Enfin, également en raison de leur grande taille, *le passage à l'échelle dans le temps* de la manipulation des traces d'exécution est de première importance, et implique le besoin de moyens efficaces pour parcourir une trace.

On peut observer que relever ces défis pour tout modèle exécutable nécessite de prendre en compte une grande quantité de LMDxs existants et futurs. Cela représente un obstacle clé supplémentaire que nous considérons dans cette thèse, qui conduit naturellement soit à des solutions génériques, soit à des solutions génératives.

Contributions

Pour relever les challenges mentionnés ci-dessus, nous étudions deux directions complémentaires. Tout d'abord, nous nous concentrons sur la représentation de *l'état d'un modèle exécuté* dans le cadre de *traces d'exécution basées sur des clones*. Une trace d'exécution contenant tous les états atteints par un modèle exécuté peut être obtenu de manière générique par *clonage* du modèle après chaque pas d'exécution. Cette façon de faire comporte des avantages en ce qui concerne la *facilité d'utilisation* (Ch#1), car la structure de données des traces d'exécution est simple et appropriée pour des manipulations génériques. En outre, les transformations de modèles et les manipulations existantes spécifiques au LMDx peuvent être appliquées directement sur les états stockés sous forme de clones. Cependant, lorsqu'il est manipulé, un modèle est représenté par un ensemble d'éléments stockés dans la mémoire: la *représentation en mémoire* du modèle. Cloner un modèle est généralement effectué en dupliquant la représentation en mémoire complète d'un modèle, ce qui peut nécessiter une quantité importante de mémoire, et donc compromettre le besoin de *passage à l'échelle en mémoire* (Ch#2). Pour faire face à ce problème, nous proposons une **approche efficace de clonage de modèle** [20] permettant de créer de grandes quantités des clones de modèles tout en épargnant utilisation de la mémoire. Notre approche est basée sur l'observation qu'une manipulation de modèle modifie rarement ce dernier dans son l'intégralité. Par exemple, dans le cas de l'exécution d'un modèle, la seule partie modifiée est l'état d'exécution qui peut être

[1] `https://www.eclipse.org/cdo/`

dispersé dans les différentes parties du modèle. Ainsi, en sachant quelles parties pourraient être modifiées, notre approche de clonage détermine ce qui peut être partagé entre les représentations en mémoire d'un modèle et de ses clones. Notre algorithme de clonage générique est paramétrable par trois stratégies différentes, chacune reposant sur un compromis entre les gains en mémoire et la facilité d'utilisation des manipulations de clones. Nous proposons une mise en œuvre de l'approche au sein de l'Eclipse Modeling Framework (EMF), ainsi qu'une évaluation de l'empreinte mémoire et de la performance de la manipulation des clones avec 100 modèles générés aléatoirement. Les résultats montrent une corrélation positive entre la proportion d'éléments partageables et les gain de mémoire, tandis que la mediane du surcoût de manipulation est de 9,5% lors de la manipulation des clones.

Ensuite, nous nous concentrons sur la *structure* des traces d'exécution qui contiennent de informations à la fois sur les états et les pas d'exécution d'un modèle. Alors que les traces d'exécution basées sur des clones montrent certains avantages, elles nécessitent une structure de données *générique* basée sur une *unique séquence* d'états. Cela a deux conséquences. Premièrement, il ya un écart sémantique entre les concepts de domaine du LMDx et la structure de données générique, ce qui compromet la *facilité d'utilisation* (Ch#1) dans le cas de manipulations de traces spécifiques à un domaine. Deuxièmement, les manipulations qui mettent l'accent sur une partie spécifique de l'état d'un modèle doivent malgré tout parcourir la trace d'exécution complète, même si cette partie a changé un petit nombre de fois. Cela compromet *le passage à l'échelle dans le temps* (Ch#3). Pour faire face à ces problèmes, nous proposons **une approche générative pour définir des métamodèles de traces d'exécution multidimensionnelles et spécifiques à un domaine** [23]. Un *métamodèle* est une structure de données définie par un modèle orienté objet qui définit un domaine particulier. Pour améliorer la facilité d'utilisation, notre première idée est d'aller de métamodèles génériques vers *une méta-approche générique* pour définir les *métamodèles de traces d'exécution spécifiques à un domaine*. Nous accomplissons cela à l'aide de la connaissance des parties d'un LMDx qui sont requises par les manipulations, en utilisant donc le même principe que la contribution précédente. Ensuite, pour améliorer le passage à l'échelle dans le temps des manipulation, notre deuxième idée est de créer des métamodèles de traces d'exécution *multidimensionnelles*, *i.e.,* métamodèles qui fournissent de nombreux chemins de navigation pour explorer une trace. Plus précisément, ces chemins permettent de suivre l'évolution des différents éléments du modèle, évitant ainsi de parcourir la trace complète pour analyser ces changements locaux. Par rapport à des traces d'exécutions basées sur des clones, les résultats montrent une simplification de la définition des manipulations de trace, un temps d'exécution plus faible, et une empreinte mémoire inférieure.

Applications

Comme nous l'avons mentionné précédemment, une grande quantité de techniques dynamiques de V&V reposent sur les traces d'exécution pour analyser les comportements de modèles exécutables. Par conséquent, après avoir étudié comment mieux construire

et manipuler des traces d'exécution, nous étudions comment mettre à profit nos contributions pour améliorer deux approches de V&V dynamiques existantes.

Nous considérons d'abord le domaine de *l'évolution de modèles*, dont le souci est d'analyser et de comprendre les modifications apportées à un modèle au fil du temps. Dans le cas d'un modèle exécutable, puisque un changement dans son contenu peut avoir un impact sur son comportement, il est nécessaire de comparer les comportements d'un modèle avant et après un changement. Cela peut être fait en utilisant la *différenciation sémantique de modèles*, qui consiste à comparer les traces d'exécution de différents modèles. Tout d'abord, des *règles de différenciation sémantique* sont définies pour un xDSML donné pour définir quelles différences entre les traces d'exécution constituent des différences sémantiques entre les modèles. Ensuite, ces règles sont utilisées pour comparer les traces d'exécution des modèles considérés, ce qui permet par conséquent de comparer leurs comportements. Toutefois, la définition de ces règles est une tâche difficile, surtout lorsque le métamodèle de traces d'exécution utilisé est *générique*, car il manque de *facilité d'utilisation* (Ch#1) pour un tel cas. De même *le passage à l'échelle dans le temps* (Ch#3) est un problème, car tous les états doivent être énumérés lorsque l'on compare des traces d'exécution ordinaires, même si les règles sont basées sur un sous-ensemble de l'information qu'elles contiennent. Pour surmonter ces problèmes, nous proposons une **amélioration de la différenciation sémantique par l'utilisation de métamodèles de traces d'exécution multidimensionnelles spécifiques à un domaine** [23]. Pour valider cette intuition, nous générons d'abord un tel métamodèle de traces pour un LMDx concret, fUML[2]. Ensuite, nous définissons une ensemble de règles de différenciation sémantique pour fUML basées sur ce métamodèle généré. Finalement, nous utilisons ces règles pour comparer les traces d'un ensemble de modèles fUML du monde réel extraits d'une étude de cas existante. Les résultats montrent une amélioration significative des performances (Ch#3) et une simplification des règles de différenciation sémantique par rapport à des règles équivalentes basées sur un métamodèle de traces d'exécution générique reposant sur des clones (Ch#1).

Nous étudions ensuite le domaine du débogage interactif, qui consiste à contrôler (*i.e.,* mettre en pause ou reprendre l'exécution) et observer une exécution afin de trouver la cause de certains comportements inattendus. Tandis que le débogage interactif permet communément d'aller seulement *vers l'avant* dans une exécution, le *débogage omniscient* est une technique prometteuse qui repose sur des traces d'exécution pour permettre le libre parcours des états atteints, ce qui inclut donc le *retour en arrière* dans l'exécution. Alors que certains langages généraux (*e.g.,* Java) possèdent déjà des outils pour le débogage omniscient, développer un tel outil complexe pour tout LMDx reste une tâche difficile propice aux erreurs. Une solution à ce problème est de définir un débogueur omniscient générique pour tout LMDx. Cependant, un support générique de tout LMDx compromet l'efficacité et la facilité d'utilisation d'une telle approche. Pour répondre à ces problèmes, nous proposons une approche de **débogage omniscient avancée et efficace pour tout LMDx** [22]. Notre contribution consiste en un débogueur omniscient partiellement générique s'appuyant sur un dispositif de gestion de

[2]Signifie *foundational UML*, qui est un sous-ensemble exécutable et standardisé de UML [122].

traces généré et spécifique au LMDx considéré. Ce dispositif inclut un métamodèle de traces d'exécution spécifique au domaine généré à l'aide de notre seconde contribution. Étant spécifique au domaine, ce dispositif est optimisé pour le LMDx considéré pour une meilleure efficacité. La facilité d'utilisation est renforcée par la mise à disposition de services de débogage omniscient *multidimensionnels*, définis à l'aide du métamodèle de traces d'exécution généré (Ch#1). Les résultats montrent que notre approche est en moyenne 3,0 fois plus efficace en mémoire (Ch#2) et 5,03 plus efficace dans le temps (Ch#3) par rapport à une solution générique qui clone le modèle exécuté à chaque étape.

Contexte de cette thèse

Cette thèse a été réalisée par le biais de plusieurs partenariats internationaux. D'abord, le travail sur les métamodèles de trace d'exécution multidimensionnels et spécifiques à un domaine [23] a été fait en collaboration avec le Business Informatics Group (BIG) de l'Université Technologique de Vienne (TU Wien), située en Autriche. Ensuite, le travail sur le débogage omniscient avancé et efficace [22] a été fait en collaboration avec le Software Engineering Group de l'Université de l'Alabama (UA), située aux Etats-Unis.

Ces collaborations ont été réalisées dans le cadre de l'Initiative GEMOC, une coopération industrielle et universitaire qui vise à permettre l'utilisation coordonnée de LMDxs, aussi appelé la *mondialisation des langages de modélisation* [34]. Les membres de l'Initiative GEMOC rassemblent et partagent une expertise complémentaire dans divers domaines, tels que la modélisation de logiciels ou la vérification et validation de logiciels. En outre, les résultats de recherche sont en permanence matérialisés dans le GEMOC Studio [33], un atelier dans lequel on peut définir des LMDxs, définir des modèles, utiliser des techniques de V&V sur ces modèles, et enfin les exécuter.

Par conséquent, nous avons intégré nos contributions dans le GEMOC Studio, ce qui nous a permis à la fois de bénéficier d'un cadre existant pour mettre en œuvre notre approches, et de recueillir des commentaires des membres du projet. De plus, nos travaux sont rendus accessibles à tous les utilisateurs actuels et futurs du GEMOC Studio.

Chapter 1

Introduction

1.1 Context

A most important challenge in software and systems engineering is the development and maintenance of *complex* systems, such as cyber-physical systems or the internet of things. Designing such systems require experts of diverse and heterogeneous domains. Because of this complexity, there are many threats to their proper development and functioning, which implies a need for appropriate methods, methodologies and tools [26].

Model-Driven Engineering (MDE) is a development paradigm that aims at coping with the complexity of systems by separating concerns through the use of *models*. A model is a representation of a particular aspect of a system, and is defined using specific abstractions provided by a Domain-Specific Modeling Language (DSML) [137]. At the core of MDE is the idea of going from *descriptive* models representing existing systems to *prescriptive* models that can be used to construct the target system [152]. In the past years, studies have shown evidence of the many benefits of MDE for the development of complex systems, such as improvements regarding the productivity of developers or regarding the quality of the systems [119, 86]. One explanatory factor is the use of models to perform *early verification and validation* (V&V) of systems (*e.g.,* [24]). Indeed, most software errors occur in the early phases of development (*i.e.,* requirements and design), and are more expensive to remove in later stages [17, 16].

While many models only represent structural aspects of systems, a large amount express behavioral aspects of the same systems. In this case, to ensure that a model is correct with regard to its intended behavior, early *dynamic* V&V techniques are required, such as omniscient debugging [38], semantic differencing [99] or runtime verification [102]. These techniques require models to be *executable*, which can be achieved by defining the execution semantics of DSMLs used to describe them. Although technically only conforming models are said executable, such languages are called *executable DSMLs* (xDSMLs). In addition to enabling early dynamic V&V, providing executability at the model level also gives the possibility to directly deploy an executable model to run on a production system.

1.2 Problem Statement

While an executable model by itself inherently expresses an *intended* behavior, dynamic V&V techniques need an *extended* representation of behavior over time. A most common representation of a model's behavior is the *execution trace*, which is a sequence containing all the relevant information about an execution over time. Such information may include the *execution states* reached during the execution, the *execution steps* that were responsible for these state changes, and the *stimuli* originating from the execution environment and the system.

All previously mentioned V&V approaches rely on execution traces: omniscient debugging relies on an execution trace to revisit a previous execution state; semantic differencing consists in comparing execution traces of two models in order to understand the semantic variations between them; runtime verification consists in checking whether or not an execution trace satisfies a property. In addition, execution traces are at the core of *behavioral equivalence checking* of xDSMLs, such as bisimulation [118], and can be used as evidence [51] shared among different combined V&V approaches [19].

Consequently, it appears that providing *execution trace management facilities* is an essential requirement to support dynamic V&V for xDSMLs. Such facilities include *acquiring*, *processing* and *visualizing* execution traces that result both from testing and deploying executable models. However, these facilities have an important prerequisite to satisfy: the definition of a *data structure* to define the content and the layout of the execution traces of an xDSML. Yet, this undertaking is not trivial for at least two reasons. First, the operational semantics of an xDSML can be arbitrarily complex, both regarding the definition of the execution state and the definition of the model transformation that changes it. As a result, structuring and adapting this information into an execution trace data structure is difficult. Second, execution traces tend to be very large artifacts: a short execution of a simple Java program of 20 classes and 3,000 lines of code can lead to 150 000 method calls to store in an execution trace [39]. Consequently, a data structure must be adapted for an efficient representation and processing of large traces.

All in all, providing execution trace management facilities can be summarized as three main inter-related challenges [21]:

Ch#1: The *usability* of an execution trace data structure must be ensured to cope with the complexity of data. More precisely, both *generic manipulations* (*e.g.,* comparing the number of different states or the amount of steps) and *domain-specific manipulations* (*e.g.,* determining how many tokens traversed a Petri net place) must be taken into account.

Ch#2: Since executing even a simple model or program can lead to very large execution traces, *scalability in memory* of executions traces must be taken into account. Indeed, while database solutions for storing models[1] (*e.g.,* execution traces) are more and more efficient, loading models directly in memory remains more efficient for large models and heavyweight manipulations [11].

[1] `https://www.eclipse.org/cdo/`

Ch#3: Finally, also because of their large size, *scalability in manipulation time* of execution traces are of primary importance, and imply the need for efficient ways to browse a trace.

It can be observed that addressing these challenges for any executable model requires to take into account a large amount of existing and potential xDSMLs. This represents an additional key obstacle that we consider in this thesis, which naturally leads to either generic or generative solutions.

1.3 Contributions

To tackle the aforementioned challenges, we investigate two complementary directions. First, we focus on the representation of the *execution state* of an executed model in the context of *clone-based* execution traces. An execution trace containing all the states reached by an executed model can be obtained in a generic way by *cloning* the model after each execution step. Such way of doing brings advantages regarding *usability* (Ch#1), since the data structure of the execution trace is simple and appropriate for generic manipulations. Moreover, existing model transformations and queries specific to the xDSML can directly be applied on execution states stored in a clone-based execution trace. Yet, at runtime, a model is represented by a set of elements stored in memory called the *runtime representation* of the model. Cloning is usually done by duplicating the complete runtime representation of a model, hence requiring an important amount of memory, compromising the need for *scalability in memory* (Ch#2). To cope with this problem, we propose **a scalable model cloning approach** [20] to create large amounts of model clones while sparing memory usage. Our approach is based on the observation that manipulations rarely modify a whole model. In the case of model execution, the only modified part is the execution state, which may be scattered in the different parts of the model. Hence, knowing which parts might get modified, our cloning approach determines what can be shared between the runtime representations of a model and its clones. Our generic cloning algorithm is parameterized with three strategies that establish a trade-off between memory savings and the usability of clone manipulations. We propose an implementation of the approach within the Eclipse Modeling Framework (EMF), along with our evaluation of memory footprints and computation overheads with 100 randomly generated models. Results show a positive correlation between the proportion of shareable elements and memory savings, while the worst median overhead is 9,5% when manipulating the clones.

Then, we focus on the *structure* of execution traces that contain information about both states and steps. While clone-based execution traces show some benefits, they are necessarily relying on a *generic* data structure based on a *unique sequence* of execution states. This has two consequences. First, there is a gap between the domain concepts of the xDSML and the generic data structure, which compromises *usability* (Ch#1) regarding domain-specific trace manipulations. Second, execution trace manipulations that focus on a specific part of the execution state has still to browse the complete trace

even if this part changed a small number of times, hence compromising *scalability in time* (Ch#3). To cope with these problems, we propose **a generative approach to define multidimensional and domain-specific execution trace metamodels** [23]. A *metamodel* is a data structure defined by an object-oriented model defining a particular domain. To enhance usability, our first idea is to go from generic trace metamodels to a *generic meta-approach* to define *domain-specific execution trace metamodels*. This is accomplished by knowing which parts of an xDSML is required by the manipulations, hence using the same principle as the previous contribution. Then, to enhance scalability in manipulation time, our second idea is to create *multidimensional* trace metamodels, *i.e.,* metamodels that provide many navigation paths to explore a trace. More precisely, these paths make possible to follow the changes of individual elements of the model, thus avoiding to browse the complete trace to analyze these changes. As compared to regular clone-based execution traces, results show a simplification of the trace manipulations definitions, a lower execution time, and a lower memory footprint.

1.4 Applications

As we previously mentioned, a large amount of dynamic V&V techniques rely on execution traces to analyze the behaviors of executable models. Therefore, after investigating how to better construct and manipulate execution traces, we study how to take advantage of our contributions to improve two existing dynamic V&V approaches.

We first consider the field of *model evolution*, whose concern is to analyze and understand the changes made to a model over time. In the case of an executable model, since a change in its content may impact its behavior, taking a change into account requires comparing the behaviors of the model before and after the change. This can be achieved using *semantic model differencing*, which consists in comparing execution traces from different models. First, *semantic differencing rules* are defined for a given xDSML, the rules indicating which differences among the execution traces constitute semantic differences among the models. Second, these rules are used to compare the execution traces of the considered models, hence comparing their behaviors. However, defining semantic differencing rules is a difficult task, especially when the used execution trace metamodel is *generic*, since it lacks *usability* (Ch#1). Likewise *scalability in time* (Ch#3) is an issue, since all states must be enumerated when comparing the execution traces, even if the rules are based on a subset of the information they contain. To overcome these problems, we propose an **an enhancement of semantic differencing using multidimensional domain-specific execution trace metamodels** [23]. We validate this intuition by generating such a metamodel for a real world xDSML, namely fUML[2]. Then, we define a set of semantic differencing rules for fUML based on this generated metamodel. Finally, we use these rules to compare traces of a set of real world fUML models extracted from an existing case study. Results show a significant performance improvement (Ch#3) and a simplification of the semantic differencing rules as compared to equivalent rules based on a generic execution trace metamodel (Ch#1).

[2]Stands for *foundational UML*, which is a standardized executable subset of UML [122].

We then consider the field of interactive debugging, whose concern is to control (*i.e.,* pause or unpause) and observe an execution in order to find the cause of some unintended behavior. While regular interactive debugging only allows to go *forward* in an execution, *omniscient debugging* is a promising technique that relies on execution traces to enable free traversal of the reached states, which includes going *backward* in the execution. While some General-Purpose Languages (GPLs) already have support for omniscient debugging, developing such a complex tool for any xDSML remains a challenging and error prone task. A solution to this problem is to define a generic omniscient debugger for all xDSMLs. However, generically supporting any xDSML both compromises the efficiency and the usability of such an approach. To address these problems, we propose **an advanced and efficient omniscient debugging approach for xDSMLs** [22]. Our contribution consists in a partly generic omniscient debugger supported by generated domain-specific trace management facilities. These facilities include a multidimensional domain-specific execution trace metamodel, obtained using our second contribution. Being domain-specific, these facilities are tuned to the considered xDSML for better efficiency. Usability is strengthened by providing multidimensional omniscient debugging, which is achieved using our generated execution trace metamodel (Ch#1). Results show that our approach is on average 3.0 times more efficient in memory (Ch#2) and 5.03 more efficient in time (Ch#3) when compared to a generic solution that clones the model at each step.

1.5 Context of this Thesis

This thesis was done through several beneficial international partnerships. First, the work on multidimensional and domain-specific execution trace metamodels [23] was done in collaboration with the Business Informatics Group (BIG) from the Vienna University of Technology (TU Wien), located in Austria. Second, the work on advanced and efficient omniscient debugging [22] was done in collaboration with the Software Engineering Group from the University of Alabama (UA), located in the USA.

These collaborations were done in the context of the GEMOC Initiative, an academic and industrial effort that aim at supporting the coordinated use of DSMLs, also called the *globalization of modeling languages* [34]. The members of the GEMOC Initiative gather and share complementary expertise from various domains, such as software modeling or software verification and validation. In addition, research result are continuously materialized in the GEMOC Studio [33], a language and modeling workbench in which one can define xDSMLs, define models, use V&V techniques on these models, and finally execute them.

Consequently, we implemented and integrated our contributions within the GEMOC Studio, which allowed us both to benefit from an existing framework to implement our approaches, and to gather direct feedback from the project members. In addition, our work is made accessible to all present and future users of the GEMOC Studio.

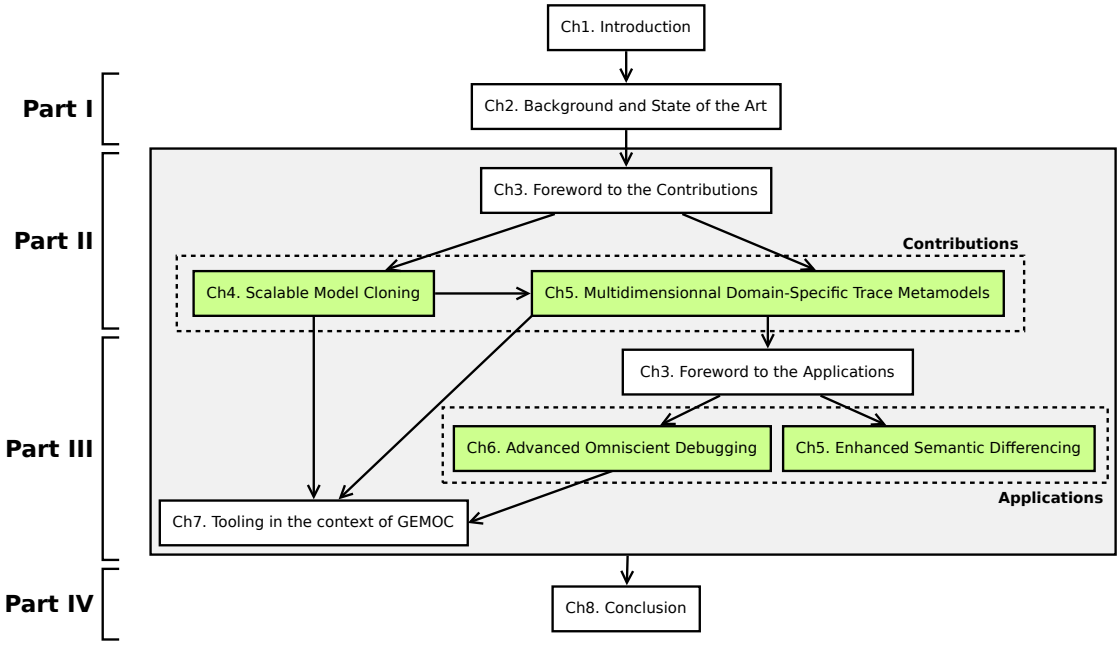


Figure 1.1: Graphical representation of the outline of the thesis. Chapters in green contain the core of the scientific contributions.

1.6 Outline

Figure 1.1 shows an overview of the structure of the thesis. Arrows define the reading partial order. We present the different chapters thereafter.

Part I — State of the Art

Chapter 2 introduces the foundations and the state of the art of model-driven engineering, executable metamodeling, execution trace management, and model debugging.

Part II — Contributions

Chapter 3 is a foreword to both contributions. We present some observations regarding trace manipulations, a synthesis of the state of the art regarding execution trace data structures, and the proposal that led to our contributions.

Chapter 4 presents our first contribution, which is a scalable model cloning approach through data sharing. We give a more detailed motivation regarding the different usages of model cloning, a description of the algorithm and of the cloning strategies, and an evaluation of memory gain using random metamodels.

Chapter 5 presents our second contribution, which is a generative approach to define multidimensional domain-specific execution trace metamodels for xDSMLs. We explain the advantages of a domain-specific data structure as compared to a generic one, then explain our generation algorithm and discuss the benefits of the approach.

Part III — Applications and Tooling

Chapter 6 is a foreword to both applications. We explain why we rely on these applications to evaluate our second contribution, then we present an overview, and lastly we describe the fUML case study considered for both evaluations.

Chapter 7 shows our first application, which is an enhancement of a semantic differencing approach based on execution traces. We first the existing introduce semantic differencing approach, then we explain how it is enhanced using the generation of a multidimensional domain-specific trace metamodel, and lastly we evaluate our approach by defining and using semantic differencing rules for fUML.

Chapter 8 shows our second application, which is an omniscient debugging approach for xDSMLs relying on multidimensional domain-specific execution trace metamodels. We first introduce the challenges in omniscient debugging, then explain our approach, and finally present its evaluation on fUML which further highlights the benefits of multidimensional domain-specific execution trace metamodels.

Chapter 9 presents an overview of the software development that was achieved during this thesis, either to improve existing tools or to implement our approaches and applications. In particular, it explains the integration of our work within the GEMOC studio, which is a language and modeling workbench resulting from an academic and industrial project.

Part IV — Conclusion and Perspectives

Chapter 10 concludes the thesis by summarizing the advances that it brings to execution trace management for xDSMLs and to dynamic V&V of executable models. We end by discussing the perspectives of future research on the topic.

Part I

State of the Art

Chapter 2

State of the Art

In this chapter, we present the state of the art in the different domains covered by our contributions and applications. In Section 2.1, we first introduce model-driven engineering through a number of fundamental concepts. Then in Section 2.2, we introduce object and model duplication , and we present existing work on the topic.

Continuing, in Section 2.3, we focus more specifically on executable domain-specific modeling languages (xDSMLs), and we introduce a running example of xDSML. The main purpose of xDSMLs is to enable the dynamic verification and validation (V&V) of behavioral models, which requires the capture of execution traces describing their executions. Consequently, in Section 2.4, we define what is an execution trace, and we review the literature regarding execution trace management and execution trace data structures. In Section 2.5 we first explain what is interactive debugging of models and present existing work on model debugging, then we present omniscient debugging and different categories of omniscient debuggers.

2.1 Model-Driven Engineering

Model-driven engineering (MDE) is a development paradigm that aims at coping with the complexity of systems by separating concerns through the use of *models*. [137] While the term model is extensively used in many scientific fields, France et al. [62] give the following description in the context of MDE:

> "A model is an abstraction of some aspect of a system. The system described by a model may or may not exist at the time the model is created. Models are created to serve particular purposes, for example, to present a human understandable description of some aspect of a system or to present information in a form that can be mechanically analyzed"

Proper expressiveness is necessary to create and use meaningful models, which is accomplished through the definition of *languages*. We distinguish two main categories

of languages. On the one hand, a general-purpose language (GPL) provides substantial expressiveness to be able to handle a large variety of concerns, and can thus be used for modeling many aspects of a system. On the other hand, a domain-specific modeling language (DSML) defines specific abstractions dedicated to a particular area of expertise [137, 115]. The specificities of a DSML can also be influenced by the kind of usage (*e.g.,* visualization or simulation) and the kind of information that must be modeled (*e.g.,* architectural components or behaviors). In this thesis, we are mostly interested in the use of DSMLs to model the multiple aspects of a system.

As any other language, a DSML consists both of a *syntax*, defining what can be expressed, and *semantics*, defining the meaning of what can be expressed [80]. More precisely, the syntax is composed of both an *abtract syntax*, which defines the concepts of the DSML and the relationships between them, and a *concrete syntax*, which defines a human-readable representation to manipulate these concepts. Semantics are defined through both a *semantic domain*, and a *mapping* from concepts of the abstract syntax to the semantic domain. Note that the concrete syntax is not involved in the definition of the semantics of a DSML. Therefore, since semantics are the primary concern of our work, we focus in this thesis on the abstract syntax of a DSML and we leave aside the definition of the concrete syntax.

2.1.1 Metamodel

A most common way to define the abstract syntax of a DSML is by defining a *metamodel*. There are many definitions of this term in the literature : "*a model defining a language*" [89], "*a model to model modeling*" [120] or "*a model defining the structure and semantics of metadata*" [52]. In this thesis, we consider a metamodel to be an object-oriented model, similarly to [12, 13].

Therefore, a metamodel is essentially composed of *classes*, each being composed of *properties*. In addition, a metamodel possesses *static semantics*, which are additional structural constraints that must be satisfied by conforming models.

> **Definition 1** *A* metamodel *is an object-oriented model defining a particular domain. It is thus composed of:*
>
> - *A set of* classes *(also called* metaclasses*) that consist of properties.*
>   - *A* property *is either an* attribute *(typed by a datatype, e.g., integer) or a* reference *to another class.*
>   - *A class can be* abstract*, meaning it cannot be instantiated.*
>   - *A class can* inherit *from one or multiple classes, meaning it shares their properties.*
> - Static semantics*, which are a set of constraints that must be satisfied by conforming models (e.g., multiplicities, containment references, OCL rules).*

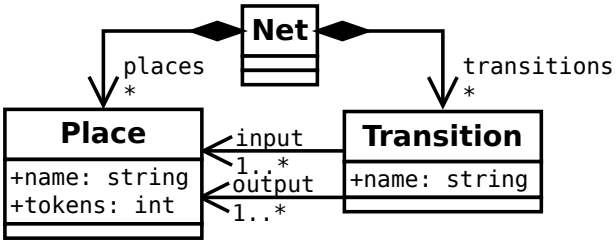


Figure 2.1: Petri net abstract syntax.

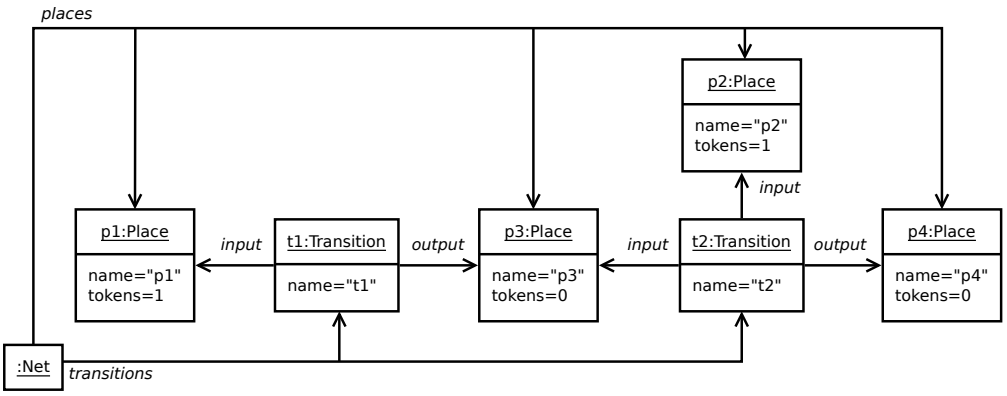
Figure 2.1 shows an example of metamodel defining the domain of Petri nets. More precisely, we consider a common subclass of Petri nets whose arcs have a weight of one. It is composed of three classes: Net, Transition and Place. Each class, through its name and the properties it contains, expresses a concept of the domain. A Petri net is composed of places and transitions, hence the class Net has a containment reference `places` pointing to the class Place, and a containment reference `transitions` pointing to the class Transition. A transition has input and output places, hence the class Transition has two references `input` and `output` pointing to the class Place. A place has a number of tokens, hence the attribute `tokens`. Likewise, both places and transitions have names, hence the attributes `name` in the corresponding classes.

A widely used standard that matches this definition of metamodel is the Essential Meta-Object Facility (EMOF) [120], introduced by the Object Management Group (OMG). It is supported by the Object Constraint Language (OCL) [121], also maintained by the OMG, for the definition of complex static semantics rules. In practice, the tool-supported Ecore language from the Eclipse Modeling Framework (EMF) [145] is considerably aligned with EMOF, and is therefore the *de facto* standard extensively used for defining metamodels.

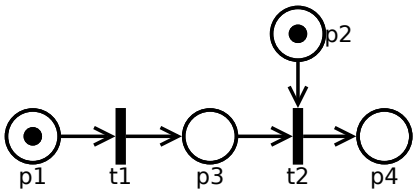
2.1.2 Model

Based on the definition of *metamodel*, we can define more precisely what we call a *model*. In the class-based object-oriented paradigm, a class can be instantiated into an *object*. Since a metamodel is a set of classes, we consider a model as a set of objects that are instances of these classes, and that satisfy the static semantics of the metamodel. This is commonly referred as the *conformity* relationship between a model and its metamodel. Note that we can also say that a model conforms to a DSML, which is simply a shorter way to express that the model conforms to the metamodel of the DSML.

> **Definition 2** *A* model *is a set of objects. Furthermore, a model conforms to a unique metamodel.* Conformity *implies that each object in the model is an instance of one class defined in the metamodel, and that the model satisfies the static semantics of the metamodel. An* object *is composed of* fields*, each representing the object's values for one property of the corresponding class.*



(a) *Represented as an object diagram.*



(b) *Represented with concrete syntax.*

Figure 2.2: Example of Petri net model.

Figure 2.2 shows an example of a Petri net model conforming to the metamodel shown in Figure 2.1. More precisely, Figure 2.2a shows the object diagram depicting all the objects of the model and their relationships, whereas 2.2b shows a concrete syntax representation of the model using the usual Petri net notation. The model is composed of one instance of the Net class, four instances of the Place class, and two instances of the Transition class. Each object has a set of fields based on the properties of its class. For instance, the transition $t1$ has a field `name` containing the string value `"t1"`, a field `input` containing a value reference to the Place $p1$, and a field `output` containing a reference value to the Place $p3$.

2.1.3 Model Transformations

While a model by itself can be used as a relevant description of a system, and can be analyzed for static inconsistencies or defects, many situations require to change or create a model in an automated way. This is done through *model transformations*, which are central operations in MDE for numerous purposes [141], such as the refactoring of a model, the generation of a new model based on an existing one, or model slicing [15]. Model transformations have been widely studied as first class artifacts [44, 48, 95] and can be defined using many paradigms, such as declarative programming (*e.g.,* ATL [91], VIATRA [42]), imperative programming (*e.g.,* Xtend/EMF, Kermeta [88]) or triple

graph grammars (*e.g.,* [140]). They are also fundamental regarding the definition of semantics of DSMLs, *e.g.,* to define operational semantics [31] (see Section 2.3).

Among others, a model transformation is composed of *transformation rules*. A rule defines a subset of the changes performed by a model transformation on the target model. Depending on the paradigm used to define the execution transformation, a rule can take different forms:

- Using declarative model transformation languages, such as VIATRA [42, 129, 130] or ATL [91], a model transformation is declared as a set of rules, each rule begin composed of a source pattern and a target pattern. The source pattern identifies a subset of the source model, while the target pattern defines how it is transformed. Executing such model transformation consists in a *pattern matching* loop that constantly tries to apply a transformation rule whose source pattern matches a part of the model.
- Using imperative model transformation languages, such as Kermeta [88] or xMOF [114], a model transformation is defined as a sequence of statements organized in different transformation rules called *operations*. One of these operations is the *entry point* of the model transformation, and is the one that is called to start the transformation. An operation may call other operations, thereby defining the order in which transformation rules are applied. Note that only operations that may change the target model can be considered as transformation rules.
- In the context of the GEMOC project, Combemale et al. [36, 35] propose to define the *model of concurrency* (*i.e.,* a set of logical clocks and constraints between them) of an xDSML in a dedicated model. Then, an external component called a *solver* relies on this model to schedule the next model transformation rule (*e.g.,* a Kermeta operation) to apply on the executed model. With this approach, a rule is here again a Kermeta operation, but the order in which they are called depends this time on the model of concurrency and the solver.

In addition, there are multiple sorts of model transformations [44]. If both the source models and target models conform to the same metamodel, they are said *endogenous*. Otherwise, they are said *exogenous*. Finally, if a model transformation directly changes a source models without creating new target models, they are said to be *in-place*.

Definition 3 *A* model transformation *is an operation that applies on one or more source models and transforms them into one or more target models. In addition:*

- *A model transformation is composed of* transformation rules*, each responsible for a subset of the transformation, i.e., executing the model transformation implies the application of a sequence of rules.*
- *A model transformation is said* endogenous *if target models conform to the same metamodel as source models, and is said* exogenous *in the case of different*

```
1  public def void fire(Transition transition) {
2
3      // Checking if input places are enabled
4      if (transition.input.forAll [place | place.tokens > 0]) {
5
6          // Removing a token from each input place
7          for (Place input : transition.input)
8              input.tokens = input.tokens - 1
9
10         // Adding a token to each output place
11         for (Place output : transition.output)
12             output.tokens = output.tokens + 1
13     }
14 }
```

Listing 2.1: In-place model transformation rule that fires a Petri net transition, written in Kermeta.

> *metamodels.*
>
> – *A model transformation is said* in-place *if models are effectively being modified directly; such transformation is endogenous.*

Listing 2.1 shows an example of transformation rule called `fire` that is part of an in-place model transformation defined using Kermeta. This transformation rule takes as an input a specific Transition object called `transition` (line 1). It first checks if input Place objects of `transition` are all enabled (line 4), *i.e.,* if all its input Place objects have at least one token. If the condition is satisfied, it removes token from each input Place object (lines 7–8), and adds a token to each output one (lines 11–12). With an imperative language such as Kermeta, such operation must be explicitly called by some other operation, starting with the entry point operation. This can for organized using a design pattern such as *visitor* or *interpreter* [65].

Model Transformation Footprints Given a model and a model transformation, an observation can be made: it is likely that only a subset of the elements of the model are involved in the model transformation. Jeanneret et al. [87] defines such subset as the *model footprint* of the application of a model transformation to a model. More precisely, as shown in Figure 2.3, there are two sorts of model footprints: *dynamic* and *static* model footprints.

First, a *dynamic model footprint* is derived from the actual execution of a model transformation on a considered model. Concretely, all elements of the model that are accessed during the execution are part of the footprint. Among others methods, such information can be obtained by directly observing the modification made to the model (*e.g.,* using EMF notifications), or by instrumenting the model transformation, or by producing a detailed execution trace of the transformation (as shown in Figure 2.3).

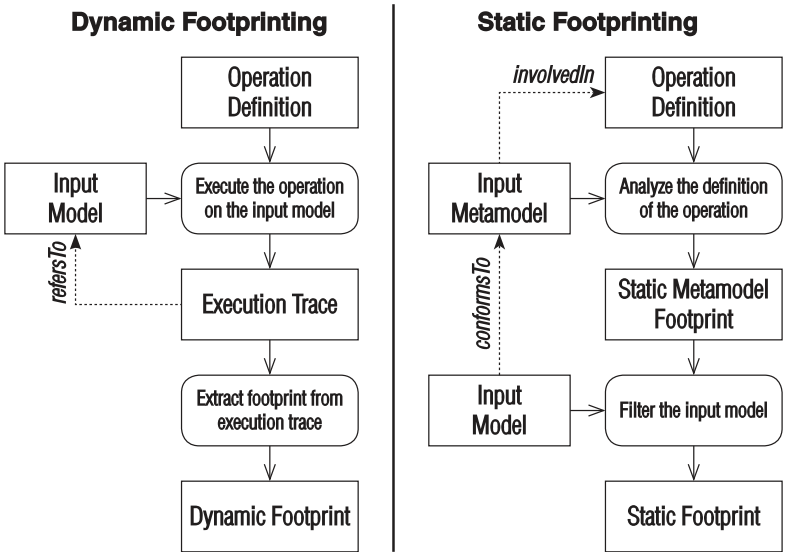


Figure 2.3: Dynamic and static model footprinting, from [87].

Definition 4 *(derived from [87])*
The dynamic model footprint *of the application of a model transformation on a model is the set of elements of the model that are manipulated by the model transformation. Thus, the dynamic model footprint contains all elements that affect the outcome of the transformation, as long as this transformation is deterministic and does not use data other than those contained in the input model.*

For instance, the dynamic model footprint of the `fire` rule shown in Listing 2.1 applied on the Transition object $t1$ from Figure 2.2 is:

$$\{\texttt{t1}, \texttt{t1.input}, \texttt{t1.outputp1}, \texttt{p1.tokens}, \texttt{p3}, \texttt{p3.tokens}\}.$$

Second, a *static model footprint* is derived by first analyzing the definition of a model transformation, then the considered model to transform. As shown in Figure 2.3, analyzing the definition of the model transformation yields the *static metamodel footprint* of the model transformation, which is the set of elements of the metamodel that are required by the model transformation.

Definition 5 *(derived from [87]) The* static metamodel footprint *of a model transformation is the set of metamodel elements involved in the definition of the transformation.*

For instance, the static metamodel footprint of the `fire` rule shown in Listing 2.1 is:

$$\{\text{Transition}, \texttt{input}, \texttt{output}, \text{Place}, \texttt{tokens}\}.$$

From there, the static model footprint can be obtained by filtering the input model using the static metamodel footprint. This filtering consists in keeping only instances of elements of the static metamodel footprint.

> **Definition 6** *(derived from [87]) The* static model footprint *of the application of a model transformation on a model is the set of elements of the model that are instances of elements found in the static metamodel footprint of the model transformation.*

For instance, the static model footprint of the `fire` rule shown in Listing 2.1 applied on the Transition object $t1$ from Figure 2.2 is:

{t1, t1.input, t1.output, p1, p1.tokens, p2, p2.tokens, p3, p3.tokens, p4, p4.tokens}.

2.2 Object and Model Duplication

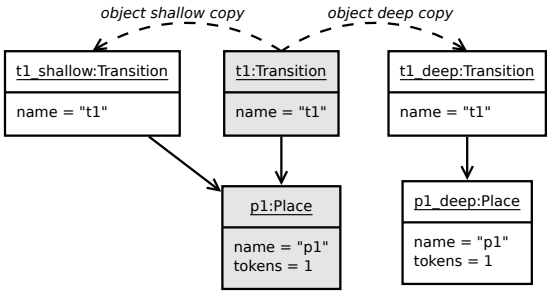
In this section, we first present object duplication and its usages, then we discuss its multiple definitions and names, and finally we present model duplication.

2.2.1 Object Duplication

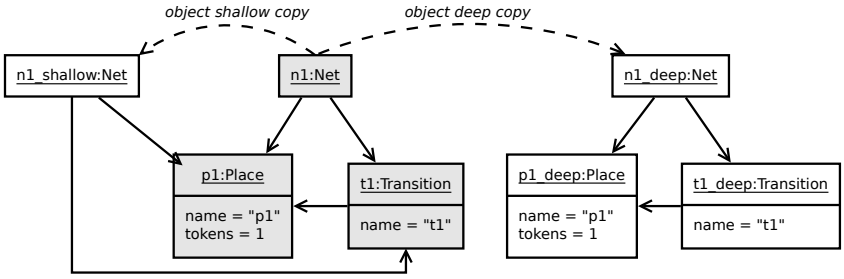
Duplicating an object is the action of creating a new and independent object identical to an existing one. It is an important activity that have been widely studied in the object-oriented programming community [74, 104, 50, 73, 105, 71]. Operators to duplicate objects can be found in many popular programming languages, such as the `clone` method of Java or the `dup` method of Ruby. Such operators are used in many situations: to avoid data sharing and aliasing problems, to duplicate data in a distributed environment, or simply to ease the construction of a complex object graph using an existing one. Most of these operators consist in taking a single input object as a parameter, and returning a single new object seemingly identical to the input one.

Smalltalk [71] was one of the first languages to provide object duplication operators, namely `shallowCopy` and `deepCopy`. Both operators gave their names to the two main ways to duplicate an object still today: *shallow copying* and *deep copying*. *Shallow copying* consists in creating an output object instance of the same class as the input object, and copying the exact same field values of the input object inside the output object. Primitive values are hence copied along with reference values, which means that a shallow copy will have references pointing to the same objects as its origin. *Deep copying*, on the other hand, does not copy reference values. Instead, all objects transitively referenced by the input object are copied. Hence, a complete object graph is copied, and there is no data sharing between the input and output objects. While being more costly in memory, it is hence safer than shallow copying.

Examples Figure 2.4 shows two examples using small subsets of the Petri net model from Figure 2.2: first when copying a Transition object, second when copying a Net object. Firstly, in Figure 2.4a, we start with the t1 and p1 objects at the center, in gray.



(a) *Copying a* Transition *object.*



(b) *Copying a* Net *object.*

Figure 2.4: Illustrations of *deep* and *shallow* copying. In each case, grey elements depict the original object graph (*i.e.,* before copying anything).

We perform a shallow copy of `t1` to the left which creates a second Transition object named `t1_shallow`. This new object has both the same `name` value, which is `"t1"`, and the same `input` value, which is a reference to `p1`. Then, we perform a deep copy of `t1` to the right, and we obtain a third Transition object called `t1_deep`, again with the same `name` value `"t1"`. This time, however, a new Place object was created, called `p1_deep`, referenced by the `input` reference value of `t1_deep`.

Figure 2.4b shows very similar situations, but with one main difference: there is a link between two objects referenced by the Net object `n1` that we want to copy. Indeed, `t1` has a reference value to `p1`. While this has no impact on shallow copying, deep copying implies managing such situations and making sure that `t1_deep` has a reference value pointing to `p1_deep`, and not to `p1`. From an implementation point of view, this requires the storage of traceability links between the original object graph to its copy.

Vocabulary and definitions There is some heterogeneity among object duplication operators, both regarding their names and their definitions. Table 2.1 illustrates this situation for a selection of programming languages. The first two columns contain respectively the name of the language and of the duplication operator; column (a) states whether or not an output object is constructed; column (b) states whether or not the attribute fields of the input object are copied in the output one; column (c) states whether or not objects transitively referenced by the input object are copied along the output

Language	Operator	(a) creates object	(b) shallow	(c) deep
Smalltalk	`deepCopy`	✓	✓	✓
Smalltalk	`shallowCopy`	✓	✓	✗
Perl	`dclone`	✓	✓	✓
Java	`clone`	✓	✓	✗
Ruby	`clone`	✓	✓	✗
Ruby	`dup`	✓[a]	✓	✗
OCaml	`copy`	✓	✓	✗
Eiffel	`copy`	✗	✓	✗
Eiffel	`deep`	✗	✓	✓
Eiffel	`twin` (former `clone`)	✓	✓	✗
Eiffel	`deep_twin`	✓	✓	✓

[a] Ruby modules are not copied, see `http://ruby-doc.org/core-2.1.5/Object.html`.

Table 2.1: Comparison of a selection of object duplication operators.

object. First, we observe many different names for object duplication operators: copy, clone, dup(licate), twin. Second, there is no seeming consistency between names and definitions. For instance, *copying* in Smalltalk can be either shallow or deep, while it is always shallow in Java and OCaml. Even more intricate is Eiffel, whose `copy` operator does not even create an output object; instead, it copies the values of the input object inside some *existing* object, which is a second input to the operation. Eiffel has however a `twin` operator, formely called `clone`, that performs a proper shallow copy by constructing an output object.

In this thesis, despite this variety of names and definitions, we mostly use the term *object copying* for shallow object duplication (*i.e.,* creating a *single* new object given a *single* input object), and *model cloning* for model duplication (*i.e.,* , creating a *single* new model given a *single* input model, presented thereafter).

2.2.2 Model Duplication

Model duplication, or model cloning, is the action of creating a new and independent model identical to an existing one — this implies that cloning a model has only one possible output, which is the identical clone.

Many model-driven engineering activities rely on model cloning. Several works rely on evolutionary computation to optimize a model with respect to a given objective [95, 70]. Optimization in this case, consists in generating model variants through cloning, mutation and crossover and selecting the most fitted. Likewise, design space exploration [136] is the exploration of design alternatives before an implementation, which requires the generation of the complete design space (*i.e.,* set of variations, which are models). Last but not least, execution trace management can rely on model cloning to capture

the states of the executed model (*e.g.,* [99]), which is discussed more thoroughly in Section 2.4.

Intricacies of Model Cloning Implementing model cloning can be hazardous for multiple reasons. First, it is not possible to reuse object copying operators. Indeed, cloning a model implies copying *multiple* objects at once. Hence, while object deep copying is a good candidate for model cloning, the chosen input must be an object that transitively references all objects of the considered model, and such object may not exist.

Second, a model clone must be independent from the original model, which means that modifying one must have no impact on the other. This may not be trivial to guarantee depending on how the model and its clones are represented in memory. For instance, while *deep cloning* consists in duplicating the complete data in memory that represent the model (introduced as the *runtime representation* of a model in Chapter 4), *partial cloning* consists in copying only a chosen subset of the same data and to rely on data sharing to save memory. In such case, ensuring the independence of the clone requires either that the shared data will never change, or to simply forbid such changes.

Existing model cloning facilities There are few languages or toolboxes providing model cloning facilities. A possible reason is that, provided some adjustments and precautions, object deep copying can be used to clone models.

The Eclipse Modeling Framework (EMF) provides facilities to implement model cloning with the Java class EcoreUtil.Copier. It provides operations to copy runtime objects that constitute the model in memory. First, the `copy` method must be called on each runtime object to copy, which will both create a copy of the object, and of all the objects transitively contained in this object (*i.e.,* objects accessible through containment value references)[1]. Traceability links are kept between original objects and copied objects. To finish the cloning, the `copyReferences` method must be called, which will initialize all the reference values of the new objects based on the traceability links, *e.g.,* similarly to deep copying in Figure 2.4b. In a nutshell, depending on the amount of chosen runtime objects that are copied, EcoreUtil.Copier makes possible to implement either partial or deep model cloning. However, in the case of partial cloning, there is no mechanism to ensure the independence of the clone.

Another cloning operator is the `deepClone` operation of Kermeta from Jézéquel et al. [88]. It is similar to an object deep copy operation, since it takes as an input a single object, and copies transitively some referenced objects. However, it only performs a deep copy of *containment* reference values (*i.e.,* that imply *ownership*), and a shallow copy of normal reference values . The reason is that Kermeta was designed for a modeling context, which requires taking into account the static semantics of the considered metamodel, including containment references. Therefore, it can be considered as a partial cloning operator. In practice, it relies simply on the `copy` method of EcoreUtil.Copier, and likewise there is no mechanism to ensure the independence of the clone.

[1] This is similar to ownership-based copying, *e.g.,* as studied in [105, 50].

Finally, the Kevoree Modeling Framework (KMF) from Fouquet et al. [60, 61] is an alternative of the EMF tuned for the Models@Runtime paradigm [14]. KMF provides a partial model cloning operator, along with facilities to declare objects as being immutable. When called, the cloning operator will clone an input model while sharing its immutable runtime objects with its clones. A runtime object tagged as immutable cannot be changed, which ensures the independence of the clone. However, the cloning operator only considers a single input object, similarly to Kermeta.

2.3 Executable Metamodeling

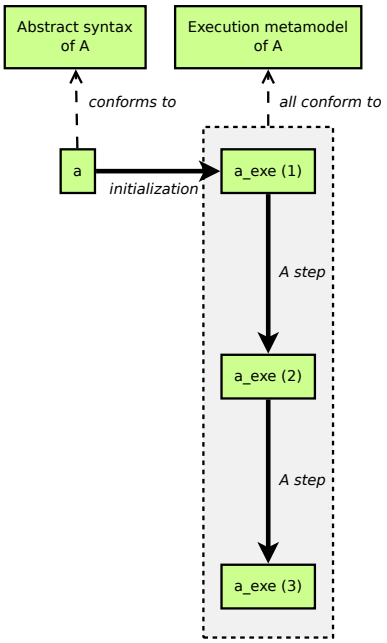
In the realm of modeling, while many models only represent structural aspects of systems, a large amount express behavioral aspects of the same systems. The idea behind *executable metamodeling* is simple: a model conforming to a DSML can express its intended behavior by being *executed*, which requires that the DSML provides *execution semantics*. Such semantics define what is an execution state, and how this execution state changes during an execution. We call *executable Domain-Specific Modeling Languages* (xDSMLs) such DSMLs that aim at supporting the execution of models, and we call *executable model* a model that conforms to an xDSML.

Model executability serves two main purposes. First, it enables the use of *dynamic verification and validation* (V&V) techniques, such as omniscient debugging [38] or semantic differencing [99], to ensure that an executable model is correct with regard to its intended behavior. Such techniques analyze the evolution of the execution state of a model over time, which is typically done using *execution traces*, as discussed in Section 2.4. Second, model executability gives the possibility to directly deploy an executable model to run on a production system. Research projects involving both academic and industrial partners such as TopCased [32, 41] or GEMOC [34] are good illustrations of the interest in providing executability to models.

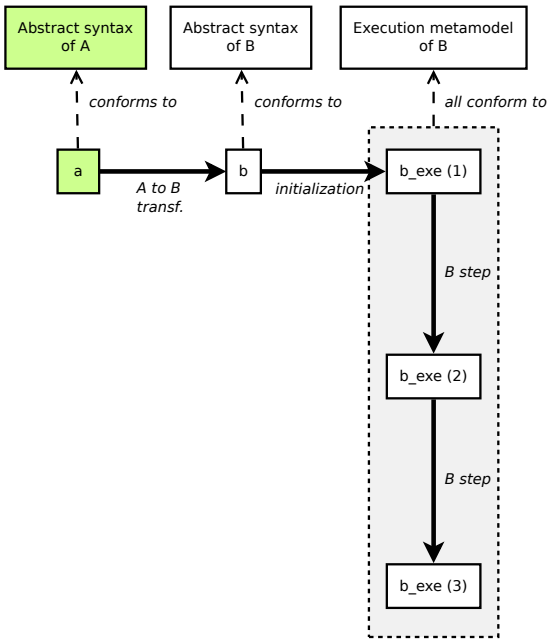
2.3.1 Operational vs. Translational Semantics

There are two general approaches to define execution semantics, namely *translational* and *operational* semantics. Figure 2.5 shows a comparison. In all subfigures, at the top left corner, a model a conforming to the abstract syntax of an xDSML A is shown. We highlight in green all elements that are related to the xDSML A, for which execution semantics must be defined. The execution then differs depending on the approach:

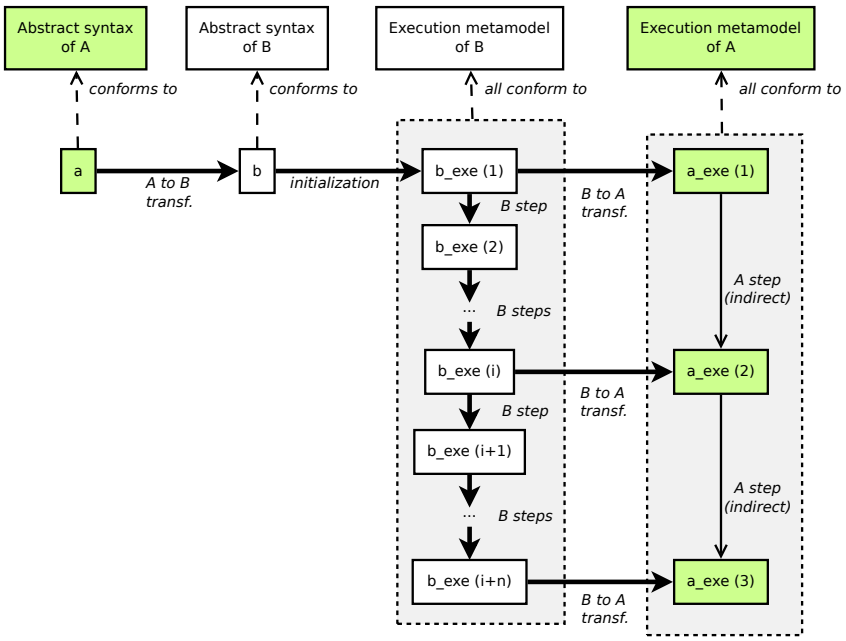
– *Operational semantics* [127, 92, 10] consist in an endogenous model transformation that changes the execution state of a. The execution state is defined through an extension of the abstract syntax called the *execution metamodel*. Figure 2.5a shows two steps performed during such semantics. First, the model a is initialized in a model a_{exe} that conforms to the execution metamodel of A. Then, the endogenous model transformation transforms a through a series of *steps*, each being the application of a *step rule* of the transformation.



(a) *Operational semantics.*



(b) *Translational semantics, using a target xDSML B.*



(c) *Translational semantics, using a target xDSML B and back-annotation.*

Figure 2.5: Translational and operational semantics for an xDSML *A*.

– *Translational semantics* [101, 63] consist in relying on the execution semantics of some target executable language to define the execution semantics of an xDSML. Figure 2.5b illustrates the process with a target xDSML B which was defined using operational semantics. First, a is translated in a model b that conforms to the abstract syntax of an xDSML B. Then, b is initialized into b_{exe} that conforms to the execution metamodel of B. Finally, the operational semantics of B are used to execute b_{exe}. While such semantics only require the definition of a translation from A to B, it makes the V&V of a more difficult. Indeed, the domain of B may have nothing in common with the domain of A, making difficult to interpret the execution from the perspective of A.

– To overcome the last mentioned issue, it is possible to augment the translational semantics with *back-annotation* [82], in order to translate back the results of the execution (*e.g.,* the execution states) in the source domain [31]. Figure 2.5c illustrates the augmented process. First, like the previous case, a is translated into b, which is initialized into b_{exe}, which is executed through a series of B steps. This time however, b_{exe} is translated back into a_{exe} each time an A step has been performed. Note that this means that a single A step can require multiple B steps, which is shown on the figure with "$\cdots$". Determining and detecting when to translate back to A is a non-trivial task. Nonetheless, this allows to observe the execution state of a_{exe} similarly to operational semantics, enabling runtime verification or the capture of an execution trace.

Regarding the implementation of xDSMLs, translational semantics would result in a *compiler* while operational semantics would result in an *interpreter*. Back-annotation results in a mechanism similar to *debug symbols* used by interactive debuggers (presented in Section 2.5) to visualize a current instruction or a stack from the perspective of a source model (*e.g.,* Java code) while a target model is being executed (*e.g.,* Java bytecode) [2].

In the remainder of thesis, we only consider operational semantics for the definition of the execution semantics of xDSMLs. More precisely, thereafter, the term xDSML only refers to xDSMLs defined using operational semantics. However, note that our work can be directly adapted to translational semantics as long as a back-annotation mechanism is provided.

2.3.2 Definition

Executable DSMLs have been widely studied under various names: dynamic metamodeling [55, 6], dynamic modeling languages [82], model execution [144, 151] or simply xDSMLs [37, 114]. Though they do not share terminology, all these approaches consider about the same design pattern to design an xDSML.

[2] Another related back-annotation activity is translating back results of an analysis performed on a target model (*e.g.,* analyses of native code [138] or counter-examples from a model checker [31, 82]) into results relevant for the source model.

Similarly to a regular DSML, the core element of an xDSML is the *abstract syntax*, which is the metamodel defining the domain of interest. In addition, providing executability requires the definition of *execution semantics*. As we explained in the previous section, we focus in this thesis on *operational semantics*. Such semantics include both the definition of the *execution state* [3] of an executed model, and of a model transformation that changes such state.

Definition 7 *An* xDSML *is defined by:*

- *An* abstract syntax*, that is a metamodel.*
- Operational semantics*, composed of:*
  - *An* execution metamodel*, that defines the execution state of executed models by extending the abstract syntax with new properties and classes using package merge, or any similar mechanism.*
  - *An* initialization transformation*, an exogenous model transformation that given a model conforming to the abstract syntax, returns a model conforming to the execution metamodel.*
  - *An* execution transformation*, an in-place model transformation that modifies a model conforming to the execution metamodel. The subset of transformation rules that are considered observable are called* step rules*.*

We explain and discuss the different parts of this definition in the following sections.

2.3.3 Execution State Definition

The first part of the operational semantics of an xDSML is the definition of the *execution state* of a model conforming to the xDSML. In theory, this can be accomplished using an arbitrarily complex data structure (*e.g.,* a stack, registers or a tape), which can be completely independent from the abstract syntax of the designed xDSML. However, in practice, the definition of an execution state is intuitively coupled with the abstract syntax. For instance, if an xDSML contains the concept of *variable*, it is very likely that an execution state contains the values of the different variables of a conforming model. In such case, it seems convenient to directly link the concept of *value* from the execution state to the concept of *variable* of the abstract syntax.

Following this idea, many existing approaches define the execution state of an xDSML by *extending* the abstract syntax with execution-only constructs:

- Hegedüs et al. [82, 84] extends the abstract syntax by defining additional classes in a *dynamic metamodel* that may contain references to classes from the abstract syntax. Bandener et al. [6] and Soden et al. [144] do the same within a *runtime metamodel*.

[3] also called *runtime data* or *dynamic data*

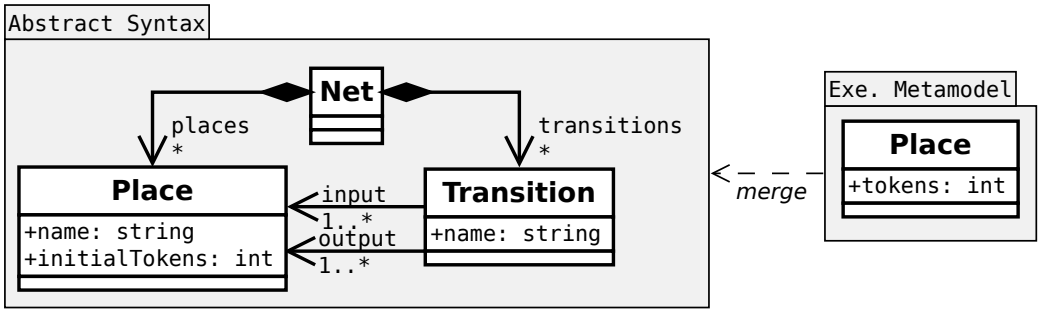


Figure 2.6: Abstract syntax and execution metamodel of the Petri Net xDSML.

```
1  @Aspect(className=Place)
2  class PlaceAspect {
3
4      /**
5      * Current amount of tokens in a Place object.
6      */
7      public int tokens;
8  }
```

Listing 2.2: Definition of the execution metamodel of Petri net through a Kermeta aspect.

- Mayerhofer et al. [114] proposes with the xMOF language to define *configuration classes* to extend the abstract syntax. A configuration class is a subclass of a class from the abstract syntax that introduces new properties specific to the execution state. Additional regular classes can be defined along these configuration classes to introduce execution-only concepts.

- Jézéquel et al. [88] provides similar facilities with the Kermeta language using *aspect weaving*. In particular, an aspect can be defined to extend a class of the abstract syntax with new properties specific to the execution state. Additional classes can also be defined for execution-only concepts.

In essence, all these approaches propose to define the execution state by adding new properties and/or new classes to the abstract syntax. We call *execution metamodel* the metamodel resulting from this extension. These approaches are very similar to an existing and well-known relationship between two metamodels called *package merge*. This relationship was introduced in the Unified Modeling Language (UML) [123], and is also part of the Meta-Object Facility (MOF) [120]. A merge relationship between two metamodels declares the intent of merging classes of one metamodel into the other. Simply put, the result of a merge is the set of all classes from both metamodels; if two classes have the same name, then they are combined in a class containing the properties from both originating classes. Package merge is conceptually very similar to the *inheritance* relationship between two classes, but as the metamodel level.

Figure 2.6 shows an example of package merge usage to define the execution state for a Petri net xDSML. At the left, the abstract syntax is a metamodel almost identical to the one from Figure 2.1, with one important difference: the `tokens` property of Place was renamed `initialTokens`. The reason for this change is to make explicit that this information does not represent the execution state of a Petri net, but simply the static initial marking before the execution. At the right, a metamodel called the *execution metamodel* has a *merge* relationship with the abstract syntax. The purpose of this new metamodel is to be an extended version of the abstract syntax, in which execution-only constructs are added. The merge means that the execution metamodel contains all the concepts of the abstract syntax (*i.e.,* Net, Place and Transition) while extending it with the new constructs it declares. Here, a single property `tokens` is declared in the existing Place class. This means that the Place class of the execution metamodel not only contains `name` and `initialTokens` that were "inherited" from the abstract syntax, but also `tokens` which defines the current amount of tokens of a Place during an execution.

Listing 2.2 shows how the exact same extension is done using Kermeta through the definition of an aspect for the Place class. Line 1 states that the aspect is for the Place class, and line 7 declares the `tokens` property.

2.3.4 Initialization Transformation

With the definition of the execution state, we obtain two distinct metamodels: the *abstract syntax* representing the domain of the xDSML, and the *execution metamodel* that extends the abstract syntax with new constructs representing the execution state. However, the model that we want to execute originally conforms to the abstract syntax, and not to the execution metamodel. This is true for all executable languages, including programming languages: a `.java` file doesn't contain a stack or a symbol table, but simply a set of Java instructions. Hence, it is necessary to initialize the execution by transforming the model to execute into a model conforming to the execution metamodel. We call such transformation an *initialization transformation*.

Figure 2.7 shows an example of initialization. At the top is depicted a model very similar to the one from 2.2, but with additional `tokens` fields to conform to the new abstract syntax. At the right, the two metamodels are shown along with the initialization transformation for Petri net. This function creates identical Net and Transition objects, and creates Place objects whose `tokens` field contains the same value as the `initialTokens` field. Note that for complex xDSMLs, an initialization transformation can have to create an arbitrarily complex set of data depending on the constructs introduced in the execution metamodel. Such initialization is specific to the xDSML and its semantics. At the bottom of the figure is shown the result of the initialization, which is a Petri net model conforming to the execution metamodel, with `tokens` fields initialized to the values of `initialTokens`.

In practice, such initialization can be partly handled generically. With Kermeta [88], any model can be generically loaded and transformed in a model conforming to the execution metamodel, with all execution-only fields set to default values. Likewise, xMOF [114] can generically transform each object of the input model in an instance of

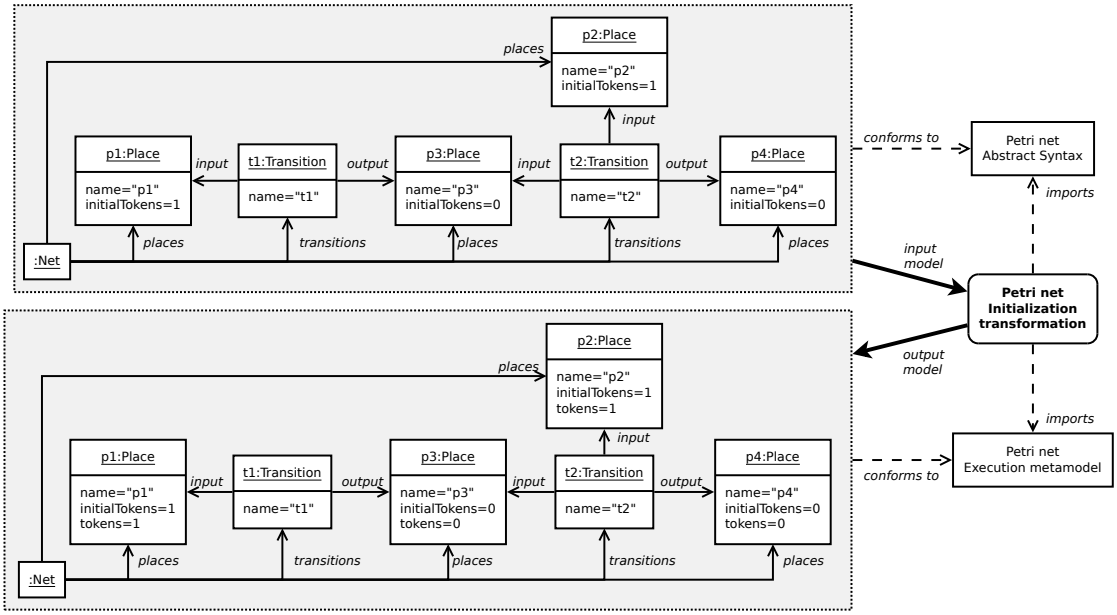


Figure 2.7: Illustration of an initialization transformation for Petri net.

corresponding configuration class with all execution-only fields set to default values. In both cases, from there, a simple xDSML-specific model transformation can be used to initialize the fields (*e.g.,* visiting all Place objects to set `tokens` to `initialTokens`).

2.3.5 Execution Transformation

We have seen how to define the execution state and the initialization transformation of an xDSML. The next step is the definition of how the execution state of a model changes over time, *i.e.,* what happens during an execution. This is accomplished by the definition of an endogenous transformation whose input and output is a model conforming to the execution metamodel. We call it the *execution transformation* [4] of the xDSML. To avoid having to duplicate most of the model for the execution, we consider this transformation to be *in-place* (*i.e.,* the executed model is directly modified). Besides, observing the modifications made to a single model is a common pattern when defining tools for xDSMLs (*e.g.,* graphical animation).

Because one of the purpose of xDSMLs is to analyze the behaviors of models, an important concern is to be able to follow the evolution of the execution state during the execution. However, by definition, it is only guaranteed that the executed model conforms to the execution metamodel *before* and *after* the transformation, and not *during* the execution transformation. Therefore, to be observable, such transformation must be specifically designed to preserve both consistency and conformity at specific instants of

[4] It can be observed that the metamodel static footprint (see Section 2.1.3) of an execution transformation should include all classes and properties introduced in the execution metamodel.

the execution. In this thesis, we consider that this is accomplished through *step rules*, which are designed rules of the execution transformation that guarantee both conformity and consistency before and after their application. These rules represent relevant changes in the model from the domain point of view; for instance, a step rule may express the firing of a Petri net transition. As a comparison, an example of non-step rule would be a simple adding of a single token to a Petri net place, since it is a small intermediate change that leads the model into an inconsistent state, as the resulting Petri net marking should never be observed.

We call *execution step* the application of a step rule. More precisely, some approaches draw a distinction between a *small step* [5] and a *big step* [6] [83, 38, 57, 40], the latter being composed of multiple execution steps.

> **Definition 8** *An* execution step *is the application of a step rule. An execution step that is not composed of other steps is called a* small step*, while an execution step composed of multiple steps is called a* big step*.*

Listing 2.3 shows the execution transformation for the Petri net xDSML using Kermeta aspects. It relies on the aspect defining the execution metamodel, previously shown in Listing 2.2. The first aspect (lines 1–21) defines two operations for the Transition class: `isEnabled` is a query to know if a transition is enabled, and `fire` is transformation rule already introduced in Listing 2.1 that fires a transition. The annotation `@Step` [7] defines that this operation is a step transformation rule, *i.e.,* we want to be able to observe the changes made by this operation (*e.g.,* in a trace, in a debugger, etc.). The second aspect (lines 23–36) defines one transformation rule called `run` for the Net class, also annotated with `@Step`. This operation calls `fire` while there are transitions that are enabled, and is used as the entry point of the overall entry point of the transformation. Therefore, a call to `run` yields a big step composed of small steps, each being the consequence of a call to `fire`.

2.3.6 Interacting with the Environment

Depending on its boundaries, its interfaces and its components, a system may interact with its *environment*, *i.e.,* to react to inputs and to produce outputs. Such systems are called *reactive systems* [79]. Consequently, either for analysis or production purposes, an executable model that represents a behavioral aspect of a reactive system should be interactive as well. Finite state machines [25] or state charts [78] are example of modeling languages designed to represent reactive systems.

From an xDSML point of view, this implies that an execution transformation may require or be influenced by *input data* during its application. Such data can be arbitrarily complex, from a boolean provided by the user to a complex model describing

[5] Also called *micro step* [83, 38].

[6] Also called *macro step* [83, 38], *combo-step* [57], or *compound step* [83].

[7] The annotation `@Step` was added to the Kermeta in the context of this thesis, as we explain later in Section 9.2 of Chapter 9.

```
@Aspect(className=Transition)
class TransitionAspect {

    def boolean isEnabled() {
        return _self.input.forall[place|place.tokens > 0]
    }

    @Step
    def void fire() {
        if (_self.isEnabled) {

            // Removes a token from each input place
            for (Place input : _self.input)
                input.tokens = input.tokens - 1

            // Adds a token to each output place
            for (Place output : _self.output)
                output.tokens = output.tokens + 1
        }
    }
}

@Aspect(className=Net)
class NetAspect {

    @Step
    def void run() {
        while (true) {
            val enabledTransition = _self.transitions.findFirst[t|t.isEnabled]
            if (enabledTransition != null)
                enabledTransition.fire
            else
                return
        }
    }
}
```

Listing 2.3: Execution transformation for the Petri net xDSML, written in Kermeta.

the environment state. Since a model transformation is an arbitrarily complex piece of executable software that manipulates models, handling external input can be accomplished by any existing input mechanism provided by model transformation languages (*e.g.,* `stdin`, socket, file, graphical interface, etc.). Interaction possibilities can also be reified into an interface to send events to the running execution. For instance, Combemale et al. [35] propose the definition of *domain-specific events* that can represent input from the environment. Lastly, regarding *output data*, it can be considered as part of the execution state of the model, provided that the latter is observable.

Nonetheless, in the scope of this thesis, we do not explicitly take into account the possible interactions of a model with its environment during its execution.

2.4 Model Execution Tracing

Model executability make possible to express behaviors with models, and therefore to verify that these behaviors are correct early in the design process using dynamic V&V approaches. However, while an executable model inherently expresses an *intended* behavior, dynamic V&V techniques need an *extended* representation of behavior over time. A most common representation of a model's behavior is the *execution trace* which is an artifact representing what happened during an execution.

A large proportion of dynamic V&V approaches use execution traces. Omniscient debugging [38, 107, 126] (see Section 2.6) consists in exploring previous states of a past or current execution, and relies on an execution trace to reconstruct previous states in the executed model. Semantic differencing [99, 112] aims at comparing models not only syntactically, but also semantically through execution traces comparison. Runtime verification [102] consists in checking whether or not an execution trace satisfies a temporal property, which can be done either online (*i.e.,* during an execution using a *monitor*) or offline Basin et al. [8] (*i.e.,* after an execution by analyzing a stored trace). Traces can also be manually manipulated to investigate the cause of a failure, using operators such as filter, slice, or merge to create relevant projections [126].

2.4.1 Execution Traces

Even though an execution trace is always a sequence containing information on the execution of a model, it appears that there is a large number of kinds of execution traces. In the context of state-based model checking, Baier et al. [5] defines an *execution* as an alternating sequence of states and actions, and a *trace* as a sequence of sets of valid atomic propositions — each set corresponding to a given state. This formal definition hence considers that a trace only contains a subset of the information that defines an execution. This is also what we observe in practice: some approaches capture all complete execution states reached by the model (*e.g.,* [99, 69, 85, 116]), other focus and the changes made to elements of the model (*e.g.,* [38]), and many are mostly concerned with events occuring during the execution (*e.g.,* [47, 113, 46, 120]).

In this chapter, we simply consider an execution trace to be a sequence containing information about the execution of a model, which include all aforementioned approaches.

Definition 9 *An* execution trace*, is a sequence containing relevant information about a particular execution over time. Such information may include:*

- execution states *reached during the execution;*
- changes *made to the execution state of the executed model (e.g., the change in a value of a field, or the creation of an object);*
- event occurrences*, including:*

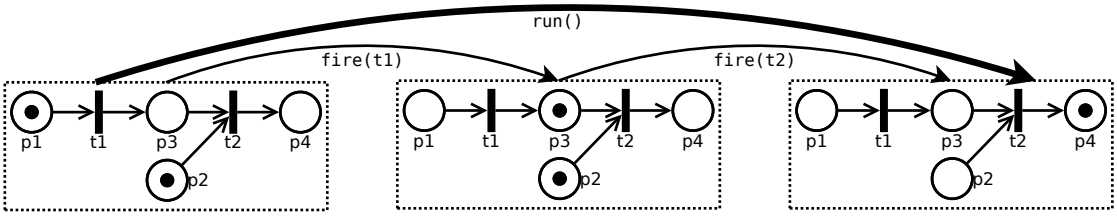


Figure 2.8: Example of Petri net execution trace represented using concrete syntax.

- input data *originating from the execution environment;*
- execution steps *that changed the execution state, which includes both* small steps *and* big steps.

Figure 2.8 shows an example of execution trace obtained by executing the Petri net model shown in Figure 2.2b using the operational semantics shown in Listing 2.3. At the bottom, three execution states are depicted using the concrete syntax representation of the xDSML. At the top, two small steps are recorded: first the application of `fire` on the transition $t1$, then on $t2$. Both are part of the big step that is the application of `run`. This execution trace gives us all the required information to understand and analyze this execution: we know how the marking of the Petri net evolved, and we know which transitions were fired and in which order.

Differences With Logging *Logging* consists in using print statements (*e.g.,* `printf` or `System.out.println`) in some executable program or model in order to understand, monitor, or analyse its behavior. Some approaches are dedicated to the analysis of log files, such as [156]. However, in the literature, the differences with tracing can be subtle, if not non-existant. Sauter et al. [135] distinguishes both terms in the following way: logging consists in printing messages, while tracing consists in capturing the events issued from specific constructs (*e.g.,* a method) in a systematic manner, including by logging the events. Following this idea, we consider that logging can be used to print any sorts of messages, which includes the possibility to print an execution trace as pure ASCII (*e.g.,* by following a trace format, see Section 2.4.3). Yet, a set of log messages doesn't necessarily constitute an execution trace.

Clone-based Execution Traces An execution trace containing all the states reached by an executed model can be obtained in a generic way by *cloning* the model after each execution step. We name such traces *clone-based execution traces*. This way of doing brings several advantages. First, the execution trace data structure is simple and appropriate for generic execution trace manipulations. Second, existing model transformations and queries specific to the xDSML can directly be applied on execution states stored in a clone-based execution trace. However, an important drawback is that each clone contains much more information than the execution state itself, *i.e.,* the elements only

defined by the abstract syntax of the xDSML. We present some clone-based execution trace data structures in Section 2.4.3.

2.4.2 Execution Trace Management

All the aforementioned dynamic V&V approaches need to be able to construct and manipulate execution traces. We name *execution trace management* the set of activities that includes:

1. *Acquiring and constructing* execution traces of model executions. This can be done through instrumentation of either a model (if its xDSML provides concepts to construct a trace, *e.g.,* `print`) or of the operational semantics of its xDSML.

2. *Manipulating* (or *processing*) execution traces, in order to analyze and understand them. This includes browsing, filtering or splitting an execution trace.

3. *Visualizing* (or *exploring*) execution traces, to be able to embrace and understand the important amount of data they contain. This includes the definition of efficient graphical representations for execution traces, which relies on trace manipulations (see previous item).

Below, we review some existing work and tools for execution trace management.

Execution Trace Exploration Tools Even though our work focuses on execution traces of models, there are many open or commercial tools to create, analyze and visualize execution traces of software programs. Vampir [8] is a popular tool-set to analyze execution traces from parallel applications. It includes VampirTrace, which is a program monitoring and execution trace acquisition tool. LTTng [9], STWorkbench [10] and Intel VTune Amplifier [11] provide facilities to analyze very "low-level" traces, with information specific to the operating system kernel or to the CPU architecture. TAU [142] [12] and Scalasca [67] [13] focus on execution traces of parallel applications. More recently, Trace Compass [14] from the Polarsys group aims at handling all kinds of program traces, though it focuses on concerns of software systems (memory/processor usage, network streams, etc).

Model Transformation Traceability *Model transformation traceability* is a research field that focuses on managing links between the source and the target models of a model transformation, for purposes of V&V or engineering. Such links are usually provided as a

[8] `http://www.vi-hps.org/projects/score-p/`
[9] `https://www.lttng.org/`
[10] `http://www.st.com/web/en/catalog/tools/PF250516`
[11] `https://software.intel.com/en-us/intel-vtune-amplifier-xe/`
[12] `https://www.cs.uoregon.edu/research/tau/home.php`
[13] `http://www.scalasca.org/`
[14] `https://www.polarsys.org/eclipse/trace-compass`

set (*e.g.,* ATL [90]) and not as a *sequence*, hence it is important not to confuse traceability with execution traces. Yet, the work of Falleri et al. [59] for Kermeta is an exception to this statement, since they propose to store traceability links as a sequence of actions performed by a model transformation. In such case, traceability links represent a way to store changes in the execution state within an execution trace.

Model Execution Trace Management In the realm of model execution, several approaches propose different kinds of execution trace management facilities.

The xMOF language [114] provides facilities to capture an execution trace from the application of the execution transformation of an xDSML. Yet, this is a trace of the transformation itself (*e.g.,* which xMOF activities from the execution transformation were called and in which order) rather than of the model being executed (*e.g.,* the sequence of transitions fired by a Petri net). Therefore, the purpose is more to analyze the operational semantics than the model itself, although it can be used to analyze the model if the modeler has a good understanding of the operational semantics.

Timesquare [45] is a model-based environment for the specification, analysis and verification of causal and temporal constraints defined using the CCSL language. While its purpose is not to execute model, it can be combined with other facilities to drive model executions [35]. Timesquare can capture execution traces obtained from executing CCSL models, both for visualization and verification purpose. Further work from Garces et al. [66] focused on reconciling Timesquare execution traces from different independent sources with synchronization instants.

Hegedüs et al. [83] propose a rather complete execution trace management approach based on the Viatra model transformation language. It consists in defining a domain-specific execution trace metamodel for an xDSML, and to use Viatra live transformation rules (*i.e.,* rules fired whenever models are changed) to construct traces. The approach also includes trace replay and a back-annotation mechanism to derive a domain trace from a formal analysis tool trace.

The TopCased toolkit [32] provides facilities to construct traces containing both external events (*i.e.,* originating from the environment) and internal events (*i.e.,* originating from the executed model). A trace that only contains external events defines a scenario, which can be used to drive an execution for testing or simulation. Traces must conform to a domain-specific trace metamodel specific to the xDSML, as developed in [37].

Some approaches focus especially on trace exploration. Maoz et al. [110, 111] give an approach to generate an execution trace of a scenario model (*e.g.,* a UML interaction diagram) according to the execution a system that should follow this scenario. The goal is to provide traces at the scenario model level for executions at the system level, and to explore these traces through a dedicated tool called *Tracer*. Another approach is the one of Aboussoror et al. [1], which relies on the creation of analytical abstraction models from execution traces for visualization purposes.

Finally, some approaches provide facilities to generate domain-specific execution trace metamodels, but without any trace management facilities to acquire the trace

of an execution or to visualize the trace. This includes the PromoBox framework [116], that provides facilities to generate a set of metamodels for a given input xDSML, including an execution trace metamodel, and the work of Gogolla et al. [69] and Hilken et al. [85] on filmstrip models. W e discuss such approaches more thoroughly in the next section.

2.4.3 A Look at Execution Trace Data Structures

To enable execution trace management, a most important requirement relies in the definition or choice of an *execution trace data structure*. Indeed, it defines the content of traces, and hence impacts both their construction and their manipulations. For instance, investigating race conditions of a multithreaded program requires information specific to parallelism. Likewise, storing only a list of execution steps may require the reconstruction of the reached execution states to perform an analysis based on the latter. In addition, a prominent requirement is the compatibility of the data structure with existing popular tools, such as the aforementioned ones.

Table 2.2a shows a comparison of some existing data structures for execution traces, and some approaches to design such structures. Figure 2.2b describes the content of each column. The table is split in four parts. The first contains execution trace data structures with specific concerns (*e.g.,* a specific xDSML). The second part contains generic trace formats that can be used for any xDSML. The third part contains so-called self-defining trace formats, which provide facilities to define custom types for elements of the trace. Finally, the last part does not contain data structures, but approaches to define execution trace metamodels. We review each part in the following paragraphs.

Structures with Specific Concerns Because originally execution traces were made to debug and understand programs conforming to GPLs, such as Java, C or C++, a large proportion of existing execution trace data structures are focused on concepts and concerns typically found in such languages. A most famous one is the Open Trace Format 2 (OTF2) [56], which is a format designed for execution traces of parallel software. OTF2 traces hence contain concepts such as “thread”, “lock”, “fork”, or “MPI” (Message Passing Interface). The format is understood by many trace-analysis tools, such as Vampir. Another example is the Compact Trace Format [77], which is a metamodel relying on ordered directed acyclic graph to compress call trees. It is designed for tracing object-oriented software, with concepts such as “class”, “method”, “object” or “thread”.

Some other execution trace data structures include also include platform concerns. For instance, KPTrace [146] was designed by ST Microelectronics for the STLinux system. Its scope is therefore the operating system, with concepts such as “system call”, “memory allocation” or “interrupt”. Another example is CUBE4 [67], which is is concerned with distributed systems, *e.g.,* a massively parallel software running in a data center. It includes concepts such as “topology”, “call path” or “system resources”.

Finally, an execution trace metamodel can be specific to an xDSML, and hence only defines execution traces for models conforming to this xDSML. This can imply a

Name	Type	Ev./St.	Concerns
Open Trace Format 2 [56]	ASCII format	Both[a]	Parallel software
MPI Trace Format [3]	Metamodel	Events	HPC [b]
Compact Trace Format [77]	Metamodel	Events	Software
KPTrace [146]	ASCII format	Events	Operating systems
CUBE4 [67]	Binary format	Both[c]	Distributed software
UML Testing Profile [75]	Metamodel	Events	Software (UML)
fUML [113]	Metamodel	Events	fUML
Scenario-Based Traces [111]	ASCII format	Events	Sequence charts
Timesquare [46]	Metamodel	Events	Time, Timesquare
Gen. Sem. Diff. [99]	Metamodel	Both	Generic
KMF Versioning [81]	Other	States	Generic
Pablo SDDF [4]	ASCII/Binary	Both	Self-defining
Pajé [139]	ASCII format	Both	Self-defining
SOC-Trace project [124]	Metamodel	Events	Self-defining
Common Trace Format [47]	ASCII/Binary	Events	Self-defining
TopCased [32, 41]	Approach	Events	Domain-specific
Hegedüs et al. [83]	Approach	Both	Domain-specific
Promobox [116]	Generative App.	Both	Domain-specific
Filmstrip models [69, 85]	Generative App.	Both	Domain-specific

(a) *Comparison table (columns description below).*

[a] External *snapshots* can be referenced from an OTF2 execution trace (*e.g.,* a Java heap dump)
[b] High Performance Computing
[c] *Measures* can be made at each operation call, which can be considered as a form of state

Name: name of the approach, format or author

Type: how the data structure is defined; one of the following:

- **ASCII format:** a textual syntax (*e.g.,* a grammar)
- **Binary format:** a binary syntax (*e.g.,* the IP packet format)
- **Metamodel:** a metamodel (including UML profiles)
- **Approach:** an approach to define an execution trace metamodel
- **Generative approach:** an generative approach to define an execution trace metamodel, *e.g.,* by deriving it from an input xDSML

Ev./St.: whether events and/or states are represented

Concerns: the concerns taken into account by the data structure, *e.g.,* the application domain(s) or the kind of information

(b) *Description of the columns of the comparison table.*

Table 2.2: Comparison of a selection of execution trace data structures

direct dependency from the execution trace metamodel to the abstract syntax or the execution metamodel of the xDSML. While this may appear as a limitation, the benefits of narrowing the scope of a language to a domain are well known [86, 153]. In the case of trace metamodels, scoping a trace metamodel to an xDSML means focusing on concepts of the xDSML itself, thereby providing proper expressiveness to capture information on conforming models. For instance, Mayerhofer et al. [113] defined an execution trace metamodel for fUML [122], an executable subset of UML. An fUML model consists of Activity objects, each being composed of ActivityNode objects. Consequently, the execution trace metamodel defines an fUML trace as a sequence of ActivityExecution objects, each of these executions being a sequence of ActivityNodeExecution objects. Most classes of the metamodel reference classes of the fUML abstract syntax; for instance, ActivityExecution has a reference to the corresponding Activity class of fUML. Another example is the execution trace metamodel of Timesquare [46], which defines execution traces of CCSL models. To that effect, it has references to the CCSL metamodel, and considers a distinction between *logical time* and *physical time* in order to represent both logical clocks values and chronometric timestamps from real-world sources.

In summary, each of these execution trace data structures is relevant for specific executable languages. Consequently, it is noteworthy that they are unlikely to be convenient to define the execution traces of a given arbitrary xDSML. For instance, a "system call" or an "fUML activity execution" are concepts that are not relevant when constructing an execution trace for a Petri net model.

Generic Data Structures To our knowledge, very few data structures are completely generic and independent from an xDSML or from specific usages. Langer et al. [99] proposed a generic semantic differencing approach, which relies on generic clone-based execution trace metamodel. It defines a Trace object as a sequence of State objects, each consisting of Object objects (*i.e.,* any objects from any model conforming to any metamodel). Such State contains a clone of the executed model. In addition, there is a Transition object in between two following State objects, labeled by an Event.

Another generic approach is the runtime model versioning feature of the Kevoree Modeling Framework (KMF) [60, 61], proposed by Hartmann et al. [81]. While being closer to a memento design pattern [65] than to a trace data structure, it provides facilities to manipulate the objects of a model not only in *space* (*e.g.,* navigate from one object to another), but also in *time*. For instance, it is possible to set an object to a specific version, or to reference a specific version of a object. In the end, all the states of the objects are captured during their lifetime, which can effectively capture an execution trace of the model. Multiple backends are available to store the elements, from a memory cache to a NoSQL database.

Self-defining Trace Formats An interesting sort of data structure for execution traces are so-called *self-defining trace formats*, or *meta-formats*. These formats define that a trace contains metadata describing the format of the trace itself. This can be compared with any language that make both possible the definition of new types (*e.g.,* a

Java class or a C struct) and the instantiation of such types. Thereby, an execution trace constructed using a self-defining trace format can be adapted to a specific usage or context through the definition of appropriate types as metadata. Examples of such trace formats include Pablo SDDF [4], Pajé [139] and the trace metamodel of the SOC-Trace project [124].

For embedded systems or operating systems tracing, a well known self-defining trace format is the Common Trace Format (CTF) [47]. A CTF trace is composed of an ASCII header written using a declarative language called the Trace Stream Description Language (TSDL), and of data in a binary format. Among other things, the header defines different kinds of *events*, each event kind having a set of fields, that can each be typed by a wide range of types. CTF specifies for all possible elements of a trace how they are represented in binary format. Because of the compact design of CTF execution traces, they can be constructed with little overhead and with little memory, hence making them very popular for tracing systems with limited resources.

However, because these formats are *meta*-formats, it means that each of them in fact defines a wide range of *potential* formats. Thus, given a self-defined trace, it is either difficult or impossible to analyze its arbitrarily complex content, which would require specific tooling. A good illustration of this situation is the following description that can be found on the homepage of the Trace Compass tool (introduced in Section 2.4.2):

> *"Trace Compass currently supports many trace formats natively (no third-party libraries needed), such as:*
>
> - *Common Trace Format (CTF), including but not limited to:*
>   - *Linux LTTng kernel traces*
>   - *Linux LTTng-UST userspace traces*
>   - *Linux Perf traces (using the out-of-tree patchset to convert to CTF)*
>   - *Bare metal traces"*

This description implies that Trace Compass does not support CTF in general, but only supports about four different formats defined using CTF. In other words, it is necessary to define specific tooling for each format defined using a meta-format.

Domain-Specific Trace Metamodel Definition Approaches Lastly, some approaches that we already presented in Section 2.4.2 propose frameworks or methodologies to *define domain-specific execution trace metamodels*. A domain-specific trace metamodel is specific to an xDSML, such as the trace metamodel for fUML [113] that we presented above. The idea is similar to self-defining trace formats (*i.e.,* to define a format relevant for a given tracing activity), with a number of technical differences. First, a model—and hence a trace model—rarely contains its metamodel, while a self-defined trace contains its format. Second, using the same language (*e.g.,* MOF) to define both the trace metamodel and the abstract syntax of the xDSML makes possible to define proper references from one metamodel to another, while self-defined trace formats are in

a different technological space than metamodels. And third, self-defined trace formats do not provide a methodology regarding how to define a format for a specific usage, *e.g.,* an xDSML.

In the context of the TopCased project [32, 41], Combemale et al. [37] propose the definition of a *trace management metamodel* specific to the model of computation of an xDSML. More precisely, they propose a simplified trace metamodel dedicated to discrete-events system modeling, which defines a Trace object as a sequence of RuntimeEvent objects. The RuntimeEvent class is abstract and must be manually inherited by all classes defining the events specific to the xDSML.

Hegedüs et al. [83] propose a generic execution trace metamodel that must be manually extended into a domain-specific trace metamodel using inheritance relationships. The provided generic trace metamodel defines a Trace object as a sequence of Step objects. A Step can either be a SimpleStep or a CompoundStep. A CompoundStep object is composed of multiple Step objects. There are multiple kinds of SimpleStep: a Snapshot is the new value of an element of the model; a Change contains both the old and the new value of an element; a Trigger is the event that triggered a state change. All step classes can be extended into domain-specific classes, *e.g.,* to define a specific sort of Change relevant for a given domain. Note that a SimpleStep doesn't match what we call a *small step*, since it represents a very fine-grained change (*e.g.,* a change in a field), whereas we name small step a relevant set of changes that leads to a consistent state. However, CompoundStep can match both small steps and big steps, since it can be composed of changes.

Few approaches propose the automatic generation of a domain-specific execution trace metamodel. The PromoBox framework [116] provides facilities to generate a set of metamodels for a given input xDSML, including an execution trace metamodel. More precisely, they provide a clone-based generic execution trace metamodel that is extended into a domain-specific metamodel by their generative approach. They define a Trace object as a sequence of State objects, each containing a set of OrderedElement objects. After generation, OrderedElement becomes the supertype of all classes of the execution metamodel of the xDSML, so that all objects of an executed model can be stored in the trace. Lastly, State objects are linked by Transition objects, each referencing the transformation rule that was applied for this state change. Gogolla et al. [69] propose a similar approach to generate so-called *filmstrip models*, which can be considered as domain-specific execution trace metamodels. A filmstrip model is composed of a Snapshot class, and by a generated class per class of the xDSML execution metamodel. Such generated class is identical to the xDSML class, with the addition of a `succ` and a `pref` references, so that it becomes possible to browse a trace according to the different versions of a specific instance of this class. However, a new object is created after a step even if it didn't change. Therefore, aside from the changes made to the classes, the obtained metamodels are similar to clone-based trace metamodels.

2.4.4 Navigation Paths

An execution trace is a sequence of information about an execution. Hence, to analyze the behavior of a system, processing an execution trace essentially consists in reading this information sequentially from the start. However, the processing task may only require to focus on the evolution of specific elements of the executed models, or on specific executions step. In this thesis, we call *navigation path* a facility to browse a subset of an execution trace. Such paths aim at avoiding to process a complete execution trace for better scalability in time (Ch#3). Among the data structures we presented, only a few provides alternate navigation paths in addition to the possibility to iterate over all the elements of the trace.

The Open Trace Format 2 [56] focuses on parallel software executions, therefore the format stores separately the information for each *thread* or *process* running in parallel. This provides a different navigation path for the events of a specific process.

As explained previously, the execution trace metamodel from TimeSquare [46] distinguishes *logical steps* from *physical steps*, physical steps being split in different *physical bases*. A physical base can for instance express the evolution of chronometric time according to some specific hardware clock, while logical steps express the overall evolution of the system. A single logical step is referenced by possibly multiple physical steps from different physical bases. This structure makes it possible to browse a trace according to a specific physical base, avoiding to enumerate all logical steps.

We presented above the approach from Gogolla et al. [69] to generate filmstrip models. Their structure makes possible to follow the evolution of a single object of a model through the provided references `succ` and a `pref`, which facilitates the analysis of specific elements. Yet, because exactly one object snapshot is stored per state of the entire model (*i.e.,* even if the object did not change), following such navigation path requires as many iterations as browsing the complete execution trace of the model.

Lastly, KMF runtime versioning [81] stores the versions of each object of a model separately, allowing to enumerate the states of a specific object of the executed model.

2.5 Interactive Debugging of Executable Models

Debugging was originally defined as the activity *"to remove a malfunction from a computer or an error from a routine"* [125]. More recently, Zeller [157] defines debugging as *"Relating a failure or an infection to a defect […] and subsequent fixing of the defect"*. Continuing, he defines a debugger as *"a tool to facilitate debugging"*. In other words, while the general goal of dynamic V&V is to check that the behavior of a system fulfills its intended purpose, debugging is more specifically concerned with both finding the cause of some identified unintended behavior (*i.e.,* a failure), and removing the defect responsible for this behavior using debuggers.

Finding the cause of a problem requires the analysis of the faulty behavior of a system, which can be accomplished using many kinds of dynamic V&V techniques. In

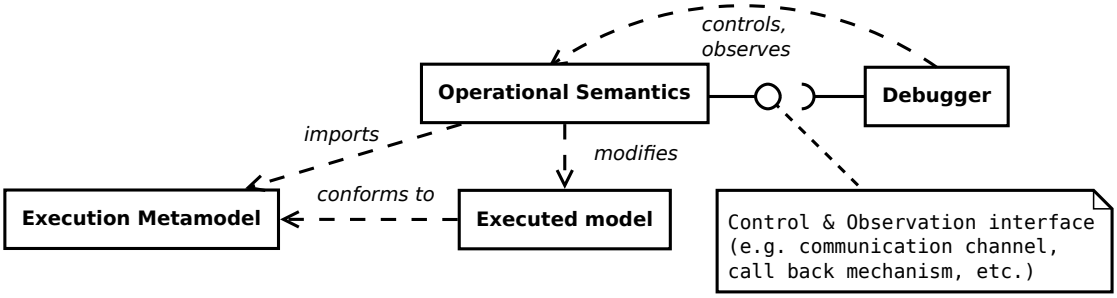


Figure 2.9: Typical architecture for interactive debugging.

particular, the term debugging is often associated with *interactive debugging*[15], which is a dynamic V&V approach that consists in both controlling and observing some execution with the help of an *interactive debugger*. Controlling an execution means being able to *pause* and *unpause* an execution in between execution steps, in order to observe the different execution states. Pausing is usually done either through the definition of *breakpoints*, which are conditions upon which the execution must pause (*e.g.,* reach a specific instruction), or through simply by *step-wise execution* (*i.e.,* pausing after a step has been performed). Applied to model execution, we define interactive debugging as follows:

> **Definition 10** Interactive debugging *of an executable model consists both in controlling the execution of the model through pausing and unpausing in between steps, and in observing the content of the current execution state of the model during pauses.*

In the remainder of this thesis, if not stated otherwise, *debugging* always refers to *interactive debugging*, and *debugger* always refers to *interactive debugger*.

2.5.1 Enabling Interactive Debugging

Debugging requires both to be able to control some execution, and to be able to observe the execution state of the model throughout this execution. Figure 2.9 shows the typical architecture used for interactive debugging. As presented in Section 2.3.2, the core element of the operational semantics of an xDSML is a model transformation that modifies the execution state of an executed model. While control over the execution could be provided by the model itself — given an xDSML expressive enough —, it is common for the operational semantics to expose an interface to pause and observe an execution. Indeed, this allows both to capitalize this interface for all models conforming to an xDSML, and to prevent from handling such concern in a model whose only purpose is to represent an aspect of a system. This interface can then be used to develop a *debugger*.

[15] Also called *breakpoint debugging* [128]

In the following paragraphs, we briefly discuss the two main scenarios using examples: first the case of process virtual machines (*i.e.,* software operational semantics), then the case of CPUs (*i.e.,* hardware operational semantics).

Process Virtual Machine A *process virtual machine* [143] is a software program that includes the operational semantics of a language. Except for languages that are compiled in native code (discussed thereafter), all languages rely directly or indirectly on process virtual machines for execution. This includes a wide range *intermediate representations* of GPLs, such as Java bytecode or Python bytecode, but also many xDSMLs (*e.g.,* [9, 98, 100]). As explained above, to make debugging possible, a virtual machine must expose an interface to control the executions it performs, such as a client-server architecture or callback mechanisms. Thereby, this interface can be used to implement a *debugger* with a relevant user interface (*e.g.,* a GUI). Such debugger is generally external to the operational semantics for better separation of concerns.

A well-known virtual machine is the Java Virtual Machine (JVM), which contains operational semantics for Java bytecode. The JVM provides the Java Platform Debugger Architecture (JPDA)[16], which is a set of interfaces to define both a unique *backend* and different *frontends* for the JVM. The backend is part of the JVM itself, and can both control an execution and inspect the current execution state. A frontend is an external component that can communicate with a backend (*e.g.,* using a network socket) to give orders (*e.g.,* set a breakpoint) or to ask for information (*e.g.,* the value of a variable). A JVM must run in debug mode to allow frontends to communicate with the backend. The most known JPDA frontend is the Java Debugger (*jdb*)[17], which can control a JVM to debug Java programs. Note that since a Java program is translated into executable bytecode, this is a case of *translational semantics*. As we have previously explained in Section 2.3.1, this requires some back-annotation mechanism to be able to follow the execution from the perspective of the source program. To that effect, *jdb* relies on *debug symbols* written during the compilation to bytecode. For instance, these symbols indicate which line of the Java program corresponds to a set of bytecode instructions, or which Java variable corresponds to a bytecode variable.

CPU A *central processing unit (CPU)* is a piece of hardware that includes the operational semantics of a specific kind of executable language called an *instruction set*. Controlling an execution performed by a CPU is a rather complex task. In a nutshell, for a x86 CPU, this is accomplished using several mechanisms. Data is directly read from registers and memory to observe the execution state. A specific flag of the CPU can be set to enable stepwise execution. Instructions of the debugged program can be replaced by *interrupts* to define breakpoints, which requires that the debugger registers itself as an interrupt handler of the debugged program to handle these breakpoints.

[16] `https://docs.oracle.com/javase/8/docs/technotes/guides/jpda/index.html`
[17] `https://docs.oracle.com/javase/8/docs/technotes/tools/unix/jdb.html`

In practice, all these facilities are abstracted by an API provided by the operating system (*e.g.,* `ptrace` on Linux). As an example, the GNU Debugger (GDB)[18] makes use of these mechanisms to debug compiled C or C++ programs. Here again, as for Java and *jdb*, debug symbols must be written in the compiled binary in order to follow the execution from the perspective of the source program.

2.5.2 Model Interactive Debugging

In this thesis, we are mostly concerned with the debugging of models conforming to xDSMLs, and thus on how to provide debuggers to xDSMLs. We present thereafter some existing interactive debugging approaches for xDSMLs: domain-specific debuggers, generic debuggers, and approaches to define domain-specific debuggers. The presented approaches are considering operational semantics if not indicated otherwise. Lastly, we briefly discuss the use of models in *model-based debugging* of systems.

Domain-Specific Debuggers Many approaches provide debuggers that are domain-specific, *i.e.,* specific to an xDSML. For instance, in the last decade, a large amount of work has been done to provide debugging for several parts of the UML [133, 96, 49, 40, 64]. We present below some recent xDSML-specific approaches.

Krasnogolowy et al. [98] manually mapped GPL debugging concepts (*e.g.,* step, instruction, variable, stack, scope) to a *story diagram* xDSML, and proposed a debugger following this mapping. In addition to breakpoints and step-wise execution, the resulting debugger provides advanced facilities such as, control flow visualization, variable modification, remote debugging and omniscient debugging (discussed in Section 2.6).

Mierlo et al. [117] defined a debugger for the *Parallel DEVS* xDSML, which is an extension to *DEVS*, a formalism for modeling complex dynamic systems using a discrete-event abstraction. They developed a specific interpreter using Statechart models, in which they defined debugging-specific operations such as pausing, breakpoints, and state manipulation. The resulting debugger is integrated within the AToMPM environment, which provides both visualization and animation of the model being executed.

Mayerhofer et al. [113] extended the standard fUML operational semantics in order to support debugging of fUML models. This includes the definition of a control API to pause or execute single steps, and an observer pattern to follow model changes. They validate their extension through the implementation of a debugger, which provides facilities such as breakpoints and stepwise execution.

In a very similar fashion, Laurent et al. [100] also extended the standard fUML operational semantics in order to support debugging of fUML models. They observed that the standard fUML operational semantics define an execution as a single execution step, with no intermediate steps nor facilities to stop or observe executions states. Hence, they proposed an extension to fUML operational semantics to make debugging possible, which includes the definition of a *controller* that centrally manages all model modifica-

[18] `https://www.gnu.org/software/gdb/`

tions as steps. This controller is extended to implement a debugger, with facilities such as breakpoints, stepwise execution and back stepping (see Section 2.6).

Generic Debuggers A model transformation is defined using a language that can manipulate models, such as Java (using the EMF), Kermeta [88], or ATL [91]. Since executing a model is the application of a model transformation, a first idea for model debugging would be the use of the existing debuggers of model transformation languages, such as the debugger provided by ATL. However, doing so would make possible to pause the execution in the middle of an execution step of the considered xDSML, which contradicts Definition 10. Moreover, the visualized execution state would be the one of the model transformation, and not of the executed model.

Combemale et al. [32] propose a model simulator in the TopCased toolkit. This simulator can execute models, and provides a GUI for interactive simulation that can be considered as a debugger. The execution can be paused in between steps, and visualization of the execution state is provided by the graphical editor used to edit models. While the presented approach is generic, the presented prototype is specific to an ad-hoc simulation engine for UML state machines.

Ráth et al. [130] propose an approach based on the Viatra language to execute and debug models conforming to xDSMLs. The execution can be paused in between steps, and the model can be edited on-the-fly during pauses (similarly to "hot code replace" proposed by some GPLs debuggers). An even more advanced feature is the possibility to add new transformation rules to the semantics during a pause. Visualization of the execution state is provided by the graphical editor used to edit models.

Bandener et al. [6] propose a tool called the *Dynamic Meta Modeling (DMM) Player*, which can drive the execution of the transformation rules that comprise the operational semantics of an xDSML. A debugger is provided as part of the tool on top of the execution engine responsible for executing the model transformation. Visualization of the execution state is provided by the graphical editor used to edit models, in which the concrete syntax representation of the model is constantly updated during the execution. The authors consider that only a subset of the transformation rules should be considered as *visual steps* that update the concrete syntax representation of the models. Therefore, they provide to the language designer a way to specify which rules are visual steps. When debugging, the execution can be paused before or after the application of any transformation rule, hence in between execution steps. In addition, *watchpoints* can be defined to pause the execution after a specific value change in the model.

Domain-Specific Debugger Definition Approaches Several approaches have been proposed regarding how to define a domain-specific debugger for an xDSML.

Wu et al. [155] propose a generative approach for grammar-based xDSMLs with translational semantics whose target language is already supported by a debugger (*e.g.,* a GPL such as Java). The approach requires traceability links between the executed model and the target model. Debugging components are generated to implement the debugging interface of the Eclipse IDE. Using these traceability links, debugging actions

at the xDSML level (*e.g.,* step forward) are translated into orders for the target language debugger (*e.g.,* set a breakpoint and continue). Similarly to what we already explained with Figure 2.5c, it is necessary to provide a mapping between the definition of a step of the xDSML and the one of the target language, in order to perform the right amount of steps in the target language for a given step of the xDSML. Likewise, a mapping between the execution state definition of the target language to the one of the xDSML is necessary to update the execution state of the executed model.

Lindeman et al. [109] present a generative approach for grammar-based xDSMLs targeting both translational and operational semantics. A language called the *debugger specification language* is used to specify when should debugging events be sent during the execution of the model. Such specification is used to automatically instrument the executable model with elements that send such events to an external component. This requires the xDSML to be expressive enough to make such event sending possible. At runtime, these events are handled to pause the execution when required.

More recently, Chiş et al. [29, 30] proposed the Moldable Debugger framework for developing domain-specific debuggers. The authors claim that generative approaches can only generate debuggers with generic debugging facilities (*e.g.,* step, step into, stack visualization, etc.), while *domain-specific facilities* should be defined for the application domain of the xDSML. They provide a framework to develop *domain-specific extensions*, each being composed of a set of *domain-specific debugging operations* and a *domain-specific debugging* view. One example is a domain-specific extension for PetitParser, an xDSML for parsing source code. The domain-specific views include dynamic representations of the produced structure and of the stream obtained from the input file (*i.e.,* the parsing progress). The domain-specific debugging operations include stepping until the stream position changes or reaches a specific position (*e.g.,* stepping until a specific line is being parsed).

Model-Based Debugging of Systems A dynamic V&V approach called *model-based diagnosis* [131, 132] in concerned with the verification of concrete systems through the use of models that represent them. More precisely, observations made of a running system are analyzed and compared with the expected behavior derived from the models. Applied to both non-interactive and interactive debugging, *model-based debugging* of systems [147, 154] consists in using models to more efficiently find the cause of a failure of a concrete system. Since we are concerned in this thesis with *early* V&V using executable models, model-based debugging is out the scope of our work.

2.6 Model Omniscient Debugging

In an empirical study of debugging stories, Eisenstadt [53] discovered that bugs are difficult to track down mostly because of the large temporal or spatial gap between the cause and the actual symptom of a bug. However, as Pothier et al. [128] disclaim: *"Unfortunately, most [interactive] debuggers provide very limited assistance for temporal navigation, so programmers frequently have to resort to mental simulation of program*

execution." Indeed, interactive debuggers have the following limitation: if a modeler notices a faulty behavior during a debugging session, he needs to restart the execution from the beginning to give a second look to the state of interest. The main reason is that, except for bidirectional model transformations [43, 94] (*e.g.,* triple graph grammars [140]), a model transformation cannot be trivially undone. Hence, restarting a virtual machine and executing the model a second time to revisit the state of interest can be costly in time. In addition, if the operational semantics are non-deterministic, then the initial faulty behavior might not show up at all.

To cope with this limitation, an interesting and convenient dynamic V&V approach that can be used is *omniscient debugging*[19] [103, 107, 128, 54]. From a modeler point of view, the idea is simple: in addition to being able to explore a series of execution states by going *forward* (*i.e.,* regular interactive debugging), additional facilities are provided to revisit states by going *backwards*. In other words, omniscient debugging makes possible to "go back in time" during a debugging session. The technique was inspired by several dynamic V&V approaches allowing to analyze execution states of a specific execution, such as log analysis [156], runtime verification [102] or record-and-replay [72] (*i.e.,* to reexecute a program in a deterministic way using a record of all its interactions with its execution environment).

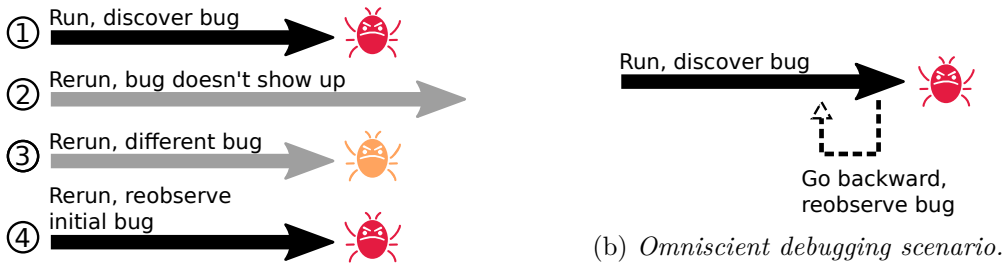
2.6.1 Omniscient Debugging Definition

Through a synthesis of aforementioned work [103, 107, 54, 126], we define omniscient debugging as follows

> **Definition 11** Omniscient debugging *is an extension of interactive debugging adding facilities to step* backwards *in the execution, i.e., to revisit previous execution states. This can include setting a* breakpoint *in the past and "executing backwards" until this breakpoint is reached, or simply* jumping *to a chosen past execution state.*

Figure 2.10 schematizes the differences between regular interactive debugging and omniscient debugging for re-observing a failure in a non-deterministic situation. Using interactive debugging (Figure 2.10a), a series of reruns must be done, which is commonly known as *cyclic debugging*. (1) The first run yields the first encounter with the failure. Now the modeler hypothetically wants to re-observe the failure to better understand its cause. (2) If there is a source of non-determinism due to the operational semantics (*e.g.,* declarative model transformation with different source patterns valid at the same time) or due to the execution environment, the initially observed bug might always not occur during reruns. Hence, multiple reruns may be required. (3) For the same reason, other bugs might occur during these reruns. The modeler may confuse them with the initial one. (4) Eventually, the initial bug occurs again and can be observed a second time. By contrast, using omniscient debugging (Figure 2.10b), a single attempt has to be

[19] Also called time-travel debugging [126], back-in-time debugging [107], reverse debugging [54], bidirectional debugging [18], or backtracking [2].



(a) *Cyclic debugging scenario using interactive debugging.*

(b) *Omniscient debugging scenario.*

Figure 2.10: Comparison of interactive debugging with omniscient debugging for re-observing a failure with non-determinism. Inspired by [54].

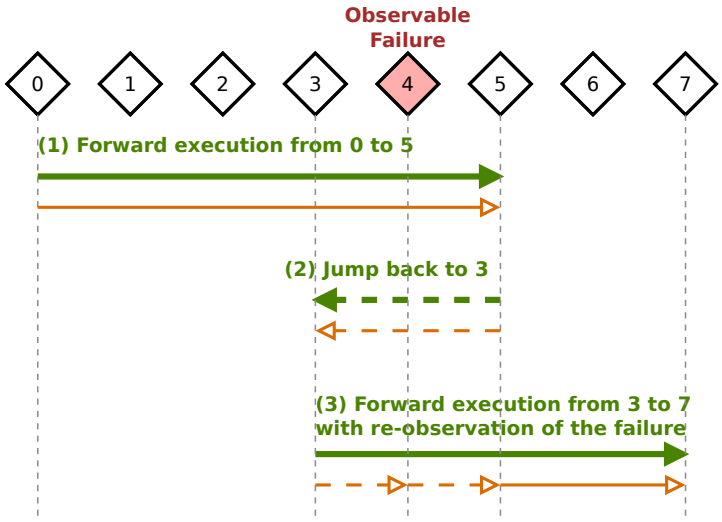
made: once the failure is observed, the modeler can go back in a previous state instantly to re-observe it, and if needed can then continue the execution of the model.

Some user studies have shown the superiority of omniscient debugging to find the cause of defects, as compared to regular interactive debugging. For instance, Lewis [103] showed that while a bug was found in over an hour by the original programmer using "conventional tools", all subjects of the study were able to identify the source of the problem within fifteen minutes.

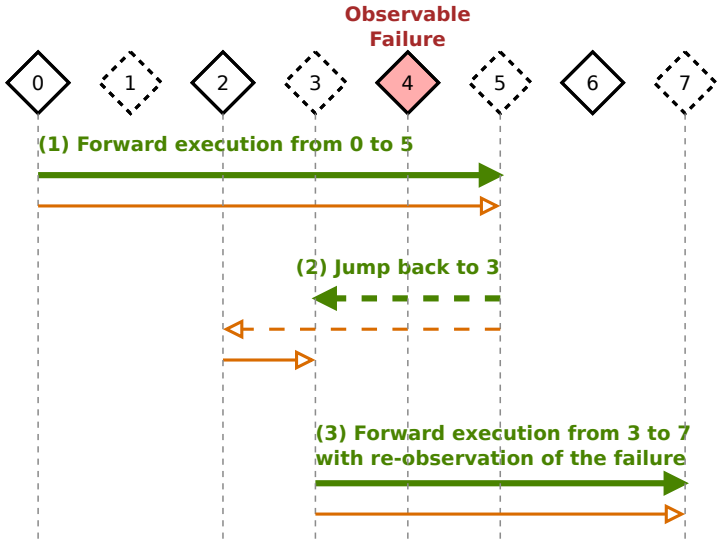
2.6.2 Omniscient Debugging Methods

Omniscient debugging can be accomplished in multiple ways. Engblom [54] reviewed existing techniques and established two main categories: *trace-based* and *reconstruction-based*. Figure 2.11 compares them with the following scenario. (1) Starting from a state `0`, the user steps forward until `5` is reached. (2) Because he observed a failure in the previous state `4`, he uses the debugger to jump back into the state `3`. (3) Finally, the user steps forward again, re-observes the failure, and decides to continue the execution until `7` is reached. For each action, what the user wants is shown using a thick green arrow, and what the debugger actually does is shown below using thin orange arrows. We present the two main methods of omniscient debugging thereafter using this example.

Trace-based omniscient debugging consists in recording in an execution trace all the necessary information to go into previous states. Therefore, this approach is independent from the operational semantics to step backwards, though it requires large execution traces. Such traces can for instance contain a list containing execution states, or reversible atomic changes made by each execution step. Figure 2.11a shows a scenario with a trace that contains all execution states reached by the executed model. For the first action (1), the debugger simply relies on the operational semantics to go forward. For the second action (2), the debugger reads the corresponding state stored in the trace, and injects it into the executed model. For the third action (3), to ensure an identical and deterministic replay, the debugger restores twice in a row a state from the trace



(a) *Trace-based omniscient debugging scenario, with a trace containing all states.*



(b) *Reconstruction-based omniscient debugging scenario, with a trace containing half of the states.*

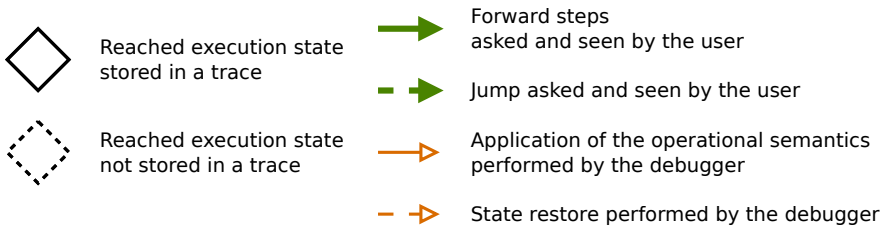


Figure 2.11: Comparison of omniscient debugging approaches. Inspired by [54].

until 5 is reached (which at this point is the last state stored in the trace), then relies on the operational semantics again.

Reconstruction-based omniscient debugging also consists in recording an execution trace, but only containing *partial* information that is not sufficient to directly go back to a previous state. This includes a selection of execution states called *checkpoints*, and information to reexecute the operational semantics deterministically[20] (*e.g.,* input from the environment, similarly to *record-and-replay* approaches). From there, going backwards is accomplished by first jumping to a checkpoint that happened before the target state, and then using the operational semantics deterministically to go forward until the target state is reached. Figure 2.11a shows an example with the same scenario. There is no difference with for the first action, which is simply the execution of the operational semantics. For the second action (2), the debugger first restores the closest checkpoint before the target state, which is 2, then uses the operational semantics to perform one step, thereby reaching the target state. For the third action (3), the debugger uses the operational semantics again until 7 is reached.

In this thesis, we focus on the capture of complete execution traces with both all execution states and all execution steps. Consequently, we only consider trace-based omniscient debugging.

2.6.3 Omniscient Debugging for xDSMLs

In the last decades, a lot of work has been done to provide omniscient debuggers for GPLs, such as for C/C++ [18], Java [103, 68, 128] or Smalltalk [107]. A recent example is the work of Barr et al. [7] on the TARDIS debugger which provides reconstruction-based omniscient debugging for C#. They claim to be *"the first affordable time-traveling debugger for managed languages"*, with a slowdown of only 14% when executing and recording information for omniscient debugging.

While most research on omniscient debugging is being done for GPLs, little work has been done to provide omniscient debugging for xDSMLs. In the following paragraphs, we present some domain-specific omniscient debuggers for xDSMLs, and we discuss generic omniscient debugging for xDSMLs.

Domain-specific Omniscient Debuggers As we already mentioned in Section 2.5.2, Krasnogolowy et al. [98] proposed a debugger for a *story diagram* xDSML. This debugger provides *back-stepping* by creating an execution trace containing all the changes made to the execution state, hence making possible to undo these changes. It can therefore be considered as a trace-based omniscient debugger. Interestingly, the execution state

[20] Note that cyclic debugging with an interactive debugger (as shown in Figure 2.10a) can be considered as a weak form of reconstruction-based omniscient debugging with only one checkpoint (the initial state), and with no deterministic replay.

of the execution transformation itself is also reset when stepping backward, which is in theory only necessary for a reconstruction-based omniscient debugger.

Also mentionned in Section 2.5.2, Laurent et al. [100] presented a debugger for the fUML xDSML, which required extending the standard operational semantics of fUML in order to be able to perform execution steps one by one. Among many features, they provide the possibility to roll-back the execution by relying on an execution trace containing the previous positions and contents of all the fUML tokens. It can therefore also be considered as a trace-based omniscient debugger. For better memory-efficiency, they only store new values when there are changes in the model.

We reviewed some trace management approaches in Section 2.4.2, such as the work of Maoz et al. [110, 111] on exploration of execution traces of scenario models. Since such approaches allow to explore previous states of an executed model, they are very similar to omniscient debugging. Yet, this accomplished offline and not during an execution, therefore this cannot technically be considered as interactive or omniscient debugging.

Generic Omniscient Debuggers for xDSMLs Unlike interactive debugging, few approaches aim at providing omniscient debugging to any xDSML. [38] propose omniscient debugging facilities for the cloud-based modeling solution AToMPM, in order to step both forward and backward in model transformations executed in an AToMPM runtime. AToMPM supports two model transformation languages, namely T-Core [149] and MoTif [148]. However, similarly to what we discussed in Section 2.5.2 with interactive debuggers of model transformation languages, using such an omniscient debugger would make possible to pause the execution in the middle of an execution step of the xDSML, which contradicts Definition 10. Also, the visualized state would be the one of the execution transformation instead of the one of the executed model.

Hegedüs et al. [83] propose a execution trace management approach for xDSMLs. In addition to an extensible execution trace metamodel (presented in Section 2.4.2), the approach includes model transformation rules to *replay* execution traces obtained from previous executions or from counter-examples generated of a model-checker. While being able to step forward and backward according an execution trace is very similar to omniscient debugging, trace replay is only offline and it is not possible to step backwards during a model execution.

Part II

Contributions

Chapter 3

Foreword to the Contributions

In this thesis, we are concerned with the management of execution traces of executable models. In other words, we aim at answering this question: given an arbitrary xDSML (*e.g.,* Petri nets), how to represent and manipulate traces of its conforming models, while taking into account a number of challenges? We introduced these challenges in Chapter 1 as *usability* (Ch#1), *scalability in memory* (Ch#2), and *scalability in time* (Ch#3).

This short chapter is an introduction to the contributions presented in two following chapters, Chapter 4 and Chapter 5, which both aim at meeting the aforementioned challenges, but for two distinct uses: *generic* and *domain-specific* trace manipulations. In Section 3.1, we explain these uses and we synthesize the corresponding state of the art subset presented in Chapter 2. Then in Section 3.2, given these observations, we explain the reasoning and the scope of both our contributions.

3.1 Observations

In the following, we present observations first regarding the differences between generic and domain-specific trace manipulations, then regarding the state of the art regarding execution trace data structures.

3.1.1 Generic vs. Domain-Specific Trace Manipulations

We make the following observation: execution trace manipulations can either be *generic* (*e.g.,* comparing the number of different states or the amount of steps, visualizing the values of all mutable properties of a state), or *domain-specific* (*e.g.,* determining how many tokens traversed a Petri net place). In the former case, manipulations are simple and the structure or content of the trace has little influence on the complexity of the analysis task. Moreover, they only have to be defined once, and can then be used for any xDSML. However, in the latter case, manipulations handle domain-specific data that can be arbitrarily complex depending on the considered xDSML. Hence, in such cases,

defining the right analysis can be error-prone and difficult, and accessing to relevant execution data of the domain becomes a critical requirement. In other words, the kind of manipulation is an important factor regarding the usability of execution trace data structures (Ch#1), and it is likely for a structure to only be adapted to a specific kind of manipulation.

3.1.2 Limitations of Existing Execution Trace Data Structures

In the previous chapter, more specifically in Section 2.4.3, we have seen different data structures to represent traces of executable models. As a synthesis, we make the following observation: a large number of existing execution trace data structures are specific to a selection of concerns, such as parallel software [56], operating systems [146], or xDSMLs [113]. Hence, each of these execution trace data structures is relevant for specific xDSMLs, and are consequently unlikely to be convenient to define the execution traces of a given arbitrary xDSML. For instance, a "system call" (from [146]) or an "fUML activity execution" (from [113]) are concepts that are not relevant when constructing an execution trace for a Petri net model, which hinders usability (Ch#1).

A first possible solution relies in generic execution trace data structures, such as [99] or [81]. These solutions allow the capture and the manipulation of traces for any kind of execution. In particular, clone-based execution traces can be captured using a simple metamodel [99]. In this case, each execution state is stored in the form of a clone. Among other advantages, existing model transformations and queries specific to the xDSML can directly be applied on execution states stored in a clone-based execution trace. Moreover, generic trace manipulations can be defined for such metamodels using reflexivity. However, domain-specific trace manipulations are not facilitated, since relevant concepts of the execution (*i.e.,* defined in the execution metamodel) are not directly accessible.

A second possible solution relies the definition of an ad-hoc execution trace data structure that is appropriate for the considered xDSML. This can be accomplished using *self-defined trace formats* (or *meta-formats*) [47, 4] allowing to both define the possible content of a trace and the trace itself in the same model. Additionally, some approaches provide a base structure along with some guidelines or examples for the definition of the execution trace data structure of an xDSML [83, 32]. Yet, there are two main problems with these approaches. First, a trace data structure specific to an xDSML requires the development of dedicated tooling, which is expensive. Second, even with appropriate meta-formats or approaches, manually defining a trace data structure is likely to be a difficult task [93], and only few approaches propose the automatic generation of a trace metamodel [116].

In parallel, another limitation of most execution trace data structures relies in the lack of alternate navigation paths to process execution traces, which is only possible with very few structures [56, 61]. Yet, being able to browse an execution trace efficiently by focusing on specific elements it contains appears as an interesting way to improve scalability in time (Ch#3).

3.2 Overview of the Contributions

In summary, we observe that both generic and domain-specific execution trace manipulations must be taken into account for any xDSML. Yet, it appears that there is no silver bullet to achieve this goal: some data structures are more appropriate for specific tasks than others, including regarding generic or domain-specific execution trace manipulations. Both categories of manipulations are independent cases that must be taken into account, and challenges stated in Chapter 1 are of importance for both categories. Consequently, we propose the following two contributions.

For generic trace manipulations, we propose in Chapter 4 *a scalable and generic approach to construct clone-based execution trace*s. Our technique relies on data sharing among runtime representations of model clones to save memory. Beyond generic execution traces, this approach is an improvement of model cloning in general, which can hence benefit other fields such as design space exploration [136] or evolutionary computation [70]. Chapter 4 is self-contained: it contains both the contribution itself, and its evaluation which relies on a custom cloning benchmarking tool based on randomly generated metamodels and models. Although the evaluation doesn't directly focus on execution traces and dynamic V&V, it is application-neutral and therefore also relevant for other applications (*e.g.,* design space exploration [136]). This work led to a publication in the proceedings of the MODELS'14 conference [20].

For domain specific trace manipulations, we propose in Chapter 5 *a generative meta-approach to construct a domain-specific execution trace metamodel of an xDSML.* Thereby, concepts from the domain of the xDSML are explicitly available to construct and manipulate execution traces, hence facilitating the definition of domain-specific manipulations. The generation being completely automatic, appropriate tooling can be generated along the trace metamodel. Because the envisioned direction of this thesis was the definition of trace management facilities specific to an xDSML, this research direction was more thoroughly studied than clone-based execution traces. Hence, contrary to Chapter 4, Chapter 5 only contains the contribution itself, while the evaluation was made through two applications to dynamic V&V, namely semantic differencing and omniscient debugging. We refer to Chapter 6 for an introduction and an overview of these applications. This work led to a publication in the proceedings of the ECMFA'15 conference [23].

Chapter 4

Scalable Armies of Model Clones through Data Sharing

In this chapter, we present our first contribution, which is an approach for scalable model cloning through data sharing [20]. In Section 4.1, we introduce the context of our contribution and our proposal. In Section 4.2, we motivate our problem by presenting a list of requirements for cloning operators, and explaining the intuition of our idea regarding existing cloning techniques. Section 4.3 defines what we call model cloning and what are runtime representations of models.

Then, Section 4.4 presents the main contribution of this chapter: a new approach for efficient model cloning through data sharing. Section 4.5 describes our evaluation, which was done using a custom benchmarking tool suite that relies on random metamodel and model generation. Finally, Section 4.6 concludes on the observed gain regarding memory consumption.

4.1 Introduction

When executing a model using an in-place model transformation, an execution trace can be captured by *cloning* the model after each execution step. Such *clone-based execution trace* contains all the reached execution states as a sequence of clones, which provides good usability for generic trace manipulations (Ch#1). Moreover, existing model transformations and queries specific to the xDSML can directly be applied on execution states stored in a clone-based execution trace.

Technically, cloning a model consists in obtaining a new and independent model identical to the original one. This operation can be implemented using the EcoreUtil.Copier class of the Eclipse Modeling Framework (EMF) [145], which consists in first creating a copy of the runtime representation of a model (*i.e.,* the set of Java objects that represent the model) and then resolving all the references between these objects. Such an implementation is also known as *deep cloning*. This implementation is effective to produce valid, independent clones. However it has very poor memory performances for opera-

tions that require manipulating large quantities of clones, such as genetic algorithms [95] or design space exploration [136]. Most importantly, in the context of this thesis, this directly opposes the need for scalability in memory when capturing traces (Ch#2).

We address the performance limitations of current cloning operations by leveraging the following observation: given a metamodel and an operation defined for this metamodel, the operation usually changes elements conforming to only a subset of this metamodel. That means that it is possible to identify the *footprint* of the write accesses of these operations on a metamodel. This footprint is the set of *mutable* parts of the metamodel, *i.e.,* elements that can be modified by an operation. We call it the *mutable subset* of a metamodel. The counterpart of these elements, the *immutable* elements, are definitively stated at the creation of objects. For instance, the immutable elements of an xDSML are defined by its abstract syntax, and the mutable subset by the set of properties introduced by the execution metamodel. Our intuition is the following: knowing the immutable elements, data could be shared between the runtime representation of a given model and its clones, saving memory when generating the clone.

In this chapter, we propose a new *model cloning algorithm*, which implements different strategies to *share immutable data between clones*. This contribution relies on a specific runtime representation of the model and its clones in order to share the data and still provide an interface that supports the manipulation of the clones independently from each other. We articulate our proposal around the following questions:

- Considering that we know which parts of a metamodel are mutable, how can we avoid duplicating immutable runtime data among cloned models?

- Can it effectively save some memory at runtime when creating a high number of clones as compared to EMF cloning implementation ?

Our goal is both to give a solution that can be implemented in various existing execution environments, and to provide concrete evidence of the efficiency of such an approach on a widely used tool set: the Eclipse Modeling Framework (EMF) [145].

Our main contribution is a new approach for efficient model cloning. The idea is to determine which parts of a metamodel can be shared, and to rely on this information to share data between runtime representations of a model and its clones. We provide a generic algorithm that can be parameterized into three cloning operators (in addition to the reference *deep cloning* one): the first one only shares objects, the second only shares fields, and the third shares as much data as possible.

We evaluated our approach using a custom benchmarking tool suite that relies on random metamodel and model generation. Our dataset is made of a hundred randomly generated metamodels and models, and results show that our approach can save memory as soon as there are immutable properties in metamodels.

4.2 Cloning Requirements and Proposal

In this section we give requirements for cloning operators, and we explain how our idea is related to existing approaches

4.2.1 Requirements for Cloning

New activities have emerged in the model-driven engineering community in recent years, which all rely on the automatic production of large quantities of models and variations of models. A clone-based execution trace consists of a sequence of clones of the executed model, each created after an execution step. These clones are all variants of the initial executed model, with only the execution state changing. Several works rely on evolutionnary computation to optimize a model with respect to a given objective [95, 70]. Optimization in this case, consists in generating large quantities of model variants through cloning, mutation and crossover and selecting the most fitted. Design space exploration [136] is the exploration of design alternatives before an implementation, which requires the generation of the complete design space (*i.e.,* set of variations, which are models).

All these new MDE techniques produce *large* sets of models that originate from few models. From a model manipulation point of view, all these techniques require the ability to *clone*—possibly many times—an original model, and to query and modify the clones as models that conform to the same metamodel as the original. More precisely, we identify five requirements for model manipulation in these contexts. We state these requirements in the form of research question for the definition of new cloning operators:

RQ#4.1 Do the new operators reduce the memory footprint of clones, compared to deep cloning?

RQ#4.2 Can a clone be manipulated with the same efficiency as the original model?

RQ#4.3 Can a clone be manipulated using the same generated API as the original model?

RQ#4.4 Can a clone be manipulated using the reflective layer (*e.g.,* as stated in the MOF Reflection package)?

RQ#4.5 Is it impossible to compromise the independence of a clone, *e.g.,* to modify data shared between the runtime representations of a model and its clones?

Our work defines novel cloning operators that reduce the memory footprint of clones, while trying to comply with the aforementioned requirements.

4.2.2 Existing Cloning Approaches and Intuition

As we already explained in Section 2.2, object copying has existed since the beginning of object-oriented programming languages [71] with the *deep* and *shallow* copy operators.

While the second operator cannot take a whole model into account and is thus not of interest, the first is at the basis of deep model cloning. Concerning models, the EMF provides a class named EcoreUtil.Copier with operations for copying sets of objects, which can be used to implement either a deep or a partial model cloning operator. Yet, as stated previously, such deep cloning operator is not memory efficient (RQ#4.1), and the partial cloning one offers no guarantees regarding clone independence (RQ#4.5). Not surprisingly, the same observations can be made for the `deepClone` operation of the Kermeta language [88], since it is directly based on EcoreUtil.Copier. Finally, the partial cloning operator of the Kevoree Modeling Framework (KMF) from Fouquet et al. [60, 61] does comply with the requirements stated above. Independence of clones is ensured thanks to the possibility to tag which parts of a model are immutable (RQ#4.5), and sharing immutable Java objects among runtime representations allows memory savings (RQ#4.1). However, it has some limitations. First, the input of this operator is a single root object, and not a set of objects (*i.e.,* a model). Second, each model must be *manually* decorated with immutable tags, which hinders usability. Third, data sharing is only done at the object level.

In terms of memory management, *copy-on-write* (a.k.a. lazy copy) is a widespread way to reduce memory consumption. The idea is the following: when a copy is made, nothing is concretely copied in memory and a link to the original element is created. At this point, both elements are identical, and accordingly reading the copy would in fact read the origin directly. But when writing operations are made on the copy, modified elements are effectively copied so that the copy keeps its own state and appears like a regular and independent element. Applied to model cloning, the runtime object configuration of a clone obtained using this technique would eventually only contain written mutable elements of the original model, which meets our need to reduce memory footprint (RQ#4.1). However, it adds a considerable amount of control flow at runtime in order to detect when copies must be done, and such copies can happen unpredictably depending on the manipulations; this contradicts the need for efficient clones (RQ#4.2). More importantly, depending on the programming language used, this technique can be very difficult to implement; for instance, Java is pass-by-value, making it impossible to dynamically change the value of a variable from a different context (*i.e.,* updating all references to an object that was just effectively copied), which is required to dynamically copy a model progressively and transparently.

Our intuition is that while deep cloning is easy to implement but memory expensive, and copy-on-write is memory-efficient but complicated and with manipulation overhead, it is possible to provide safe partial cloning operators *in between* these two extremes. Similarly to the way copy-on-write discovers *dynamically* which parts of a model are mutable when copying written elements, our idea is to *statically* determine which elements have to be copied at runtime. Such elements are opposed to the ones that can be referenced by both the original runtime representation and its clone. We present an approach based on this idea in the next sections.

4.3 On Model Cloning

The purpose of this section is to clarify what we mean by the runtime representation of a model and to precisely define what we call a clone in this work.

4.3.1 Mutable Subset of a Metamodel

We already defined what is a *metamodel* (see Definition 1 page 12) and what is a *model* (see Definition 2 page 13). Likewise, we defined the notion of *metamodel footprint* (see Definition 5 page 17), which is the set of elements of a metamodel that are used by a model operation. Inspired by this notion, our idea for this work is to focus on a subset of the metamodel footprint only concerned by *modifications* at the model level.

During its lifecycle, a model can change in two possible ways: by creating/deleting objects or by changing values of fields of objects. We designate as *mutable* elements both the elements of a model that may change over time and the metamodel parts that define these elements. Our approach considers a given object configuration in order to produce a clone, and is thus not influenced by the creation of deletion of objects.

> **Definition 12** *A property of a class of a metamodel is* mutable *if, in each object instance of this class, the value of the field corresponding to this property can change after the construction of the object. The set of all mutable properties of a metamodel is called the* mutable subset *of a metamodel. Dually, a property is said to be* immutable *if its value cannot change after construction.*

Fig. 4.1 shows a metamodel named `AB` that is composed of two classes `A` and `B`. `A` has two attributes `i` and `j` and one reference `b`. `j` is mutable as specified by `(mut)`. `B` has a single attribute `x`. Below the metamodel, a model `abb` conforms to `AB` and is composed of one object instance of `A` and two objects instance of `B`.

In the case of an xDSML, the mutable subset of an execution metamodel is generally defined by the elements that it adds to the abstract syntax. Indeed, a Java program cannot change its instructions at runtime, and a Petri net model cannot create new transitions: such concepts are immutable. However, the current instruction of a Java program or the amount of tokens of a Petri net are mutable, since the whole purpose of an execution is to change them.

4.3.2 Implementation of Metamodels and Models

Specific execution environments are necessary to use metamodels and models. The Eclipse Modeling Framework (EMF) is one of the most popular. It generates Java interfaces and classes that implement a given metamodel, providing concrete mechanisms to create *runtime representations* of models that conform to the metamodel. We define a runtime representation as follows:

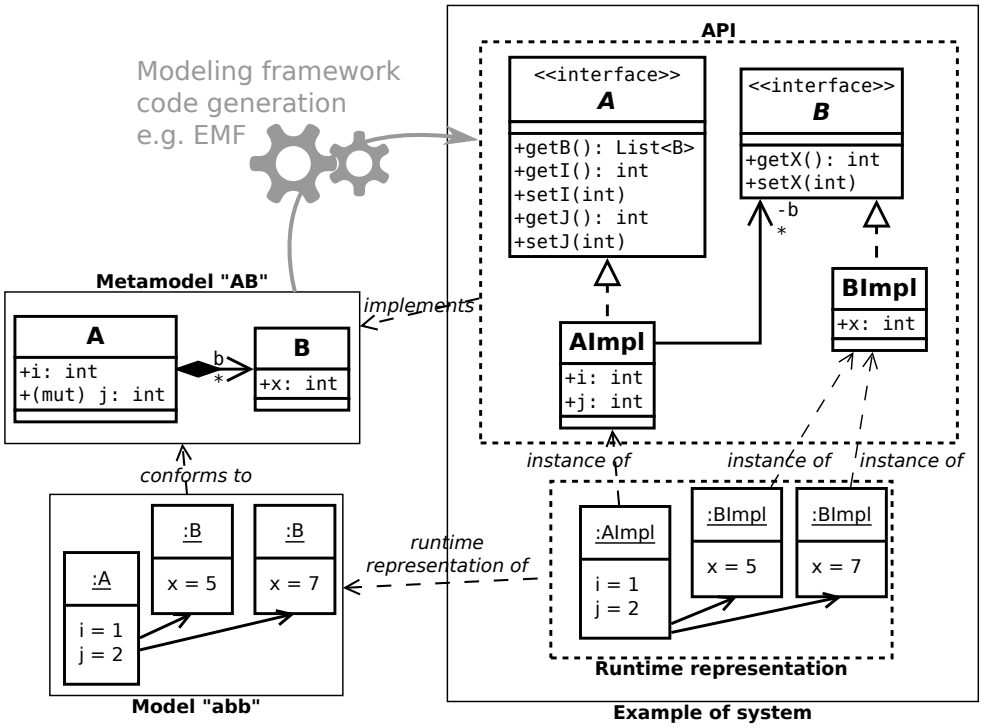


Figure 4.1: Example of modeling and EMF usage with a sample metamodel `AB` and a sample model `abb`.

> **Definition 13** *The* runtime representation *of a model is the set of runtime data that is sufficient to reflect the model data structure. It must be manipulated through an interface that is consistent with the corresponding metamodel.*

Top right of Fig. 4.1 shows the API (Java interfaces and classes) generated by the EMF generator. Interfaces A and B define services corresponding to the data structure of the original metamodel `AB`, while Java classes AImpl and BImpl implement these interfaces. These elements support the instantiation and manipulation of runtime representations—here, Java object configurations—of models that conform to the metamodel. The bottom right of the figure shows a runtime representation of m.

Note that a runtime representation that is eventually obtained using the EMF is structurally very similar to the original model: each object is represented by a Java object; each reference is represented by a Java reference; and each attribute is represented by a Java field. Yet runtime representations could theoretically take any form, as long as they are manipulated through an API that reflect the metamodel. One could imagine "empty" objects that get data from a centralized data storage component, or the use of a prototype-based programming language to create consistent runtime representations without defining classes.

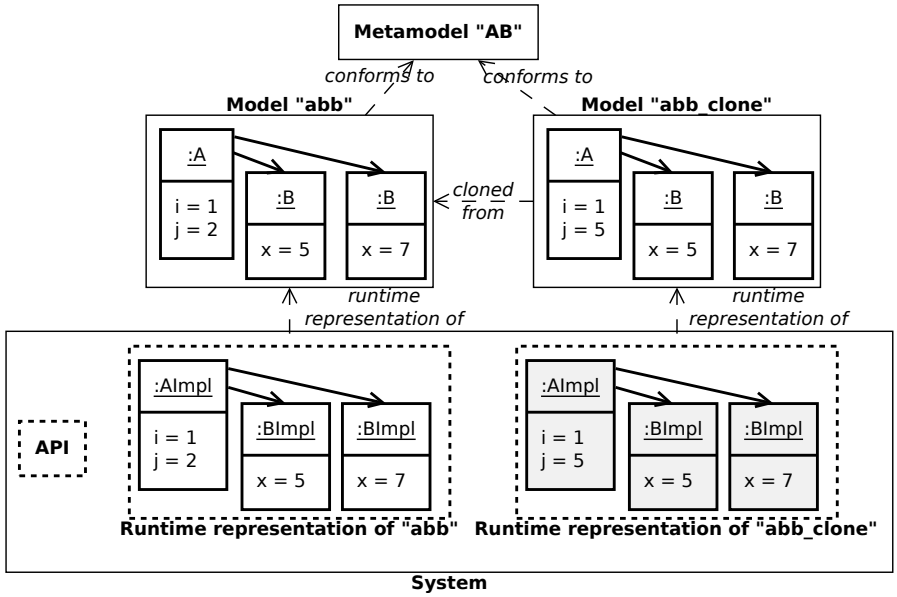


Figure 4.2: Following Fig. 4.1, *deep cloning* of the model `abb`, which created a new model `abb_clone` along with a new runtime representation in memory. Then `abb_clone` diverged from `abb` by changing its `j` value.

4.3.3 Cloning

Cloning is at the intersection of two main ideas: the duplication of elements and the independence of the obtained clone. Applied to models, a clone is therefore an independent duplication of some existing model. While we already introduced the notion in Section 2.2.2, we define a clone as follows:

> **Definition 14** *A* clone *is a model that is, when created, identical to an existing model called the* origin*. Both models conform to the same metamodel and are independent from one to another*

Cloning a model is a deterministic procedure that has a unique possible output (*i.e.,* a model identical to the original model). However there are multiple ways to implement this procedure for a given runtime environment. For instance, as long as independence is ensured, objects may be shared between a model and its clones. We therefore introduce the idea of *cloning operator* as follows:

> **Definition 15** *A* cloning operator *is an operator that takes the runtime representation of a model as input and returns the runtime representation of the clone of the model.*

Fig. 4.2 gives an example of cloning: the model `abb_clone` is a clone that was created at some point from the model `abb`. The moment the clone was created is important,

since it is an independent model that can completely diverge from its origin; on this example, `abb_clone` already changed and has a different `j` value.

At the bottom right of Fig. 4.2, the runtime representation of `abb_clone` was obtained using the *deep cloning* operator. However, as stated in the previous section, runtime representations of models can virtually take any form, as long as it can be manipulated through an API consistent with the metamodel. This is what we investigate in the next section, where we present our main contribution: cloning operators that reduce the memory footprint of runtime representations of clones through data sharing.

4.4 Memory Efficient Cloning Operators

In this section we present our main contribution: an approach for memory efficient cloning through data sharing among runtime representations. For this work, we consider that input runtime representations were obtained using the EMF, *i.e.,* each input runtime representation is identical to its model. Moreover, for our clones to be compliant with EMF, we ensure that each object of a clone is implemented by exactly one runtime object.

4.4.1 Data Sharing Strategies

When using the deep cloning operator, each object of a runtime representation is duplicated, which means twice as many objects and fields in memory. Our intuition is that since we know which parts of a metamodel are immutable, it must be possible to avoid duplicating some runtime objects and fields by safely (RQ#4.5) using them for both the runtime representations of a model and its clones. Given a model conforming to a metamodel, we call *shareable* both the elements that can be shared between the runtime representations of the model and its clones, and the parts of the metamodel that define these elements.

In Section 4.2, we defined RQ#4.2 (efficient manipulation of clones) and RQ#4.4 (ability to define generic operations). However, sharing objects and fields between runtime representations necessarily breaks one or both of these requirements. First, if the same runtime object is shared between two runtime representations, it is supposed to represent two distinct objects—one per model. Therefore, it is possible for each of these objects to have a different *container*, since both objects are conceptually separate. The problem is that the MOF Reflection package states that each object must provide a `container()` operation that returns the unique container of an object, which is implemented in an operation of EMF `EObject` called `eContainer()`. Unfortunately, when a shared EMF runtime object is used, there is no way to know in which context (*i.e.,* model) this manipulation occurs, and this operation thus cannot always return a unique container as expected. Therefore, generic operations that rely on this operation cannot be used on clones, which contradicts our RQ#4.4. Second, we rely on a *proxy* design pattern to share the fields of runtime objects: a runtime object with a shareable field can be copied into a new runtime object without this field, but with a reference

pointing to the original runtime object to provide access to this field. However, there is an overhead when accessing shared data through these proxy objects, which can be an issue with respect to RQ#4.2.

Data sharing is essential to reduce the memory footprint of clones, which is our primary objective. Consequently, we designed several strategies that establish trade-offs between memory savings and satisfaction of RQ#4.2 and RQ#4.4. Modelers can then decide how to tune the cloning algorithm with respect to their specific needs. Since only immutable data is shared, independence of clones is guaranteed (RQ#4.5). We provide four strategies that implement different interpretations of shareable metamodel elements:

DeepCloning Nothing is shareable.

ShareFieldsOnly Only immutable attributes are shareable.

ShareAll Shareable elements are immutable attributes, classes whose properties are all shareable, and immutable references pointing to shareable classes.

ShareObjOnly Same shareable classes as ShareAll, while properties are not.

If implementing the *DeepCloning* and *ShareFieldsOnly* strategies is quite straightforward, *ShareAll* and *ShareObjOnly* are more complicated because of a double recursion: shareable properties depend on shareable classes, and conversely. This can be solved using a fixed-point algorithm, or using the Tarjan algorithm [150] to compute strongly connected components of a metamodel seen as a graph. We choose Tarjan in our implementation. Our approach to memory management through data sharing is quite close to the *flyweight* design pattern from Gamma et al. [65], which consists in identifying mostly immutable objects in order to share them between multiple objects. The main difference is that this pattern specifies that the mutable part of shared objects must be a parameter of all the operations of the objects, which contradicts our first requirement since the API of the clones hence differs from the one of the original model.

4.4.2 Generic Cloning Algorithm

Before defining our algorithms for model cloning, we introduce data structures and primitive functions on which the algorithms rely. We use pseudo-code inspired from prototype-based object-oriented programming [106], *i.e.,* creating and manipulating objects without defining classes. The goal is to define the algorithms independently from any API that may be generated by a particular modeling framework. We consider the following structures and operations:

a runtime object *o* is created completely empty (*i.e.,* no fields) using the *createEmptyObject*() operation. Fields can be added using *addField*(*name*,*value*), and can be retrieved using *getFields*().

	Objects not shared (RQ#4.4 ok)	Objects shared (RQ#4.4 not ok)
Fields not shared (RQ#4.2 ok)	***DeepCloning***	***ShareObjOnly***
Fields shared (RQ#4.2 not ok)	***ShareFieldsOnly***	***ShareAll***

Table 4.1: Cloning operators obtained, one per strategy.

a strategy is an object that implements one of the strategies given Section 4.4.1 with three operations:

- ***isFieldShareable***(f) returns true if, at the metamodel level, there is a shareable property represented by f.
- ***isObjShareable***(o) returns true if, at the metamodel level, the class of the object that match this runtime object is shareable.
- ***isObjPartShareable***(o) does the same, but for *partially shareable* classes, *i.e.,* non-shareable classes with shareable properties.

copyObject(o) returns a copy of a runtime object o, i.e., a new object with the same fields and the same values. This is equivalent to the operation *copy* of EMF EcoreUtil.Copier

a runtime representation is a set of runtime objects. It can be created empty with *createEmptyRR*(), and it can be filled with objects using *addObject*(o).

a map is a data structure that contains a set of ⟨key,value⟩ pairs. It can be created with *createEmptyMap*() and be filled with *addKeyValue*(*key*, *value*).

resolveReferences (*map*) is an operation that, given a *map* whose keys and values are runtime objects, will create references in the values based on the references of the keys. This is equivalent to the operation *copyReferences* of EMF EcoreUtil.Copier.

The operation *copyObjectProxy*(o,*strategy*) is presented as Algorithm 1. It is parameterized by a strategy and an original object o, and it copies in a new object all the fields of o, except those considered shareable by the strategy. The last line of the operation creates a link to the original object in order to keep a way to access to the shareable data. Fig. 4.3 illustrates this operation with a simple object o that has two fields x and y: x is not copied in p, but can still be accessed using the reference *originObj*.

The second operation is *cloning*(rr, *strategy*), the cloning algorithm itself, presented as Algorithm 2. It takes a runtime representation rr as input and a considered strategy, and returns a runtime representation rr_{clone} of a clone of the model of rr. Depending on the strategy outputs, each object is processed differently. If the object o is shareable, it is simply added in rr_{clone}, and is thus shared between rr and rr_{clone}. If o is partially shareable (not shareable but with shareable fields), a proxy copy of o is added to rr_{clone}. Finally, if o is not shareable at all, a regular copy is put in rr_{clone}.

Algorithm 1: *copyObjectProxy*

Data:
o, a runtime object
strategy, the strategy used (*i.e.,* what is shareable)
Result: *p*, a proxy copy of *o*

```
1  begin
2      p ← createEmptyObject()
3      for f ∈ getFields(o) do
4          if ¬ strategy.isFieldShareable(f) then
5              p.addField(f.name, f.value)
6      p.addField("originObj", o)
```

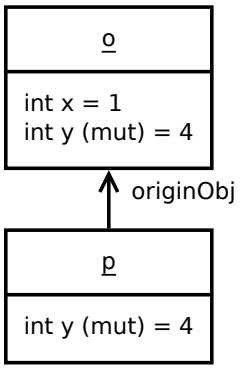
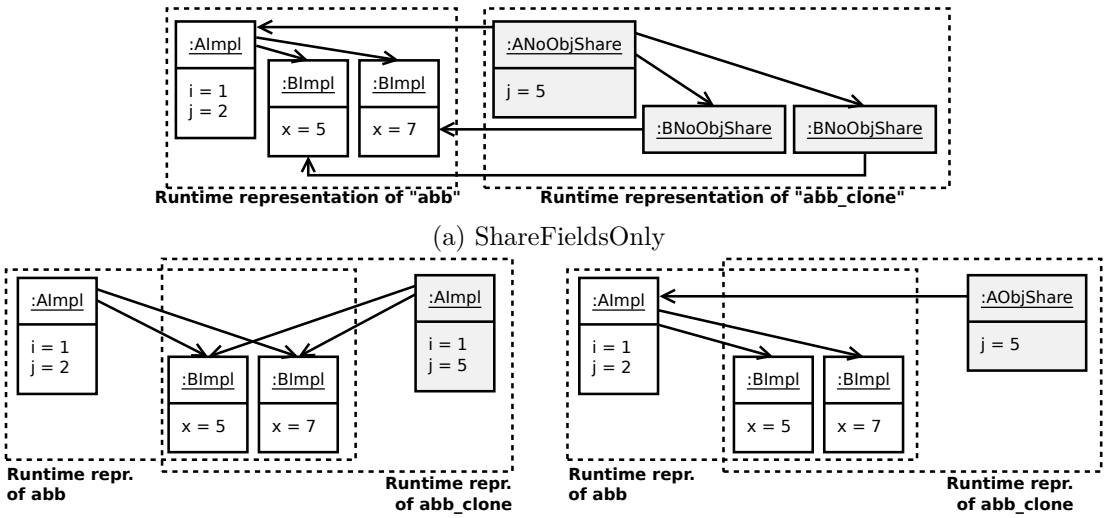


Figure 4.3: Example of proxy object: *p* is a copy of *o*.

Algorithm 2: *cloning*

Data:
rr, a runtime representation of a model
strategy, the strategy used (*i.e.,* what is shareable)
Result: rr_{clone}, a runtime representation of the clone

```
1  begin
2      rr_clone ← createEmptyRR()
3      copyMap← createEmptyMap()
4      for o ∈ rr do
5          if strategy.isObjShareable(o) then
6              rr_clone.addObject(o)
7              copyMap.addKeyValue(o, o)
8          else if strategy.isObjPartShareable(o) then
9              copy ← copyObjectProxy(o,strategy)
10             rr_clone.addObject(copy)
11             copyMap.addKeyValue(o, copy)
12         else
13             copy ← copyObject(o)
14             rr_clone.addObject(copy)
15             copyMap.addKeyValue(, o, copy)
16     resolveReferences(copyMap)
```



(a) ShareFieldsOnly

(b) ShareObjOnly

(c) ShareAll

Figure 4.4: Runtime representations of models `abb` and `abb_clone` of Fig. 4.2 obtained with the different cloning operators.

4.4.3 Family of Cloning Operators

From our single cloning algorithm, we eventually obtain four cloning operators depending on the strategy used. We sum up the possibilities in Table 4.1, and we illustrate them with examples in Fig. 4.4. *DeepCloning* clones without any form of data sharing. *ShareFieldsOnly* clones using proxy objects to share as many fields as possible; Fig. 4.4a shows an example where each runtime object has a reference to the runtime object from which it originates. In the clone, the A runtime object contains a field `j` because the corresponding property is mutable, and hence cannot be shared. *ShareObjOnly* clones with object sharing only; Fig. 4.4b shows an example where B runtime objects are referenced by both models. Finally, *ShareAll* clones with both objects and fields sharing; Fig. 4.4c shows an example where only `j` is kept by the A runtime object.

In section 4.4.1, we listed five research questions to evaluate our cloning operators. Without proper benchmarking, we cannot answer the memory consumption (RQ#4.1) question yet. Concerning the efficiency when manipulating clones (RQ#4.2), we do not expect *ShareFieldsOnly* and *ShareAll* to comply because of proxy objects. As they rely on of object sharing, *ShareObjOnly* and *ShareAll* are not compatible with generic operators that use the MOF `container()` reflective operation (RQ#4.4). However, our clones perfectly comply with the need to be manipulable by operations defined for the metamodel of the original model (RQ#4.3). This is illustrated by our implementation, which allows each clone to be manipulated using the EMF Java API generated for the metamodel. Likewise, since only immutable data is shared, the independence of the clones is ensured (RQ#4.5).

4.4.4 EMF-Based Implementation

We implemented our approach in Java with as much EMF compatibility as possible, which required us to face two main challenges. First, we had to extend EMF libraries — including implementations of EObject and Resource — to ensure that *containment* references are handled consistently in each model. Second, our approach relies on proxy objects, which are easy to create dynamically using a prototype-based object oriented language. However, with a class-based object oriented language such as Java, the fields of an object are determined by its class at design-time. We thus have to generate appropriate classes beforehand, which we do with a java-to-java transformation using EMF and MoDisco [28] to remove non-shareable properties of generated EMF implementations. More details about the implementation can be found in Section 9.1 of Chapter 9.

4.5 Evaluation and Results

This section presents our evaluation. First we describe our dataset, then what we measure and the metrics considered for our metamodels, and finally the obtained results and how they relate to the requirements stated in Section 4.2. Figure 4.5 depicts the complete evaluation process, that we describe throughout the section.

4.5.1 Dataset

To evaluate this work, we need both various metamodels and models that conform to these metamodels. For the metamodels part, we developed a random Ecore model generator, shown as (1) in Figure 4.5. We parameterized it the following way: a maximum number of 100 classes per metamodel, 250 properties per class and 50 mutable properties (which are properties with a `_m` suffix) per class. We use weighted randomness to create different kinds of properties, with the following weights: 30% of integers, 30% of booleans, 30% of strings, and 10% of references. For the models part, we generate for

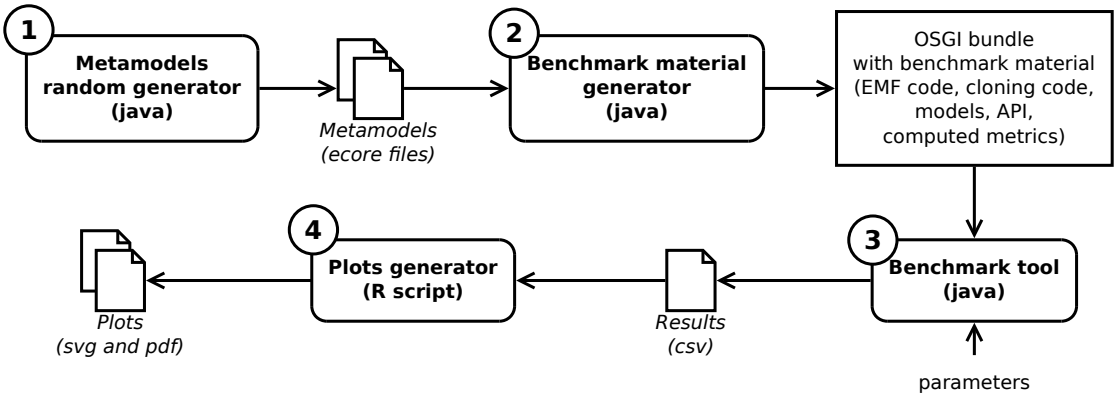


Figure 4.5: Evaluation process through random metamodel generation.

each metamodel a single model in a deterministic way that covers the whole metamodel. It starts from the roots, navigates through each composition and creates a maximum of two objects per encountered class. Then, all attributes are initialized with random values and references with random objects. We could have generated more models per metamodel, but our goal was to illustrate how our operators behave with varying *metamodels*, each with different shareable parts.

4.5.2 Measures

To verify that we reached our main objective, we must measure the memory consumption of the runtime representations of the clones, and more precisely the memory gain compared at the *DeepCloning* operator. For precise memory measures, we create a heap dump at the end of each evaluation run, and we analyze it using the Eclipse Memory Analyzer (MAT)[1]. The second measure we make is the read-access performance of the runtime representations of clones, compared to the one of the original model. We expect to see some performance decrease when proxy runtime objects are involved. We proceed by measuring the amount of time required to navigate 10 000 times through each object of a model while accessing each of their properties.

Since our implementation requires a design-time step to generate required proxy and copier classes (see Section 9.1 of Chapter 9), measures are made in two steps. As shown in Figure 4.5, we first generate an OSGI bundle (2) with everything required for the evaluation (EMF generated code, cloners, models, etc.). Then we provide this bundle to our benchmark tool to actually run the evaluation (3) with the right parameters (cloning strategies to use, number of clones, etc.)

4.5.3 Metrics

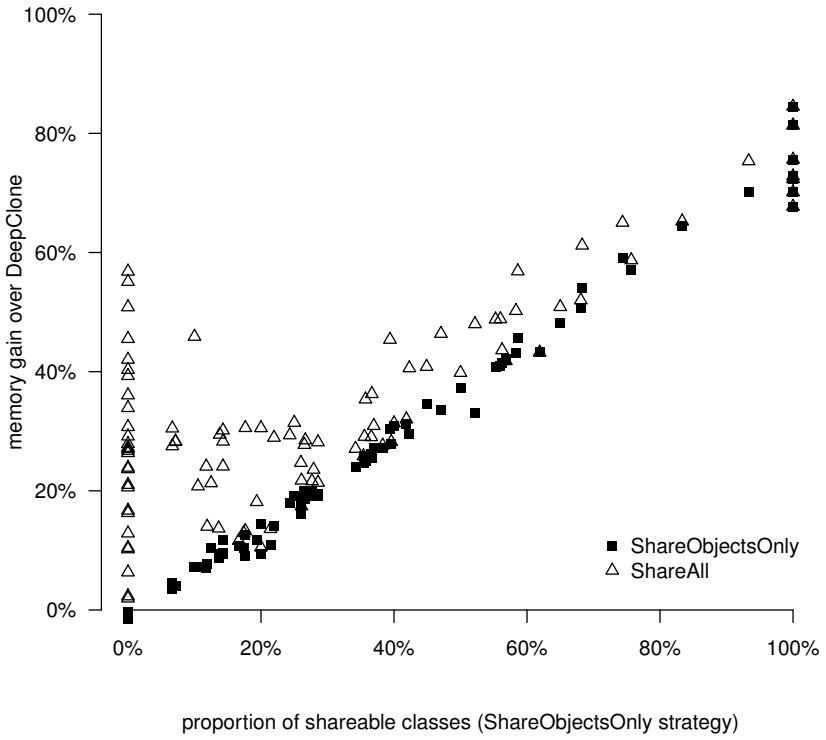
To embrace the variety of metamodels, we consider two metrics: the proportion of shareable classes when using either the *ShareObjOnly* or the *ShareAll* strategy, and the density of shareable properties within partially shareable classes when using the *ShareFieldsOnly* strategy. The first metric most likely correlates with the memory gain for operators that share objects, and the second for the operator that only shares fields.

4.5.4 Results

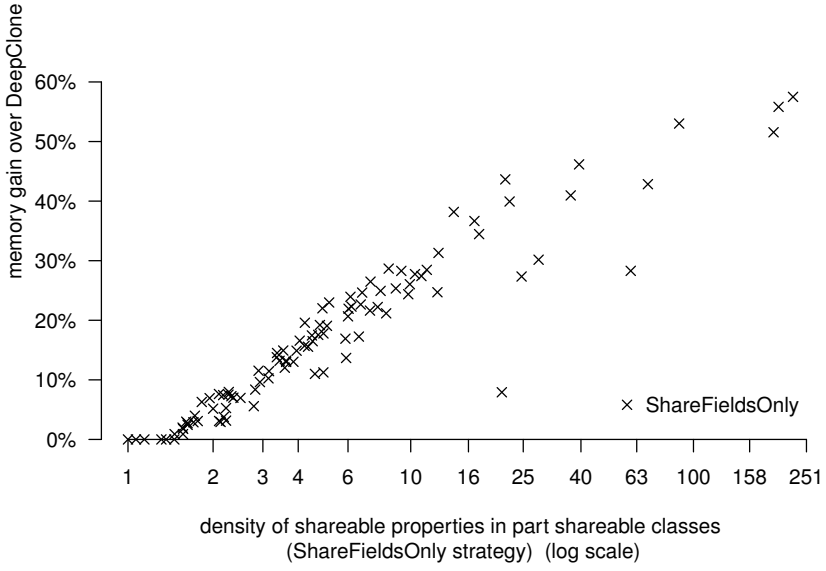
Each measure was done by creating the model of the metamodel, cloning it 1000 times with the chosen operator, and measuring both the memory footprint and the efficiency of one of the clones. As shown in Figure 4.5, raw numbers are obtained by running the benchmark tool (3), and plots are automatically obtained through an R script (4).

Fig. 4.6a shows the memory gain of the *ShareObjOnly* and *ShareAll* operators over the *DeepCloning* operator with varying proportion of shareable classes. We can see that the more shareable classes there are, the more memory gain there is. This relation appears linear for *ShareObjOnly*, and less regular for *ShareAll*. This is quite normal

[1] `http://www.eclipse.org/mat/`



(a) *Memory gain for the* ShareObjOnly *and* ShareAll *operators, with varying proportion of shareable classes.*



(b) *Memory gain with* ShareFieldsOnly *against density of shareable properties in part. shareable classes (log scale).*

Figure 4.6: Memory gain results obtained for 1000 clones.

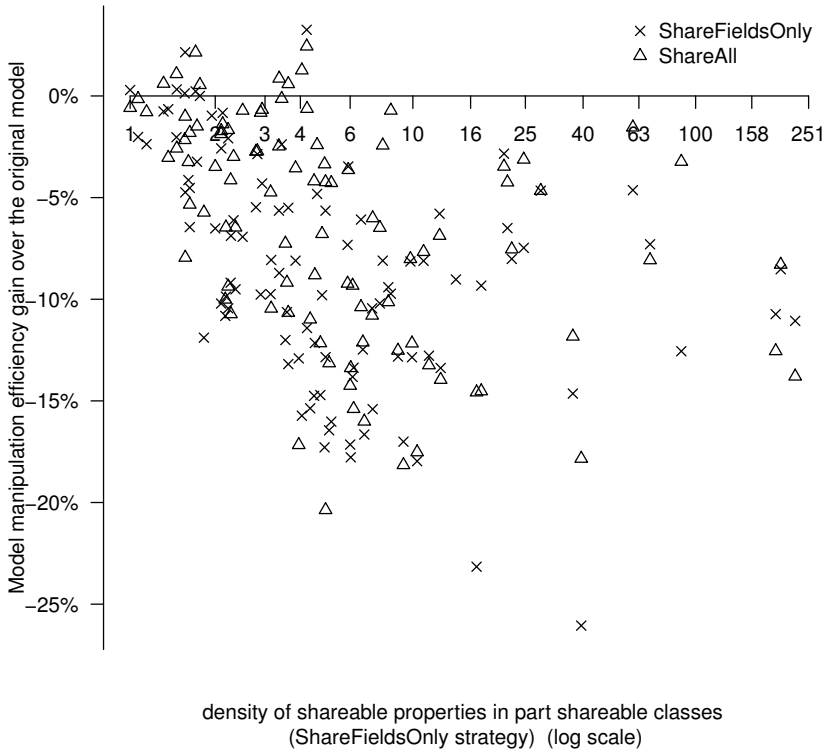


Figure 4.7: Manipulation time gain for the *ShareFieldsOnly* and *ShareAll* operators, with varying density of shareable properties in part. shareable classes (log scale).

since the first operator only relies on object sharing, while the second is also influenced by the amount of shareable properties that can be shared through proxies. We also observe that *ShareAll* is always better that *ShareObjOnly*, which was expected since it shares fields in addition to objects. Some points may look surprising at position 0%, however they are simply caused by metamodels with very few classes and a high amount of shareable properties. Thus, sharing fields of such metamodels quickly gives very high gains.

Fig. 4.6b shows the memory gain of the *ShareFieldsOnly* operator over the *Deep-Cloning* operator with varying density of shareable properties within partially shareable classes. We observe a correlation between gain and the metric, and the gain raises up to approximately 40%. This operator gives overall worse results than the *ShareObjOnly* and *ShareAll* operators, but can give better results in some situations (*e.g.,* metamodels with mostly partially shareable classes).

Finally, Fig. 4.7 presents the model manipulation efficiency gain over the runtime representation of the model originally cloned. We observe that, as expected because of the proxy design pattern, the operators *ShareFieldsOnly* and *ShareAll* both suffer from a little performance decrease. The median overhead is -9,5% for *ShareFieldsOnly* and -5.9% for *ShareAll*.

Overall, the results match our expectations. On the one hand, memory gain measures show that our operators are as good as *DeepCloning* when no parts are shareable, and are better and better as the quantity of shareable parts raises. Therefore, all our operators satisfy the need to reduce the memory footprint of clones (RQ#4.1). On the other hand, manipulation efficiency measures show that there is a little overhead when manipulating clones obtained by our operators *ShareFieldsOnly* and *ShareAll*. Thus, as we foresaw, these operators do not comply with the efficiency requirement (RQ#4.2).

4.5.5 Threats to Validity

We identified two main threats to our evaluation. First, using random metamodels, we hope to cover as many situations as possible in terms of metamodel design. Yet, have no way to be sure that our dataset contains enough "realistic" designs, as we have no metric for this criterion. Second, we use only one model per metamodel, which even if it covers the whole metamodel and is thus appropriate to evaluate our approach regarding metamodels characteristics, may overshadow some situations. For instance, if the objects of the model are mostly instances of non-shareable classes despite the fact that most classes are shareable, memory gain would not correlate with this metric as much as we observe.

4.6 Conclusion

Model cloning is an operation to duplicate an existing model that can be used in many kinds of applications. In particular, clone-based execution traces are a convenient way to capture information about a model execution. We identified five requirements for cloning operators: to be able to apply domain operators on clones, to have some memory gain over deep cloning, to ensure that clones are independent from the original model, to be able to apply generic operators on clones, and to be able to manipulate clones as efficiently as their original model, and to ensure the independence of the clones. Our goal was to provide cloning operators compliant with the first three requirements while satisfying the last two if possible.

The approach we presented consists in sharing both runtime objects and fields between runtime representations of a model and its clones. We give four possible strategies

	RQ#4.1 (mem.)	RQ#4.2 (effic.)	RQ#4.3 (API manip.)	RQ#4.4 (reflect. manip.)	RQ#4.5 (indep.)
DeepCloning	✗	✓	✓	✓	✓
ShareFieldsOnly	+	- -	✓	✓	✓
ShareObjOnly	++	✓	✓	✗	✓
ShareAll	+++	-	✓	✗	✓

Table 4.2: Summary of the characteristics of the cloning operators.

to determine which parts of a metamodel are shareable, and we use these strategies to parameterize a generic cloning algorithm. We obtain four cloning operators, each being more appropriate for a specific situation. Table 4.2 summarizes the different characteristics of the operators with respect to the research questions. *DeepCloning* is the most basic operator with no memory footprint reduction, but that can be used in all situations where memory consumption is not an issue. *ShareFieldsOnly* shares fields of immutable attributes, which reduces the memory footprint of the clones but also introduces an overhead when manipulating them. *ShareObjOnly* shares objects to reduce significantly the memory footprint, but produced clones are not compatible with generic operations that rely on the `container()` specified in the MOF Reflection package. Finally, *ShareAll* shares both objects and remaining shareable fields, which saves even more memory, but with the weaknesses of the two previous operators. All our operators can be used when the generated API of the considered metamodel is used (RQ#4.3), and all guarantee that clones are independent (RQ#4.5). Our evaluation was done using a hundred randomly generated metamodels, and results show both memory gain over *DeepCloning* for all three other operators, and a loss of manipulation efficiency for *ShareObjOnly* and *ShareAll* operators.

Regarding trace management, our approach provides facilities to generically capture clone-based execution traces while addressing the scalability in memory challenge (Ch#2). It can be used to reduce memory consumption of existing approaches relying on clone-based execution traces [99, 116], and is convenient for generic execution trace manipulations. The following chapter present our second approach, which consists in generating of multidimensional domain-specific trace metamodels that are both convenient and efficient for domain-specific trace manipulations.

Chapter 5

A Generative Approach to Define Multidimensional Domain-Specific Execution Trace Metamodels

In this chapter, we present our second contribution [23], which is a generative approach to define multidimensional domain-specific execution trace metamodels. Section 5.1 introduces the context and the main idea of the contribution. Section 5.2 motivates the problem domain and present our proposal. Section 5.3 refines some required concepts that we previously introduced, such as the mutable subset of a metamodel and execution traces.

Continuing, Section 5.4 presents our contribution, which is an approach to generate rich domain-specific trace metamodels. Finally, Section 5.5 discusses related work and Section 5.6 concludes the chapter. The work presented in this chapter is the result of a collaboration with Tanja Mayerhofer from TU Wien.

5.1 Introduction

As shown in Section 2.4.3 of Chapter 2, considerable effort has been made to design execution trace data structures to represent traces of programs or models. However, most of these data structures cannot take the domain-specific concepts of an xDSML explicitly into account, which makes the development of domain-specific analyses of execution traces more difficult (Ch#1). Moreover, redundancy of both immutable data and mutable data (*e.g.,* such as with clone-based execution traces), induced by some data structures, yields poor scalability in memory (Ch#2). Finally, most existing trace data structures only offer to explore a trace by enumerating all states and steps one by one, which can only scale linearly at best (Ch#3). To cope with that, a *domain-specific* trace metamodel that is specific to the considered xDSML can be defined [113], and alternate navigation paths can be provided to browse an execution trace [81]. Yet,

designing such a domain-specific metamodel is a time consuming and error-prone task [93], and providing alternate navigation paths is non-trivial.

In this chapter, we propose a new way to define domain-specific trace metamodels for xDSMLs through two contributions: (1) a generic approach to automatically derive a domain-specific trace metamodel for a given xDSML by analyzing its definitions of execution states and steps; (2) facilities to navigate efficiently within a trace conforming to such a generated metamodel by providing a variety of navigation paths.

We evaluated this work through two applications to existing dynamic V&V techniques: *semantic differencing* presented in Chapter 7, and omniscient debugging presenter in Chapter 8. The results show a simplification of the definition of domain-specific trace manipulations (*e.g.,* semantic differencing rules), and large improvements both in scalability in time and scalability in memory as compared to the usage of a clone-based generic trace metamodel[1].

5.2 Motivation and Proposal

As we presented in Section 2.4.3, there is a large number of execution trace data structures to represent traces of models. However, while they may have interesting characteristics (modeling of logical time [45], handling of distributed systems [56], etc.), and may be compatible with existing trace analysis tools, most of them do not answer to the challenges stated in Chapter 1. First, there is necessarily a gap between the concepts defined in an existing trace data structure and the domain concepts of a particular xDSML. Indeed, existing formats are either generic [99], or focus on specific concerns [56, 146] or languages [113, 111]. Consequently, the concepts they consider are unlikely to be adequate for an arbitrary xDSML (*e.g.,* Petri nets), especially for defining domain-specific trace manipulations. This semantic gap has a significant impact on usability (Ch#1). Second, redundancy of both immutable data and mutable data (*e.g.,* such as with clone-based execution traces), induced by some data structures, yields poor scalability in memory (Ch#2). Third, as discussed in Section 2.4.4, most do not provide facilities to process traces efficiently: the only way to navigate in a trace is by enumerating each captured execution state one by one (Ch#3). Moreover, most of these formats only capture events that occurred (see Table 2.2 page 36), such as steps, and lack a representation of the execution state, such as the values of the variables of a program. This is due to the large size of traces, which leads to limiting the amount of information stored in them. Yet, we focus in this chapter on execution traces containing both states and steps, since traces containing only steps need to be replayed in order to reconstruct the states, whereas traces containing states allow direct analyses.

The first underlying intuition of the approach we propose is the following: considering that the benefits of narrowing the scope of a language to a domain are well known [86, 153], defining a trace metamodel specific to a language should bring similar advantages. In particular, by providing concepts of the xDSML directy in the trace metamodel, the

[1] without using our scalable cloning approach from Chapter 4

usability of the trace should be improved. Mayerhofer et al. [113], followed this idea by defining manually a complete trace metamodel for fUML, which shows many benefits for analyzing past executions of fUML models. Yet, defining such metamodel can be tedious and error-prone [93], and we observe redundancies between the trace metamodel and the concepts defined in fUML. These redundancies are simply explained: the definition of an xDSML specifies what the state of a model is during its execution as part of the xDSML's semantics, and a trace metamodel directly requires such a notion of state. Hence, a first difficulty is the definition of a domain-specific trace metamodel, which can possibly be mitigated by analyzing how the execution state is defined in the xDSML. A second difficulty is that while existing trace data structures can benefit from existing trace analysis and visualization tools, domain-specific ones require specific tooling. Therefore, our first idea is to go from *generic* trace metamodels to a *generic meta-approach* to define domain-specific trace metamodels. More precisely, we propose to automatically derive a complete domain-specific trace metamodel using the definitions of execution state and steps of an xDSML. Such a generic generative approach would allow both to avoid the difficulty of defining domain-specific trace metamodels, and to automatically provide suitable tools for manipulating domain-specific traces.

The second intuition is that while a trace is generally only seen as a *sequence* of states and steps, there are in fact many imaginable ways to browse a trace. Having more navigation paths at disposal could be a great way to browse traces more efficiently. An example is finding the next value change of a given model element regardless of any other state changes in the model. Such query can be done easily by traversing the complete trace, yet reifing it as a *navigation path* dedicated to the investigated model element would avoid browsing the whole trace. Henceforth, our second idea is to create *multidimensional* trace metamodels, *i.e.,* metamodels that provide many navigation paths to explore a trace.

In a nutshell, our proposal is an approach to automatically generate *multidimensional* and *domain-specific* trace metamodels for an existing xDSML.

5.3 From Executable Metamodeling to Execution Traces

We already presented and defined what is an xDSML in Section 2.3 of Chapter 2. In the following paragraphs, we define additionally what is the mutable subset of an xDSML, and what sort of execution traces we consider.

Mutable Subset of an xDSML In Section 2.1.3 of Chapter 2, we defined the concept of metamodel footprint of a model operation, which is the set of concepts of a metamodel that are manipulated by a model operation. Then, in Section 4.3.1 of Chapter 4, we defined the mutable subset of a metamodel as the set of concepts that can be changed at the model level through model transformations.

In this chapter, we consider that the only part of an executed model that can change during an execution is its execution state. In other words, we consider that the mutable subset of the execution metamodel of an xDSML is the set of concepts it adds on top

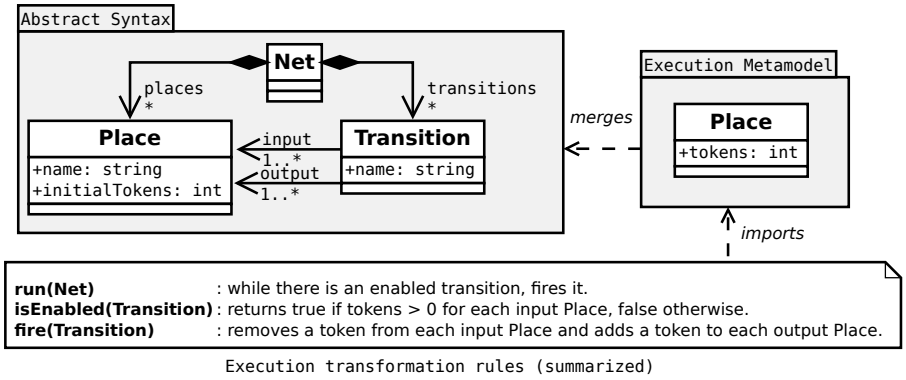


Figure 5.1: Petri net xDSML (reminder from Chapter 2).

of the abstract syntax. Hence, a property introduced by the execution metamodel is *mutable*, and a property originally defined in the abstract syntax is *immutable*.

> **Definition 16** *We call* immutable *a property introduced in the abstract syntax. At the model level, we also call immutable an object's field based on an immutable property. We call* mutable *a property introduced in the execution metamodel. At the model level, we also call mutable an object's field based on a mutable property.*

We introduced an example of xDSML in Section 2.1.3 of Chapter 2, namely Petri net. We consider in this chapter the exact same xDSML as a running example. Figure 5.1 shows a reminder and a summary of all its components. On the top left corner, its abstract syntax is depicted with three classes Net, Place and Transition. Next to the abstract syntax, the execution metamodel is shown. It extends the class Place using *package merge* with a new mutable property `tokens`. At the bottom, the transformation rules defining the operational semantics are depicted. An application of `run` is a big step composed of small steps, each being an application of `fire`.

Execution Trace We introduced a broad definition of execution trace in Section 2.4 of Chapter 2 (Definition 9 page 31). While execution traces can take various forms, we consider in the work presented in this chapter that an execution trace is a sequence of states and steps. Thereby, an execution state contains all the values of all the mutable fields of a model, *i.e.,* the values of the fields defined by properties introduced in the execution metamodel. After each small step, the execution state of the model changes, and each step is recorded in the trace along states.

> **Definition 17** *An* execution trace *is a sequence of execution states and execution steps (both small steps and big steps) responsible for the state changes.*

5.4 Generating Multidimensional Domain-Specific Trace Metamodels

We propose a generative approach to define multidimensional and domain-specific trace metamodels that provide facilities for efficiently processing traces. In this section, we present this approach by first presenting the challenges we had to overcome, second explaining our generation procedure based on the introduced Petri net xDSML, third discussing the resulting benefits of the approach, and fourth providing details on our implementation.

5.4.1 Observations and Technical Challenges

There are many possible ways to generate a domain-specific trace metamodel for an xDSML. Regarding the execution states, a simple yet working idea is to reuse the complete execution metamodel of the xDSML in the trace metamodel. As the executed model conforms to the execution metamodel, we can *clone* it at each execution step and store it as a state in the trace. We introduced such traces as *clone-based execution traces* in Section 2.4 of Chapter 2. However, this solution has multiple drawbacks. First, by duplicating the whole model to store each execution state, we create redundancies between the states for both immutable fields (as they never change) and mutable fields (as they may not change in each step). This impacts both usability (Ch#1) and memory consumption (Ch#2), although the scalable model cloning approach that we presented in Chapter 4 would mitigate this issue by sharing immutable data among clones. Second, the mutable fields we are interested in are scattered among the immutable fields, which may require complex queries to access them within a state. These issues compromise usability regarding domain-specific trace manipulation (Ch#1). Lastly, such a trace metamodel does not provide any efficient way to browse a trace, since the only possibility is to enumerate each state one by one. Thus it would be, for instance, tedious and inefficient to look for the next value of a given mutable field, compromising both scalability in time (Ch#3) and usability (Ch#1). From these observations, we identified three technical challenges (TC):

(TC#1) Narrowing the concepts introduced in a trace metamodel, *e.g.,* by focusing on the mutable properties of the execution metamodel.

(TC#2) Avoiding redundancy in traces, *e.g.,* by not storing the same value twice consecutively for a given mutable field.

(TC#3) Providing alternative navigation paths, *e.g.,* among the sequence of values of a specific mutable field.

5.4.2 Trace Metamodel Generation

Algorithm 3 shows our trace metamodel generation procedure. It relies on a recursive procedure *createStepClass* (called line 21), that is defined in Algorithm 4. Note that

Algorithm 3: Trace metamodel generation (simplified)

Input:
mm_{as} : the abstract syntax
mm_{exe} : the execution metamodel
os : the operational semantics
Result:
mm_{trace} : the trace metamodel

```
1  begin
2      c_trace, c_exeState, c_step, c_smallStep, c_bigStep ← createBaseGenericClasses()
3      mm_trace ← {c_trace, c_exeState, c_step, c_smallStep, c_bigStep}
4      foreach c_exe ∈ {c | containsMutableProperties(c)} do
5          c_traced ← createClass()
6          mm_trace ← mm_trace ∪ {c_traced}
7          c_trace.createReferenceTo(c_traced, [0..*], unordered)
8          if containsImmutableProperties(c_exe) then
9              c_orig ← getClassFromAbstractSyntax(c_exe)
10             c_traced.createReferenceTo(c_orig, [1..1])
11         foreach p ∈ getMutablePropertiesOf(c_exe) do
12             c_value ← createClass()
13             mm_trace ← mm_trace ∪ {c_value}
14             c_value.properties ← { copyProperty(p) }
15             c_traced.createReferenceTo(c_value, [0..*], ordered)
16             c_value.createReferenceTo(c_traced, [1..1])
17             c_exeState.createReferenceTo(c_value, [0..*], unordered)
18             c_value.createReferenceTo(c_exeState, [1..1])
19     foreach r ∈ os do
20         map_steps ← createMap()
21         createStepClass(f, map_steps, mm_trace, c_smallStep, c_bigStep)
22     replaceReferencesToExecutionMM(mm_trace, mm_as, mm_exe)
```

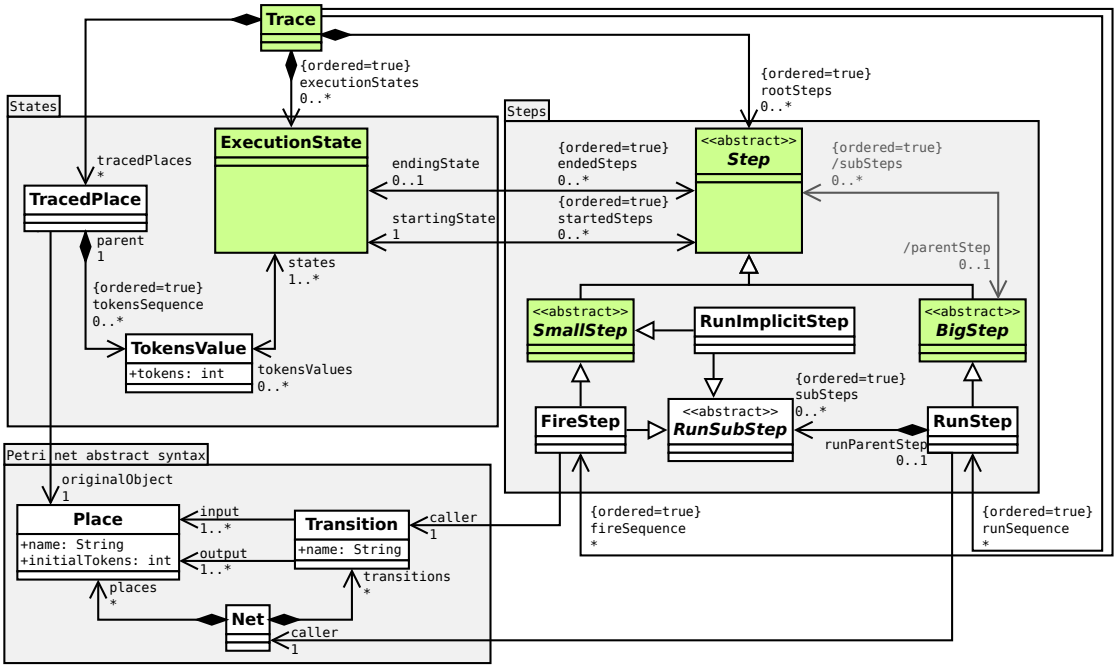


Figure 5.2: Execution trace metamodel generated for the Petri net xDSML. Classes in green are always generated.

the algorithm is simplified for illustration purposes, meaning that some parts are reduced to functions, and that special cases, such as abstract classes, are not considered. The inputs of the procedure are the abstract syntax (mm_{as}), the execution metamodel (mm_{exe}) and the operational semantics (os) of an xDSML. The procedure is independent from executable models, since the obtained metamodel is valid for any execution trace of any model of the considered xDSML. Note that the classes Trace, ExecutionState, Step, SmallStep and BigStep (shown in green color in Figure 5.2) are always created (lines 2–3). In the following paragraphs, we explain the generation procedure based on the Petri net xDSML, starting with trace concepts for capturing the smallest unit of an execution state, *i.e.,* an object's field values, up to the concepts for capturing the complete execution state of a model. The trace metamodel generated for the Petri net xDSML is shown in Figure 5.2.

Capturing the Values of Fields (lines 11–14 of Algorithm 3). At any given point in time, all mutable fields of an object of the executed model have a *value*. To represent such a value in a trace, we create one class per mutable property of the execution metamodel, and we copy this mutable property into this new class (lines 12–14). This enables us to capture each value of a mutable field as an instance of this generated class. For Petri nets this means creating one class called TokensValue for the property `tokens`. Thereby, we precisely narrow the trace metamodel to the mutable part of the execution metamodel (TC#1).

Capturing the States of Objects (lines 4–10, 15–16 of Algorithm 3). The state of an object of the executed model at any point in time is defined by the values of all its mutable fields. To represent all states reached by an object, we create one class for each class of the execution metamodel containing at least one mutable property (lines 4–5). In addition, we make all instances of these generated classes accessible through a single instance of the class Trace. For Petri nets this means creating a class TracedPlace for the class Place, and a reference `tracedPlaces` from the class Trace.

An instance of such a generated class shall contain all values reached by all mutable fields of an object of the considered type in chronological order. This is achieved by creating an ordered unbounded reference to each corresponding generated value class discussed previously (line 15). For Petri nets this means generating a reference `tokensSequence` for the class TracedPlace to the class TokensValue. When creating an execution trace, one TracedPlace object will be created per Place object, each storing a sequence `tokensSequence` of all the values reached by the `tokens` field of the respective Place object. A first benefit of this structure is that we avoid redundancy by creating a single object per value change of a mutable field (TC#2). A second benefit is that such sequences provide additional navigation paths in the trace, making it possible to directly access all changes of one specific mutable field (TC#3).

The last concern for capturing the state of an object is that the object may also contain *immutable* fields, which remain an important piece of information. Since the corresponding immutable properties are all defined in a class introduced in the abstract syntax, our solution is to create a reference to this class (lines 8–10). For Petri nets this means adding a reference `originalObject` for the traced class TracedPlace to the class Place of the abstract syntax. A TracedPlace object is thus linked to the Place object whose states it captures.

Capturing the State of the Model (lines 17–18 of Algorithm 3). An execution state can be seen as the n-tuple of the values of all mutable fields in an executed model at a given point in time. However, n is not xDSML-specific, but *model*-specific, as the number of mutable fields depends on the number of objects in the executed model. For instance, in our Petri net xDSML, n equals the number of `tokens` fields of one given model, *i.e.,* the number of Place objects.

In addition, n can change during the execution, as new objects can be created for classes introduced in the execution metamodel. To represent this n-tuple, we create a bidirectional reference between each generated value class and the class ExecutionState, which represents one execution state of a model. By that means, an execution state references an unbounded set of values of mutable fields. For Petri nets this means introducing the references `tokensValue` and `states` between the classes ExecutionState and TokensValue.

Capturing Steps (lines 19–21 of Algorithm 3, and whole Algorithm 4). A step may occur between two execution states if its step transformation rule was responsible for the respective state change. More precisely, multiple steps can start or end at an

Algorithm 4: createStepClass (simplified)

Input:

r	:	the transformation rule to transform into a step class
map_{steps}	:	a map with the step class of each processed function
mm_{trace}	:	the trace metamodel in construction
c_{trace}	:	the trace class
$c_{\text{smallStep}}$	:	the small step abstract class
c_{bigStep}	:	the big step abstract class

```
1  begin
2      if r ∉ dom(map_steps) then
3          c_step ← createClass()
4          mm_trace ← mm_trace ∪ c_step
5          map_steps ← map_steps ∪ (r ↦ c_step)
6          foreach p ∈ r.parameters do
7              c_step.properties ← copyProperty(p)
8          if getStepRulesCalledBy(r) = ∅ then
9              c_step.superTypes ← c_smallStep
10         else
11             c_step.superTypes ← c_bigStep
12             c_sub ← createClass()
13             mm_trace ← mm_trace ∪ c_sub
14             c_step.createReferenceTo(c_sub, [0..*], ordered)
15             c_sub.createReferenceTo(c_step, [1..1])
16             foreach called ∈ getStepRulesCalledBy(f) do
17                 createStepClass(called, map_steps, mm_trace, c_smallStep, c_bigStep)
18                 c_called ← map_steps(called)
19                 c_called.superTypes ← c_called.superTypes ∪ c_sub
20             if containsImplicitSteps(f) then
21                 c_fill ← createClass()
22                 mm_trace ← mm_trace ∪ c_fill
23                 c_fill.superTypes ← {c_smallStep, c_sub}
24         c_trace.createReferenceTo(c_step, [0..*], ordered)
```

```
1  def void run() {
2      ... // code with model change (1)
3      someTransition.fire()
4      ... // code with model change (2)
5  }
```

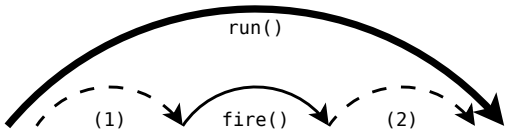


Figure 5.3: Illustration of implicit steps with a simplified `run` step rule.

execution state: an unbounded number of big steps and at most a single small step. This is represented by the references `startingState`, `startedSteps`, `endingState-State endedStepsStep` between the classes ExecutionState and Step. In addition, each step transformation of the operational semantics is reified into a class of the same name (lines 3–5 of Algorithm 4), in which all parameters of the rule are copied (lines 6–7 of Algorithm 4). The resulting class inherits either SmallStep if the rule doesn't call another step rule (line 8–9 of Algorithm 4), or BigStep otherwise (line 10–11 of Algorithm 4). For Petri nets, this means creating the classes FireStep inheriting from SmallStep, and RunStep inheriting from BigStep. By copying the parameters of the rules, each of these two classes is given a reference `caller`, to be able to point to the Net or Transition object concerned by the rule.

A step can be part of a big step, which is represented by the derived references `parentStep` and `subSteps`. More precisely, a big step is the root of a *tree* whose nodes are steps and whose leaves are small steps. To match the operational semantics as precisely as possible (TC#1), we restrict the steps contained into a big step to the ones that may occur within its corresponding model transformation through the creation of a dedicated abstract class (lines 12–13 of Algorithm 4). In addition, we rely on containment references to enforce the tree structure that are induced by big steps (lines 14–15 of Algorithm 4). For Petri nets, this means creating a class RunSubStep representing all sorts of steps that may occur during a RunStep step, and two references `subSteps` and `runParentStep`.

Then, step classes of all called step rules are created through a recursive call of the step class creation procedure (Algorithm 4), and through the use of a map that associate each rule to its step class (lines 16–19 of Algorithm 4). The first line of the algorithm is the stopping criterion to handle the recursion: we only create once the class corresponding to a rule. For Petri nets, this means that the class FireStep is defined as a subclass of RunSubStep, since this is the only operation called by `run`.

Another problem is that it is possible for the `run` operation to be responsible of other model changes in between the calls to `fire`. Figure 5.3 depicts such situation through a simplified and hypothetical `run` step rule: before and after calling `fire`, the code of `run` might be responsible for model changes, annotated (1) and (2). Even though such changes are not *explicitly* isolated within dedicated transformation rules, they must be considered as *implicit* small steps nonetheless, which is done by creating a dedicated class (lines 20–23 of Algorithm 4). For Petri nets, this means having a class RunImplicit-Step inheriting both from SmallStep and RunSubStep. Note that such generation could be avoided provided an analysis of the `run` operation that would verify that no changes

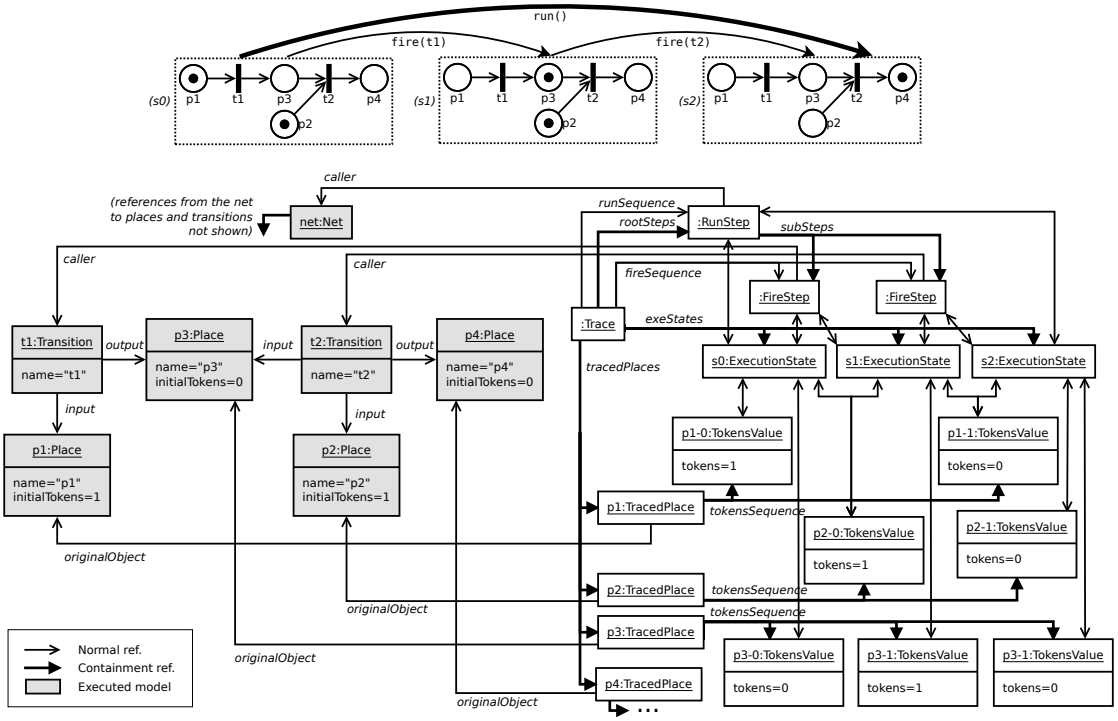


Figure 5.4: Example of Petri net model and multidimensional domain-specific trace.

are made to the model apart from the calls to `fire`. We represent this analysis by a procedure called *containsImplicitSteps* (line 20 of Algorithm 4). Yet, to better illustrate the algorithm, we consider that we don't have such an analysis for Petri nets operational semantics, and thus that this procedure returns *true*.

Finally, in the same manner as for values, all steps are stored chronologically within the unique Trace object (line 24). For Petri nets this means having an ordered reference `fireSequence` in the Trace class to the class FireStep, and a similar reference `runSequence` to the class `Run`. This gives direct access to all steps of a specific transformation rule in chronological order, which is an interesting additional navigation path for a trace (TC#3).

Replacing References to the Execution Metamodel (line 22). When mutable properties and step classes were copied in the trace metamodel, this included copying references to classes of the execution metamodel. Yet, such classes may contain mutable properties that were already copied in the trace metamodel. To avoid having twice the same concept in the trace metamodel (TC#1) or twice the same value stored in a trace (TC#2), our solution is to replace all references to the execution metamodel by references either to the abstract syntax or to classes representing the states of objects (*e.g.,* TracedPlace). This is indicated by the function *replaceReferencesToExeMM* (line 22).

Example Trace. Figure 5.4 shows a multidimensional domain-specific trace of a Petri net model. Note that to construct such a trace, one must instrument the semantics of an xDSML, which is out of the scope of this contribution. In the upper part, we use the concrete syntax of Petri nets to show the execution. In the lower part, we use an object diagram to show the content of the executed model and of the trace at the end of the execution. In the example model, the transitions $t1$ and $t2$ are fired, leading to a trace with three states, two small steps, and one big step. To represent the states, three ExecutionState objects are linked to a set of TokensValue objects, which represent the marking of the Petri net. They are linked to FireStep objects, which represent the firing of $t1$ and $t2$, some are linked to the RunStep object that represents the complete Petri net run. There is one `tokensSequence` list per `tokens` field: $(1, 0)$ for $p1$ and $p2$, $(0, 1, 0)$ for $p3$ and $(0, 2)$ for $p4$ (not shown). These sequences constitute alternative navigation paths that facilitate queries, *e.g.,* we can find the maximum number of tokens reached by $p1$ by reading only two values. Moreover, we can go from one such sequence back to the complete trace, *e.g.,* to find all states in which $p4$ had at least two tokens. Regarding steps, we have access to the list of the fired transitions by browsing the fireSequence list, *e.g.,* to find states following directly a firing of $t2$. Likewise, we have access to the list of runned nets with runSequence.

Note that this example does not illustrate the creation or deletion of objects within an execution. Such case is handled with the help of the variable number of references from a ExecutionState element to values. Hence, an object created just before a state means that this state and the following ones have references to the values of this object. Likewise, an object deleted just before a state means that this state and the following ones have no references to its values.

5.4.3 Resulting Benefits

Among all the concepts we create in a trace metamodel, some are generic (*e.g.,* Trace), but the others are specific to the xDSML (*e.g.,* TokensValue). Also, we make sure not to have any redundancy of concepts. In other words, we *precisely define* the structure of execution traces of models conforming to an xDSML. Thereby, domain-specific analyses of traces have direct access to these concepts, and do not have to rely on complex queries or introspection to use domain-specific data. We aim by that means to provide good usability (Ch#1).

In addition, we provide several *navigation paths* for browsing traces. Indeed, we create for each mutable property (*e.g.,* `tokens`) and each step definition (*e.g.,* FireStep) of an xDSML a dedicated navigation path (*e.g.,* `tokensSequence` and `fireSequence`). This allows to enumerate each value of a particular field, or each step definition of a particular step, without having to enumerate all the states of the trace. Moreover, all values and steps are connected through execution states, allowing to go from one navigation path to another. These navigation facilities offer better usability and scalability in time (Ch#3 and Ch#1)

5.4.4 EMF Implementation

We implemented our approach for the Eclipse Modeling Framework (EMF). The resulting execution trace metamodel generator is written using EMF and Xtend. It is defined in two components: one to extract information from the operational semantics, and a second to generate the execution trace metamodel using this information. While the first component is specific to the considered model transformation language (*e.g.,* xMOF, Kermeta), the second is completely generic. Along the trace metamodel, the generator also produces a trace manager with basic operations to construct and manipulate traces. More details about the implementation can be found in Section 9.4 of Chapter 9.

5.5 Related Work

In Section 2.4.3 of Chapter 2, we reviewed a number of existing work on the topic of model execution trace data structures. In the following, we look back at approaches we had presented that are related to our solution, and we discuss the differences we observe. We first focus on methods for defining domain-specific trace metamodels, then we look at existing work on multidimensional trace data structures and finally we examine how self-defining trace formats can be related to our work.

5.5.1 Domain-Specific Trace Data Structures

Hegedüs et al. [83] propose a generic execution trace metamodel that must be manually extended into a domain-specific trace metamodel using inheritance relationships. They consider a trace to be a sequence of both changes and snapshots of objects of the model, with no representation of the complete execution state. We can summarize three main differences with our approach. First, the structure is different from ours, both because we take into account the complete execution states of the model, and because we only consider high-level changes (*i.e.,* steps) corresponding to relevant subsets of the execution model transformation. Second, their approach consists in extending a generic execution trace metamodel using inheritance, while we generate a complete metamodel with customized classes and properties for the considered xDSML. Thereby, we aim to avoid both type checks and casting, and to be closer to the considered domain. Lastly, their approach is manual, while ours is generative and automatized.

In the context of the TopCased project [32, 41], Combemale et al. [37] propose the definition of a *trace management metamodel* specific to the model of computation of an xDSML. Such trace metamodel is only concerned with events occurrences, while our approach considers execution states. In addition, like Hegedüs et al. [83], their approach is manual while ours is generative and automatized.

Gogolla et al. [69] generate *filmstrip models* from UML class diagrams. Such filmstrip models consist of UML classes, and match what we call domain-specific trace metamodels. However, the generated classes are almost identical to the ones from the input metamodel, hence leading to a trace metamodel equivalent to a clone-based one. This

induces the same limitations as the ones we identified in Section 5.4.1, which included redundancy, poor usability and and poor efficiency.

Meyers et al. [116] introduced the ProMoBox framework , which generates a set of metamodels from an annotated xDSML, including a property metamodel and a trace metamodel. More precisely, they provide a clone-bases generic execution trace metamodel that is extended into a domain-specific metamodel by their generative approach. While being generative like ours, their approach differs on multiple aspects. First, they consider an abstract syntax whose properties are annotated either as *runtime* or *event* to identify mutable elements and event-related elements, while we consider the abstract syntax and the execution metamodel to be separated. Indeed, such separation makes possible a better separation of concerns and interchangeability of semantics. Second, the obtained trace metamodel is clone-based, since each state is a complete snapshot of the execution metamodel, with the same limitations as Gogolla et al. [69]. Finally, similarly to Hegedüs et al. [83], they use inheritance links to extend a base trace metamodel, while we generate new classes to avoid having to rely on introspection and casting when manipulating traces.

5.5.2 Multidimensional Trace Data Structures

As we presented in Section 2.4.4 of Chapter 2, few approaches propose multidimensional facilities to follow the evolution of specific model elements within an execution trace. Two approaches show significantly related to ours.

Filmstrip models from Gogolla et al. [69]— mentioned above for their domain-specific aspect— provide a structure that makes possible to follow the evolution of a single object of a model through added references, which facilitates the analysis of specific elements. This is very similar to the *dimensions* we propose in our approach. However, because a new snapshot of an object is created at each execution step, following such navigation path requires as many iterations as browsing the complete execution trace of the model. Moreover, we consider a dimension to be at the level of a *field*, while they consider the level of an *object*.

KMF runtime versioning [81] stores the versions of each object of a model separately, allowing to enumerate the states of a specific object of the executed model. Their approach does allow to navigate among the states of a model from the perspective of a specific element of the model, hence with much fewer iterations. However, their approach is generic and does not capture a domain-specific execution trace metamodel. Moreover, similarly to Gogolla et al. [69], they consider changes at the level of an *object*.

5.5.3 Self-defining Trace Formats

Lastly, we also presented in the Section 2.4.3 of Chapter 2 *self-defining trace formats*, which are formats allowing to define the data structure of the trace within metadata stored within the trace itself. This can be compared to a model that would embed its own metamodel. While a self-defining trace format cannot directly be used to construct the traces of an xDSML, it could potentially be an interesting alternative to MOF

for the definition of trace metamodels. For instance, adapting our approach to generate domain-specific metadata for the Common Trace Format (CTF) [47] would make possible to benefit from a very memory efficient binary format. However, since we consider xDSMLs to be defined using metamodels defined with MOF, using such meta-formats for execution traces would make difficult the proper definition of links with the executable model, both at the metamodel and at the model level. Moreover, to our knowledge, self-defining trace formats do not provide multidimensional navigation facilities.

5.6 Conclusion

Dynamic V&V of models requires the ability to model executions traces. We identified two important requirements regarding the definition of a *trace metamodel* for an xDSML: it must provide good *scalability in time* when manipulating traces, and good *usability* to analyze traces containing domain-specific data and steps. Generic trace metamodels are not adequate because of their distance to the domain of an xDSML and because of their lack of alternative trace exploration means. The approach we presented consists in generating a *multidimensional* and *domain-specific* trace metamodel of an xDSML, using its definition of what the execution state of a model is, and which steps may occur during an execution. We reify the mutable properties of the execution metamodel into classes, allowing both to reduce redundancy and to narrow the trace metamodel. We also provide navigation paths both to follow the evolution of each mutable field of the model over time, and to follow the steps of each step definition. This allows an efficient navigation of traces, *i.e.,* an exploration without visiting each state of the trace.

The following chapters present and evaluate two applications of our generative approach to two existing dynamic V&V techniques, namely semantic differencing, and omniscient debugging.

Part III

Applications and Tooling

Chapter 6

Foreword to the Applications to Dynamic V&V

In the previous chapter, we presented our second contribution, which is a generative approach to define multidimensional execution trace metamodels. The current chapter is a short introduction to the two applications of this contribution to existing dynamic V&V approaches, namely *semantic differencing* (Chapter 7) and *omniscient debugging* (Chapter 8). Section 6.1 explains the incentive and objectives of these applications. Section 6.2 gives an overview of these applications. Finally, Section 6.3 introduces the considered xDSML for the evaluation of both applications, namely fUML.

6.1 Objectives

As we mentioned in Chapter 3, we aim in this thesis at providing trace management facilities for both generic and domain-specific trace manipulations. Yet, in the context of early dynamic V&V, the main focus of this thesis is the definition of trace management facilities specific to an xDSML. This eventually led to our second contribution: generating multidimensional domain-specific execution trace metamodels (Chapter 5).

Consequently, in the following chapters, we focus on illustrating the concrete benefits of this solution in particular. To this end, we apply it to two existing dynamic V&V approaches, namely *semantic differencing* and *omniscient debugging*. Doing so has two main advantages. (1) Because our contribution is a meta-approach, it is difficult to evaluate while taking into account all the impacts at the application level. Therefore, having multiple concrete applications shows that the approach is working and relevant in different contexts. In addition, evaluating these applications allows us to indirectly evaluate our the meta-approach, *e.g.,* to illustrate the benefits in scalability. (2) Applying our meta-approach is beneficial to existing V&V techniques. Hence, it leads to contributions to the field of dynamic V&V itself, such as advanced and efficient omniscient debugging of executable models [22].

6.2 Overview

To illustrate and evaluate our second contribution (Chapter 5), we applied it in two different contexts: one that requires the manual definition of trace manipulations (*i.e.,* manually written code), and the other that relies on the automatic generation of trace manipulations (*i.e.,* generated code).

We present in Chapter 7 an application to *semantic model differencing*, which is a dynamic V&V activity that consists in analyzing the semantic differences among different versions of an executable model being developed. More precisely, we enhance an existing approach that relies on the comparison of execution traces [99] by adding a preliminary step to generate a multidimensional domain-specific execution trace metamodel of the considered xDSML. This approach requires the definition of *semantic differencing rules*, which define when the traces of two models conforming to the same xDSML are equivalent. In other words, these rules are manually defined trace manipulations that are specific to the considered xDSML. Studying the complexity of these rules gives the possibility to evaluate the usability (Ch#1) of the generated trace metamodel for domain-specific trace manipulations. Moreover, these manipulations can benefit from the additional navigation paths, which gives the possibility to evaluate the gain in execution time due to the multidimensional structure (Ch#3). This work is an extension of the evaluation presented in our ECMFA'15 publication [23].

We then present in Chapter 8 an application to *model omniscient debugging*, which is a dynamic V&V activity that consists in controlling and observing the execution of a model in order to find the cause of a defect. We propose an approach to define a partly generic advanced omniscient debugger relying on generated domain-specific trace management facilities. The generic part includes the debugger logic. The generated part includes a multidimensional domain-specific trace metamodel for the considered xDSML, but also a state manager and a trace constructor. The latter two components consist of trace manipulations defined for the generated trace metamodel. Since these manipulations are generated and not manually defined, the usability of the trace metamodel cannot be evaluated in this case. However, execution time (Ch#3) and memory consumption (Ch#2) can be measured. This work led to a publication in the proceedings of the SLE'15 conference [22].

We evaluate both these applications using the same real-world xDSML, namely *fUML*. In addition, both evaluations rely on the same dataset of real-world models from the case study of Maoz et al. [112]. We present thereafter both the xDSML and the dataset.

6.3 Case Study: fUML

fUML (foundational UML) [122] is a subset of UML [123] for which precise and complete operational semantics are defined within a standard. The considered subset focuses on two well-known parts of UML: *class diagrams* to define the structure, and *activity diagrams* to define the behavior. The resulting xDSML is a modeling language very

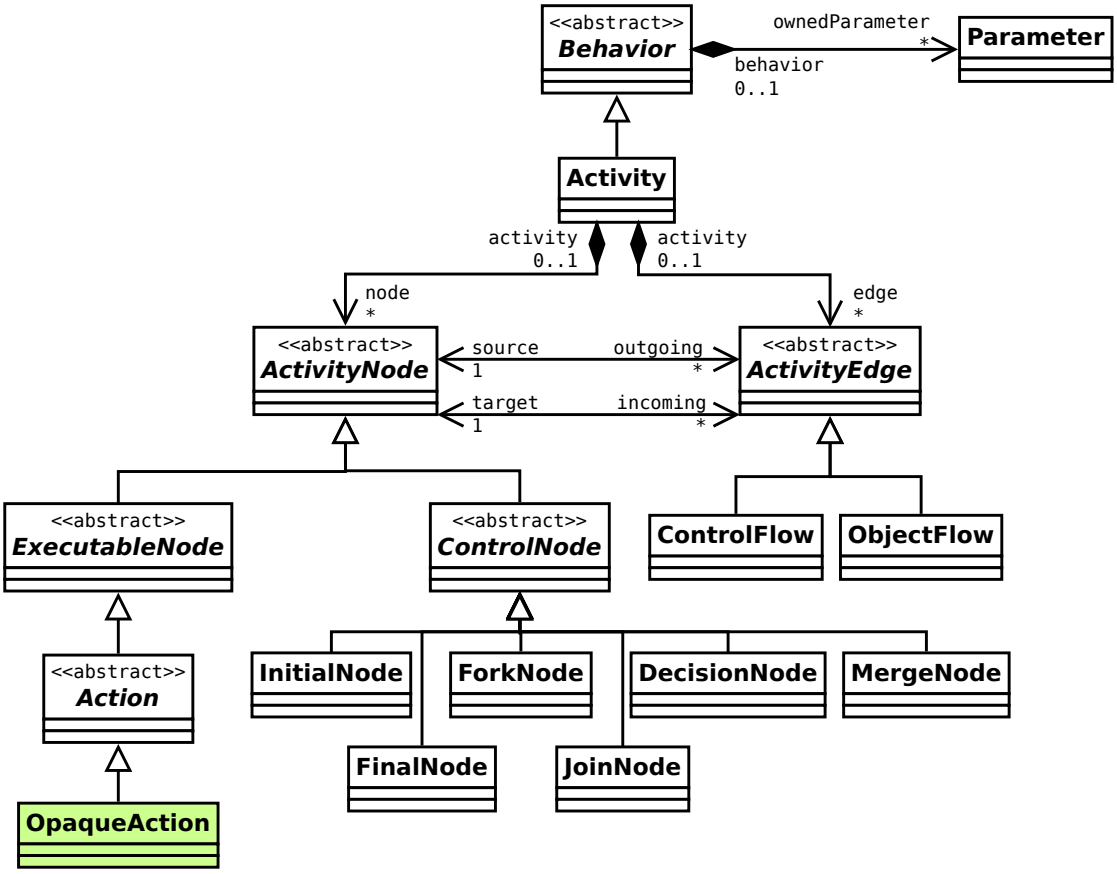


Figure 6.1: Excerpt of the extended fUML abstract syntax (focus on Activity).

similar to object-oriented programming languages such as Java or C#. In addition, a reference implementation[1] of the fUML operational semantics can be used to execute models.

Both the abstract syntax and the execution metamodel of fUML contain an important number of classes. For this case study, we only rely on the behavioral part of fUML, *i.e.,* activities. In the following paragraphs, we summarize the parts from both the abstract syntax and the operational semantics that concern fUML activities.

6.3.1 Abstract syntax

Figure 6.1 shows an excerpt of the fUML abstract syntax focusing on activities. In most cases, an Activity object represents the implementation of an operation of a fUML Class object (not shown). It has a set of parameters through the property `ownedParameters`, and is composed both of ActivityNode objects, and ActitityEdge objects. There are two main types of *nodes* defined by two classes: Action, ControlNode. Actions define how

[1] `https://github.com/ModelDriven/fUML-Reference-Implementation`

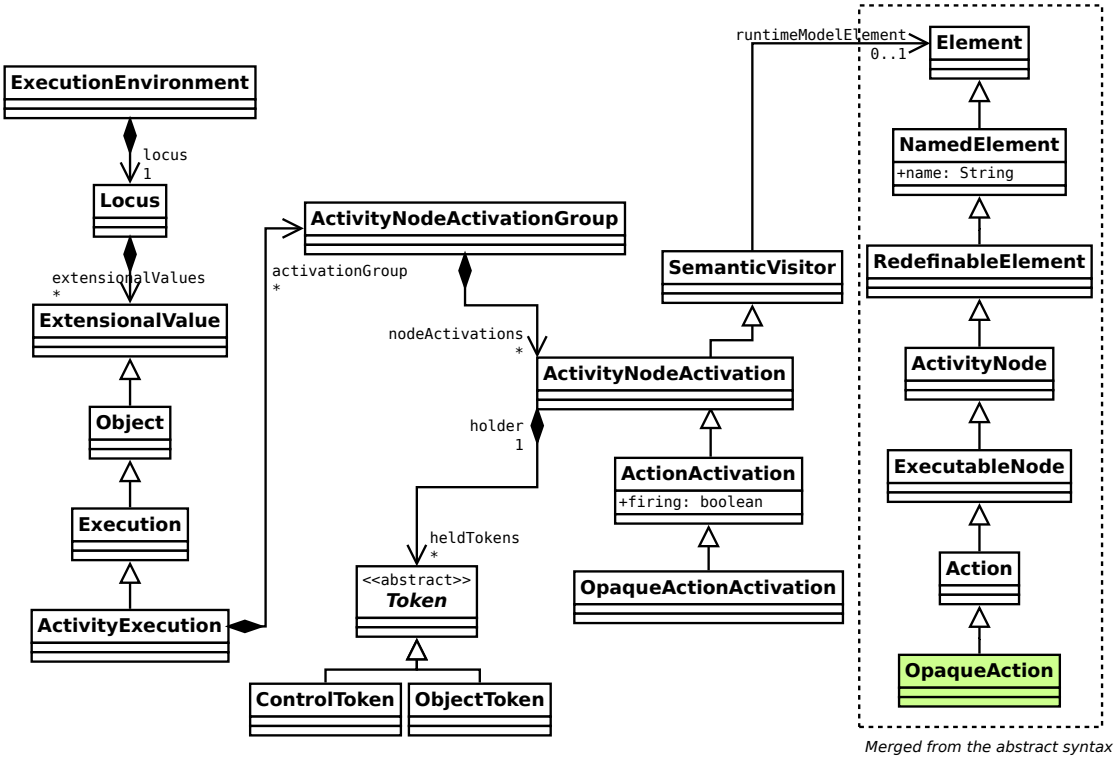


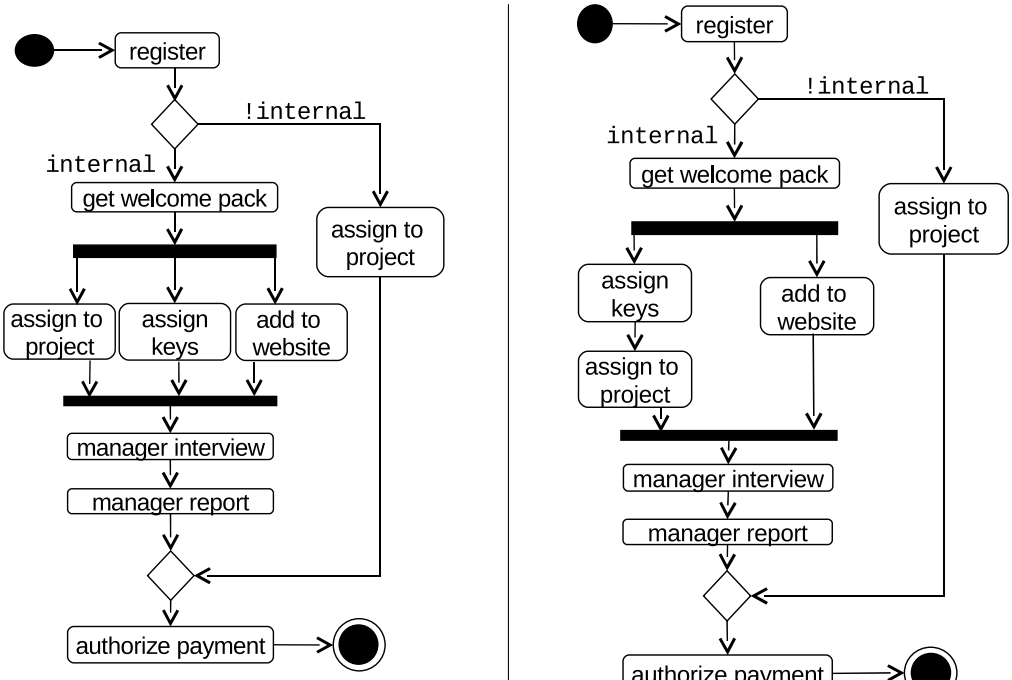
Figure 6.2: Excerpt of our fUML execution metamodel (focus on ActionActivation).

fUML objects and data can be manipulated within an activities: creating instances of classes, assigning values to variables, invoking other activities, etc. Control nodes define the begining and the end of an activity, as well as conditionals or concurrency among nodes. To connect nodes, there are two types of *edges* defined by two classes: ControlFlow and ObjectFlow. Control flow edges define the flow of control among nodes (*i.e.,* in which order and under which conditions are nodes executed), and object flow edges define the flow of data (*e.g.,* to parameterize operation calls). Note that for this case study, we only focus on the control flow of fUML activities, and we do not consider any concept related to the object flow.

In addition, to be able to reuse models from the case study of Maoz et al. [112] (see Section 6.3.4), we extend fUML with one additional class borrowed from UML: OpaqueAction (depicted in green). Indeed, their case study originally does not rely on fUML, but on a variant of UML activity diagrams containing the OpaqueAction class. While in UML this class represents an unspecified behavior, we consider in our case study that executing an OpaqueAction does not have any effect.

6.3.2 Operational semantics

Figure 6.2 shows an excerpt of the fUML execution metamodel that we consider. It is mostly based on the operational semantics provided in the fUML specification [122].



(a) *Version 1*

(b) *Version 2*

Figure 6.3: Example of two different versions of a fUML activity. First *version 1* was developed, then was modified to obtain *version 2*. Figure taken from [112].

Classes called *semantic visitors* are defined to decorate classes of the abstract syntax with execution data using references. They are defined as subclasses of the SemanticVisitor class, which has a property `runtimeModelElement` to refer to the element that is decorated. For instance, ActivityNodeActivation defines the state of an ActivityNode object. Among other things (not shown), it defines the tokens that are held by the node through the reference `heldTokens`. Token objects are contained within nodes, and drive both the control flow (with the ControlToken subclass) and the data flow (with the ObjectToken subclass). Hence, somewhat similarly to Petri nets, executing an activity consists in a flow of tokens from the initial node to the final node. Finally ActionActivation is a subclass of ActivityNodeActivation, and defines whether or not an action is being executed through the boolean `firing`.

The complete execution state is contained in a single ExecutionEnvironment object, which contains a single Locus object, which among others contains the state of all Activity objects through ActivityExecution objects. Each ActivityExecution contains the states of all nodes of the corresponding activity using ActivityNodeActivation objects.

6.3.3 Example of Model

Figure 6.3 depicts an example of a real world activity taken from the case study of Maoz et al. [112] (presented thereafter). More precisely, Figure 6.3a shows the first version of the model, and Figure 6.3b shows how it was modified into a second version. All actions are opaque actions. The activity describes the process of managing a new employee in a company. At the bottom, the *initial node* is a control node indicating where the activity starts. A *control edge* links it to an *action* named register. Then a second control edge links it to a *decision node*, whose guard only relies on a single boolean parameter of the activity called `internal`. If this parameter is *true*, then the left part of the activity diagram is executed, starting with get welcome pack. If it is *false*, it directly goes to assign to project.

Continuing with the left part of the first version (Figure 6.3a), a *fork node* starts three actions in parallel: assign to project, assign keys and add to website. Then, the three control flows meet in a *join node* that only continues the flow when the three actions are finished. It then leads to the action manager interview, then manager report, and then to a *merge node* which merges the two possible control flow originating from the decision node at the start. Finally, after a last action authorize payment, the *final node* is reached, and the execution of the activity is over.

The second version of the model (Figure 6.3b) is almost identical to the first version, except for the action assign to project which is moved after assign keys.

6.3.4 Considered Dataset

For evaluating our applications, we used real world models taken from the case study of Maoz et al. for evaluating their semantic differencing operator *ADDiff* [112]. More precisely, the study contains different sequences of models, each sequence containing different versions created from one original model. These models may be found at `http://www.se-rwth.de/materials/semdiff/`. Both models that we have shown in Figure 6.3 are part of this study.

The choice of this existing case study was made to help establish a benchmark, facilitate comparison with future work, and because the models were drawn from industrial sources. To constitute our dataset, we selected 40 models whose sizes range from 36 to 51 objects. This required the a manual conversion of the models so that they conform to the fUML metamodel instead of the UML metamodel. We plan to integrate larger models to the dataset for a future study, but are confident in the current ones to provide initial meaningful comparison.

Chapter 7

Efficient Semantic Model Differencing

In this chapter, we present an application of multidimensional domain-specific execution trace metamodels (presented in Chapter 5) to *semantic differencing* of executable models. Section 7.1 introduces the context of model evolution, the problem of defining semantic match rules, and the research questions we consider. Section 7.2 presents an existing semantic differencing approach based on execution traces, which relies on a generic clone-based execution trace metamodel.

Continuing, Section 7.3 shows how we enhance this approach through the use of *multidimensional domain-specific trace metamodels*. Section 7.4 presents our evaluation, which relies on the fUML case study presented in the previous chapter. Finally, Section 7.5 concludes.

The work presented in this chapter is an extension of the evaluation originally performed in [23], for the contribution described in Chapter 5. Likewise, it is the result of a collaboration with Tanja Mayerhofer from TU Wien, who was author of the semantic differencing approach [99] that we consider and enhance.

7.1 Introduction

The field of model evolution is concerned with analyzing and understanding the changes made to a model during its development. Most of the existing approaches compare two models *syntactically* [27, 108], *i.e.,* by computing correspondences and differences among their model elements. Yet, in the case of an executable model, a change in its content may impact its behavior, thus requiring to compare the behaviors of the model before and after the change to properly take into account their differences. This is the principle of *semantic model differencing*. A particular way to perform semantic differencing is to compare execution traces of the models of interest [99]. First, *semantic differencing match rules* are defined for a given xDSML, the rules indicating which differences among the execution traces constitute semantic differences among the models. The definition

of these rules directly depends on the considered execution trace metamodel. Second, these rules are used to compare the execution traces of the considered models, hence comparing their behaviors.

However, defining semantic differencing match rules is a difficult task, especially when the execution trace metamodel is *generic*, since it lacks usability. Likewise *scalability in time* is an issue, since all states must be enumerated when comparing the execution traces, even if the rules are based on a subset of the information they contain. For instance, the approach of Langer et al. [99] relies on a generic execution trace metamodel, and is hence affected by both these problems. To overcome these obstacles, we propose to enhance semantic differencing by relying on *multidimensional domain-specific execution trace metamodels*, as introduced in Chapter 5. To achieve this, we modify the approach from Langer et al. [99] by adding a prior step to generate such an execution trace metamodel for the considered xDSML. The expected benefits are both a simplification of the semantic match rules due to the fact that the trace metamodel is *domain-specific*, and an improvement of the performance thanks to the *multiple dimensions* of the execution traces. We evaluate our approach according to the following research questions:

RQ#7.1 Can a multidimensional domain-specific trace metamodel provide better execution times for semantic model differencing as compared to a generic clone-based trace metamodel[1]?

RQ#7.2 Can a multidimensional domain-specific trace metamodel simplify the definition of semantic match rules as compared to a generic trace clone-based metamodel?

To validate our approach, we generate a multidimensional domain-specific trace metamodel for a real world xDSML, namely fUML. Then, we define a set of semantic differencing rules for fUML based on this generated metamodel. Finally, we use these rules to compare traces of a set of real world fUML models extracted from an existing case study. Results discussed in Section 7.4 show a significant performance improvement and a simplification of the semantic differencing rules as compared to equivalent rules based on a generic execution trace metamodel.

7.2 Semantic Model Differencing

7.2.1 Existing Approach

As we explained in the introduction, semantic differencing of models is concerned with identifying behavioral differences among distinct versions of models. While some approaches are *non-enumerative* and synthesize an aggregate model representing the semantic differences between two models [58], some approaches are *enumerative* and produce a list of *witnesses* [112]. Each witness makes explicit one specific semantic difference between two models.

[1] without using our scalable cloning approach from Chapter 4

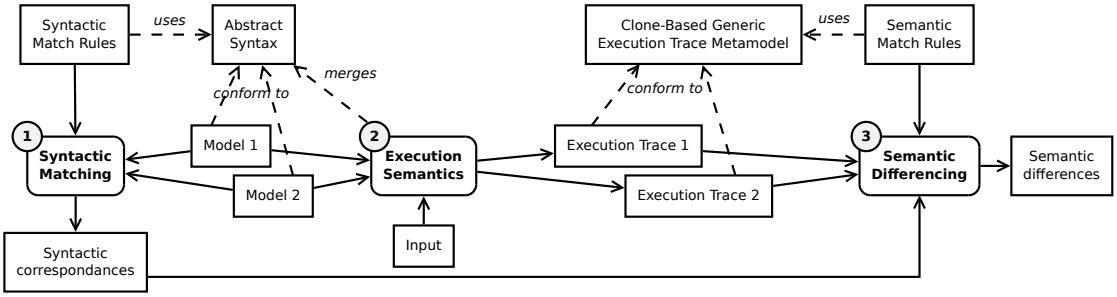


Figure 7.1: Overview of the semantic differencing approach from Langer et al. [99].

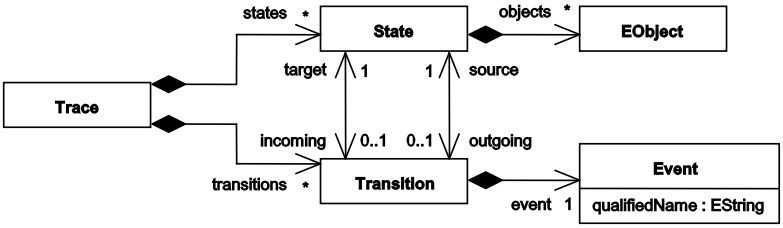


Figure 7.2: Generic clone-based execution trace metamodel, from Langer et al. [99].

In particular, Langer et al. [99] have proposed a generic enumerative semantic differencing approach for xDSMLs that is based on the analysis of execution traces. In this approach, execution traces obtained from relevant executions of two models are compared according to different *equivalence criteria*. For instance, two Petri nets may be considered semantically equivalent if they continuously have the same marking throughout their executions. These criteria, which are specific to the considered xDSML, are defined as *match rules* [97] indicating which syntactic differences among the traces constitute semantic differences among the models.

Figure 7.1 shows an overview of their approach, which consists in three steps.

1. First, **syntactic matching** match rules are used to identify syntactic correspondences between two models. For instance, a transition of a first Petri net model may match a transition in a second Petri net model if it has the same name.

2. Then, both models are **executed** using the execution semantics of the considered xDSML and using some relevant and identical input. Langer et al. [99] propose in their approach to rely on symbolic execution to identify a set of relevant input values to cover most behaviors. Execution traces are produced through an observation of the execution (*e.g.,* instrumented models or semantics).

3. Finally, **semantic differencing** match rules are used to compare the obtained traces, which yield semantic differences between the models.

This semantic differencing approach utilizes a *clone-based generic trace metamodel* for capturing execution traces, which is depicted in Figure 7.2. A trace conforming to

```
1  rule MatchOpaqueActions
2    match left : OpaqueAction
3    with right : OpaqueAction
4  {
5    compare : compareActions(left, right)
6  }
7
8  operation compareActions(left : OpaqueAction, right : OpaqueAction) : Boolean {
9    return (leftOpaqueAction.name = rightOpaqueAction.name);
10 }
```

Listing 7.1: Syntactic match rule for fUML OpaqueAction objects, written in ECL.

this metamodel is a sequence of clones of the model, each stored in a State object created after a step causing a state change.

7.2.2 Application to fUML

We presented the fUML language in the previous chapter (see Section 6.3 of Chapter 6). To illustrate this approach, we consider an equivalence criterion for which two fUML activities are equivalent if all sequences of *action executions* possible in one activity are also possible in the other. More precisely, as we have seen in the execution metamodel, the start and the end of executing an action is captured in an object ActionActivation by a boolean mutable property called `firing`. This property is set to *true* when the execution starts and reset to *false* when the execution ends. Hence, checking the criteria consists in computing the order in which the actions of the activity diagrams are executed (*i.e.,* when `firing` is set to *true*), and in comparing the resulting sequences with each other. Note that multiple actions may be executed in the same execution state, hence an element of a sequence is *set* of actions. Also, for this example, we focus on a single type of actions represented by the OpaqueAction class. Match rules are implemented using the Epsilon Comparison Language (ECL)[2].

Syntactic Match Rule Listing 7.1 shows the single syntactic match rule `MatchOpaqueActions` required for this fUML equivalence criterion. Since we are only interested in OpaqueAction objects, we define that two of them are equivalent if they have the same name.

Semantic Match Rule Listings 7.2 and 7.3 show the semantic match rule `MatchTraces` defined for the generic clone-based trace metamodel shown previously. It is defined as a call to the helper function `compareTraces` (lines 1–6). The latter is defined as a comparison of the sequences of sets of executed actions (lines 8–12). These sequences are computed by the main helper function `getFiringActionsSequence`, which browses the whole sequence of State objects and extracts the actions of interest (lines 14–33).

[2] `https://www.eclipse.org/epsilon/doc/ecl/`

```
rule MatchTraces
match left : Trace
with right : Trace
{
  compare : compareTraces(left, right)
}

operation compareTraces(left : Trace, right : Trace) : Boolean {
  var leftFiringActions : OrderedSet = left.getFiringActionsSequence();
  var rightFiringActions : OrderedSet = right.getFiringActionsSequence();
  return leftFiringActions.matches(rightFiringActions);
}

operation Trace getFiringActionsSequence() : OrderedSet {
  var firingActionsSequence : OrderedSet = new OrderedSet();
  var i : Integer = 0;
  while (i < self.states.size() - 1) {
    var state : State = self.states.at(i);
    var firingActions : Set = state.getFiringActions();
    if (firingActions.size() > 0) {
      if (firingActionsSequence.size() == 0) {
        firingActionsSequence.add(firingActions);
      } else {
        var previousState : State = self.states.at(i-1);
        if (not compareStates(state, previousState)) {
          firingActionsSequence.add(firingActions);
        }
      }
    }
    i = i + 1;
  }
  return firingActionsSequence;
}

operation State getFiringActions() : Set {
  var firingActionActivations : Set = new Set();
  var activityExecution : ActivityExecution = self.getActivityExecution();
  if (activityExecution <> null) {
    firingActionActivations = activityExecution.getFiringActions();
  }
  return firingActionActivations;
}

operation ActivityExecution getFiringActions() : Set {
  var firingActions : Set = new Set();
  if (self.activationGroup <> null) {
    for (nodeActivation : ActivityNodeActivation in self.activationGroup.nodeActivations) {
      if (nodeActivation.isKindOf(OpaqueActionActivation)) {
        var actionActivation : ActionActivation = nodeActivation;
        if (actionActivation.firing) {
          firingActions.add(actionActivation.runtimeModelElement);
        }
      }
    }
  }
  return firingActions;
}
...
```

Listing 7.2: Semantic differencing match rule for fUML using a generic clone-based execution trace metamodel, written in ECL (part 1/2).

```
...

operation State getActivityExecution() : ActivityExecution {
  var activityExecution : ActivityExecution = null;
  var executionEnvironment : ExecutionEnvironment = self.getExecutionEnvironment();
  var locus : Locus = executionEnvironment.locus;
  activityExecution = locus.getActivityExecution();
  if (activityExecution <> null and activityExecution.runtimeModelElement = null) {
    locus = self.getLocus();
    activityExecution = locus.getActivityExecution();
  }
  return activityExecution;
}

operation State getExecutionEnvironment() : ExecutionEnvironment {
  var executionEnvironment : ExecutionEnvironment = null;
  for (object : Any in self.objects) {
    if (object.isKindOf(ExecutionEnvironment)) {
      executionEnvironment = object;
    }
  }
  return executionEnvironment;
}

operation State getLocus() : Locus {
  var locus : Locus = null;
  for (object : Any in self.objects) {
    if (object.isKindOf(Locus)) {
      locus = object;
    }
  }
  return locus;
}

operation Locus getActivityExecution() : ActivityExecution {
  for (extensionalValue : ExtensionalValue in self.extensionalValues) {
    if (extensionalValue.isKindOf(ActivityExecution)) {
      return extensionalValue;
    }
  }
  return null;
}
```

Listing 7.3: Semantic differencing match rule for fUML using a generic clone-based execution trace metamodel, written in ECL (part 2/2).

Order	Version 1 (*false*)	Version 2 (*false*)
1	register	register
2	assign to project	assign to project
3	authorize payment	authorize payment

Table 7.1: Action execution order of the models of Figure 6.3, with a *false* input value.

Version 1 (*true*)	Version 2 (*true*)
register	register
get welcome pack	get welcome pack
assign to project, assign keys, add to website	assign keys, add to website
∅	assign to project
manager interview	manager interview
manager report	manager report
authorize payment	authorize payment

Table 7.2: Comparison of the action execution order of the models of Figure 6.3, with a *true* input value. Each cell contains a *set* of executed actions.

The remainder of the listings only consists of helper functions defined to reach the values of interest within the clone stored in a State object of the trace. In particular, we are interested in the `firing` fields of ActionActivation objects. The latter are stored within a ActivityExecution object, which is contained in a unique Locus object, which itself can be either found in a unique ExecutionEnvironment object or at the root of the clone depending on the situation. All this is handled by the remaining functions: `getFiringActions`, `getActivityExecution`, `getExecutionEnvironment`, `getLocus` and `getActivityExecution`.

We can already observe that because of the complexity of the execution metamodel, defining this match rule requires a considerable of lines of code to navigate to the part of interest. In addition, the rule enumerates all State objects of the execution state, while the activation of actions only concerns a small subset of them.

Usage Let us consider the two versions of the fUML activity shown in Figure 6.3. First, syntactic matching creates correspondances between all action nodes, *e.g.,* register from version 1 is matched with register from version 2. Then, the models are executed with some identical input. In this case, the only parameter of the activity is the boolean `internal`, hence with two possibilities: *true* or *false*. Execution traces conforming to the generic clone-based execution trace metamodel are captured during the execution, and are then analyzed by the semantic match rule.

Table 7.1 shows a comparison of the action execution order of both versions when the input value is *false*. In this situation, both orders are the same, hence the semantic match rule considers both executions equivalent. Then, Table 7.2 shows the execution orders

when the input value is *true*. In such case, a difference is found starting at position 3, because assign to project is executed later in the second version.

7.2.3 Observations

As we have seen, the usage of a generic clone-based trace metamodel has two key implications on the trace analysis:

1. As a state is simply a collection of objects of any type, type checks and type casting are required to analyze the captured execution data. Moreover, it is necessary to navigate among the complete and potentially complex execution metamodel. This implies complex rules that are hard to read and comprehend. Listings 7.2 and 7.3 are a good illustration of this problem: two pages of code are required to implement a quite simple match rule.

2. Analyzing state changes of an executed model requires the traversal of all execution states captured in a trace. This implies an execution time that scales at best linearly to the number of captured states. Such iteration can be seen at line line 17 of Listing 7.2.

In the next section, we propose to enhance this approach by mitigating these issues through the use of generated multidimensional domain-specific trace metamodels.

7.3 Efficient Semantic Model Differencing

7.3.1 Enhancement of the Existing Approach

As proposed above, we have adapted the semantic model differencing approach from Langer et al. [99] so that it relies on execution traces conforming to a generated multidimensional and domain-specific trace metamodels instead of a generic trace metamodel. By doing so, we expect to make semantic differencing match rules easier to write by taking advantage of fact that execution traces arethe domain-specific. In addition, we expect to improve scalability in time by taking advantage of the multiple dimensions of the execution traces.

Figure 7.3 shows an overview of our approach, which adds one step to the original approach shown in Figure 7.1. We have highlighted in green the parts that are changed, new, or affected by our enhancement. We explain the differences below.

- A new prior step is added (labeled (0) in Figure 7.1), which is the generation of a multidimensional and domain-specific execution trace metamodel, as proposed in Chapter 5. The obtained metamodel replaces the former clone-based execution trace metamodel.

- This of course affects the capture of the execution traces, since they must conform to the generated execution trace metamodel.

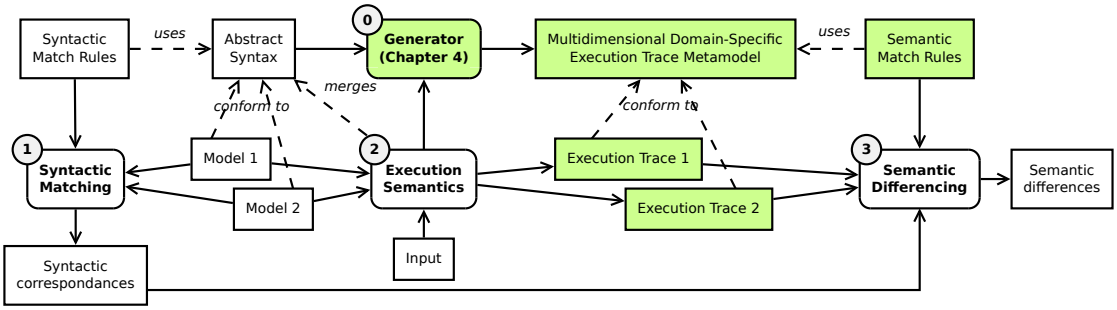


Figure 7.3: Overview of our efficient semantic differencing approach using a generated multidimensional execution trace metamodel. Elements in green are either new or changed as compared to Figure 7.1.

– More importantly, semantic match rules must be adapted to analyze execution traces conforming to the generated multidimensional domain-specific execution trace metamodel. Hence, rules can directly access to domain concepts reified within this trace metamodel, and can iterate on the provided dimensions to avoid exploring the complete execution trace.

7.3.2 Application to fUML

Like for the original approach, we use illustrate our approach using fUML and the models from Figure 6.3. Note that since we consider the same xDSML, we use the same syntactic match rule (shown in Listing 7.1). Moreover, applying our approach yields the exact same results as the original approach, since the same match rule was implemented.

Generation of the Trace Metamodel The first step of our approach is the generation of a multidimensional domain-specific trace metamodel for fUML by following the procedure that we previously described in Section 5.4.2 of Chapter 5. Because many concepts are reified into classes and properties, the resulting execution trace metamodel is quite large, with 56 classes for mutable values and 58 classes for traced objects. Yet, we only need a very small subset of this metamodel for the definition of the match rule.

Figure 7.4 shows an excerpt of the multidimensional domain-specific trace metamodel generated for fUML, by focusing on the parts required for the definition of the match rules. Few classes and properties are required, even though an important number of classes must be shown due to inheritance relationships. The root Trace object contains a sequence of ExecutionState objects, each referencing all corresponding values of mutable fields, *e.g.,* `firingValues`. More importantly, other navigation paths are also provided, such as one TracedOpaqueActionActivation object per OpaqueActionActivation of the executed model. Each one of them contains all the values reached by the corresponding `firing` mutable property within a `firingSequence` inherited from TracedActionActivation. Another property called `runtimeModelElementTrace` is inher-

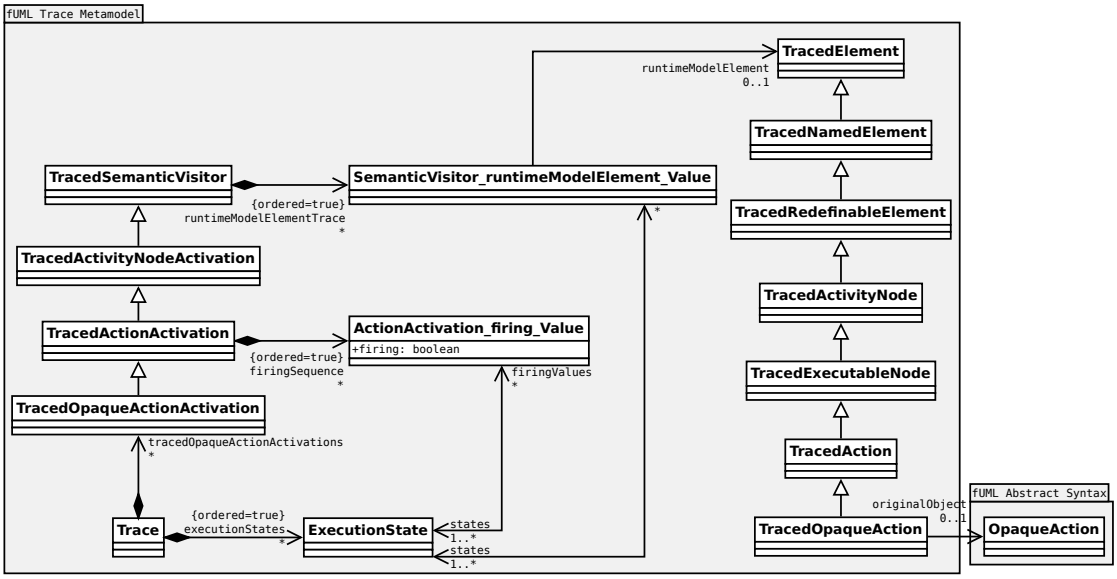


Figure 7.4: Excerpt of the multidimensional domain-specific trace metamodel generated for fUML. Properties names have been simplified for better readability. Classes shown are those used in the match rules from Listing 7.4

ited from TracedSemanticVisitor, allowing to eventually reach the original OpaqueAction object that was executed.

Semantic Match Rule Listing 7.4 shows the semantic match rule defined for the generated multidimensional domain-specific trace metamodel of fUML. The beginning is identical to the match rule from Listing 7.2, with the definition of the rule itself and of the helper function `compareTraces` to compare the action orders (lines 1–12). Again, a helper function `getFiringActionsSequence` is defined to find the action execution order of a given Trace object (lines 14–35).

However, `getFiringActionsSequence` is specific to the generated trace metamodel. Instead of enumerating all ExecutionState objects, the function relies on the `firingSequence` property (line 18), that stores the values taken by the `firing` attribute of each OpaqueActionActivation object. Thereby, only ExecutionState objects in which there were changes in `firing` fields are visited. The main intricacy relies on the fact that each `firingSequence` field may contain the values of `firing` for arbitrary execution states, (*e.g.,* the first and the last one), depending on when the action was executed. Since these sequences are browsed one by one, it is therefore necessary to sort the gathered actions to match the correct global order of execution states. This is done by creating a map (line 15) whose keys are indexes of execution states, and whose values are sets of actions. Later, these keys are sorted (line 30) to reconstruct the correct execution order.

Finally, the last helper function `getAction` retrieves the OpaqueAction object corresponding to a TracedOpaqueActionActivation object (lines 37–47). To do so, it follows

```
rule MatchTraces
match left : Trace
with right : Trace
{
  compare : compareTraces(left, right)
}

operation compareTraces(left : Trace, right : Trace) : Boolean {
  var leftFiringActions : OrderedSet = left.getFiringActionsSequence();
  var rightFiringActions : OrderedSet = right.getFiringActionsSequence();
  return leftFiringActions.matches(rightFiringActions);
}

operation Trace getFiringActionsSequence() : OrderedSet {
  var firingActionsMap : Map = new Map();
  for (tracedAction : TracedOpaqueActionActivation in self.tracedObjects.
       ↪basicActions_tracedOpaqueActionActivations){
    var action : OpaqueAction = tracedAction.getAction();
    for (firingSequence : ActionActivation_firing_Value in tracedAction.firingSequence) {
      if (firingSequence.firing) {
        var state : ExecutionState = firingSequence.states.at(0);
        var stateIndex : Integer = self.executionStates.indexOf(state);
        if (!firingActionsMap.hasKey(stateIndex)) {
          firingActionsMap.put(stateIndex, new Set());
        }
        firingActionsMap.get(stateIndex).add(action)
      }
    }
  }
  var firingActions : OrderedSet = new OrderedSet();
  var sortedStateIndexes : OrderedSet = firingActionsMap.keySet().sortBy(f | f);
  for (index : Integer in sortedStateIndexes) {
    firingActions.add(firingActionsMap.get(index));
  }
  return firingActions;
}

operation TracedOpaqueActionActivation getAction() : OpaqueAction {
  var action : OpaqueAction = null;
  for (runtimeModelElementState : SemanticVisitor_runtimeModelElement_State in self.
       ↪runtimeModelElementTrace) {
    var runtimeModelElement : TracedElement = runtimeModelElementState.runtimeModelElement;
    if (runtimeModelElement <> null and runtimeModelElement.isKindOf(TracedOpaqueAction)) {
      var tracedOpaqueAction : TracedOpaqueAction = runtimeModelElement
      action = runtimeModelElement.originalObject;
    }
  }
  return action;
}
```

Listing 7.4: Semantic differencing match rule for fUML using a generated multidimensional execution trace metamodel, written in ECL.

the runtimeModelElementTrace property— which always contains a single element, since it never changes despite the fact that it is introduced in the execution metamodel—and the `originalObject` property.

7.4 Evaluation

In the following, we present the results of the evaluation and discuss how they give answers to the research questions stated in Section 7.1.

7.4.1 Set-up

We presented in the previous chapter our case study based on fUML and on real world models taken from the case study of Maoz et al. (see Section 6.3 of Chapter 6). For this evaluation, we implemented the operational semantics of fUML using xMOF [114], which required the extension of one class and the definition of 57 new classes. Using these operational semantics, we generated a multidimensional domain-specific execution trace metamodel for fUML.

Our evaluation relies on fUML using the same equivalence criterion as in our illustrations: two fUML activity diagrams are trace equivalent if all sequences of *action executions* possible in one activity diagram are also possible in the other. As we have previously explained, we developed two variants of match rules implementing this criterion: one for performing the analysis on trace models conforming to the generic trace metamodel (Listings 7.2 and 7.3), and one for performing the analysis on trace models conforming to the generated domain-specific trace metamodel (Listing 7.4).

7.4.2 Complexity Reduction (RQ#7.2).

To assess the gain in *usability*, we estimate the complexity of the developed match rules through different metrics: the number of lines of code required, the number of statements, the number of operations, the number of operation calls, the number of loops, the number of type checks and the number of conditionals. Table 7.3 shows the

Elements	**Generic**	**Domain-specific**
Lines of code	90	44
Statements	35	16
Operations	8	3
Operation calls	35	24
Loops	5	4
Type checks	4	1
Conditionals	11	3

Table 7.3: Complexity of the semantic differencing rules of fUML defined for the generic (G) and multidimensional domain-specific (DS) trace metamodel.

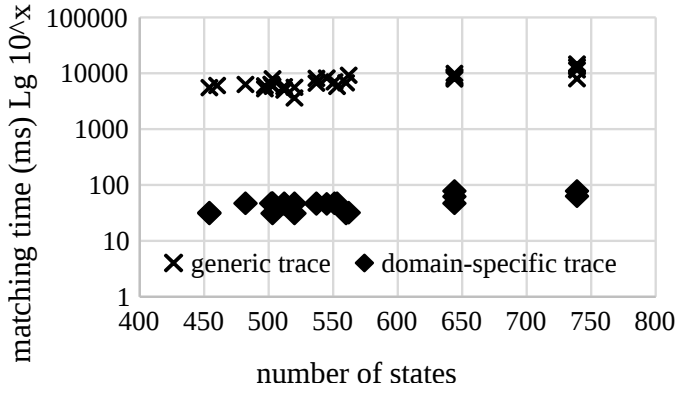


Figure 7.5: Execution time of the semantic differencing rules of fUML for generic and multidimensional domain-specific traces. Each point is a comparison of two execution traces of two versions of a model.

obtained numbers for both sets of rules (generic or multidimensional domain-specific metamodel).

For all elements, we observe a significant reduction of the complexity of the semantic differencing rules. This gain is mainly due to the structure of the generated multidimensional domain-specific trace metamodel. In contrast to the generic trace metamodel, there is no need to traverse the complex data structure of the execution metamodel of fUML, but instead the actions and the evolution of their values can be directly accessed. Only three loops are required for traversing them as well as the trace of the property `firing`, and another loop is required for sorting the action executions chronologically. Other improvements are due to the fact that the trace metamodel is domain-specific, such as type checks that become almost obsolete, since the trace structure precisely captures the concepts required in the trace. These results allow us to answer RQ#7.2 as follows: multidimensional domain-specific trace metamodels seem to simplify the definition of domain-specific trace analyses.

7.4.3 Performance Improvement (RQ#7.1).

To assess the gain in *performance*, we measure the time required to execute both sets of semantic differencing rules on the considered dataset of fUML activities (see Section 6.3.4 of Chapter 6). More precisely, given the sequences of model versions from the dataset, we applied the rules to pairs of models originating from the same model and whose version numbers follow one another. Figure 7.5 shows the execution times obtained for applying the rules on the traces of the considered example models. This experiment was performed on an Intel Core i7-4600U CPU, 2.10GHz, 2.69GHz, with 12GB RAM, running Windows 8.1 Pro. The X-axis of Figure 7.5 shows the number of states contained by the generic and domain-specific traces. The Y-axis shows the measured execution time to perform a comparison of two execution traces, on a logarithmic scale. Each execution time was measured ten times and the arithmetic mean values are shown in the figure.

As can be seen from the measurements, the rules analyzing traces conforming to the domain-specific trace metamodel outperform the match rules analyzing generic traces since they are between 170 and 400 times faster with an average of 250. As we had previously highlighted, the main reason for this result is the multidimensional structure of the domain-specific trace metamodel allowing to efficiently explore the trace through dedicated navigation paths related to specific model elements. Thereby, analyzing the trace does not require the enumeration of all execution states, and only requires to iterate as many times as the elements of interest changed during the execution. These results allow us to answer RQ#7.1 as follows: multidimensional domain-specific trace metamodels seem to enable better execution times for trace manipulations as compared to a generic trace metamodel.

7.5 Conclusion

Developing executable models requires facilities to track the changes that are made to them, and to understand the impact of these changes. In particular, *semantic model differencing* consists in analyzing how changes made to models impact their behavior. This can be accomplished by comparing execution traces captured during the executions of different versions of a model. Such comparisons are based on *equivalence criteria* specific to the considered xDSML, and can be accomplished by defining *semantic differencing match rules*. These rules are domain-specific trace manipulations that specify which differences among execution traces constitute semantic differences among the models.

A set of semantic differencing rules must be defined according to a specific execution trace metamodel. While a generic execution trace metamodel is simple to understand and can be used for any xDSML, it doesn't provide a direct access to the concepts of the xDSML, which hinders the complexity of semantic differencing rules. In addition, it generally requires each execution state to be enumerated even when the rules are based on a subset of the information they contain. To cope with these limitations, we propose the use of multidimensional domain-specific execution trace metamodels, as defined in our previous contribution presented in Chapter 5. Thereby, domain concepts are directly accessible, and execution traces can be explored through a variety of navigation paths. We evaluated our approach using a real-world xDSML, fUML, and a selection of real-world models. Results show that semantic differencing rules are much less complex to define, and that performance is improved by at least 170 times.

Chapter 8

Efficient and Advanced Omniscient Debugging

In this chapter, we present a second application of multidimensional domain-specific execution trace metamodels (presented in Chapter 5) to *omniscient debugging* of executable models [22]. Section 8.1 introduces the context and the objectives of this work. Then, Section 8.2 defines the considered scope of model execution and model debugging through a comparison of the features of interactive debugging approaches, and an enumeration of the services expected by a multidimensional omniscient debugger.

Continuing, Section 8.3 presents our approach to provide generic multidimensional omniscient debugging to any xDSML, which relies on the generation of efficient domain-specific trace management facilities. Section 9.5 briefly presents a prototype supporting the technique in the GEMOC Studio. Section 8.4 discusses the evaluation of our approach. Finally, Section 8.6 concludes the chapter. The work presented in this chapter is the result of a collaboration with Jonathan Corley and Jeff Gray from the University of Alabama.

8.1 Introduction

As we have seen in Section 2.3 of Chapter 2, many efforts aim at providing facilities to design executable Domain-Specific Modeling Languages (xDSMLs) in order to analyze behavioral properties early in the development process. In particular, we introduced *interactive debugging* in Section 2.5 as a common dynamic facility to observe and control an execution in order to look for the cause of a defect. However, standard debugging only provides facilities to pause and step *forward* during an execution, hence requiring developers to restart from the beginning to give a second look at a state of interest. To cope with this issue, we presented *omniscient debugging* in Section 2.6, which is a promising technique that can rely on execution traces to enable free traversal of the states reached by a system, thereby allowing developers to "go back in time."

While most general-purpose languages (GPLs) already have their own efficient standard debugger (*e.g.,* Java[1]) or omniscient debugger (*e.g.,* also Java [128]), developing such a complex tool for any xDSML remains a difficult and error prone task. Despite the specificities of each xDSML, it is possible to identify a common set of debugging facilities for all xDSMLs. Thus, to avoid manual creation of each debugger, a possible solution is to define a generic omniscient debugger that would work for any xDSML. However, handling any xDSML has two main consequences: (1) There is necessarily a trade-off between genericity and efficiency of the debugging operations, since supporting any xDSML requires the use of expensive introspection, conditionals, or type checks to support a wide variety of abstract syntax and runtime data structures. Moreover, since debugging is an interactive activity, *responsiveness* is of primary importance. Hence, a first concern is the *efficiency* of a generic debugger. (2) The execution data structure defined in an xDSML can be arbitrarily complex (*e.g.,* a large object-oriented structure), and therefore difficult to comprehend in a debugging session, especially if the execution leads to a large amount of states. Hence, a second concern is the *usability* of omniscient debugging for xDSMLs; *i.e.,* specific *advanced* facilities are required to manage the complexity and size of the executions. To summarize, the following are key objectives that drive the focus of the work presented in this chapter:

O#1: Providing efficient omniscient debugging facilities, to ensure responsiveness of the debugger.

O#2: Offering advanced omniscient debugging facilities, to improve the usability.

To address O#1, we propose to go from a *generic* omniscient debugger to a *generic meta-approach* to define omniscient debuggers. Such a generative approach can provide an efficient and finely tuned omniscient debugger for any xDSML. Yet, considering a generic set of debugging services for all xDSMLs, both the interface and some underlying logic of a debugger can remain generic without compromising efficiency. Hence, our contribution relies on a *partially generic* omniscient debugger supported by *generated domain-specific* trace management facilities. The trace management facilities include a *domain-specific trace metamodel* that precisely captures the execution state of a model conforming to the xDSML, and a *domain-specific trace manager* providing all the required services to manipulate the execution trace generically. We rely on our contribution presented in Chapter 5 for the generation of execution trace metamodels. Because the trace manager is domain-specific, it is finely tuned to the considered xDSML and to the generated trace metamodel, and hence more efficient than a generic one. To address O#2, our contribution provides *multidimensional omniscient debugging* services, which mix both omniscient debugging services, and advanced facilities to navigate among the *values* of specific elements of the executed model.

We implemented our approach as part of the GEMOC Studio, a language and modeling workbench; and we conducted an empirical evaluation. To evaluate the efficiency

[1] http://docs.oracle.com/javase/8/docs/technotes/tools/unix/jdb.html

of our solution, we assessed its quality with regard to both memory consumption and the time required to run omniscient debugging operations. We compared our approach with two generic omniscient debuggers: one that simulates omniscient debugging by resetting the execution engine and re-executing until the target state is reached, and one that copies the model at each execution step. Obtained results show that our approach is on average 3.0 times more efficient in memory when compared to the second debugger, and respectively 54.1 and 5.03 times more efficient in time when compared respectively to the first and the second debugger.

8.2 Multidimensional Omniscient Debugging

We have already introduced executable Domain-Specific Modeling Languages (xDSMLs) and model execution in Section 2.3 of Chapter 2. We then presented interactive debugging and omniscient debugging in Section 2.5 of the same chapter. In the following section, we first define the scope of our approach regarding the field of omniscient debugging, then we enumerate the expected services of our approach as an extension of omniscient debugging, and finally we present an example scenario.

8.2.1 Comparison of Interactive Debugging Approaches

Interactive debugging of an executable model involves *controlling* the model's execution and observing the states traversed. Figure 8.1 shows four approaches to achieve this, with different levels of control over the execution.

First, regular *interactive debugging* only traverses forward the states reached by the model through application of the model transformation defining the execution seman-

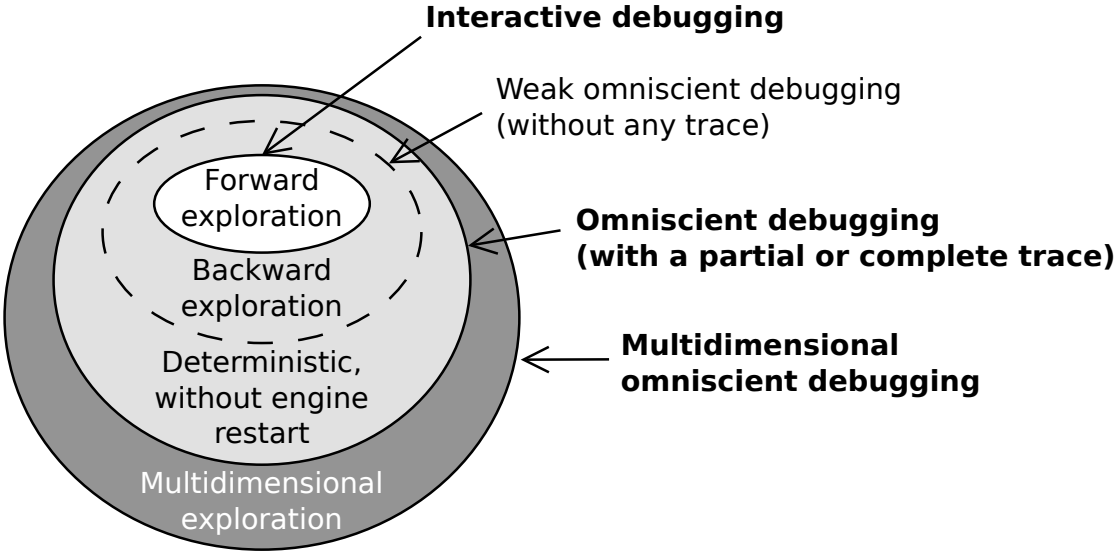


Figure 8.1: Feature comparison of interactive debugging approaches.

tics. Second, we call *weak omniscient debugging* the possibility to go backward in the exploration of the states through a restart of the model transformation until the target prior state is reached again. Note that as we explained in Section 2.6 of Chapter 2, this can be accomplished manually with any interactive debugger through *cyclic debugging* (see Figure 2.10a page 47). Moreover, no execution trace is required.

Third, *omniscient debugging* is an extension of interfactive debugging that relies on capturing an execution trace to by able to revert the executed model into a prior state. Such trace can be partial in the case of reconstructive omniscient debugging, or complete in the case of traced-based omniscient debugging. This makes the procedure deterministic (*i.e.,* the exact same states are visited) even if the model or the operational semantics are non-deterministic.

Finally, our proposal relies on *multidimensional omniscient debugging*, which extends omniscient debugging with facilities to navigate among the values of mutable fields of the model.

8.2.2 Multidimensional Omniscient Debugging Services

While interactive and omniscient debugging can be broadly defined as facilities to control the execution of a model (see Definition 10 page 41 and Definition 11 page 46), precise common facilities are expected to be found in an interactive or omniscient debugger. In the following, we first summarize such common facilities as sets of provided services, then we introduce the additional ones we propose for our approach. Note that all these services are only valid when the execution is *paused*; *i.e.,* when the model transformation waits for instruction before continuing.

Standard Debugging Most debuggers only provide *interactive debugging*, which includes the following forward exploration services:

- ***breakpoint***: pause the execution when a specified condition is true (*e.g.,* a transformation rule is reached).
- ***stepInto***: resume execution and pauses after either executing a single *small step* or moving to the next step encountered in the following *big step*.
- ***stepOver***: resume execution and pause when the next step is completed (including the contained steps, if this is a *big step*).
- ***stepOut***: resume execution and pause when the first step not contained within the current *big step* is reached.
- ***play***: resume execution.
- ***visualization of the current state***: display the values of relevant mutable elements.

Omniscient Debugging To provide exploration of previously visited states, *omniscient debugging* relies on the construction of an execution trace to extend standard debugging with the following services:

- ***jump***: revert the model to a specified state.
- ***backInto***: revert a single *small step* or moves to the last step encountered in a *big step*.
- ***backOver***: revert the last encountered step (including the contained steps, if the last step is a *big step*).
- ***backOut***: revert all the remaining steps within the current *big step*.
- ***playBackwards***: continuously revert execution until the execution is paused or the initial state is reached.
- ***visualization of the trace***: display an interactive representation of the reached execution states and show which state is current.

Multidimensional Omniscient Debugging With the ability to go both forward and backward, a developer can explore any state of a model's execution. Yet, large traces are difficult to navigate practically, and information stored within a state can be arbitrarily complex, compromising usability (O#2). To cope with this issue, we investigate *multidimensional omniscient debugging*; *i.e.,* facilities to navigate among the *values* of the mutable fields of the model:

- ***jumpValue***: jump to the first state in which a given mutable field has a given value.
- ***stepValue***: given a mutable field, jump to the next value of this field.
- ***backValue***: given a mutable field, jump to the previous value of this field.
- ***visualization of the value sequences***: display an interactive representation of the reached values of the mutable fields and show which values are the current ones.

8.2.3 Example Debugging Scenario

Consider a complete execution and debugging scenario with a Petri Net model conforming to the Petri net xDSML introduced in Section 2.3 of Chapter 2, and later summarized in Figure 5.1 of Chapter 5 (page 78). The initial state of the considered Petri Net model is depicted at the left of Figure 8.2 with the label A. First, we set a *breakpoint* in order to pause the execution right after it starts. Then we start the execution and reach the first A state. From there, the next step is an application of `run`. We perform a first *stepInto* (1), which does not change the current state, but presents us with a new next step, which is an application of `fire` on `t1`. We then use *stepInto* a second time (1), which applies

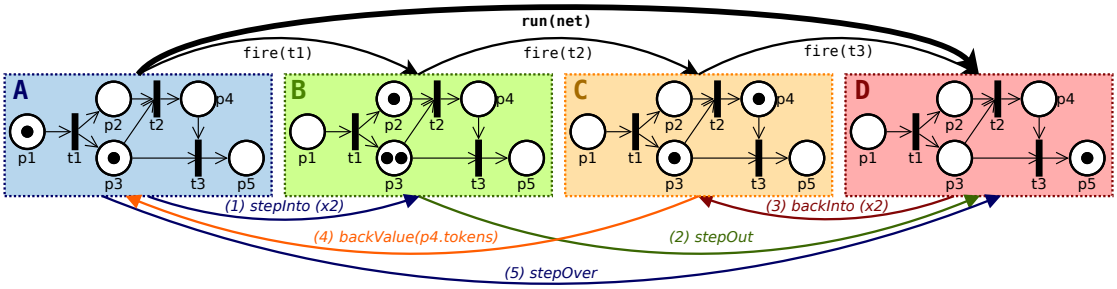


Figure 8.2: Example of Petri Net execution trace annotated with the use of a selection of debugging services.

the `fire` *small step* and brings us to the B state. From there, we use *stepOut* (2) to get out from the current *big step* (*i.e.,* `run`), which brings us to the D state. At this point, the trace is fully constructed, and no additional transformation rules will be applied.

Then, similar to the beginning of the scenario, we apply twice *backInto* (3) to reach the C state. We then use *backValue* (4) to go back to the previous value reached by the `tokens` field of the p4 Place object. While p4 has one token in the C state, its previous amount was zero, which started in the A state. Hence, we reach the A state again. Finally, this time we use *stepOver* (5) to directly follow the first step (*i.e.,* `run`) and we reach the D state again. Note that in this case, *stepOver* should not apply execute any part of the model transformation, but simply read information from the execution trace to directly revert the executed model into the stored D state.

8.3 Efficient and Advanced Omniscient Debugging for xDSMLs

We presented in the previous section the services that define a multidimensional omniscient debugger. This section presents our approach that provides efficient and advanced omniscient debugging for xDSMLs using a partially generic, multidimensional omniscient debugger, supported by generated domain-specific trace management facilities – including a trace metamodel obtained with approach from Chapter 5.

8.3.1 Overview of the Approach

Defining an xDSML implies the definition of a number of domain-specific facilities to edit or analyze a model conforming to the language. In particular, one method to provide a visual animation of a model execution is to observe the model and react to changes. Because such a pattern is common when defining tools for xDSMLs, our approach is designed to have a *single instance* of the executed model loaded at any given time that can be modified throughout the execution and the debugging session.

Figure 8.3 shows an overview of our approach. The idea is to obtain a complete trace-based omniscient debugger for a considered input xDSML. We consider that the

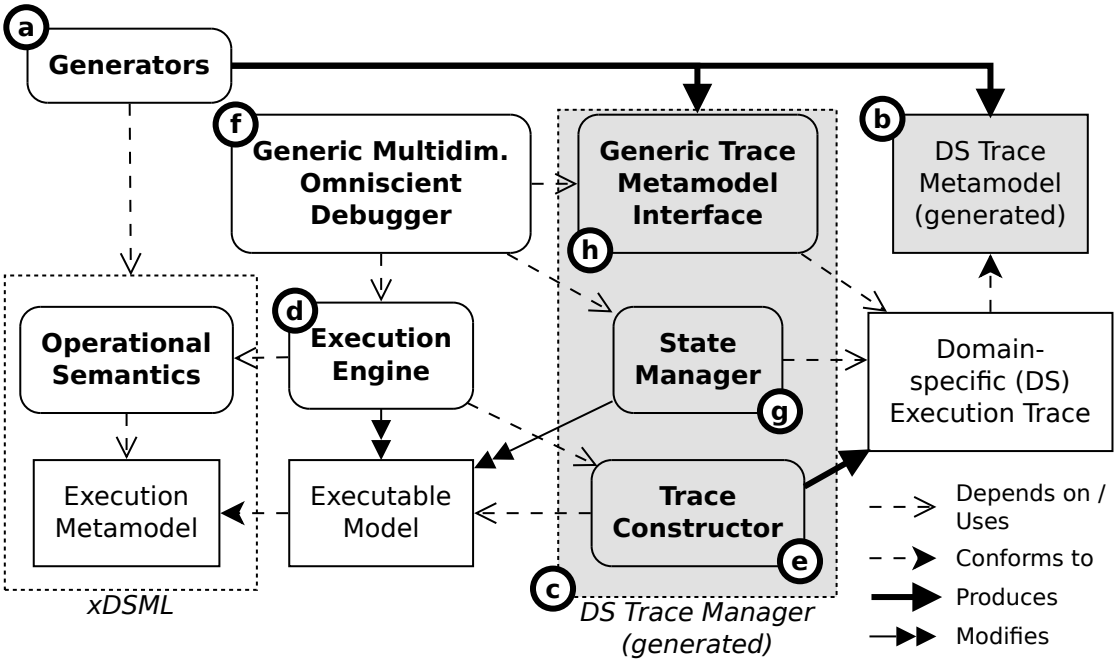


Figure 8.3: Overview of the advanced and efficient omniscient model debugging approach.

initialization function of the xDSML was already applied to an input model, creating the *executable model*. The first step of our approach relies on *generators* (a), which take the considered xDSML as input to produce two domain-specific components: a *trace metamodel* (b) and a *trace manager* (c). The second step is the execution and the debugging of the model. The *execution engine* (d) applies the *operational semantics* to change the model and uses the *trace constructor* (e) from the trace manager to construct a domain-specific trace. The *generic multidimensional omniscient debugger* (f) provides all the services described in Section 8.2 by controlling the execution engine and relying on the *state manager* (g) to revert the model into previous states. Additionally, the debugger relies on the *generic trace metamodel interface* (h) to manipulate the trace.

To illustrate a subset of the interactions between the components shown in Figure 8.3, Figure 8.4 shows a sequence diagram that sketches what happens when a *small step* must be computed and stored in the trace. Duration bars depicted in gray represent changes made in the affected element. First, the *engine* (d) determines the next rule to apply then notifies the *trace constructor* (e) that a small step will occur. As a result, the trace constructor reads the *executed model*, and updates the *domain-specific trace* with new elements accordingly (*e.g.,* add a new *small step* and, if the model was altered, a new state). Finally, the execution engine applies the rule and modifies the executed model accordingly.

We present all of these components in more detail in the remainder of this section.

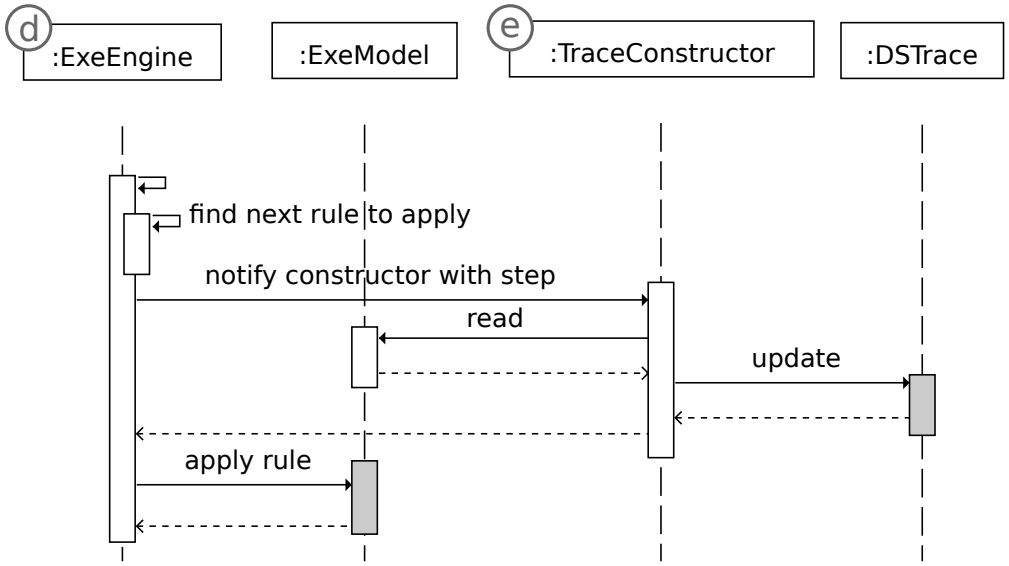


Figure 8.4: Interactions when a *small step* is to be computed and added to the trace.

8.3.2 Execution Engine

First and foremost, an omniscient debugger must provide precise control over the execution of a model, such as the ability to pause during execution or traverse the trace in a controlled manner. As we explained in Section 2.5 of Chapter 2, this requires that the operational semantics of the considered xDSML provides a *control interface*. Because defining such interface and underlying control mechanism for each xDSML is a tedious and error prone task, we propose the definition of a *generic execution engine* (`d` in Figure 8.3) valid for any xDSML and responsible for the application of transformation rules of the operational semantics.

Such component must adhere to certain specifications. The engine must be able to drive the execution of the model (*i.e.,* initialization, start, stop), and to provide to the debugger some control over the execution. This includes the ability to pause the execution at a specific state during execution, and the ability to resume the execution from a paused state. We assume that the engine provides at least the following services:

- ***pauseWhen****:* order to suspend the execution in between two steps as soon as a given predicate is `true`.
- ***isPaused***: return `true` if the engine is paused.
- ***resume***: resume execution (*i.e.,* cancel a pause).

As presented thereafter in Section 9.5 of Chapter 9, the implementation of our approach relies on the execution engine of the GEMOC Studio, which encompasses the aforementioned services.

8.3.3 Multidimensional Domain-Specific Trace Metamodel

Our approach relies on the generation of a *domain-specific trace manager* to create and manipulate execution traces. The first and most central component of this manager is the *multidimensional domain-specific execution trace metamodel* (b in Figure 8.3) of the input xDSML. To obtain this metamodel, we rely on the generation procedure we defined in Section 5.4.2 of Chapter 5.

In the context of omniscient debugging, relying on a multidimensional domain-specific trace metamodel has multiple benefits. First, since it precisely captures the structure of the execution traces of the considered xDSML, it reduces the risk of creating an invalid trace. Second, traces can be stored and used for ulterior domain-specific execution traces manipulations. Third, the multidimensional structure greatly facilitates the definition of multidimensional debugging services in an efficient way.

8.3.4 Trace Constructor

To provide omniscient debugging, we must construct an execution trace during the execution of the model. We have defined the following set of operations to be provided by the *trace constructor* (e in Figure 8.3):

- ***initialize***: create the base elements of the trace.
- ***addState***: add a new state in the trace if a mutable field of the model changed, or if instances of classes introduced in the execution metamodel are created/deleted.
- ***addSmallStep:*** add a *small step* in the trace.
- ***bigStepStarted***: notify that a *big step* has started.
- ***bigStepEnded***: notify that a *big step* has ended.

As we explained in Section 2.3.5 of Chapter 2, the execution of a model consists of a sequence of execution steps, each step originating from the application of a subset of the execution transformation. To capture an execution state that matches a model conforming to the execution metamodel, the operation *addState* must be called just before or after a step.

Since a *big step* is simply a sequence of *small steps*, we only need to capture states before and after *small steps*. However, we also need to capture when steps occur, hence *addSmallStep* must be called at *small step*, while *bigStepStarted* and *bigStepEnded* must be called before and after a *big step*, respectively. In summary, all the calls required to construct the trace are as follows:

- Just before the first *small step*: *initialization*
- Just before a *small step*: *addState*, *addSmallStep*
- After the last *small step*: *addState*

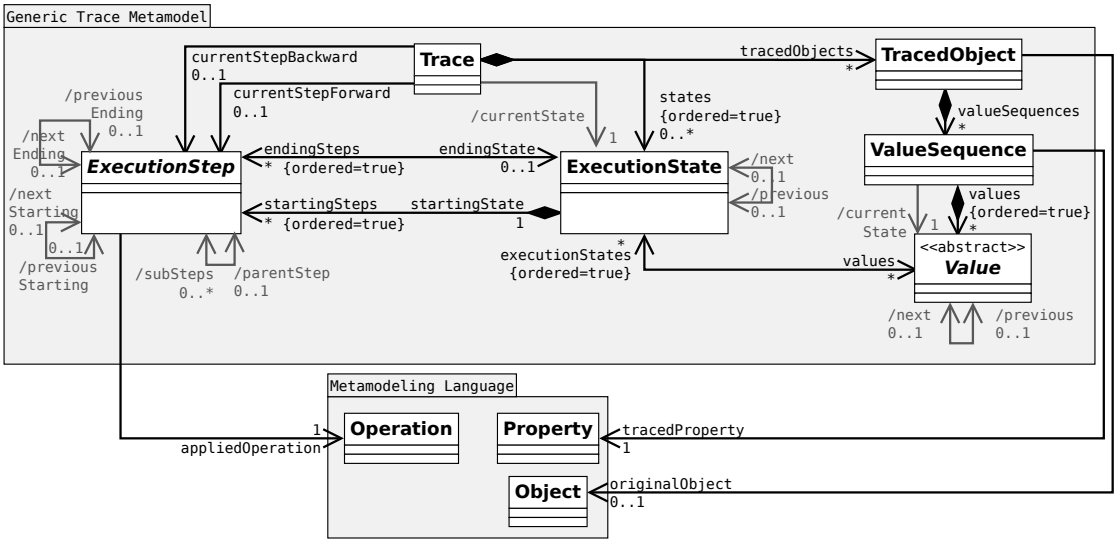


Figure 8.5: Generic trace metamodel interface (simplified).

– Just before a *big step*: *bigStepStarted*

– Just after a *big step*: *bigStepEnded*

8.3.5 Generic Trace Metamodel

Our approach relies on the generation of a domain-specific trace metamodel for the considered xDSML. Since the debugger is generic, an interface must also be defined to manipulate traces in a generic way despite their various possible data structures. For this approach, we defined this structural interface as a *generic trace metamodel* (h in Figure 8.3) specifying all the information that should be accessible within a domain-specific trace. Thus, it has a similar structure to generated domain-specific trace metamodels, except it contains less classes and properties.

Figure 8.5 shows the generic trace metamodel interface. To summarize, we have the same base classes (Trace and ExecutionState) as generated domains specific trace metamodels (*e.g.,* the Petri net trace metamodel shown in Figure 5.2, page 81 of Chapter 5), and classes to represent both steps (ExecutionStep) and values (TracedObject, ValueSequence, Value). Primitive types that extend the Value class (*e.g.,* IntegerValue) are not shown due to space limitations. We use references to elements of the execution metamodel, operational semantics, and executed model: `appliedRule` to specify which rule was applied, `originalObject` to specify which object of the original model is traced by a TracedObject, and `tracedProperty` to specify the property traced by a ValueSequence. Also note that derived properties are defined to facilitate the navigation among the trace, such as `nextState`. Finally, ExecutionStep objects are ordered either by starting time, or by ending time, hence the derived properties `nextStarting` and `previousStarting` for the starting time then `nextEnding` and `previousEnding` for the ending time.

In order to go back and forth through the execution states and steps, a Trace has a reference `currentStepForward` to the ExecutionStep object that represents the next forward execution step, and a similar reference `currentStepBackward` for the next backward step (*e.g.,* to *backOver* the last step handled by the debugger). The current state is accessible with `currentState`, which is derived from `currentStepForward`. Similarly, the property `currentValue` of ValueSequence is indirectly derived from `currentState`.

To provide this interface, our solution relies on the generation of a one-way model transformation from the domain specific trace metamodel to the generic trace metamodel. Thereby, we have a generic read-access to the trace. Regarding write-accesses, we store the debugging state (*e.g.,* `currentState`) in a separate generic structure, hence avoiding the need to modify the domain-specific trace.

8.3.6 State Manager

An omniscient debugger must be able to revisit a previous state by reverting the executed model into the state stored in the execution trace. The operation enabling a debugger to return to a past state is provided by the *state manager* (g in Figure 8.3), which we specified with a single service:

– ***restoreModelToState***: restore the executed model into a given execution state.

The idea is similar to the well-studied *memento* design pattern, albeit at the model level. The *originator* is the model being executed; the *memento* is an execution state of the trace; and the *caretaker* is both the trace and the trace manager.

8.3.7 Domain-Specific Trace Manager

To implement both the trace constructor and the state manager and to generically expose as much information as stated in the generic trace metamodel, our approach relies on the generation of a *domain-specific trace manager* (c in Figure 8.3). The reason for generating this component is *efficiency* (O#1), because trace manipulations can be tuned for both the considered xDSML and the generated domain-specific trace metamodel (introduced in Section 8.3.3).

Consequently, the domain-specific trace manager generation is coupled with the domain-specific trace metamodel generation. Since all generated operations manipulate a trace conforming to this metamodel, a set of traceability links obtained from the generation of the domain-specific trace metamodel is provided to the generator. From there, the main steps of the generation are as follows:

1. Since the systematic base structure of the generated trace metamodels is known from the domain-specific trace metamodel generator, *initialize* can be generated;

2. Since the mutable fields of the execution metamodel and the corresponding classes in the trace metamodel are known, *addState* can be generated. An implementation of this service includes looking for changes among mutable fields then creating a

Service	Definition

jumpToState
(*state* : ExecutionState)
	if *state* ≠ *trace*.`currentState` **then**
	  manager.restoreModelToState(*state*)
	if *state*.`startingSteps` ≠ ∅ **then**
	  trace.`currentForwardStep` ← *state*.`startingSteps`.first()
	else
	  trace.`currentForwardStep` ← `null`
	if *state*.`endingSteps` ≠ ∅ **then**
	  trace.`currentBackwardStep` ← *state*.`endingSteps`.last()
	else
	  trace.`currentBackwardStep` ← `null`

jumpBeforeStep (*step* : ExecutionStep)	jumpToState(*step*.`startingState`) *trace*.`currentForwardStep` ← *step*

jumpAfterStep
(*step* : ExecutionStep)
	if *step*.`endingState` ≠ `null` **then**
	  jumpToState(*step*.`endingState`)
	  trace.`currentBackwardStep` ← *step*

backInto()	jumpAfterStep(*trace*.`currentBackwardStep.previousEnding`)
backOver()	jumpBeforeStep(*trace*.`currentBackwardStep`)

backOut()	**if** *trace*.`currentBackwardStep.parent` ≠ `null` **then**
	  trace.`currentBackwardStep` ← *trace*.`currentBackwardStep.parent`
	  backOver()

playBackwards()	**while** *trace*.`currentBackwardStep.previousEnding` ≠ `null` ∧ ¬*engine*.isPaused() **do** backInto()

Table 8.1: Omniscient debugging services definitions.

state and new values if any change is detected. Likewise, *revertModelToState* can be generated, which relies on links from the trace to the model to restore values and re-create objects.

3. Since the operational semantics and the corresponding step classes in the trace metamodel are known, step creation can be generated. While *addSmallStep* is straightforward, *bigStepStarted* requires stacking big steps that are in progress, and to unstack them in *bigStepEnded*.

4. Finally, since the systematic shape of generated trace metamodels is known, a generic trace metamodel interface can be provided, as defined in Section 8.3.5.

8.3.8 Generic Multidimensional Omniscient Debugger

The last component to define is the *generic multidimensional omniscient debugger* (f in Figure 8.3) that relies on the execution engine to control the current execution, on the state manager to restore previous states, and on the generic trace metamodel interface to manipulate traces.

Tables 8.1 to 8.3 provides a precise definition of each service required for multidimensional omniscient debugging using the services of the three aforementioned required

Service	**Definition**
toggleBreakpoint (p : Predicate)	*engine*.pauseWhen(p)

stepInto()	**if** *trace*.`currentForwardStep.nextStarting` = `null` **then**
	  steppedInto ← *trace*.`currentForwardStep`
	  engine.pauseWhen(*steppedInto*.`nextStarting` ≠ `null`)
	  engine.resume()
	else
	  jumpBeforeStep(*trace*.`currentForwardStep.nextStarting`)

stepOver()	**if** *trace*.`currentForwardStep.endingState` = `null` **then**
	  steppedOver ← *trace*.`currentForwardStep`
	  engine.pauseWhen(*steppedOver*.`endingState` ≠ `null`
	  engine.resume()
	else
	  jumpAfterStep(*trace*.`currentForwardStep`)

stepOut()	**if** *trace*.`currentForwardStep.parent` ≠ `null` **then**
	  trace.`currentForwardStep` ← *trace*.`currentForwardStep.parent`
	  stepOver()

play()	**while** *trace*.`currentForwardStep.nextStarting` ≠ `null` ∧ ¬*engine*.isPaused() **do** stepInto() *engine*.resume()

Table 8.2: Interactive debugging services definitions.

Service	**Definition**
jumpToValue(v : Value)	jumpToState(v.`executionStates`.first())

stepValue
(*valueSeq* : ValueSequence)
	if *valueSeq*.`current.nextValue` = `null` **then**
	  previousValue ← *valueSeq*.`current`
	  engine.pauseWhen(*previousValue*.`nextValue` ≠ `null`
	  engine.resume()
	else
	  jumpToValue(*valueSeq*.`current.nextValue`)

backValue (*valueSeq* : ValueSequence)	jumpToValue(*valueSeq*.`current.previousValue`)

Table 8.3: Multidimensional omniscient debugging services definitions.

components. These components are represented by three singletons: *engine* represents the execution engine, *trace* represents the root element of a model conforming to the generic trace metamodel, and *manager* represents the state manager. In the following paragraphs, we explain the definitions of all the services provided by the debugger defined in the tables.

Jump services Table 8.1 starts with the definition of the most important omniscient debugging service, which is the ability to *jump* to a prior state. Jumping consists of going back to a chosen state in the execution trace, and is accomplished via the *jumpToState* service. First, it uses the *restoreModelToState* service from the state manager to modify the model, then updates the debugger state represented by `currentForwardStep` and `currentBackwardStep`. Additionally, we need to be able to jump back either right *before* or *after* an execution step, which is provided by the services *jumpBeforeStep* and *jumpAfterStep*.

Other omniscient debugging services Next, we define the remaining omniscient debugging services. *backInto*, *backOver* and *backOut* directly rely on *jumps* to reach the correct state. The last service, *playBackwards*, is a loop backwards until either the initial state is reached or the engine is paused.

Standard debugging services Continuing, Table 8.2 define the standard debugging services; *i.e.,* breakpoints and forward stepping. *toggleBreakpoint* provides a generic way to define a breakpoint through a predicate, that can be defined on the model state (*e.g.,* watching for a specific instruction to be reached) or on the trace (*e.g.,* verifying a temporal property or watching for a specific step to be applied). It is defined using the *pauseWhen* service that must be provided by the execution engine. The next services are the standard step operations: *stepInto*, *stepOver*, and *stepOut*. There are two cases to consider: (1) When the current step is at the end of the trace, we rely on *pauseWhen* and *resume* to apply the operational semantics up until the correct situation is reached (*e.g.,* waiting for the current big step to be finished with *stepOver*). (2) When the execution state is at a past state (*e.g.,* after a jump backwards), *jump* services are called (even though these step services are not specific to omniscient debugging) while the engine remains paused.

Multidimensional omniscient debugging services Lastly, Table 8.3 define the final set of services providing multidimensional omniscient debugging facilities. The goal of these services is to provide the capacity to debug a model by following the sequences of values of specific mutable fields, thereby improving the usability of omniscient debugging for xDSMLs (O#2). Implementing these services is simplified by the structure of the trace metamodel providing access to each of the value sequences. Thus, *jumpToValue* is a use of *jumpToState*; and *backValue* directly uses *jumpToValue*; while *stepValue* is very similar to *stepOver*.

8.3.9 Implementation in the GEMOC Studio

We applied our approach to implement a proof-of-concept prototype of generic multidimensional omniscient debugger. More precisely, this prototype is part of the GEMOC Studio, an Eclipse package atop the Eclipse Modeling Framework (EMF) including both a language workbench to design and implement tool-supported xDSMLs as well as a modeling workbench where the xDSMLs are automatically deployed to allow system designers to edit, execute, simulate, and animate their models. Our debugger relies on the execution engine of the GEMOC Studio, and is implemented as a set of *addons* for the engine. A GEMOC engine addon receives information about the execution progress, allowing both to construct the trace and to trigger a pause when it is required. More details about the implementation can be found in Section 9.5 of Chapter 9.

8.4 Evaluation

In this section, we first present the design and results of an empirical study providing an initial evaluation of the efficiency of our approach. Then, we discuss the benefits of multidimensional omniscient debugging.

8.4.1 Efficiency of the Approach

To evaluate the efficiency of our approach (O#1), we considered the following research questions:

RQ#8.1 Is our approach more efficient in memory as compared to a clone-based omniscient debugger?

RQ#8.2 Is our approach more efficient in time for omniscient debugging as compared to a weak omniscient debugger and to a clone-based omniscient debugger[2]?

Thus, our evaluation of efficiency is the comparison of three omniscient debuggers as presented in Section 8.2 and in Figure 8.1. First, *WeakDebugger* is a weak generic omniscient debugger. Such a debugger is expected to be efficient in memory, because there is no trace to store; and inefficient in time, because the execution engine must be restarted at each jump backward. Second, *CloneBasedDebugger* is a clone-based generic omniscient debugger, that constructs a generic trace using *deep cloning*[2] and implements *jumps* using the model differencing library EMF Compare[3]. Because this debugger relies on an execution trace, it is expected to be less efficient in memory and more efficient in time than *WeakDebugger*. Finally, *MultiDimDebugger* is the prototype multidimensional omniscient debugger applying our approach. All three debuggers were implemented in the GEMOC Studio.

[2] without using our scalable cloning approach from Chapter 4

[3] `https://www.eclipse.org/emf/compare/`

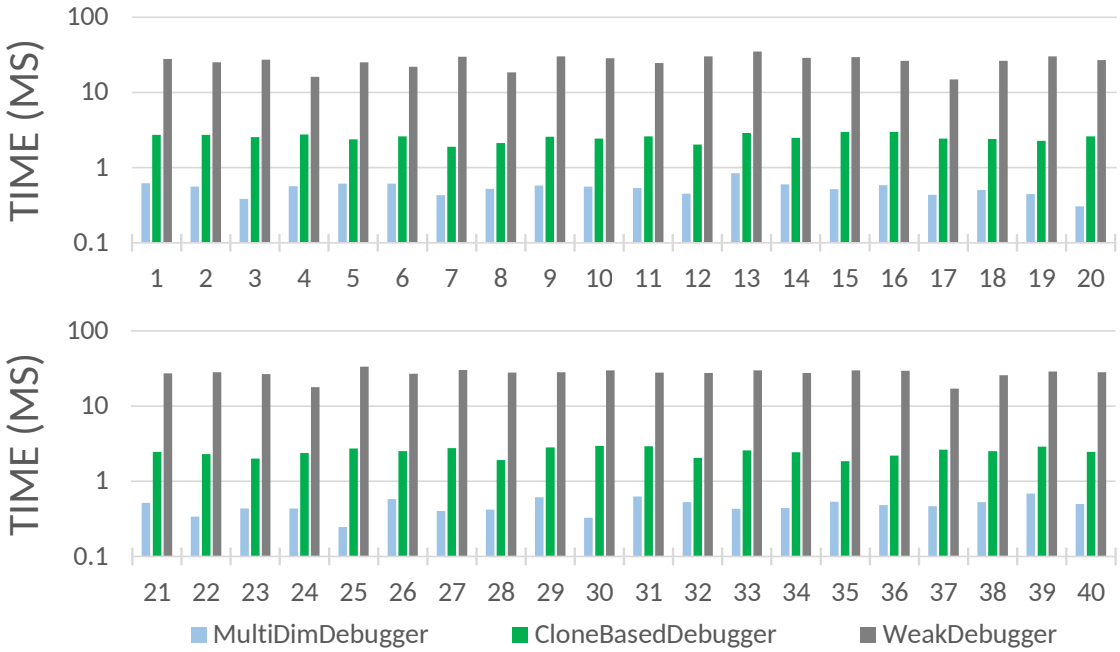


Figure 8.6: Time required to perform a *jumpToState*.

Similarly to the evaluation of our semantic differencing approach described in Chapter 7, we consider a subset of a real-world xDSML, namely fUML [122]. We presented this case study in Section 6.3 of Chapter 6. This time, the xDSML was implemented with the GEMOC Studio using Ecore for the abstract syntax and Kermeta for the operational semantics. In addition, there also, we use models taken from the case study of Maoz et al. [112].

Data Collection and Analysis To compare efficiency in *memory*, instead of observing the memory usage of the complete environment (*e.g.,* execution engine and loaded model), we measured the memory used only by the debugger. More precisely, for each of the considered models, we collected the amount of memory required to store the execution trace at the end of its execution by making precise memory measurements using heap dumps and Eclipse MAT[4].

To compare efficiency in *time*, we focused on the main operation used by all omniscient debugging services: *jumpToState*. More precisely, for each of the considered models, we measured the average amount of time required to perform a *jumpToState* by jumping to each previously visited state once and in a random order. Measures were done using Java's operation `System.nanoTime`.

Data was collected in a reproducible way through a programmatic use of GEMOC Studio's engine. Each result is an average value computed from five identical measurements made using an Intel i7-3720QM CPU with 8GB of RAM.

[4] `https://www.eclipse.org/mat/`

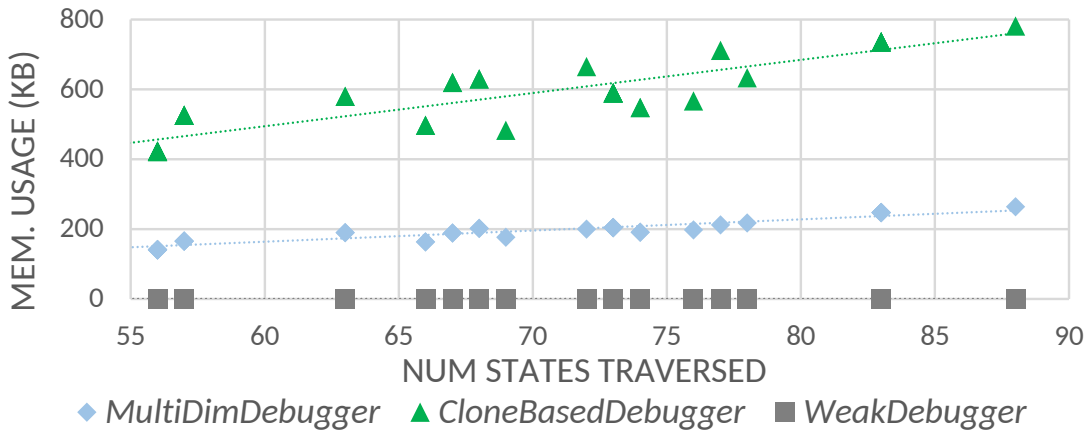


Figure 8.7: Memory used by the execution trace.

RQ#8.1: Efficiency in memory Figure 8.7 shows the results obtained regarding the memory required to store an execution trace. The x-axis shows the number of elements in the trace, while the y-axis shows the amount of memory used in kB. First, *WeakDebugger* does not use memory, because it does not store a trace. Second, we observe that our approach is always more efficient in terms of memory usage than the *CloneBasedDebugger* debugger with 3.0 times improvement on average. We hypothesize this is due to the domain-specific traces obtained with our approach that are designed to only contain the evolution of the mutable fields of the model with minimal redundancy, whereas cloning implies significant redundancy. In addition, we note that our approach has a gentler slope than *CloneBasedDebugger*, which suggests better scalability with large traces. To summarize and answer RQ#8.1, we observe that our approach is more efficient in memory than a clone-based approach.

RQ#8.2: Efficiency in time Figure 8.6 presents the results obtained regarding the average amount of time required to perform a *jumpToState*. The x-axis shows the identifier of the executed model, while the y-axis shows the amount of time in *ms*. First, we observe that trace-based debuggers are always better than *WeakDebugger* (right), with in particular *MultiDimDebugger* (left) being 54.1 times faster than *WeakDebugger*. This is explained by the time required to reset the execution engine. Second, we observe that *MultiDimDebugger* is more efficient than *CloneBasedDebugger* (center) with 5.03 times improvement on average. We hypothesize this is due to the generated trace manager, which contains code specific and tuned to both the xDSML and the generated domain-specific trace metamodel. To summarize and answer RQ#8.2, we observe our approach is more efficient in time than the traceless approach and clone-based approach.

8.4.2 Benefits of Multidimensional Facilities

To ensure the usability of omniscient debugging (O#2), our approach provides *multi-dimensional omniscient debugging*; *i.e.,* facilities to navigate among values of mutable fields of an executed model. In essence, we believe that providing explicit visualization of the dimensions of a trace (see the result in the GEMOC Studio Figure 9.10, page 148 of Chapter 9) and means to traverse such trace according to specific dimensions (*e.g.,* *stepValue*), has a significant positive impact on usability (O#2). To completely validate O#2 requires user experiments to empirically assess the expected benefits of multidimensional facilities. We defer this task to future work.

8.5 Related Work

We presented existing work on model interactive and omniscient debugging in Section 2.5 of Chapter 2. In this section, we focus on approaches that are related to our solution, and we discuss both similarities and differences with our technique.

8.5.1 Generic Omniscient Debugging in MDE

Hegedüs et al. [83] present generic trace exploration tools for executable models which contain similar facilities to an omniscient debugger. However, these techniques are defined for post-mortem analysis rather than use during live sessions, whereas our technique supports live debugging sessions.

Corley et al. [38] propose omniscient debugging facilities for the cloud-based modeling solution AToMPM, in order to step both forward and backward in model transformations executed in an AToMPM runtime. Hence it is focused on debugging the languages supported by the runtime, which are T-Core [149] and MoTif [148]. While such debugger can be used to debug the model transformation of an xDSML, it is not appropriate for debugging models conforming to this xDSML, since it make possible to pause the execution in the middle of an execution step. Moreover, the visualized state would be the one of the execution transformation instead of the one of the model.

8.5.2 Trace Visualization and Debugging in MDE

Existing work on trace visualization, such as MetaViz by Aboussoror et al. [1], or the work of Maoz et al. [110, 111], would be strongly complimentary with our approach. Indeed, while we focus on the backend concern of omniscient debugging, trace visualization is required for the frontend.

The work of Chiş et al. [29, 30] on a Moldable Debugger can be interestingly compared to our work. Indeed, while we provide *generic* debugging operations supported by *domain-specific* trace management facilities, they provide a framework to define *domain-specific* debugging operations and user interfaces. Also, our approach is completely automatic given a well-formed xDSML, whereas manual work is required to extend the

Moldable Debugger to support an xDSML. Yet, both approaches tackle different and independent challenges, and provide very complementary results.

8.6 Conclusion

Omniscient debugging is a promising dynamic V&V approach for xDSMLS that enables free traversal of the execution of a system. While most GPLs already have efficient debuggers, bringing omnicient debugging to any xDSML is a tedious and error-prone task. A solution is to define a purely generic debugger, but this requires managing both *efficiency* and *usability* issues that emerge. The approach we presented relies on generated domain-specific trace management facilities for improved efficiency and provides multidimensional omniscient debugging facilities for improved usability. The debugger relies on an execution engine to control the execution and a generated domain-specific trace manager to provide omniscient services. The states reached during an execution are stored in a trace conforming to a generated domain-specific trace metamodel. We provide a prototype within GEMOC Studio, a language and modeling workbench, and an evaluation performed using the fUML language. We observed an improvement regarding both the memory consumption and the time to perform a *jump*, when compared to two generic omniscient debugger variants.

Chapter 9

Tool Support in the Context of GEMOC

In this chapter, we present the software development that was achieved during this thesis either to improve existing tools or to implement our approaches and applications. Each section presents a different realization. Section 9.1 explains the implementation of our scalable model cloning approach that we presented in Chapter 4. Section 9.2 describes the addition we made to the Kermeta language, which consists of facilities to manage step transformation rules.

Continuing, we present all the work that was integrated in the GEMOC Studio. Section 9.3 presents the studio and an architectural change that was achieved to manage different execution engines. Section 9.4 explains the implementation of our generative approach to define domain-specific multidimensional execution trace metamodels that we presented in Chapter 5. Lastly, Section 9.5 describes the implementation of our multidimensional omniscient debugging approach that we presented in Chapter 8.

9.1 Implementation of Scalable Model Cloning for EMF

This section presents the implementation of our scalable model cloning approach that we presented in Chapter 4. The resulting prototype is an extension to the Eclipse Modeling Framework (EMF) that both generates the required cloning material (*i.e.,* Java classes) to create proxy objects, and provides the cloning operators themselves. Both the source code (EPL 1.0 licensed) and the Eclipse plugins can be found at the following webpage: `http://moclodash.gforge.inria.fr/`.

In the following sections, we first present the challenges that must be faced for implementing this approach, then explain the extension we had to make for EMF, next we present the design-time part of the implementation, and finally we present the run-time part and we summarize the developped Eclipse plugins.

9.1.1 From Theory to Practice

Implementing our cloning approach within an existing modeling framework using a common object-oriented programming language presents three main challenges: extending the framework consistently, determining what are the shareable parts of the metamodel, and creating proxy objects. We explain these challenges in the following paragraphs:

Ensuring consistency Concerning objects creation and manipulation, the EMF considers that, at all time, there is exactly one unique runtime object per object of a model. This strong assumption is used by the framework to implement the behavior of *containment* references, which are themselves important for lots of other behaviors (storage in resources, serialization, etc.). We thus need to customize some parts of EMF libraries in order to make everything work correctly.

Determining shareable parts We defined in Chapter 4 what are shareable properties and (partially) shareable classes of a metamodel. However, these definitions are "mutually recursive", since a class is shareable depending on its properties, and references are shareable depending on the class they point to. We thus need either a fixed-point algorithm or well designed passes over the metamodel in order to compute the shareable parts, which is required for both *ObjShare* and *NoObjShare* strategies.

Creating proxy objects In order to share fields of objects, our approach relies on proxy objects that only contain non-shareable fields and can access to the shareable ones using a reference. Using a prototype-based object oriented language such as JavaScript, as we did when presenting our approach (see Section 4.4.2, page 65 of Chapter 4), one can customize both the fields and the behavior of individual objects at runtime. However, with a class-based object oriented language, such as Java, the fields of an object are determined definitively by its class at design-time. Since we find ourselves in the second case, we need to generate before-hand the classes of the proxy objects.

Additionally, we found no EMF entity that directly matches the concept of *runtime representation* of a model (see Definition 13, page 62 of Chapter 4). One possible candidate is the Resource class, which represents a set of model elements stored in a file. Resource instances must then be gathered in a ResourceSet instance, which is a second candidate. Since it is likely to store a model into multiple files, and that each file yields a Resource, we consider that the runtime representation of a model is represented by a ResourceSet containing all the Resource elements containing all the rutime objects that constitute the runtime representation of a model.

9.1.2 Extending EMF Librairies

As we mentionned in the previous section, to make EMF work with our cloning approach, we need to make it possible for a runtime object to be implied in multiple

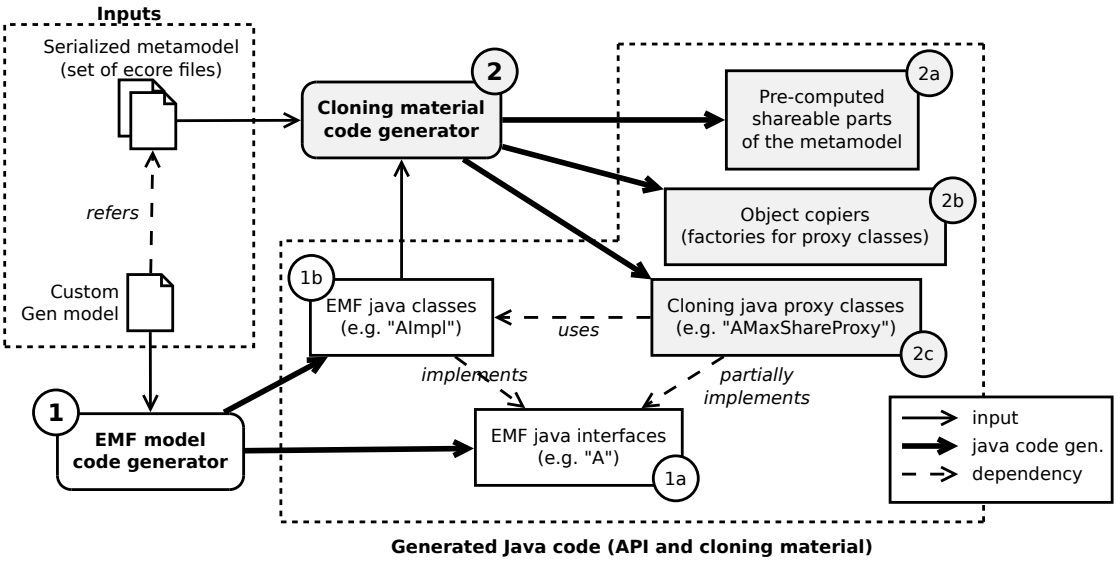


Figure 9.1: Design-time process to generate cloning material for a given metamodel.

runtime representations at the same time, and therefore potentially in multiple containment relationships at the same time. We achieved this by extending some EMF classes through inheritance. Firstly, AbstractShareableEObject is a class that extends the implementation of EObject (reflective API for all objects), in order for the runtime object to possess multiple containers. More precisely, an AbstractShareableEObject can have one container per runtime representation (*i.e.,* ResourceSet) instead of a unique one, and to behave correctly within each runtime representation. Secondly, we implemented the LooseResource class an an extension of the implementation of Resource. This new kind of resource is capable to contain runtime objects that are already contained in other Resource objects, which is also required for our approach to work.

9.1.3 Design-time: Analysis and Code Generation

In order to be able to create proxy objects as required by our cloning approach, we need to generate appropriate Java *proxy classes* beforehand. Figure 9.1 illustrates our design time process to accomplish this generation. At the beginning, at the top left corner, we have the metamodel of interest defined in one or more serialized Ecore models (`.ecore` files), and an EMF generator model (`.genmodel` file) that configures the generation of the Java API corresponding to this metamodel. The `.genmodel file` must be configured to generate implementation classes that all extend our custom AbstractShareableEObject class. From there, the steps of the process are the following (as annotated in Figure 9.1):

1. The EMF model code generator is called with the `.genmodel file` and the serialized metamodel, and generates the following artifacts:

a) Java interfaces, which provide services to manipulate runtime representations of models that conform to the metamodel.

b) Java classes, which implement the interfaces to enable the creation and manipulation of runtime representations.

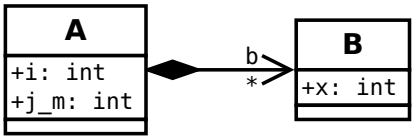
2. Our cloning material generator is called. Its first step is an analysis of the metamodel in order to determine which parts are shareable. In our prototype, mutable properties of the metamodel are specified in the Ecore files using a suffix `_m`, but we plan in the future to externalize this knowledge out of the serialized metamodel. The "mutually recursive" shareable class/property problem mentionned previously is solved by seeing the metamodel as a graph (with classes as vertices and references as edges) and relying on the Tarjan algorithm [150] to compute Strongly Connected Components (SCC). From there, if a SCC contains a mutable property, it means that all its classes are at most partially shareable, and thus that none of its internal references are shareable. The remaining elements are then easily processed. The second step of the cloning material generator is the generation of the following artifacts:

   a) Since we computed which parts of the metamodel are shareable, and since this information is required at runtime, we store this information in a static way into Java classes. This eventually reduces the amount of computation required at runtime.

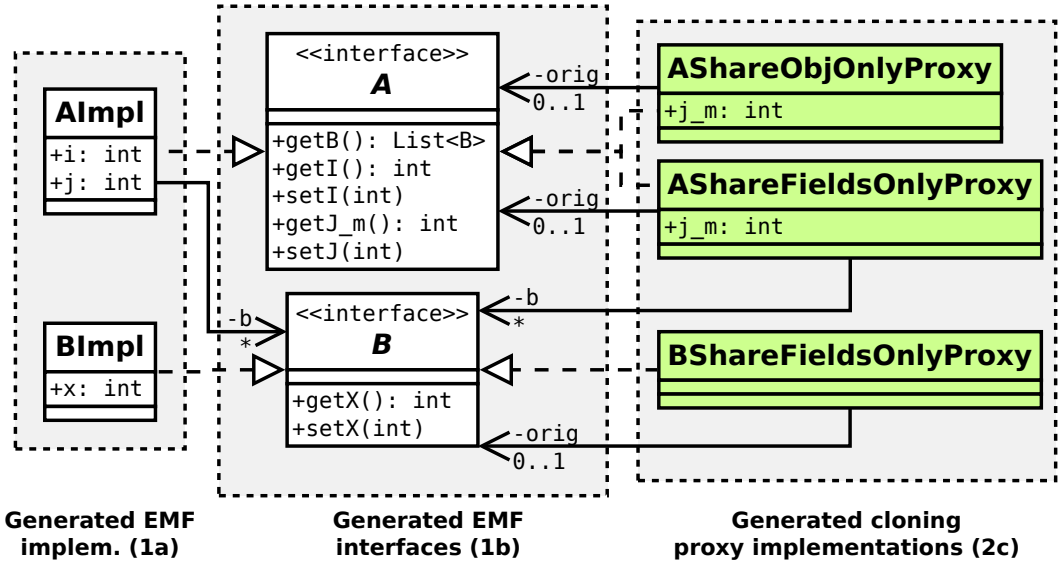
   b) Then, and most importantly, proxy classes are computed by implementing the interfaces previously generated by the EMF generator. Figure 9.2 shows the Java proxy classes that are generated given the same metamodel AB as the one considered in Chapter 4. A class AShareObjOnlyProxy is generated to create proxy A objects when using the *ShareAll* operator, while classes AShareFieldsOnlyProxy and BShareFieldsOnlyProxy do the same for the *ShareFieldsOnly* operator. Figure 9.3 shows a simplified version of the Java code of these classes. Each class contains a property `orig` in order to point to the class containing the immutable properties. A getter that should return an immutable value (*e.g.,* `getI`) is implemented with a call to the original object (*e.g.,* `orig.getI()`). A setter of an immutable property is disabled, since the value should not change at runtime.

   c) To be able to produce proxy objects, we implement the equivalent of the `copyObjectProxy` operation from our approach (see Algorithm 1, page 67 of Chapter 4) in different dedicated copier classes that are able to instantiate proxy classes.

The proxy classes generation is implemented by a Java-to-Java transformation using EMF and the MoDisco toolbox [28]. More precisely, this generation takes as input the Java implementations generated by EMF, and produces modified versions in which non-shareable properties are removed and replaced by a proxy call.



(a) *Input Ecore metamodel AB.*



(b) *Output Java classes and interfaces.*

Figure 9.2: Example of implementation classes after calling both the EMF code generator and our cloning material generator on the metamodel AB.

9.1.4 Runtime: Cloning

We implemented the cloning algorithm itself (see Algorithm 2, page 67 of Chapter 4) within a dedicated class Cloner, which is completely generic and common to all metamodels. The cloning operation is parameterized by a CloningMaterial instance specific to the considered metamodel and cloning strategy. This cloning material includes both the information on which classes and properties are shareable or partially shareable, and a reference to the copier to use for the metamodel and cloning strategy.

9.1.5 Resulting Plugins and Usage

In the end, our tool consists of two sets of eclipse plugins:

- one set for design-time, which includes:
  - the cloning material (*i.e.,* Java code) generator;
  - the graphical interface for the generator, which is an Eclipse *Run Configuration* (shown in Figure 9.4) ;

```
public class AShareObjOnlyProxy extends AbstractShareableEObject implements A {
    ...
    private A orig;
    public int getI()           { return orig.getI(); }
    public int setI(int value)  { /* Disabled: immutable */ }
    public int getJ()           { return this.j_m; }
    public int setJ(int value)  { this.j_m = value; }
    public List<B> getB()       { return orig.getB(); }
    ...
}
public class AShareFieldsOnlyProxy extends AbstractShareableEObject implements A {
    ...
    // Same as AShareObjOnlyProxy, except for getB:
    public List<B> getB()       { return this.b; }
    ...
}
public class BShareFieldsOnlyProxy extends AbstractShareableEObject implements B {
    ...
    private B orig;
    public int getX()           { return orig.getX(); }
    public int setX(int value)  { /* Disabled: immutable */ }
    ...
}
```

Figure 9.3: Code of the generated Java proxy classes (simplified).

– one set for runtime, which includes:

  - the API implemented by the generated code;
  - EMF extensions (*e.g.,* AbstractShareableEObject) used by the generated code;
  - the generic Cloner class.

```
Cloner deepCloningCloner = new ClonerImpl(MyMMDeepCloningMaterial.getInstance());
ResourceSet clone1 = deepCloningCloner.clone(mymodel, "deepClones");

Cloner shareFieldsOnlyCloner = new ClonerImpl(
MyMMShareFieldsOnlyCloningMaterial.getInstance());
ResourceSet clone2 = shareFieldsOnlyCloner.clone(mymodel, "shareFieldsOnlyClones");

Cloner shareObjOnlyCloner =
    new ClonerImpl(MyMMShareObjOnlyCloningMaterial.getInstance());
ResourceSet clone3 = shareObjOnlyCloner.clone(mymodel, "shareObjOnlyClones");

Cloner shareAllCloner = new ClonerImpl(MyMMShareAllCloningMaterial.getInstance());
ResourceSet clone4 = shareAllCloner.clone(mymodel, "shareAlldeepClones");
```

Listing 9.1: Example of usages of the generated cloning material and of the Cloner API, written in Java. The string given to `clone` is the folder in which a clone can be serialized.

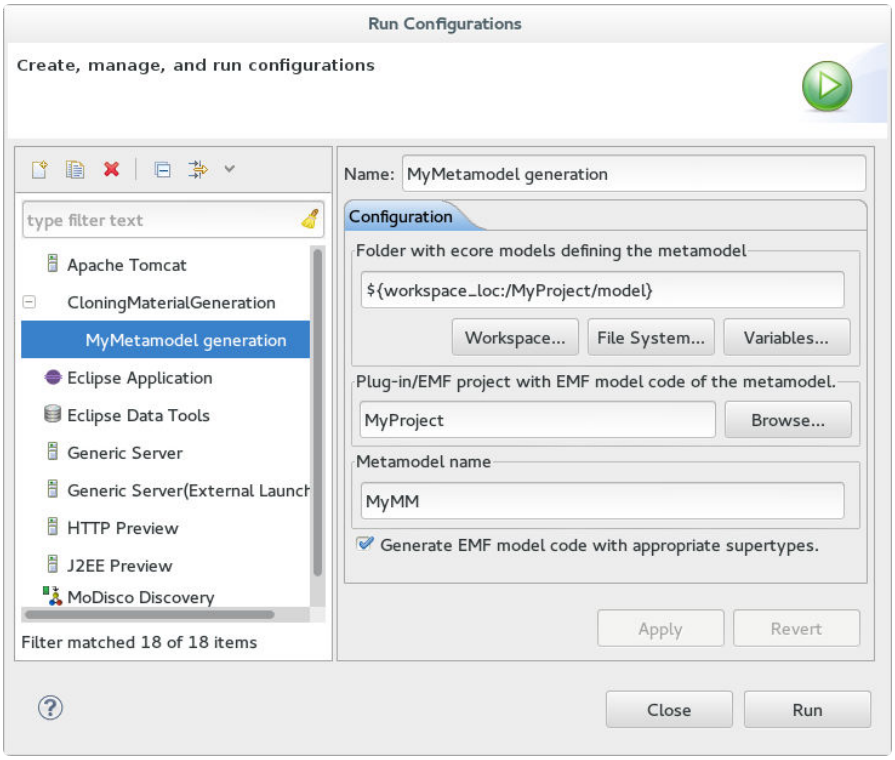


Figure 9.4: Screenshot showing the Eclipse Run Configuration to generate the cloning material of a given annotated metamodel.

These plugins can be loaded in an Eclipse installation and can be used to clone models. At *design-time*, as shown in Figure 9.4, the user can use the graphical interface of the generator to create all the cloning material specific to the considered metamodel. At *runtime*, as shown in Listing 9.1, the user can use the generated cloning material and of the Cloner API to clone models conforming to the considered metamodel.

9.2 Step Management Facilities for Kermeta

In this section, we present an addition that was made to Kermeta during this thesis in order to manage step transformation rules. We first give a more technical description of the implementation of Kermeta, then we explain our extension in the form of a new annotation called `@Step`.

9.2.1 Kermeta: an Extension of Xtend

We already shortly presented Kermeta in Section 2.1.3 of Chapter 2 as a model transformation language. Then we used it in Section 2.3 to define the operational of an xDSML, using aspects both to define the execution metamodel and the execution transformation.

From a more technical point of view, Kermeta is a language designed as an extension of the Xtend language[1], which is part of the Eclipse project. Xtend is dialect of Java that compiles into readable Java source code. It provides new facilities such as type inference, template expressions or operator overloading. Xtend is extensible through so-called *active annotations*. An active annotation is an Xtend mechanism to allow developers to participate in the translation process of Xtend source code to Java code (*e.g.,* to change create additional helper classes, to change the content of methods, etc.). It can also be used as metadata, similarly as Java annotations.

Kermeta consists of a set of Xtend active annotations. The most important annotation is probably `@Aspect`, which allows to reopen existing classes in order to weave additional properties and methods in them. Other annotations aim among others at aligning Kermeta with EMOF concepts, such as `@Composite` that allows to declare an Xtend property as a *containment* reference.

9.2.2 Adding Execution Steps Declaration and Management

We defined in Section 2.3.2 of Chapter 2 the notion of *step transformation rule*, which is an observable rule of the operational semantics of an xDSML. This notion is crucial both for the generation of a multidimensional domain-specific trace metamodel (Chapter 5), and for properly defining stepwise omniscient degugging services (Chapter 8).

However, Kermeta has two main limitations. First, there there is no way to declare which transformation rules of a model transformation should be considered as step rules. Second, as we have shown in the definition of our omniscient debugger in Section 8.3 of Chapter 8, it is necessary for some component (*e.g.,* an execution engine) to be able to perform specific actions *in between* execution steps. And likewise, Kermeta does not provide facilities to delegate such control to an external component.

Adding `@Step` active annotation To cope with these issues, we developed an additional active annotation for Kermeta named `@Step`. While `@Aspect` is targeted at classes, `@Step` is targeted at methods, and allows a language engineer to choose which transformation rules of the operational semantics of an xDSML are step rules. This additional piece of metadata can be used by generative approaches as a source of information, such as for our execution trace metamodel generation procedure from Chapter 5.

Furthermore, since an active annotation allows to customize the translation process of Xtend source code to Java code, we introduced step management facilities to give the possibility to delegate the execution of a step rule to another software component (*e.g.,* an execution engine). This is achieved using two design patterns: *singleton* to have a global registry of *step managers*, and *command* to encapsulate the content of the step method in order to entrust it to a step manager.

Listing 9.2 shows an example of Java code generation using our active annotation `@Step`. An input Kermeta aspect called FooAspect contains a simple method `bar` with the annotation `@Step` (1). The Java code generated for `bar` (2) first creates a StepCommand

[1] `https://www.eclipse.org/xtend/`

```
@Aspect(className="Foo")
class FooAspect {
    @Step
    public def bar() {
        println("Hello world")
    }
}
```

(1) *Example of Xtend aspect containing a method* `bar` *annotated with* `@Step`.

```
@Step
public static void bar(final Foo _self) {
    StepCommand command = new StepCommand() {
        @Override
        public void execute() {
            InputOutput.<String>println("Hello world");
        }
    };
    IStepManager manager= StepManagerRegistry.getInstance().findStepManager(_self);
    if (manager != null) {
        manager.executeStep(_self, command, "bar");
    } else {
        command.execute();
    }
}
```

(2) *Resulting Java code for* `bar`*, with a StepCommand delegated to an IStepManager.*

Listing 9.2: Example of Java code generation from a Kermeta `@Step` annotation.

containing the actual content of `bar`, then uses the StepManagerRegistry singleton to find a IStepManager that is registered as being able to handle the object `_self` that called the method. If a manager was found, it is asked to execute the command. Otherwise, the command is executed, which means that the code still works even without any manager.

9.3 Enhanced GEMOC Studio Execution Framework

In this section, we present an architectural change that was made to the GEMOC Studio during this thesis in order to enable the definition of multiple execution engines. In particular, this change lead to the definition of an execution engine relying on the Kermeta step management facilities that we also developed (see previous section). First we ...

9.3.1 Presentation of the GEMOC Studio

The GEMOC Studio[2] is an Eclipse package atop the Eclipse Modeling Framework (EMF) [145]. Figure 9.5 shows an overview of the two workbenches that composes it. As shown on the upper part, the *language workbench* is used to design and implement

[2] `http://gemoc.org/studio`

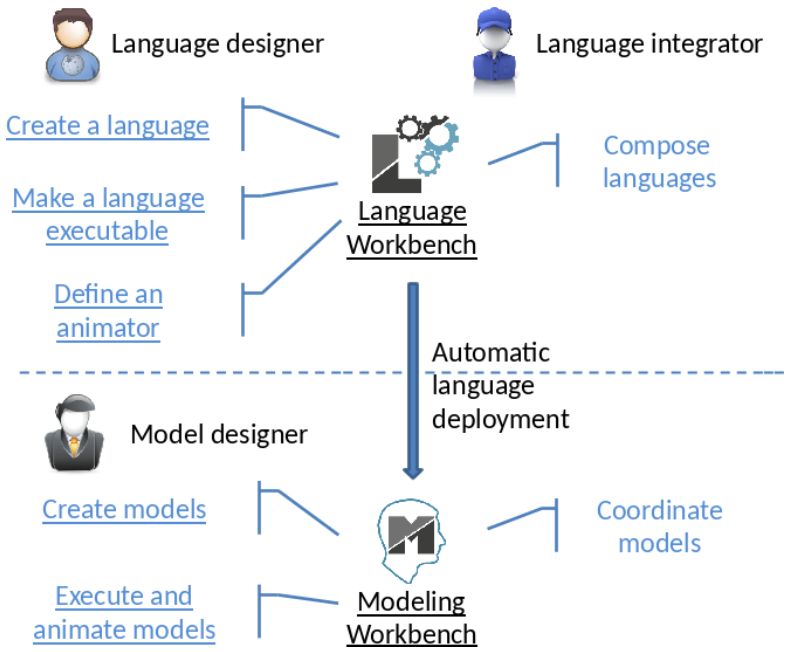


Figure 9.5: Overview of the GEMOC Studio.

tool-supported xDSMLs. This includes defining the abstract syntax (using Ecore), the operational semantics (using Kermeta and/or MoCCML[3]) and the concrete syntax (using Sirius Animator[4]) of an xDSML. In addition, the GEMOC Initiative aims at enabling the composition of xDSMLs by defining how conforming models are coordinated. As shown on the lower part, the *modeling workbench* is where the xDSMLs are automatically deployed to allow system designers to edit, execute, simulate, and animate their models. The modeling workbench includes an advanced *execution engine* that can be used to execute any model conforming to an xDSML defined within the language workbench. It is part of an *execution framework*, which also includes animation and addons facilities.

Execution Framework Figure 9.6 shows an overview of the underlying architecture of the execution framework. On the left, the xDSML designed in the language workbench is depicted. Among other things, it is composed of an abstract syntax, a concrete syntax and operational semantics. At the middle, the model being executed is shown. It conforms to the execution metamodel of the xDSML.

We present in different colors the different parts of the execution framework. At the bottom, in yellow, the *animator* relies on the concrete syntax to display the model to the user continuously. More precisely, the view is updated as soon as the model changes, such

[3]MoCCML is a tool-supported meta-language dedicated to the specification of a Model of Concurrency and Communication (MoCC) and its mapping to a specific abstract syntax and associated execution functions of a modeling language.

[4]`https://www.eclipse.org/sirius/`

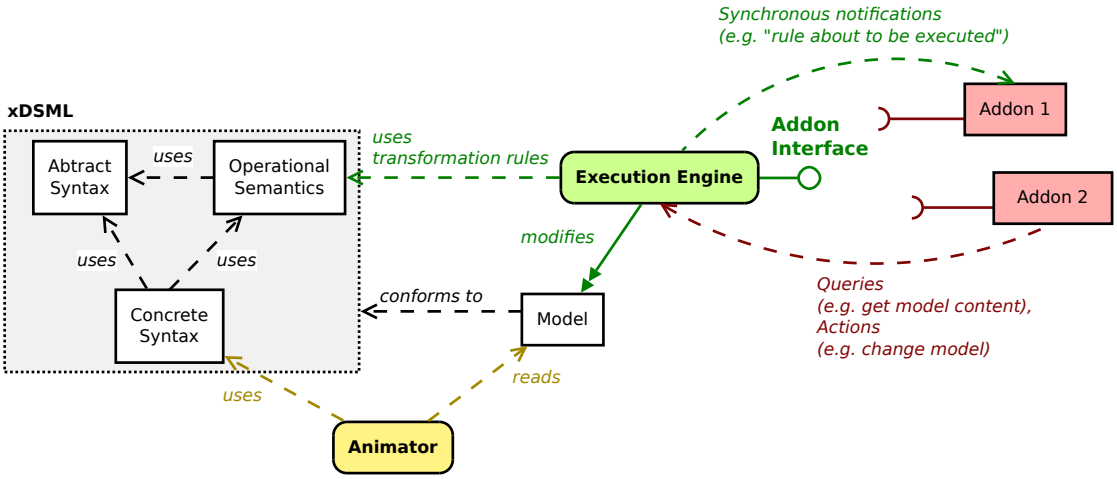


Figure 9.6: Overview of the GEMOC modeling workbench execution framework.

as a modification of the user or an execution step. At the middle, in green, the execution engine applies the transformation rules of the operational semantics to modify the state of the executed model. In addition, it provides an interface to define engine addons, which are mandatory or optional components that get notified by the engine of the progress of the execution (*e.g.,* beginning of the execution, start of a step, end of a step, etc.). Lastly, at the right, in red, addons can be defined to support the execution. By reacting the engine notifications, an addon may query the engine to ask for information, which can be used to provide a view that gets updated when it is necessary, or to log information, or can even control the execution of the model. In particular, because these notifications are synchronous, they make possible for addons to pause an execution when handling a notification, and to modify the model in between execution steps (*e.g.,* for implementing a debugger).

9.3.2 From One to Multiple Execution Engines

The GEMOC execution engine was originally designed for a specific kind of execution transformations defined using two languages: Kermeta for the transformation rules, and MoCCML for model of concurrency. In a nutshell, to execute a model, this execution engine processes the MoCCML model using a solver called Timesquare [45] to compute the series of events that occur in the execution, and may call a specific Kermeta transformation rule at each event occurrence to change the state of the model.

However, as we discussed in Section 2.1.3 of Chapter 2, there are many approaches to define the execution transformation of an xDSML. Each one of these approaches have different characteristics, such as how to initialize and a model transformation programmatically, or how to control the transformation in between execution steps. In order to manage all these different situations, our solution was to enhance the GEMOC execution framework by replacing a unique approach-specific execution engine with an API to de-

fine execution engines. This API defines an engine as a component with two operations: `initialize` to load an xDSML and a mode, and to prepare the transformation; and `execute` to run the transformation. The component is responsible for sending notifications to the different addons during the execution. Using this API, we implemented two main execution engines:

- For operational semantics defined using Kermeta and MoCCML, the `execute` operation consists in a loop that continuously asks for the next event occurrence to the Timesquare solver, and calls the corresponding Kermeta operation when it is required. Notifications are sent to addons during this execution loop.
- For operational semantics that are entirely defined using Kermeta (*e.g.,* the Petri net xDSML defined in Chapter 2), the `execute` operation simply consists in starting the entry point rule of the model transformation (*i.e.,* similarly to a `main` operation). In addition, the engine relies on the Kermeta step management facilities introduced in Section 9.2 (*i.e.,* the `@Step` annotation) in order to register itself as *step manager*. Thereby, the engine can manage the execution of each step, which makes possible for it to interweave notifications to addons in between execution steps.

9.4 Execution Trace Manager Generator in the GEMOC Studio

This section presents the implementation of the execution trace metamodel generation approach that we presented in Chapter 5. We implemented our generator for the Eclipse Modeling Framework (EMF) using the Xtend programming language. The source code (EPL 1.0 licensed) is available at the project web page: `https://gforge.inria.fr/projects/lastragen/`

Figure 9.7 shows an overview of the process. The input is an xDSML defined using Ecore for the abstract syntax, and any supported transformation language for the operational semantics. We currently support Kermeta and xMOF. The output is an Eclipse

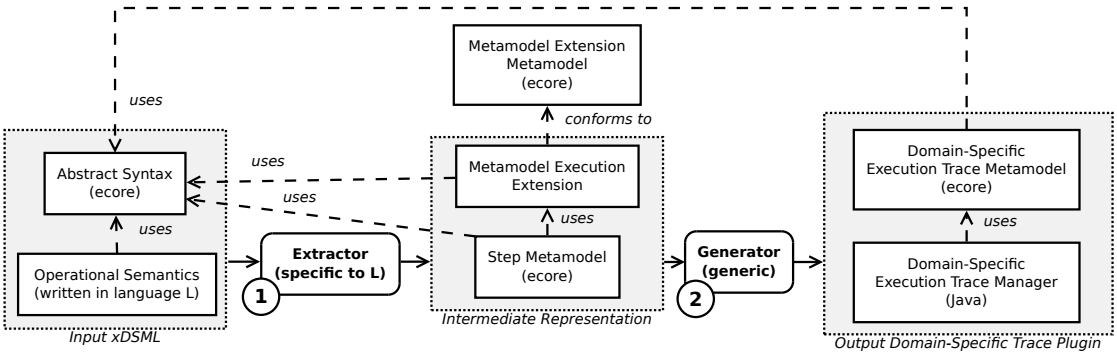


Figure 9.7: Overview of the execution trace management addon generation process.

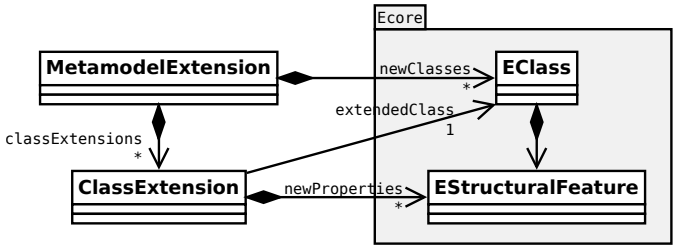


Figure 9.8: Metamodel extension metamodel (simplified).

plugin containing the execution trace metamodel defined using Ecore, and an execution trace manager written in Java. The manager can construct an execution trace and restore an executed model into a prior state. The generator is composed of two steps: one to extract generic information from operational semantics, and a second to generically generate the plugin. We present the two steps thereafter.

9.4.1 Extracting Data from the Operational Semantics

The first step of our process consists in analyzing the considered xDSML in order to extract a intermediate representation containing both the mutable part of the execution metamodel, and the definition of the step transformation rules. While we consider the abstract syntax to be defined using the *de-facto* standard Ecore, there is a large number of different model transformation languages that can be used to implement operational semantics. Hence, a specific information extractor must be implemented for each considered model transformation language (*e.g.,* an analyzer of Kermeta code). For illustration purposes, the considered transformation language is named L in Figure 9.7.

The output of the extraction is composed of two models.

- The first model is a generic representation of the mutable constructs introduced the execution metamodel of the xDSML. Figure 9.8 shows the metamodel that such representation must conform to. The root element is an MetamodelExtension object, which contains both ClassExtension objects to represent mutable properties added to existing classes of the abstract syntax, and EClass objects to represent new classes. For instance, in the case of Kermeta, defining an aspect on a class will yield a ClassExtension object. In the case of xMOF, configuration class that extends a class of the abstract syntax will yield a ClassExtension object. For our implementation to manage a new language, only a new extractor must be provided.
- The second model is a generic representation of the step transformation rules defined in the execution transformation of the xDSML. In essence, it is a metamodel obtained by implementing and using most of Algorithm 4 (page 83 of Chapter 5), which defined how to generate the step classes of the trace metamodel. Similarly to the right part of the Petri net trace metamodel shown is Figure 5.2 (page 81 of Chapter 5), the result is composed of a set of classes, each representing a step transformation rule and the relationships it has with other rules.

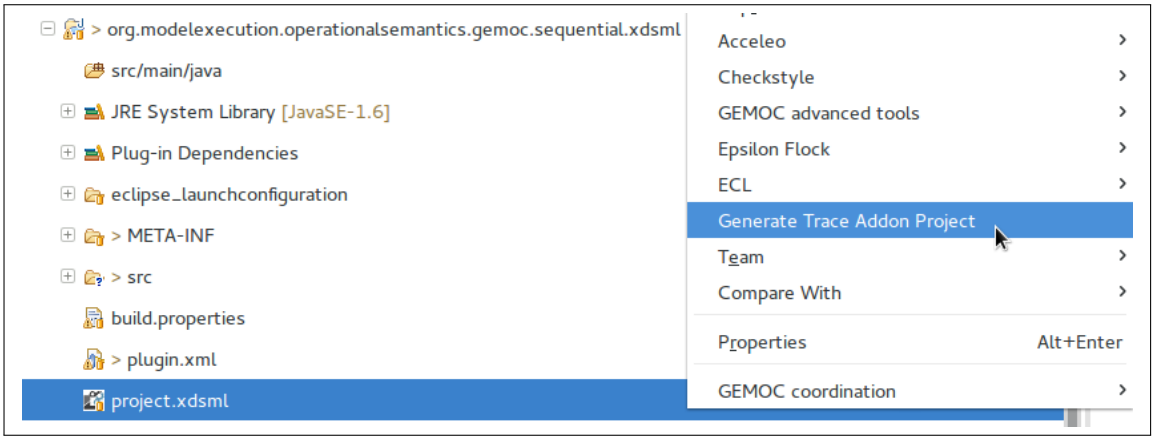


Figure 9.9: Context menu in the GEMOC Studio to generate the domain-specific trace plugin of an xDSML.

9.4.2 Generic Generation of the Trace Metamodel and Manager

The second step of our process consists in generating the execution trace metamodel along with the execution trace manager specific to this trace metamodel. The considered input is composed of the two models produced by the extraction step: the set of mutable properties added by the execution metamodel, and the almost complete step metamodel. In a nutshell, this generator implements Algorithm 3 (page 80 of Chapter 5), which is the procedure to generate a multidimensional domain-specific execution trace metamodel. In addition, it generates a set of Java classes implementing the trace manager, which is a set of operations that manipulate a model conforming to the generated execution trace metamodel. Among others, these operations include `addState` to add a new state in the execution trace given an executed model, and `restoreState` to restore an executed model in a state stored in the trace. Operations are also available to query the trace generically, *e.g.,* to provide a visualization.

9.4.3 Integration in the GEMOC Studio

Our generator has been integrated into the GEMOC Studio, and was made available through a a graphical user interface to trigger the generation for a language defined in the studio. Figure 9.9 shows this interface, which is a context menu that can be triggered on the file defining an xDSML in the language workbench. This file references all the information required to start the generation, *i.e.,* the location of the abstract syntax and the location of the operational semantics.

9.5 Omniscient Model Debugging in the GEMOC Studio

This section explains how we applied a subset the approach we presented in Chapter 8 (*i.e.,* the generative part and a debugger with basic operations) to offer a proof-of-

concept prototype multidimensional omniscient debugger. The prototype is implemented and integrated in GEMOC Studio by relying on its execution engine, and by using the addon mechanism that we introduced in Section 9.3.1. In the following, we present the different components of our prototype: the trace engine addon generator, the generic omniscient debugging addon, and the generic omniscient debugging view addon.

9.5.1 Execution Trace Manager Addon Generator

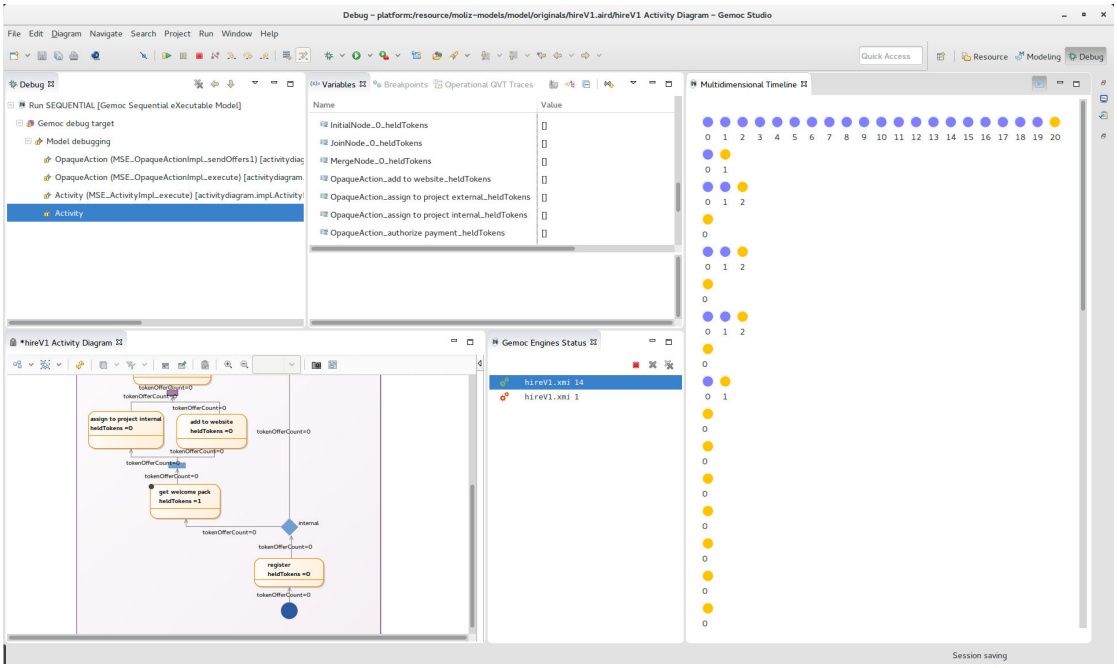
The generative part of our approach takes the form of a *trace engine addon generator* that takes as input an xDSML and that produces a GEMOC engine addon. This generator first relies on the plugin generator that we presented in Section 9.4 to create an Eclipse plugin containing a trace metamodel and a trace manager for the considered xDSML. The trace manager plugin already provides all the services required to implement most of the parts specified in our approach: the *state manager*, the *generic trace interface*, and the *trace constructor* (see Figure 8.3 page 119 of Chapter 8).

However, for the *trace constructor* part to automatically construct a trace during an execution, it must be notified of the execution progress. To that end, additional Java code in generated to configure the trace manager plugin as an addon for the GEMOC execution engine. By handling the notifications sent by the engine, the resulting addon can construct the trace in between execution steps using the trace manager.

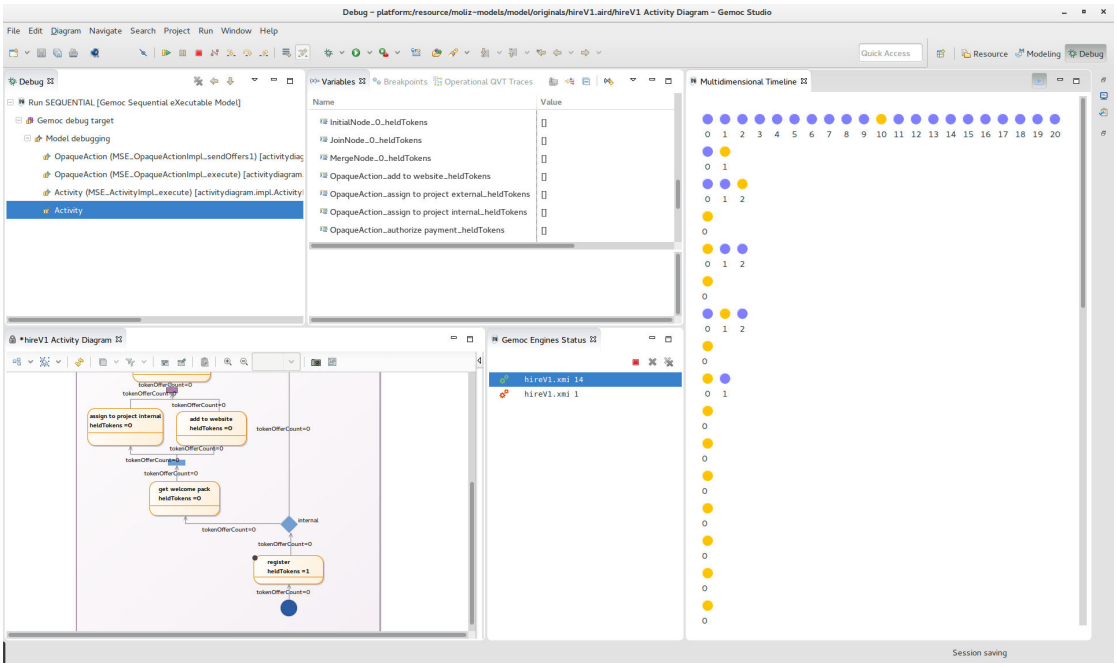
9.5.2 Generic Omniscient Debugging Addons

The generic part of our implementation consists of two generic GEMOC engine addons. The first generic addon contains the debugging logic of the debugger, *i.e.,* the implementation of services described in Tables 8.1 to 8.3 (pages 124 and 125 of Chapter 8). Our prototype provides *toggleBreakpoint* with only one kind of predicate (*i.e.,* a model element is targeted by a step), *stepInto*, *jumpToState*, and *jumpToValue*. By handling notifications sent by the execution engine (*e.g.,* "step started", "step ended"), this addon can decide to pause the execution in between execution steps if a breakpoints is reached or if a step service (*e.g.,* step over) has finished. While paused, the addon can also perform a *jump* to a former execution state by relying on the trace manager. In addition, this addon is integrated with the Eclipse debug view, allowing to benefit from the different buttons to resume, stop, or step into/over/out. Moreover, a stack shows the steps that are being executed. Figure 9.10 is a screenshot showing the GEMOC Studio debugging an fUML activity. At the top-left, the user can visualize the stack of steps being executed while the executed is paused.

The second generic addon is an interactive graphical representation of the stored execution trace. The resulting graphical widget can be seen at the right of the screenshots shown in Figure 9.10. The first line of circles represents the execution trace of the whole model, with each circle representing one execution state reached by the model. The yellow circle shows the current state, while the blue circles show the other states that can possibly be reached either by jumping or by stepping. Double clicking on a blue circle triggers a *jump* action using the generic addon that implements the debugger logic. For



(a) *After multiple forward steps, state 20 is reached and the current node is* get welcome pack.



(b) *After a jump to the state 10, the current node is back to* register.

Figure 9.10: Screenshots showing the GEMOC Studio running an fUML activity with the Eclipse debug view and with the trace visualization addon.

example, Figure 9.10a shows the activity diagram during its execution, which reached the state 20 in the trace, corresponding to the node get welcome pack. Then, Figure 9.10b shows the same execution, but after a *jump* made to the state 10, corresponding to the node register. In addition, when hovering the mouse cursor on a circle, a tooltip shows the all the current values in the state along with all the steps that finished or started in this state. This can also be observed in the variables view of Eclipse debug.

Continuing, all the other lines of circles represent the multiple dimensions of the execution trace. Each one of these lines is the sequence of values reached by a single mutable field of the executed model (*e.g.,* the collection of tokens within one activity node). Similarly to the model state line, a yellow circle represent the current value in time reached by the mutable field. Double clicking on a blue circle of one of these lines triggers a *jumpValue* action, which is again accomplished using the generic addon that implements the debugger logic. Lastly, being an addon, the view is efficiently refreshed only when the engine notifies that a step either started or ended.

Part IV

Conclusion and Perspectives

Chapter 10

Conclusion and Perspectives

10.1 Conclusion

Early dynamic V&V requires models to be *executable*. To that effect, the execution semantics of the DSMLs used to describe them must be defined. The resulting languages are called *executable DSMLS* (xDSMLs). In addition, most dynamic V&V approach require the capture of execution traces from the execution of models. Consequently, providing execution trace management facilities is a major prerequisite to enable the use of dynamic V&V approaches for xDSMLs. Such facilities include the definition of a data structure to represent execution traces, *e.g.,* an execution trace metamodel.

We identified three main challenges regarding execution trace data structures. First, because the execution state of a model can be arbitrarily complex, usability (Ch#1) must be taken into account in order to facilitate the definition of execution trace analyses. Second, scalability in memory (Ch#2) is required because of the large size of execution traces. Third, scalability in time (Ch#3) must be considered for the same reason.

In parallel, we made a number of observations. First, there are different possible kinds of execution trace manipulations. In particular, execution traces can either be manipulated in a generic way, in order to define analyses that are relevant for any xDSML, or in a domain-specific way, in order to define analyses specific to an xDSML. Second, existing execution trace data structures or approaches are either generic or domain-specific, each more appropriate for the corresponding sort of manipulation stated above. Yet, clone-based generic approaches suffer from poor scalability in memory, and the few existing approaches to create domain-specific trace metamodels do not take into account usability or scalability.

In this thesis, we aimed at providing execution trace management facilities while addressing the identified challenges and taking into account these observations. Hence, we made the following two contributions. First, to improve the scalability in memory of clone-based execution traces, we proposed a *scalable model cloning approach*, which relies on data sharing to reduce redundancy among runtime representations of clones. Furthermore, this contribution benefits to model cloning in general, and can thus be

used in many other activities such as design space exploration. Second, to improve usability and scalability of domain-specific execution traces, we proposed *an approach to generate multidimensional execution trace metamodels*. Generating a trace metamodel specific to an xDSML reduces the semantic gap between the trace and the domain, hence improving usability. In addition, a precise capture of the concepts required for execution traces ensures that there is no redundancy nor irrelevant elements in a trace, hence improving scalability in space. Lastly, a multidimensional structure provides additional navigation paths to browse a trace, thereby improving scalability in time.

Next, to validate our second contribution in the context of dynamic V&V, we made two applications to existing V&V approaches and we evaluated them using the fUML language and real-world models from a case study of the literature. First, we enhanced an existing semantic differencing approach in order to rely on a generated multidimensional domain-specific trace metamodel instead of a generic clone-based one. Results show that semantic differencing rules — which are manually written domain-specific trace manipulations — are less complex, and that scalability in time is improved by taking advantage of the multidimensional structure. Second, we proposed a complete advanced omniscient debugging approach for xDSMLs, which relies on a partly generic debugger supported by generated trace management facilities. Such facilities include a multidimensional domain-specific trace metamodel. Results show that the memory footprint is less important when using domain-specific traces is than when using generic clone-based traces. In addition, generated domain-specific trace management facilities are more efficient in time than generic facilities.

Overall, we addressed the challenges we considered in two different contexts, and our contributions improve the state of the art regarding execution trace management. In addition, our two applications not only illustrate the concrete benefits of our second contribution, but also constitute contributions to the fields of semantic differencing and omnicient model debugging.

10.2 Perspectives

This work can be pursued in many different aspects. We present thereafter some direct perspectives in model cloning, execution trace metamodel generation, and model omniscient debugging.

10.2.1 Model Cloning

Helping with the Cloning Operation Choice In Chapter 4, we presented four different model cloning operators, each with different characteristics. A possible research direction is the automation of the choice of a cloning operator. For instance, it must be possible using static analysis of operations to determine whether the reflexive layer is used or not, and more precisely to detect the use of EMF `eContainer()`. This would give the possibility to automatically disable cloning operators that forbid the use of this operation. Another possibility would be the definition of a dynamic analysis to measure

the amount of accesses made to shared elements, since these are the ones responsible for the loss of efficiency. This would provide feedback to the user for choosing the right operator.

Generation of Runtime Classes at Runtime The implementation of our model cloning approach relies on the automatic generation at design time of all the required runtime classes for the operators that share the fields of objects. To improve the usability of our approach, this generation could be done on-the-fly at runtime when the cloning operation is called. Then, generated runtime classes could be compiled and loaded by the execution environment (*e.g.,* using the JVM class loader). This would however have a significant impact on the execution time required by the cloning operation itself.

10.2.2 Execution Trace Metamodel Generation

Customized Domain-specific Execution Trace Metamodels In Section 2.4 of Chapter 2, we showed diverse execution trace data structures, most of which being designed for specific concerns and usages. Yet, the approach we presented in Chapter 5 generates a unique execution trace metamodel specific to some input xDSML.

Although this trace metamodel is more appropriate for domain-specific trace manipulations in general, it could be further customized for a specific set of trace manipulations. For instance, if only a subset of the mutable properties of the xDSML required to define a set of trace manipulations, then not only can execution traces be lightened by not capturing the values of these properties, but the execution trace metamodel itself can be reduced by removing the associated concepts. This would result in a smaller trace metamodel, thereby improving usability for defining the considered set of trace manipulations. Selecting which mutable properties must be considered in the trace metamodel could be accomplished manually, but also automatically by computing the static metamodel footprint [87] of a set of existing trace manipulations.

Use of Domain-Specific Property Languages Multidimensional Domain-specific execution traces metamodels facilitate the definition of domain-specific execution trace manipulations. Yet, these manipulations must still be defined using a generic model transformation language, which is generally designed to be able to describe any kind of model manipulation for any modeling language. Moreover, taking advantage of the multiple dimensions may not always be straightforward.

To improve usability, Rumpe et al. [134] proposed the definition of *domain-specific transformation languages*, each providing concepts to define the transformation of a specific DSML. In a similar fashion, but in the context of dynamic V&V, Meyers et al. [116] proposed the definition of *domain-specific property languages*, each designed for expressing temporal properties for models conforming to a specific DSML. Temporal properties can be used for *offline runtime verification* [102], which is the activity of checking whether or not an execution trace satisfies a temporal property. This can be

accomplished by generating or defining an execution trace manipulation that browses all the states of the trace while continuously checking the compliance with the property.

To improve scalability in time of such manipulations, and to improve usability when implementing a temporal property as an execution trace manipulation, a possible research direction is the combination of domain-specific property languages with multidimensional domain-specific trace metamodels. More precisely, given a domain-specific property, we could generate a trace manipulation taking advantage of a multidimensional domain-specific execution trace metamodel. By detecting relevant pattern within temporal properties, the multiple dimensions of the execution traces could be used. Moreover, since both languages are domain-specific, the semantic gap is considerably reduced, hence facilitating the definition of the generation procedure.

Providing Generic Interface through Metamodel Subtyping Although defining a generic trace manipulation for a domain-specific trace metamodel is possible, it is not possible to directly reuse it for another xDSML. In Chapter 8, to cope with this problem when defining the generic degugging logic, we relied on a *generic multidimensional trace metamodel interface* (see Figure 8.5 page 122). Defining such interface is straightforward because of the similar structure shared by all trace metamodels generated using our approach. However, using such interface in practice in order to manipulate domain-specific traces is not trivial, and required manually written code specific to our omniscient debugger. To generalize the use of this interface, a promising approach would be the definition of a *subtyping relationship* between the generic and the generated domain-specific trace metamodels [76]. With such relationship, a domain-specific trace could be typed as generic trace, and could then automatically benefit from generic trace manipulations defined for the generic trace metamodel.

Representing Interactions with the Environment As we explained in Section 2.3.6 of Chapter 2, in this thesis, we did not consider the possible interactions of a model with its execution environment. Yet, if the operational semantics of an xDSML explicitly defines the possible external stimuli that a conforming model can handle, this information could be taken into account in the corresponding domain-specific execution trace metamodel. This would allow to capture even more precisely the possible information contained in the domain-specific execution traces of a considered xDSML. For instance, a step triggered by some external stimulus could be labeled accordingly.

Branches and State Space In many cases, executing a model multiple times yields different executions, and therefore different execution traces. This is dependent on the operational semantics of the xDSML used to describe the model. In particular, the semantics may handle *input stimuli*, or may contain a *concurrency model* [35]. Yet, different executions may share a common *prefix* before diverging. Consequently, a possibility would be to use a single execution trace to represent a set of executions sharing a common prefix, instead of representing a single one. Storing only once the common prefix of different executions in a single trace would result in a reduction of the memory

footprint. This would require an appropriate trace metamodel allowing the construction of different *branches*, each starting from an execution state of another branch. Hence, a possible research direction is improving the multidimensional domain-specific execution trace metamodel generation procedure to include the idea of branch.

In addition to reducing the memory footprint, branches could have concrete uses in different applications. A first example is model omniscient debugging, for which it would make possible to go back in a previous state, and to decide to take another branch. By pushing the idea even further, a second example is state space exploration, which is the enumeration of the complete transition system corresponding to an executable model. In this case, however, the execution would not be a tree with branches, but a more general *graph* where nodes are execution states, and edges are execution steps.

10.2.3 Model Omniscient Debugging

Domain-Specific Debugging Services In Chapter 8, we presented a model omniscient debugging approach based on *generic* services valid for any xDSML. While the notion of execution state and of execution step seem both universal and revelant whichever the considered domain, additional domain-specific debugging services could be defined for an xDSML. To this end, Chiş et al. [29, 30] proposed the Moldable Debugger framework, which provide facilities to define domain-specific debuggers. As we mentioned in Section 2.5 of Chapter 2, the authors claim that generative approaches can only generate debuggers with generic debugging facilities (e.g., step, step into, stack visualization, etc.), while domain-specific facilities should be defined for the application domain of the xDSML. Hence, following this idea, a possible research direction is to rely on multidimensional domain-specific execution trace metamodels to facilitate the definition of domain-specific omniscient debugging facilities. This would lead to model omniscient debugging facilities that would be domain-specific both regarding its frontend (services), and regarding its backend (execution trace metamodel).

User Study to Evaluate Multidimensional Omniscient Debugging Our omniscient debugging approach proposes advanced facilities to explore an execution according to the multiple dimensions of the trace. We believe that providing explicit visualization of the dimensions of an execution trace and means to traverse such trace according to specific dimensions, has a significant positive impact on usability. Yet, validating this hypothesis requires user experiments to empirically assess the expected benefits of multidimensional facilities, and a research direction is the realization of such user study.

List of Figures

1.1 Graphical representation of the outline of the thesis. Chapters in green contain the core of the scientific contributions. 6

2.1 Petri net abstract syntax. 13
2.2 Example of Petri net model. 14
2.3 Dynamic and static model footprinting, from [87]. 17
2.4 Illustrations of *deep* and *shallow* copying. In each case, grey elements depict the original object graph (*i.e.,* before copying anything). 19
2.5 Translational and operational semantics for an xDSML A. 23
2.6 Abstract syntax and execution metamodel of the Petri Net xDSML. 26
2.7 Illustration of an initialization transformation for Petri net. 28
2.8 Example of Petri net execution trace represented using concrete syntax. . . . 32
2.9 Typical architecture for interactive debugging. 41
2.10 Comparison of interactive debugging with omniscient debugging for re-observing a failure with non-determinism. Inspired by [54]. 47
2.11 Comparison of omniscient debugging approaches. Inspired by [54]. 48

4.1 Example of modeling and EMF usage with a sample metamodel `AB` and a sample model `abb`. 62
4.2 Following Fig. 4.1, *deep cloning* of the model `abb`, which created a new model `abb_clone` along with a new runtime representation in memory. Then `abb_clone` diverged from `abb` by changing its `j` value. 63
4.3 Example of proxy object: p is a copy of o. 67
4.4 Runtime representations of models `abb` and `abb_clone` of Fig. 4.2 obtained with the different cloning operators. 68
4.5 Evaluation process through random metamodel generation. 69
4.6 Memory gain results obtained for 1000 clones. 71
4.7 Manipulation time gain for the *ShareFieldsOnly* and *ShareAll* operators, with varying density of shareable properties in part. shareable classes (log scale). . 72

5.1 Petri net xDSML (reminder from Chapter 2). 78
5.2 Execution trace metamodel generated for the Petri net xDSML. Classes in green are always generated. 81
5.3 Illustration of implicit steps with a simplified `run` step rule. 84
5.4 Example of Petri net model and multidimensional domain-specific trace. . . . 85

6.1 Excerpt of the extended fUML abstract syntax (focus on Activity). 95
6.2 Excerpt of our fUML execution metamodel (focus on ActionActivation). . . . 96
6.3 Example of two different versions of a fUML activity. First *version 1* was developed, then was modified to obtain *version 2*. Figure taken from [112]. . 97

7.1 Overview of the semantic differencing approach from Langer et al. [99]. . . . 101
7.2 Generic clone-based execution trace metamodel, from Langer et al. [99]. . . . 101
7.3 Overview of our efficient semantic differencing approach using a generated multidimensional execution trace metamodel. Elements in green are either new or changed as compared to Figure 7.1. 107
7.4 Excerpt of the multidimensional domain-specific trace metamodel generated for fUML. Properties names have been simplified for better readability. Classes shown are those used in the match rules from Listing 7.4 108
7.5 Execution time of the semantic differencing rules of fUML for generic and multidimensional domain-specific traces. Each point is a comparison of two execution traces of two versions of a model. 111

8.1 Feature comparison of interactive debugging approaches. 115
8.2 Example of Petri Net execution trace annotated with the use of a selection of debugging services. 118
8.3 Overview of the advanced and efficient omniscient model debugging approach. 119
8.4 Interactions when a *small step* is to be computed and added to the trace. . . 120
8.5 Generic trace metamodel interface (simplified). 122
8.6 Time required to perform a *jumpToState*. 128
8.7 Memory used by the execution trace. 129

9.1 Design-time process to generate cloning material for a given metamodel. . . . 135
9.2 Example of implementation classes after calling both the EMF code generator and our cloning material generator on the metamodel AB. 137
9.3 Code of the generated Java proxy classes (simplified). 138
9.4 Screenshot showing the Eclipse Run Configuration to generate the cloning material of a given annotated metamodel. 139
9.5 Overview of the GEMOC Studio. 142
9.6 Overview of the GEMOC modeling workbench execution framework. 143
9.7 Overview of the execution trace management addon generation process. . . . 144
9.8 Metamodel extension metamodel (simplified). 145
9.9 Context menu in the GEMOC Studio to generate the domain-specific trace plugin of an xDSML. 146

9.10 Screenshots showing the GEMOC Studio running an fUML activity with the Eclipse debug view and with the trace visualization addon. 148

List of Tables

2.1 Comparison of a selection of object duplication operators. 20
2.2 Comparison of a selection of execution trace data structures 36

4.1 Cloning operators obtained, one per strategy. 66
4.2 Summary of the characteristics of the cloning operators. 73

7.1 Action execution order of the models of Figure 6.3, with a *false* input value. . 105
7.2 Comparison of the action execution order of the models of Figure 6.3, with a *true* input value. Each cell contains a *set* of executed actions. 105
7.3 Complexity of the semantic differencing rules of fUML defined for the generic (G) and multidimensional domain-specific (DS) trace metamodel. 110

8.1 Omniscient debugging services definitions. 124
8.2 Interactive debugging services definitions. 125
8.3 Multidimensional omniscient debugging services definitions. 125

List of Listings

2.1 In-place model transformation rule that fires a Petri net transition, written in Kermeta. 16
2.2 Definition of the execution metamodel of Petri net through a Kermeta aspect. 26
2.3 Execution transformation for the Petri net xDSML, written in Kermeta. . 30
7.1 Syntactic match rule for fUML OpaqueAction objects, written in ECL. . . 102
7.2 Semantic differencing match rule for fUML using a generic clone-based execution trace metamodel, written in ECL (part 1/2). 103
7.3 Semantic differencing match rule for fUML using a generic clone-based execution trace metamodel, written in ECL (part 2/2). 104
7.4 Semantic differencing match rule for fUML using a generated multidimensional execution trace metamodel, written in ECL. 109
9.1 Example of usages of the generated cloning material and of the Cloner API, written in Java. The string given to `clone` is the folder in which a clone can be serialized. 138
9.2 Example of Java code generation from a Kermeta `@Step` annotation. . . . 141

Author's publications

[19] Erwan Bousse. "Combining Verification and Validation techniques". In: *PhD Students Workshop of the ECMFA, ECOOP and ECSA 2013 Doctoral Symposium*. 2013 (cit. on pp. xii, 2).

[20] Erwan Bousse, Benoit Combemale, and Benoit Baudry. "Scalable Armies of Model Clones through Data Sharing". In: *International Conference on Model Driven Engineering Languages and Systems (MODELS)*. Vol. 8767. LNCS. Springer International Publishing, 2014, pp. 86–301. DOI: 10.1007/978-3-319-11653-2_18 (cit. on pp. xiii, 3, 55, 57).

[21] Erwan Bousse, Benoit Combemale, and Benoit Baudry. "Towards Scalable Multi-dimensional Execution Traces for xDSMLs". In: *Workshop on Model-Driven Engineering, Verification, and Validation (MoDeVVa)*. Vol. 1235. CEUR-WS. CEUR, 2014, pp. 13–18 (cit. on pp. xii, 2).

[22] Erwan Bousse, Jonathan Corley, Benoit Combemale, Jeff Gray, and Benoit Baudry. "Supporting Efficient and Advanced Omniscient Debugging for xDSMLs". In: *International Conference on Software Language Engineering (SLE)*. ACM, 2015 (cit. on pp. xv, xvi, 5, 93, 94, 113).

[23] Erwan Bousse, Tanja Mayerhofer, Benoit Combemale, and Benoit Baudry. "A Generative Approach to Define Rich Domain-Specific Trace Metamodels". In: *European Conference on Modeling Foundations and Applications (ECMFA)*. Vol. 9153. Springer International Publishing, 2015, pp. 45–61. DOI: 10.1007/978-3-319-21151-0 (cit. on pp. xiv–xvi, 4, 5, 55, 75, 94, 99).

[24] Erwan Bousse, David Mentré, Benoît Combemale, Benoît Baudry, and Takaya Katsuragi. "Aligning SysML with the B method to provide V&V for systems engineering". In: *Workshop on Model-Driven Engineering, Verification and Validation (MoDeVVa)*. ACM, 2012. ISBN: 9781450318013 (cit. on pp. xi, 1).

[33] Benoit Combemale, Julien Deantoni, Olivier Barais, Arnaud Blouin, Erwan Bousse, Cédric Brun, Thomas Degueule, and Didier Vojtisek. "A Solution to the TTC'15 Model Execution Case Using the GEMOC Studio". In: *Transformation Tool Contest (TTC)*. 2015 (cit. on pp. xvi, 5).

Bibliography

[1] El Arbi Aboussoror, Ileana Ober, and Iulian Ober. "Seeing errors: model driven simulation trace visualization". In: *International Conference on Model Driven Engineering Languages and Systems (MODELS)*. Vol. 7590. LNCS. Springer Berlin Heidelberg, 2012, pp. 480–496. DOI: 10.1007/978-3-642-33666-9_31 (cit. on pp. 34, 130).

[2] Hiralal Agrawal, Richard a. DeMillo, and Eugene H. Spafford. "Debugging with Dynamic Slicing and Backtracking". In: *Software - Practice and Experience* 23.6 (1993), pp. 589–616. ISSN: 00380644. DOI: 10.1002/spe.4380230603 (cit. on p. 46).

[3] Luay Alawneh and Abdelwahab Hamou-Lhadj. "MTF: A scalable exchange format for traces of high performance computing systems". In: *International Conference on Program Comprehension (ICPC)*. IEEE, 2011, pp. 181–184. ISBN: 9780769543987. DOI: 10.1109/ICPC.2011.15 (cit. on p. 36).

[4] Ra Aydt. *The Pablo self-defining data format*. Tech. rep. Department of Computer Science at the University of Illinois at Urbana-Champaign, 1992 (cit. on pp. 36, 38, 54).

[5] Christel Baier and Joost-Pieter Katoen. *Principles Of Model Checking*. Vol. 950. MIT Press Cambridge, 2008, pp. I–XVII, 1–975. ISBN: 9780262026499. DOI: 10.1093/comjnl/bxp025 (cit. on p. 31).

[6] Nils Bandener, Christian Soltenborn, and Gregor Engels. "Extending DMM Behavior Specifications for Visual Execution and Debugging". In: *Software Language Engineering (SLE)*. Vol. 6563 LNCS. Springer Berlin Heidelberg, 2010, pp. 357–376. ISBN: 9783642194399. DOI: 10.1007/978-3-642-19440-5_24 (cit. on pp. 24, 25, 44).

[7] Earl T. Barr and Mark Marron. "TARDIS: Affordable Time-Travel Debugging in Managed Runtimes". In: *International Conference on Object Oriented Programming Systems Languages & Applications (OOSPLA)*. ACM, 2014. ISBN: 9781450325851. DOI: 10.1145/2660193.2660209 (cit. on p. 49).

[8] David Basin, Germano Caronni, Sarah Ereth, Matúš Harvan, Felix Klaedtke, and Heiko Mantel. "Scalable Offline Monitoring". In: *Runtime Verification (RV)*. Vol. 8734. LNCS. Springer International Publishing, 2014, pp. 31–47. ISBN: 978-3-319-11164-3. DOI: 10.1007/978-3-319-11164-3_4 (cit. on p. 31).

[9] Réda Bendraou, Benoit Combemale, Xavier Crégut, and Marie Pierre Gervais. "Definition of an executable SPEM 2.0". In: *Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2007, pp. 390–397. ISBN: 0769530575. DOI: 10.1109/APSEC.2007.38 (cit. on p. 42).

[10] Réda Bendraou, Jean-Marc Jézéquel, and Franck Fleurey. "Combining Aspect and Model-Driven Engineering Approaches for Software Process Modeling and Execution". In: *International Conference on Software Process (ICSP)*. Vol. 5543. LNCS. Springer Berlin Heidelberg, 2009, pp. 148–160. ISBN: 9783642016790. DOI: 10.1007/978-3-642-01680-6_15 (cit. on p. 22).

[11] Amine Benelallam, Massimo Tisi, and David Launay. "Neo4EMF, a Scalable Persistence Layer for EMF Models". In: *European Conference on Modeling Foundations and Applications (ECMFA)*. Vol. 8569. LNCS. Springer International Publishing, 2014, pp. 230–241. ISBN: 9783319091945. DOI: 10.1007/978-3-319-09195-2_15 (cit. on pp. xiii, 2).

[12] Jean Bézivin, Olivier Gerbé, J. Bezivin, and O. Gerbe. "Towards a precise definition of the OMG/MDA framework". In: *International Conference on Automated Software Engineering (ASE)*. ASE 2011. IEEE, 2001, pp. 273–280. ISBN: 0-7695-1426-X. DOI: 10.1109/ASE.2001.989813 (cit. on p. 12).

[13] Jean Bézivin, Frédéric Jouault, and David Touzet. "Principles, standards and tools for model engineering". In: *International Conference on Engineering of Complex Computer Systems (ICECCS)*. IEEE, 2005, pp. 28–29. ISBN: 0-7695-2284-X. DOI: 10.1109/ICECCS.2005.68 (cit. on p. 12).

[14] Gordon Blair, Nelly Bencomo, and Robert B. France. "Models@ run.time". In: *Computer* 42.10 (2009), pp. 22–27. ISSN: 0018-9162. DOI: 10.1109/MC.2009.326 (cit. on p. 22).

[15] Arnaud Blouin, Benoît Combemale, Benoit Baudry, and Olivier Beaudoux. "Kompren: modeling and generating model slicers". In: *Software and Systems Modeling (SoSyM)* 14.1 (2012), pp. 321–337. ISSN: 16191366. DOI: 10.1007/s10270-012-0300-x (cit. on p. 14).

[16] Barry W. Boehm, Robert K Mcclean, and D. B. Urfrig. "Some experience with automated aids to the design of large-scale reliable software". In: *IEEE Transactions on Software Engineering* SE-1.1 (1975), pp. 125–133. ISSN: 03621340. DOI: 10.1145/390016.808430 (cit. on pp. xi, 1).

[17] Barry Boehm and Victor R. Basili. "Software Defect Reduction Top 10 List". In: *Computer* 34.1 (2001), pp. 135–137. DOI: 10.1109/2.962984 (cit. on pp. xi, 1).

[18] Bob Boothe. “Efficient algorithms for bidirectional debugging”. In: *Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2000, pp. 299–310. ISBN: 1-58113-199-2. DOI: 10.1145/358438.349339 (cit. on pp. 46, 49).

[25] Daniel Brand and Pitro Zafiropulo. “On Communicating Finite-State Machines”. In: *Journal of the ACM* 30.2 (1983), pp. 323–342. ISSN: 00045411. DOI: 10.1145/322374.322380 (cit. on p. 29).

[26] Manfred Broy. “The ’Grand Challenge’ in Informatics: Engineering Software-Intensive Systems”. In: *Computer* 39.10 (2006), pp. 72–80. ISSN: 00189162. DOI: 10.1109/MC.2006.358 (cit. on pp. xi, 1).

[27] Cédric Brun and Alfonso Pierantonio. “Model Differences in the Eclipse Modeling Framework”. In: *UPGRADE, The European Journal for the Informatics Professional* 9.June (2008), pp. 28–34. ISSN: 15322157. DOI: 10.1016/j.ejso.2009.08.008 (cit. on p. 99).

[28] Hugo Bruneliere, Jordi Cabot, Frédéric Jouault, and Frédéric Madiot. “MoDisco: A Generic And Extensible Framework For Model Driven Reverse Engineering”. In: *International conference on Automated Software Engineering (ASE)*. ACM, 2010, pp. 173–174. ISBN: 9781450301169. DOI: 10.1145/1858996.1859032 (cit. on pp. 69, 136).

[29] Andrei Chiş, Tudor Gîrba, and Oscar Nierstrasz. “The Moldable Debugger: a Framework for Developing Domain-Specific Debuggers”. In: *International Conference on Software Language Engineering (SLE)*. Vol. 8706. LNCS. Springer International Publishing, 2014, pp. 102–121. DOI: 10.1007/978-3-319-11245-9_6 (cit. on pp. 45, 130, 157).

[30] Andrei Chiş, Tudor Gîrba, Oscar Nierstrasz, and Marcus Denker. “Practical domain-specific debuggers using the Moldable Debugger framework”. In: *Computer Languages, Systems & Structures* 44.Part A (2015), pp. 89–113. ISSN: 14778424. DOI: 10.1016/j.cl.2015.08.005 (cit. on pp. 45, 130, 157).

[31] Benoit Combemale, Xavier Crégut, Pierre-Loïc Garoche, and Xavier Thirioux. “Essay on Semantics Definition in MDE - An Instrumented Approach for Model Verification”. In: *Journal of Software* 4.9 (2009), pp. 943–958. DOI: 10.4304/jsw.4.9.943-958 (cit. on pp. 15, 24).

[32] Benoit Combemale, Xavier Crégut, Jean-Patrice Giacometti, Pierre Michel, and Marc Pantel. “Introducing Simulation and Model Animation in the MDE Topcased Toolkit”. In: *European congress on Embedded Real Time Software and Systems (ERTS)*. 2008 (cit. on pp. 22, 34, 36, 39, 44, 54, 87).

[34] Benoit Combemale, Julien Deantoni, Benoit Baudry, Robert B. France, Jeff Gray, Jean Marc Jezequel, and Jeff Gray. “Globalizing modeling languages”. In: *Computer* 47.6 (2014), pp. 68–71. ISSN: 00189162. DOI: 10.1109/MC.2014.147 (cit. on pp. xvi, 5, 22).

[35] Benoit Combemale, Julien Deantoni, Matias Vara Larsen, Frédéric Mallet, Olivier Barais, Benoit Baudry, and Robert France. "Reifying Concurrency for Executable Metamodeling". In: *International Conference on Software Language Engineering (SLE)*. Vol. 8225. LNCS. Springer International Publishing, 2013, pp. 365–384. DOI: 10.1007/978-3-319-02654-1_20 (cit. on pp. 15, 30, 34, 156).

[36] Benoit Combemale, Cécile Hardebolle, Christophe Jacquet, Frédéric Boulanger, and Baudry Benoit. "Bridging the Chasm between Executable Metamodeling and Models of Computation". In: *International Conference on Software Language Engineering (SLE)*. Vol. 7745. LNCS. Springer Berlin Heidelberg, 2012, pp. 184–203. DOI: 10.1007/978-3-642-36089-3_11 (cit. on p. 15).

[37] Benoît Combemale, Xavier Crégut, and Marc Pantel. "A Design Pattern to Build Executable DSMLs and associated V&V tools". In: *Asia-Pacific Software Engineering Conference (APSEC)* (2012), pp. 282–287. DOI: 10.1109/APSEC.2012.79 (cit. on pp. 24, 34, 39, 87).

[38] Jonathan Corley, Brian P Eddy, and Jeff Gray. "Towards Efficient and Scalabale Omniscient Debugging for Model Transformations". In: *Workshop on Domain-Specific Modeling (DSM)*. ACM, 2014, pp. 13–18. DOI: 10.1145/2688447.2688450 (cit. on pp. xi, 1, 22, 29, 31, 50, 130).

[39] Bas Cornelissen, Danny Holten, Andy Zaidman, Leon Moonen, Jarke J. Van Wijk, and Arie Van Deursen. "Understanding execution traces using massive sequence and circular bundle views". In: *International Conference on Program Comprehension (ICPC)*. IEEE, 2007, pp. 49–58. ISBN: 0769528600. DOI: 10.1109/ICPC.2007.39 (cit. on pp. xii, 2).

[40] Michelle L. Crane and Juergen Dingel. "Towards a UML Virtual Machine: Implementing an Interpreter for UML 2 Actions and Activities". In: *Conference of the center for advanced studies on collaborative research: meeting of minds (CASCON)*. ACM, 2008, p. 96. DOI: 10.1145/1463788.1463799 (cit. on pp. 29, 43).

[41] Xavier Crégut, Benoit Combemale, Marc Pantel, Raphaël Faudoux, and Jonatas Pavei. "Generative technologies for model animation in the TopCased platform". In: *European Conference on Modeling Foundations and Applications (ECMFA)*. Vol. 6138. LNCS. Springer Berlin Heidelberg, 2010, pp. 90–103. ISBN: 3642135943. DOI: 10.1007/978-3-642-13595-8_9 (cit. on pp. 22, 36, 39, 87).

[42] Gyorgy Csertan, Gabor Huszerl, Istvan Majzik, Zsigmond Pap, Andras Pataricza, and Daniel Varro. "VIATRA - visual automated transformations for formal verification and validation of UML models". In: *International Conference on Automated Software Engineering (ASE)*. IEEE, 2002, pp. 267–270. ISBN: 0-7695-1736-6. DOI: 10.1109/ASE.2002.1115027 (cit. on pp. 14, 15).

[43] Krzysztof Czarnecki, J Nathan Foster, Zhenjiang Hu, and James F Terwilliger. "Bidirectional Transformations: A Cross-Discipline Perspective". In: *International Conference on Model Transformation (ICMT)*. Vol. 5563. LNCS. Springer Berlin

Heidelberg, 2009, pp. 260–283. DOI: 10.1007/978-3-642-02408-5_19 (cit. on p. 46).

[44] Krzysztof Czarnecki and Simon Helsen. "Feature-based survey of model transformation approaches". In: *IBM Systems Journal* 45.3 (2006), pp. 621–645. ISSN: 0018-8670. DOI: 10.1147/sj.453.0621 (cit. on pp. 14, 15).

[45] Julien DeAntoni and Frédéric Mallet. "TimeSquare: Treat your Models with Logical Time". In: *International Conference on Objects, Models, Components, Patterns (TOOLS)*. Vol. 7304. LNCS. Springer Berlin Heidelberg, 2012, pp. 34–41. ISBN: 9783642305603. DOI: 10.1007/978-3-642-30561-0_4 (cit. on pp. 34, 76, 143).

[46] Julien DeAntoni, Frédéric Mallet, Frédéric Thomas, Gonzague Reydet, Jean-Philippe Babau, Chokri Mraidha, Ludovic Gauthier, Laurent Rioux, and Nicolas Sordon. "RT-simex: retro-analysis of execution traces". In: *International Symposium on the Foundations of Software Engineering (FSE)*. ACM, 2010, pp. 377–378. ISBN: 9781605587912. DOI: 10.1145/1882291.1882357 (cit. on pp. 31, 36, 37, 40).

[47] Mathieu Desnoyers. *Common Trace Format (CTF) Specification (v1.8.2)*. 2013. URL: http://git.efficios.com/?p=ctf.git;a=blob_plain;f=common-trace-format-specification.md;hb=master (cit. on pp. 31, 36, 38, 54, 89).

[48] Davide Di Ruscio, Romina Eramo, and Alfonso Pierantonio. "Model Transformations". In: *International School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM)*. Vol. 7320. LNCS. Springer Berlin Heidelberg, 2012, pp. 91–136. ISBN: 978-3-642-30981-6. DOI: 10.1007/978-3-642-30982-3_4 (cit. on p. 14).

[49] Dolev Dotan and Andrei Kirshin. "Debugging and testing behavioral UML models". In: *International Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 2007, pp. 838–839. ISBN: 9781595938657. DOI: 10.1145/1297846.1297915 (cit. on p. 43).

[50] Sophia Drossopoulou and James Noble. "Trust the clones". In: *Formal Verification of Object-Oriented Software (FoVeOOS)*. 2011 (cit. on pp. 18, 21).

[51] Matthew B. Dwyer and Sebastian Elbaum. "Unifying verification and validation techniques: relating behavior and properties through partial evidence". In: *FSE/SDP Workshop on Future of software engineering research (FoSER)*. ACM, 2010, pp. 93–98. ISBN: 9781450304276. DOI: 10.1145/1882362.1882382 (cit. on pp. xii, 2).

[52] Marina Egea and Vlad Rusu. "Formal executable semantics for conformance in the MDE framework". In: *Innovations in Systems and Software Engineering* 6.1 (2010), pp. 73–81. ISSN: 16145046. DOI: 10.1007/s11334-009-0108-1 (cit. on p. 12).

[53] Marc Eisenstadt. "My Hairiest Bug War Stories". In: *Communications of the ACM* 40.4 (1997), pp. 30–37. DOI: 10.1145/248448.248456 (cit. on p. 45).

[54] Jakob Engblom. "A review of reverse debugging". In: *SoC and Silicon Debug Conference (S4D)*. IEEE, 2012, pp. 1–6 (cit. on pp. 46–48).

[55] Gregor Engels, Jan Hendrik Hausmann, Reiko Heckel, and Stefan Sauer. "Dynamic Meta-Modeling: A Graphical Approach to the Operational Semantics of Behavioral Diagrams in UML". In: *International conference on the Unified Modeling Language (UML)*. Vol. 1939. LNCS. Springer Berlin Heidelberg, 2000, pp. 323–337. ISBN: 978-3-540-41133-8. DOI: 10.1007/3-540-40011-7_23 (cit. on p. 24).

[56] Dominic Eschweiler, Michael Wagner, Markus Geimer, Andreas Knüpfer, Wolfgang E Nagel, and Felix Wolf. "Open Trace Format 2: The Next Generation of Scalable Trace Formats and Support Libraries." In: *14th Int. Conf. on Parallel Computing*. Vol. 22. Advances in Parallel Computing. IOS Press, 2011, pp. 481–490. DOI: 10.3233/978-1-61499-041-3-481 (cit. on pp. 35, 36, 40, 54, 76).

[57] Shahram Esmaeilsabzali and Nancy a. Day. "Prescriptive semantics for big-step modeling languages". In: *International Conference on Fundamental Approaches to Software Engineering (FASE)*. Vol. 6013. LNCS. Springer Berlin Heidelberg, 2010, pp. 158–172. ISBN: 3642120288. DOI: 10.1007/978-3-642-12029-9_12 (cit. on p. 29).

[58] Uli Fahrenberg, Axel Legay, and Andrzej Wasowski. "Vision Paper: Make a Difference! (Semantically)". In: *International Conference on Model Driven Engineering Languages and Systems (MODELS)*. Vol. 6981. LNCS. Springer Berlin Heidelberg, 2014, pp. 490–500. DOI: 10.1007/978-3-642-24485-8_36 (cit. on p. 100).

[59] Jean-Rémy Falleri, Marianne Huchard, and Clémentine Nebut. "Towards a Traceability Framework for Model Transformations in Kermeta". In: *ECMDA Traceability Workshop*. 2006. ISBN: 82-14-04030-2 (cit. on p. 34).

[60] François Fouquet, Gregory Nain, Brice Morin, and Erwan Daubert. "An Eclipse Modeling Framework alternative to Meet the Models@Runtime Requirements". In: *International Conference on Model Driven Engineering Languages and Systems (MODELS)*. Vol. 7590. LNCS. Springer Berlin Heidelberg, 2012, pp. 87–101. DOI: 10.1007/978-3-642-33666-9_7 (cit. on pp. 22, 37, 60).

[61] François Fouquet, Grégory Nain, Brice Morin, Erwan Daubert, Olivier Barais, Nöel Plouzeau, and Jean-Marc Jézéquel. *Kevoree Modeling Framework (KMF): Efficient modeling techniques for runtime use*. Tech. rep. TR-SnT-2014-11. University of Luxembourg, Interdisciplinary Centre for Security, Reliability and Trust (SNT), 2014 (cit. on pp. 22, 37, 54, 60).

[62] Robert France and Bernhard Rumpe. "Model-driven Development of Complex Software: A Research Roadmap". In: *Future of Software Engineering (FOSE)*. IEEE Computer Society, 2007, pp. 37–54. DOI: 10.1109/FOSE.2007.14 (cit. on p. 11).

[63] Lars Fredlund. "An implementation of a translational semantics for an imperative language". In: *CONCUR '90 Theories of Concurrency: Unification and Extension*. Vol. 458. LNCS. Springer Berlin Heidelberg, 1990, pp. 246–262. DOI: 10.1007/BFb0039045 (cit. on p. 24).

[64] Lidia Fuentes, Jorge Manrique, and Pablo Sánchez. "Execution and simulation of (profiled) UML models using pópulo". In: *International workshop on Models in Software Engineering (MiSE)*. ACM, 2008, pp. 75–81. ISBN: 9781605580258. DOI: 10.1145/1370731.1370749 (cit. on p. 43).

[65] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1994. ISBN: 9781605580791. DOI: 10.1145/1368088.1368202 (cit. on pp. 16, 37, 65).

[66] Kelly Garces, Julien Deantoni, and Frederic Mallet. "A Model-Based Approach for Reconciliation of Polychronous Execution Traces". In: *Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, 2011, pp. 259–266. ISBN: 978-1-4577-1027-8. DOI: 10.1109/SEAA.2011.47 (cit. on p. 34).

[67] Markus Geimer, Felix Wolf, Brian J N Wylie, Erika Ábrahám, Daniel Becker, and Bernd Mohr. "The Scalasca performance toolset architecture". In: *Concurrency Computation: Practice and Experience* 22.6 (2010), pp. 702–719. ISSN: 15320626. DOI: 10.1002/cpe.1556 (cit. on pp. 33, 35, 36).

[68] Andy Georges, Mark Christiaens, Michiel Ronsse, and Koenraad De Bosschere. "JaRec: a portable record/replay environment for multi-threaded Java applications". In: *Software: Practice and Experience* 34.6 (2004), pp. 523–547. ISSN: 0038-0644. DOI: 10.1002/spe.579 (cit. on p. 49).

[69] Martin Gogolla, Lars Hamann, Frank Hilken, Mirco Kuhlmann, and Robert B France. "From Application Models to Filmstrip Models: An Approach to Automatic Validation of Model Dynamic". In: *Modellierung*. Vol. 225. LNI. GI, 2014, pp. 273–288 (cit. on pp. 31, 35, 36, 39, 40, 87, 88).

[70] Sherri Goings, Heather J. Goldsby, Betty H. C. Cheng, and Charles Ofria. "An ecology-based evolutionary algorithm to evolve solutions to complex problems". In: *Artificial Life* 13 (2012), pp. 171–177. DOI: 10.7551/978-0-262-31050-5-ch024 (cit. on pp. 20, 55, 59).

[71] Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., 1983. ISBN: 0201113716 (cit. on pp. 18, 59).

[72] Lorenzo Gomez, Iulian Neamtiu, Tanzirul Azim, and Todd Millstein. "RERAN: Timing- and touch-sensitive record and replay for Android". In: *International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 72–81. DOI: 10.1109/ICSE.2013.6606553 (cit. on p. 46).

[73] Peter Grogono and Patrice Chalin. "Copying, Sharing, and Aliasing". In: *Colloquium on Object Orientation in Databases and Software Engineering (COODBSE)*. 1994 (cit. on p. 18).

[74] Peter Grogono and Markku Sakkinen. "Copying and Comparing: Problems and Solutions". In: *European Conference on Object-Oriented Programming (ECOOP)*. Vol. 1850. LNCS. Springer Berlin Heidelberg, 2000, pp. 226–250. DOI: 10.1007/3-540-45102-1_11 (cit. on p. 18).

[75] Object Management Group. *UML Testing Profile (UTP), v1.2*. 2013. URL: http://www.omg.org/spec/UTP/1.2/ (cit. on p. 36).

[76] Clément Guy, Benoit Benoît Combemale, and Steven Derrien. "On Model Subtyping". In: *European Conference on Modeling Foundations and Applications (ECMFA)*. Vol. 7349. LNCS. Springer Berlin Heidelberg, 2012, pp. 400–415. DOI: 10.1007/978-3-642-31491-9 (cit. on p. 156).

[77] Abdelwahab Hamou-Lhadj and Timothy C. Lethbridge. "A metamodel for the compact but lossless exchange of execution traces". In: *Software and Systems Modeling (SoSyM)* 11.1 (2010), pp. 77–98. DOI: 10.1007/s10270-010-0180-x (cit. on pp. 35, 36).

[78] David Harel, Hagi Lachover, Amnon Naamad, Amir Pnuelli, Michal Politi, Rivi Sherman, Aharon Shtull-trauring, and Mark Trakhtenbrot. "STATEMATE: a working environment for the development of complex reactive systems". In: *IEEE Transactions on software engineering* 16.4 (1990), pp. 403–414. DOI: 10.1109/ICCSSE.1988.72235 (cit. on p. 29).

[79] David Harel and Amil Pnueli. "On the development of reactive systems". In: *Logics and Models of Concurrent Systems*. Vol. F13. NATO ASI. 1985, pp. 477–498. DOI: 10.1103/PhysRevB.84.144110 (cit. on p. 29).

[80] David Harel and Bernhard Rumpe. "Meaningful Modeling: What's the Semantics of "Semantics"?" In: *Computer* 37.10 (2004), pp. 64–72. DOI: 10.1109/MC.2004.172 (cit. on p. 12).

[81] Thomas Hartmann, Francois Fouquet, Gregory Nain, Brice Morin, Jacques Klein, Olivier Barais, and Yves Le Traon. "A Native Versioning Concept to Support Historized Models at Runtime". In: *International Conference on Model Driven Engineering Languages and Systems (MODELS)*. Vol. 8767. LNCS. Springer International Publishing, 2014, pp. 252–268. DOI: 10.1007/978-3-319-11653-2_16 (cit. on pp. 36, 37, 40, 54, 75, 88).

[82] Ábel Hegedüs, Gábor Bergmann, István Ráth, and Dániel Varró. "Back-annotation of Simulation Traces with Change-Driven Model Transformations". In: *International Conference on Software Engineering and Formal Methods (SEFM)*. IEEE, 2010, pp. 145–155. DOI: 10.1109/SEFM.2010.28 (cit. on pp. 24, 25).

[83] Ábel Hegedüs, István Ráth, and Dániel Varró. *Back-annotation framework for Simulation Traces of Discrete Event-based Languages*. Tech. rep. BME, 2010. URL: https://inf.mit.bme.hu/sites/default/files/publications/Hegedus_TechRep_201004.pdf (cit. on pp. 29, 34, 36, 39, 50, 54, 87, 88, 130).

[84] Ábel Hegedüs, István Ráth, and Dániel Varró. "Replaying Execution Trace Models for Dynamic Modeling Languages". In: *Periodica Polytechnica - Electrical Engineering and Computer Science* 56.3 (2013), pp. 71–82 (cit. on p. 25).

[85] Frank Hilken, Lars Hamann, and Martin Gogolla. "Transformation of UML and OCL models into filmstrip models". In: *International Conference on Model Transformation (ICMT)*. Vol. 8568. LNCS. Springer International Publishing, 2014, pp. 170–185. DOI: 10.1007/978-3-319-08789-4-13 (cit. on pp. 31, 35, 36).

[86] John Hutchinson, Jon Whittle, Mark Rouncefield, and Steinar Kristoffersen. "Empirical Assessment of MDE in Industry". In: *International Conference on Software Engineering (ICSE)*. ACM, 2011, pp. 471–480. DOI: 10.1145/1985793.1985858 (cit. on pp. xi, 1, 37, 76).

[87] Cédric Jeanneret, Martin Glinz, and Benoit Baudry. "Estimating footprints of model operations". In: *International Conference on Software engineering (ICSE)*. ACM, 2011, pp. 601–610. ISBN: 9781450304450. DOI: 10.1145/1985793.1985875 (cit. on pp. 16–18, 155).

[88] Jean-Marc Jézéquel, Benoit Combemale, Olivier Barais, Martin Monperrus, and François Fouquet. "Mashup of metalanguages and its implementation in the Kermeta language workbench". In: *Software and Systems Modeling (SoSyM)* 14.2 (2015), pp. 905–920. DOI: 10.1007/s10270-013-0354-4 (cit. on pp. 14, 15, 21, 26, 27, 44, 60).

[89] Jean-Marc Jézéquel, Benoit Combemale, and Didier Vojtisek. *Ingénierie Dirigée par les Modèles : des concepts à la pratique*. Références sciences. Ellipses, 2012. ISBN: 9782729871963 (cit. on p. 12).

[90] Frédéric Jouault. "Loosely coupled traceability for ATL". In: *ECMDA Traceability Workshop*. Vol. 91. 2005, pp. 29–37 (cit. on p. 34).

[91] Frédéric Jouault and Ivan Kurtev. "Transforming models with ATL". In: *Workshop on Model Transformations in Practice (MTiP)*. Vol. 3844. LNCS. Springer Berlin Heidelberg, 2006, pp. 128–138. DOI: 10.1007/11663430_14 (cit. on pp. 14, 15, 44).

[92] Gabor Karsai, Aditya Agrawal, Feng Shi, and Jonathan Sprinkle. "On the Use of Graph Transformation in the Formal Specification of Model Interpreters". In: *Journal of Universal Computer Science* 9.11 (2003), pp. 1296–1321. ISSN: 0958695X (ISSN) (cit. on p. 22).

[93] Steven Kelly and Risto Pohjonen. "Worst Practices for Domain-Specific Modeling". In: *IEEE Software* 26.4 (2009), pp. 22–29. DOI: 10.1109/MS.2009.109 (cit. on pp. 54, 76, 77).

[94] Mickaël Kerboeuf and Jean-Philippe Babau. "A DSML for reversible transformations". In: *Workshop on Domain-Specific Modeling (DSM)*. ACM, 2011, pp. 1–6. DOI: 10.1145/2095050.2095057 (cit. on p. 46).

[95] Marouane Kessentini, Houari Sahraoui, and Mounir Boukadoum. "Model Transformation as an Optimization Problem". In: *International Conference on Model Driven Engineering Languages and Systems (MODELS)*. Vol. 5301. LNCS. Springer Berlin Heidelberg, 2008, pp. 159–173. DOI: 10.1007/978-3-540-87875-9_12 (cit. on pp. 14, 20, 58, 59).

[96] Andrei Kirshin, Dolev Dotan, and Alan Hartman. "A UML Simulator Based on a Generic Model Execution Engine". In: *Workshop on Multi-Paradigm Modeling (MPM)*. Vol. 4364. LNCS. Springer Berlin Heidelberg, 2006, pp. 324–326. DOI: 10.1007/978-3-540-69489-2_40 (cit. on p. 43).

[97] Dimitrios S Kolovos, Davide Di Ruscio, Alfonso Pierantonio, and Richard F Paige. "Different Models for Model Matching: An analysis of approaches to support model differencing". In: *Workshop on Comparison and Versioning of Software Models (CVSM)*. IEEE, 2009, pp. 1–6. DOI: 10.1109/CVSM.2009.5071714 (cit. on p. 101).

[98] Alexander Krasnogolowy, Stephan Hildebrandt, and Sebastian Wlitzoldt. "Flexible Debugging of Behavior Models". In: *International Conference on Industrial Technology (ICIT)*. IEEE, 2012, pp. 331–336. DOI: 10.1109/ICIT.2012.6209959 (cit. on pp. 42, 43, 49).

[99] Philip Langer, Tanja Mayerhofer, and Gerti Kappel. "Semantic Model Differencing Utilizing Behavioral Semantics Specifications". In: *International Conference on Model Driven Engineering Languages and Systems (MODELS)*. Vol. 8767. LNCS. Springer International Publishing, 2014, pp. 116–132. DOI: 10.1007/978-3-319-11653-2_8 (cit. on pp. xi, 1, 21, 22, 31, 36, 37, 54, 74, 76, 94, 99–101, 106).

[100] Yoann Laurent, R Bendraou, and Mp Gervais. "Executing and debugging UML models: an fUML extension". In: *Symposium on Applied Computing (SAC)*. ACM, 2013, pp. 1095–1102. DOI: 10.1145/2480362.2480569 (cit. on pp. 42, 43, 50).

[101] Ernest Lepore and Barry Loewer. "Translational Semantics". In: *Synthese* 48.1 (1981), pp. 121–133. DOI: 10.1007/BF01064631 (cit. on p. 24).

[102] Martin Leucker and Christian Schallhart. "A brief account of runtime verification". In: *The Journal of Logic and Algebraic Programming* 78.5 (2009), pp. 293–303. DOI: 10.1016/j.jlap.2008.08.004 (cit. on pp. xi, 1, 31, 46, 155).

[103] Bill Lewis. "Debugging backwards in time". In: *International Workshop on Workshop on Automated and Algorithmic Debugging (AADEBUG)*. 2003 (cit. on pp. 46, 47, 49).

[104] Paley Li, Nicholas Cameron, and James Noble. "Cloning in Ownership". In: *International conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*. ACM, 2011, pp. 63–65. DOI: 10.1145/2048147.2048175 (cit. on p. 18).

[105] Paley Li, Nicholas Cameron, and James Noble. "Sheep Cloning with Ownership Types". In: *International Workshop on Foundations of Object-Oriented Languages (FOOL)*. 2012 (cit. on pp. 18, 21).

[106] H Lieberman. "Using prototypical objects to implement shared behavior in object-oriented systems". In: *International Conference on Object Oriented Programming Systems Languages & Applications (OOSPLA)*. ACM, 1986, pp. 214–223. DOI: 10.1145/28697.28718 (cit. on p. 65).

[107] Adrian Lienhard, Tudor Gîrba, and Oscar Nierstrasz. "Practical Object-Oriented Back-in-Time Debugging". In: *European Conference on Object-Oriented Programming (ECOOP)*. Vol. 5142. LNCS. Springer Berlin Heidelberg, 2008, pp. 592–615. DOI: 10.1007/978-3-540-70592-5_25 (cit. on pp. 31, 46, 49).

[108] Yuehua Lin, Jeff Gray, and Frédéric Jouault. "DSMDiff: a differentiation tool for domain-specific models". In: *European Journal of Information Systems* 16.4 (2007), pp. 349–361. DOI: 10.1057/palgrave.ejis.3000685 (cit. on p. 99).

[109] Ricky T Lindeman, Lennart C L Kats, and Eelco Visser. "Declaratively Defining Domain-Specific Language Debuggers". In: *International Conference on Generative Programming and Component Engineering (GPCE)*. ACM, 2012, pp. 127–136. DOI: 10.1145/2189751.2047885 (cit. on p. 45).

[110] Shahar Maoz. "Model-based traces". In: *Workshop on Models@runtime (MRT)*. Vol. 5421. LNCS. Springer Berlin Heidelberg, 2009, pp. 109–119. DOI: 10.1007/978-3-642-01648-6_12 (cit. on pp. 34, 50, 130).

[111] Shahar Maoz and David Harel. "On tracing reactive systems". In: *Software and Systems Modeling (SoSyM)* 10.4 (2011), pp. 447–468. DOI: 10.1007/s10270-010-0151-2 (cit. on pp. 34, 36, 50, 76, 130).

[112] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. "ADDiff: Semantic Differencing for Activity Diagrams". In: *8th Joint European Software Engineering Conference (ESEC) and Symposium on the Foundations of Software Engineering (FSE)*. ACM, 2011, pp. 179–189. DOI: 10.1145/2025113.2025140 (cit. on pp. 31, 94, 96–98, 100, 128).

[113] Tanja Mayerhofer, Philip Langer, and Gerti Kappel. "A runtime model for fUML". In: *Workshop on Models@run.time (MRT)*. ACM, 2012, pp. 53–58. DOI: 10.1145/2422518.2422527 (cit. on pp. 31, 36–38, 43, 54, 75–77).

[114] Tanja Mayerhofer, Philip Langer, Manuel Wimmer, and Gerti Kappel. "xMOF: Executable DSMLs based on fUML". In: *International Conference on Software Language Engineering (SLE)*. Vol. 8225. LNCS. Springer, 2013, pp. 56–75. DOI: 10.1007/978-3-319-02654-1_4 (cit. on pp. 15, 24, 26, 27, 34, 110).

[115] Marjan Mernik, Jan Heering, and Anthony M. Sloane. "When and how to develop domain-specific languages". In: *ACM Computing Surveys* 37.4 (2005), pp. 316–344. DOI: 10.1145/1118890.1118892 (cit. on p. 12).

[116] Bart Meyers, Romuald Deshayes, Levi Lucio, Eugene Syriani, Hans Vangheluwe, and Manuel Wimmer. "ProMoBox: A Framework for Generating Domain-Specific Property Languages". In: *International Conference on Software Language Engineering (SLE)*. Vol. 8706. LNCS. Springer, 2014, pp. 1–20. DOI: 10.1007/978-3-319-11245-9_1 (cit. on pp. 31, 35, 36, 39, 54, 74, 88, 155).

[117] Simon Van Mierlo, Yentl Van Tendeloo, Bruno Barroca, Sadaf Mustafiz, and Hans Vangheluwe. "Explicit Modeling of a Parallel DEVS Experimentation Environment". In: *Symposium on Theory of Modeling and Simulation - DEVS (TMS/DEVS)*. Society for Computer Simulation International, 2015, pp. 860–867 (cit. on p. 43).

[118] Robin Milner. *Communication and Concurrency*. Prentice-Hall, Inc., 1989. ISBN: 0131149849 (cit. on pp. xii, 2).

[119] Parastoo Mohagheghi and Vegard Dehlen. "Where is the proof? - A review of experiences from applying MDE in industry". In: *European conference on Model Driven Architecture – Foundations and Applications (ECMDA-FA)*. Vol. 5095. LNCS. Springer Berlin Heidelberg, 2008, pp. 432–443. DOI: 10.1007/978-3-540-69100-6-31 (cit. on pp. xi, 1).

[120] Object Management Group. *Meta Object Facility (MOF) Core Specification, Version 2.5*. 2014. URL: http://www.omg.org/spec/MOF/2.5 (cit. on pp. 12, 13, 26, 31).

[121] Object Management Group. *Object Constraint Language (OCL) Version 2.4*. 2014. URL: http://www.omg.org/spec/OCL/2.4 (cit. on p. 13).

[122] Object Management Group. *Semantics of a Foundational Subset for Executable UML Models (fUML), V 1.1*. 2013. URL: http://www.omg.org/spec/FUML/1.1 (cit. on pp. xv, 4, 37, 94, 96, 128).

[123] Object Management Group. *Unified Modeling Language (UML) Version 2.5*. 2015. URL: http://www.omg.org/spec/UML/2.5 (cit. on pp. 26, 94).

[124] Generoso Pagano, Damien Dosimont, Guillaume Huard, Vania Marangozova-Martin, and Jean-Marc Vincent. "Trace Management and Analysis for Embedded Systems". In: *International Symposium on Embedded Multicore Socs (MCSoC)*. IEEE, 2013, pp. 119–122. DOI: 10.1109/MCSoC.2013.28 (cit. on pp. 36, 38).

[125] Peggy Aldrich Kidwell. "Stalking the Elusive Computer Bug". In: *IEEE Annals Of The History Of Computing* 20.4 (1998), pp. 3–7. DOI: 10.1109/85.728224 (cit. on p. 40).

[126] Khoo Yit Phang, Jeffrey S. Foster, and Michael Hicks. "Expositor: Scriptable time-travel debugging with first-class traces". In: *International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 352–361. DOI: 10.1109/ICSE.2013.6606581 (cit. on pp. 31, 46).

[127] Gordon D Plotkin. *A Structural Approach to Operational Semantics*. Tech. rep. DAIMI FN-19. University of Aarhus, Faculty of Science, Department of Computer Science (Daimi), 1981 (cit. on p. 22).

[128] Guillaume Pothier and Éric Tanter. "Back to the future: Omniscient debugging". In: *IEEE Software* 26.6 (2009). DOI: 10.1145/1105755.1105759 (cit. on pp. 41, 45, 46, 49, 114).

[129] István Ráth, Gábor Bergmann, András Ökrös, and Dániel Varró. "Live Model Transformations Driven by Incremental Pattern Matching". In: *International Conference on Model Transformation (ICMT)*. Vol. 5063. LNCS. Springer Berlin Heidelberg, 2008, pp. 107–121. DOI: 10.1007/978-3-540-69927-9_8 (cit. on p. 15).

[130] István Ráth, Dávid Vágó, and Dániel Varró. "Design-time simulation of domain-specific models by incremental pattern matching". In: *Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 2008, pp. 219–222. DOI: 10.1109/VLHCC.2008.4639089 (cit. on pp. 15, 44).

[131] Raymond Reiter. "A theory of diagnosis from first principles". In: *Artificial Intelligence* 32.1 (1987), pp. 57–95. DOI: 10.1016/0004-3702(87)90062-2 (cit. on p. 45).

[132] Joel Riedesel. "Diagnosing Multiple Faults in SSM/PMAD". In: *International Energy Conversion Engineering Conference (IECEC)*. 1987. IEEE, 1990, pp. 284–290. DOI: 10.1109/IECEC.1990.716896 (cit. on p. 45).

[133] Dirk Riehle, Steven Fraleigh, Dirk Bucka-Lassen, and Nosa Omorogbe. "The architecture of a UML virtual machine". In: *International Conference on Object Oriented Programming Systems Languages and Applications (OOSPLA)*. ACM, 2001, pp. 327–341. DOI: 10.1145/504311.504306 (cit. on p. 43).

[134] Bernhard Rumpe and Ingo Weisemoeller. "A domain specific transformation language". In: *Workshop an Models and Evolution (ME)*. 2011 (cit. on p. 155).

[135] Robert Sauter, Olga Saukh, Oliver Frietsch, and Pedro José Marrón. "TinyLTS: Efficient network-wide Logging and Tracing System for TinyOS". In: *International Conference on Computer Communications (INFOCOM)*. IEEE, 2011, pp. 2033–2041. DOI: 10.1109/INFCOM.2011.5935011 (cit. on p. 32).

[136] Tripti Saxena and Gabor Karsai. "MDE-Based Approach for Generalizing Design Space Exploration". In: *International Conference on Model Driven Engineering Languages and Systems (MODELS)*. Vol. 6394. LNCS. Springer Berlin Heidelberg, 2010, pp. 46–60. DOI: 10.1007/978-3-642-16145-2_4 (cit. on pp. 20, 55, 58, 59).

[137] Douglas C Schmidt. "Guest Editor's Introduction: Model-Driven Engineering". In: *Computer* 39.2 (2006), pp. 25–31. DOI: 10.1109/MC.2006.58 (cit. on pp. xi, 1, 11, 12).

[138] Jürgen Schnerr, Oliver Bringmann, Alexander Viehl, and Wolfgang Rosenstiel. "High-performance timing simulation of embedded software". In: *Design Automation Conference (DAC)*. IEEE, 2008, pp. 290–295. DOI: 10.1109/DAC.2008.4555825 (cit. on p. 24).

[139] Lucas M Schnorr, Oliveira Stein, and Jacques Chassin. *Paje trace file format, version 1.2.5*. 2013. URL: http://paje.sourceforge.net/download/publication/lang-paje.pdf (cit. on pp. 36, 38).

[140] Andy Schürr. "Specification of graph translators with triple graph grammars". In: *International Workshop on Graph-Theoretic Concepts in Computer Science (WG)*. Vol. 903. LNCS. Springer Berlin Heidelberg, 1994, pp. 151–163. DOI: 10.1007/3-540-59071-4_45 (cit. on pp. 15, 46).

[141] Shane Sendall and Wojtek Kozaczynski. "Model transformation: the heart and soul of model-driven software development". In: *IEEE Software* 20.5 (2003). DOI: 10.1109/MS.2003.1231150 (cit. on p. 14).

[142] Sameer S. Shende and Allen D. Malony. "The Tau Parallel Performance System". In: *International Journal of High Performance Computing Applications* 20.2 (2006), pp. 287–311. DOI: 10.1177/1094342006064482 (cit. on p. 33).

[143] James E. Smith and Ravi Nair. "The architecture of virtual machines". In: *Computer* 38.5 (2005), pp. 32–38. DOI: 10.1109/MC.2005.173 (cit. on p. 42).

[144] Michael Soden and Hajo Eichler. "Towards a model execution framework for Eclipse". In: *Workshop on Behaviour Modelling in Model-Driven Architecture (BD-MDA)*. ACM, 2009. DOI: 10.1145/1555852.1555856 (cit. on pp. 24, 25).

[145] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework, 2nd Edition*. Eclipse Series. Addison-Wesley Professional, 2008. ISBN: 0321331885 (cit. on pp. 13, 57, 58, 141).

[146] STMicroelectronics. *KPTrace Specification*. 2012. URL: http://www.stlinux.com/stworkbench/interactive_analysis/stlinux.trace/kptrace_traceFormat.html (cit. on pp. 35, 36, 54, 76).

[147] Markus Stumptner and Franz Wotawa. "Debugging functional programs". In: *International Joint Conference on Artificial Intelligence (IJCAI)*. Vol. 2. 1999, pp. 1074–1079 (cit. on p. 45).

[148] Eugene Syriani and Hans Vangheluwe. "A modular timed graph transformation language for simulation-based design". In: *Software and Systems Modeling (SoSyM)* 12.2 (2013), pp. 387–414. DOI: 10.1007/s10270-011-0205-0 (cit. on pp. 50, 130).

[149] Eugene Syriani, Hans Vangheluwe, and Brian LaShomb. "T-Core: a framework for custom-built model transformation engines". In: *Software and Systems Modeling (SoSyM)* 14.3 (2013), pp. 1215–1243. DOI: 10.1007/s10270-013-0370-4 (cit. on pp. 50, 130).

[150] Robert Tarjan. "Depth-First Search and Linear Graph Algorithms". In: *SIAM Journal on Computing* 1.2 (1972), pp. 146–160. DOI: 10.1137/0201010 (cit. on pp. 65, 136).

[151] Jérémie Tatibouët, Arnaud Cuccuru, Sébastien Gérard, and François Terrier. "Formalizing Execution Semantics of UML Profiles with fUML Models". In: *International Conference on Model Driven Engineering Languages and Systems (MODELS)*. Vol. 8767. LNCS. Springer International Publishing, 2014, pp. 133–148. DOI: 10.1007/978-3-319-11653-2_9 (cit. on p. 24).

[152] Markus Voelter. *DSL Engineering*. CreateSpace Independent Publishing Platform, 2013. ISBN: 978-1481218580. URL: http://dslbook.org/ (cit. on pp. xi, 1).

[153] Jon Whittle, John Hutchinson, and Mark Rouncefield. "The State of Practice in Model-Driven Engineering". In: *IEEE Software* 31.3 (2014), pp. 79–85. DOI: 10.1109/MS.2013.65 (cit. on pp. 37, 76).

[154] Franz Wotawa. "On the relationship between model-based debugging and program slicing". In: *Artificial Intelligence* 135.1-2 (2002), pp. 125–143. DOI: 10.1016/S0004-3702(01)00161-8 (cit. on p. 45).

[155] Hui Wu, Jeff Gray, and Marjan Mernik. "Grammar-driven generation of domain-specific language debuggers". In: *Software: Practice and Experience* 38.10 (2008), pp. 1073–1103. DOI: 10.1002/spe.863 (cit. on p. 44).

[156] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. "SherLog: error diagnosis by connecting clues from run-time logs". In: *International Conference on Architectural Support for Programming Languages and Operating Systems (APLO)*. ACM, 2010, pp. 143–154. DOI: 10.1145/1735971.1736038 (cit. on pp. 32, 46).

[157] Andreas Zeller. *Why Program Fail – 1st Edition*. Elsevier, 2004. ISBN: 9780080481739. URL: http://www.whyprogramsfail.com/ (cit. on p. 40).

Abstract

Model-Driven Engineering (MDE) is a development paradigm that aims at coping with the complexity of systems by separating concerns through the use of *models*, each being defined using specific abstractions provided by a Domain-Specific Modeling Language (DSML).

A subclass of DSMLs aim at supporting the *execution* of models, namely executable Domain-Specific Modeling Languages (xDSMLs). An xDSML includes *execution semantics* that manipulate the concepts of the considered domain. To ensure that an executable model is correct with regard to its intended behavior, *dynamic verification and validation* (V&V) techniques are required, such as omniscient debugging. Yet, to analyze an executable model, these techniques need an *explicit* representation of its behavior over time.

Among dynamic V&V techniques, a most common representation of a model's behavior is the *execution trace*, which is a sequence containing all the relevant information about an execution over time. However, the execution semantics of an xDSML can be arbitrarily complex, hence making both difficult the definition of an appropriate data structure to construct execution traces, and the development of efficient and adapted tooling to manipulate them. First, the *usability* of an execution trace data structure must be ensured to cope with the complexity of data. Second, since executing even a simple model can lead to very large execution traces, both *scalability in memory* of executions traces and *scalability in processing time* of execution trace manipulations are of primary importance.

Therefore, to enable dynamic V&V of executable models of any possible xDSML, it is crucial to provide efficient facilities to construct and manipulate all kinds of execution traces. To that effect, we first focused on the representation of the *execution state* of an executed model, and proposed a *scalable model cloning* approach to conveniently construct generic execution traces using model clones. We then focused on the *structure* of execution traces, and designed *a generative approach to define multidimensional and domain-specific execution trace metamodels*. Such a metamodel precisely captures the content of the execution traces of a specific xDSML, while providing efficient navigation paths to follow the evolution of different mutable parts of a conforming model.

In addition, we made two applications of multidimensional domain-specific execution trace metamodels to existing dynamic V&V techniques. First, we defined a set of *semantic differencing* rules to analyze a set of fUML models. Second, we developed a complete *advanced omniscient debugging approach* for xDSMLs. Overall, we show that a domain-specific structure provides good usability and scalability in memory, and that multiple dimensions enable good scalability in processing time while also enhancing usability.

Our contributions make possible both to construct and to efficiently manipulate execution traces of models conforming to any xDSML, including for dynamic V&V. All our work have been implemented and integrated within the GEMOC Studio, which is a language and modeling workbench resulting from an academic and industrial project. Many research directions are possible to pursue this work, such as taking into account the *environment* influencing an execution or the explicit *concurrency* expressed in the execution semantics.