AIX-MARSEILLE UNIVERSITE

ED 184 - ECOLE DOCTORALE EN MATHEMATIQUES ET INFORMATIQUE DE MARSEILLE

UFR SCIENCES D'AIX-MARSEILLE UNIVERSITE

LABORATOIRE DES SCIENCES DE L'INFORMATION ET DES SYSTEMES

Thèse présentée pour obtenir le grade universitaire de docteur

Discipline : Informatique

Aznam YACOUB

Une approche de vérification formelle et de simulation pour les systèmes à événements : Application à PROMELA

An approach for formal verification and simulation of discrete-event systems: A PROMELA Application

Soutenue le 08/12/2016 devant le jury :

Vincent ALBERT	Université Paul Sabatier	Examinateur
Florence SEDES	Université Paul Sabatier	Examinatrice
Bernard P. ZEIGLER	University of Arizona	Rapporteur
Eric RAMAT	Université du Littoral Côte d'Opale	Rapporteur
Claudia FRYDMAN	LSIS	Directrice de thèse
Maâmar el-amine HAMRI	LSIS	Co-Directeur de thèse
Jacques PINATON	ST Microelectronics	Encadrant

An approach for formal verification and simulation of discrete-event systems

A PROMELA Application

Abstract

For many years, the new technological advances have enabled the development of software and physical systems which considerably help people in their daily tasks or in manufacturing processes. Although they look like simple objects, those signboards, those manufacturing chains and even those autopilots are become more and more complex to understand and to master. Because they implied more interactions than before, with more and more components which communicate with each other, these systems require more vigilance from their designers and from their engineers. Those ones must ensure that their products do not generate unexpected and maladjusted behaviours. This finding is the same than the one that we do when we are interesting in modeling and simulation of complex physical phenomena, such as a climatic event, or the behaviour of the human brain which also involves intricate interactions between dozens, hundreds and even thousands of different entities.

Nevertheless, if the advances have been tremendous in these technological fields, and even in scientific fields, the domains of Verification and Validation have also known a significant progress, with the emergence of new concepts and new automatic methods that ensure reliability of systems. Among all these techniques, we can find two great families of tools : the Formal Methods and the Simulation. For a long time, these two families have been considered as opposite to each other. However, recent work tries to reduce the border between them. In this context, this thesis proposes a new approach in order to integrate Discrete-Event Simulation in Formal Methods. The main objective is to improve existing model-checking tools by combining them with simulation, in order to allow them detecting errors that they were not previously able to find, and especially on timed systems. This approach led us to develop a new formal language, called DEv-PROMELA. This new language, which relies on the PROMELA and on the DEVS formalism, is like both a verifiable specifications language and a simulation formalism. By combining a traditional model-checking and a discrete-event simulation on models expressed in DEv-PROMELA, it is therefore possible to detect and to understand dysfunctions which could not be found by using

only a formal checking or only a simulation. This result is illustrated through the different examples which are treated in this work.

Keywords : Verification and Validation, Modeling and Simulation, Discrete-Event Systems, Formal Methods, Timed Systems, PROMELA, DEv-PROMELA.

Résumé

Depuis quelques années, les nouvelles avancées technologiques ont permis la mise au point de systèmes physiques et logiciels qui aident considérablement l'Homme dans ses tâches, que ce soit dans la vie quotidienne ou dans des processus industriels. Aux allures simples, ces panneaux publicitaires, ces chaînes de fabrication automatisées ou même ces pilotes automatiques sont devenus, au fil du temps, de plus en plus complexes à comprendre et à maîtriser, entraînant en permanence de nouveaux lots de problématiques et de questions. Du fait qu'ils impliquent davantage d'interactions qu'auparavant, avec de plus en plus de composants qui communiquent entre eux, ces systèmes demandent encore plus de vigilances à leurs concepteurs qui doivent s'assurer que ces systèmes ne génèrent pas de comportements inadaptés ou imprévus. Ce constat, que nous faisons pour les systèmes et les processus que nous développons, est identique au constat réalisé dès lors que nous souhaitons modéliser et simuler un phénomène physique complexe, tel qu'un évènement météorologique ou le fonctionnement du cerveau humain, qui impliquent aussi des interactions compliquées entre des dizaines, des centaines voire des milliers d'entités différentes.

Mais si les avancées ont été énormes dans les champs technologiques et scientifiques en général, les domaines de la Vérification et de la Validation ont également connu un bond significatif avec la mise au point de nouveaux concepts et de nouvelles méthodes, automatiques ou non, pour répondre à ce besoin de fiabilité. Parmi elles, se dégagent notamment deux grandes familles : celle de la Vérification Formelle et celle de la Simulation. Longtemps considérées comme à l'opposée l'une de l'autre, les recherches récentes essaient de rapprocher ces deux grandes familles de méthodologies. C'est dans ce cadre que les travaux de cette thèse proposent une nouvelle approche pour intégrer la Simulation dites à Evènements Discrets aux Méthodes Formelles. L'objectif d'une telle approche est alors d'améliorer les méthodes formelles existantes, en les combinant à la simulation, afin de leur permettre de détecter de potentielles erreurs qu'elles ne pouvaient déceler avant, notamment sur des systèmes temporisés. Cette approche nous a conduit à la mise au point d'un nouveau langage formel, le DEv-

PROMELA. Ce nouveau langage, créé à partir du PROMELA et du formalisme DEVS, est à mi-chemin entre un langage de spécifications formelles vérifiables et un formalisme de simulation. En combinant alors un model-checking traditionnel et une simulation à évènements discrets sur le modèle exprimé dans ce nouveau langage, il est alors possible de détecter et de comprendre des dysfonctionnements qu'un model-checking seul ou qu'une simulation seule n'auraient pas permis de trouver. Ce résultat est notamment illustré à travers les différents exemples étudiés dans ces travaux.

Mots clés : Vérification et Validation, Modélisation et Simulation, Systèmes à Evènements Discrets, Méthodes Formelles, Systèmes temporisés, PROMELA, DEv-PROMELA.

Remerciements

La valeur d'un homme tient dans sa capacité à donner et non dans sa capacité à recevoir.
Albert Einstein

La reconnaissance est la mémoire du coeur.
Hans Christian Andersen

En premier lieu, je tiens à remercier celui sans qui rien de tout cela ne serait arrivé et qui se reconnaîtra sans que je le nomme.

Je tiens à exprimer mes plus vifs remerciements à mes deux directeurs de thèse, Madame Claudia FRYDMAN, et Monsieur Amine HAMRI, qui m'ont donné la chance d'effectuer cette thèse dans d'excellentes conditions, ainsi que pour leur disponibilité, leur attention et leur rigueur scientifique. Leur apport a été inestimable et j'ai beaucoup appris d'eux. Ils ont été et resteront des modèles dans mon travail de chercheur.

J'exprime également ma gratitude à Monsieur Bernard P. ZEIGLER et à Monsieur Eric RAMAT pour avoir accepté d'être rapporteurs de cette thèse, ainsi qu'à Monsieur Vincent ALBERT et Madame Florence SEDES pour avoir accepté d'être membre du jury. C'est un grand honneur pour moi.

Je remercie également Monsieur Mustapha OULADSINE pour m'avoir accueilli au sein de l'unité de recherche, et je remercie également tous les membres du personnel du LSIS et de l'Université d'Aix-Marseille qui m'ont permis de passer cette thèse dans d'agréables conditions, ainsi que toutes les personnes formidables que j'ai rencontrées grâce au laboratoire. Un grand merci également à tous les doctorants, ainsi qu'à toute l'équipe MoFED, à Radhia et Rabah, à notre chef d'équipe Madame Isabel DESMONGODIN, et également à Monsieur Norbert GIAMBIASI dont les conseils m'ont été précieux pour éclaircir ce travail. Grand merci aussi à mes collègues qui m'ont accueilli à bras ouverts durant ces trois années et qui ont contribué à adoucir les moments de stress.

Egalement, un grand merci à ST Microelectronics et Monsieur Jacques PINATON sans qui cette thèse n'aurait surement pas existé, mais aussi au Professeur Bernard P. ZEIGLER et son équipe qui m'ont accueilli et avec qui ce fut un très grand plaisir de collaborer.

Je voulais également exprimer ma gratitude envers tous les enseignants et professeurs que j'ai eu durant l'ensemble de ma scolarité et qui ont très largement contribué à me transmettre le goût du travail acharné, et qui d'une certaine façon, ont permis l'existence de ces travaux de thèse. Merci également à Monsieur Patrice TORGUET et Madame Minica PANCHETTI qui m'ont accueilli lors de mon passage à l'IRIT, mais surtout un grand merci à Monsieur Noel NOVELLI qui est sans doute l'enseignant qui m'a le plus appris et qui a le plus influencé ma façon de penser, en particulier dans le domaine du Génie Logiciel, et qui a donc indirectement influencé l'orientation prise dans cette thèse. Un grand merci aussi à M. VAN CANEGHEM et M. MORIN qui m'ont fait confiance lors de mon passage à l'Université et qui m'ont donné l'occasion de personnaliser mon cursus. Un grand merci aussi à l'ensemble de mes étudiants d'AMU et de Polytech, mais aussi à mes amis et amies, en particulier Paul, Valentin, Alexandre, Alexia, Ludivine, Delphine, Roxanne, Alexia, Sandhya, Claire, Marina, Lucie, Léonore, Ibrahima et Kandel, pour m'avoir donné du courage dans les périodes difficiles et pour m'avoir changé les idées dans les moments délicats. Et je n'oublie évidemment pas Romain, Sylvain, Gregory, Thibault et Thibaut, Toni, Nathalie et tous mes anciens collègues d'Exkee et d'Ubisoft auprès de qui j'ai beaucoup appris, ainsi que toutes les personnes que je n'ai pas citées mais que je n'oublie pas, y compris mes anciens enseignants, Monsieur Christophe BOULLAY et Madame Nathalie BOUQUET, ainsi que mes anciens camarades de l'EPITA et de l'Université. C'est grâce à toute l'expérience que j'ai accumulée auprès de vous que ce travail a été rendu possible.

Enfin, les mots les plus simples étant les plus forts, je voudrais simplement dédier ce travail à mon grand-père Amirdine YACOUB qui m'a certainement transmis, d'une façon ou d'une autre, le goût pour la recherche. Merci pour tout. Sans oublier mes parents, mon frère Taher et ma soeur Chamila à qui j'adresse toute mon affection. Merci beaucoup pour votre soutien, votre amour et votre patience, et merci de m'avoir supporté durant toutes ces années, et en particulier ces trois dernières années.

Contents

Abstract 5

Résumé 7

Remerciements 10

List of Figures 15

List of Tables 16

Introduction 23

1 State of the Art 25
- 1.1 Introduction 25
- 1.2 Definitions of Primal Concepts 26
- 1.3 Theory of Modelling and Simulation 30
  - 1.3.1 M&S as activies: Why Modelling ? Why Simulate ? 30
  - 1.3.2 System Specification Hierarchy and Morphisms 31
  - 1.3.3 Entities 33
  - 1.3.4 Timed Models 36
  - 1.3.5 The DEVS Formalism 38
  - 1.3.6 Hierarchy of Simulation Formalisms 40
- 1.4 Overview of Verification and Validation 43
  - 1.4.1 Verification and Validation in Project, Quality and Risk Management 43
  - 1.4.2 Verification and Validation in Software Engineering 45
  - 1.4.3 Verification and Validation of Simulation Models 50
  - 1.4.4 Model Accreditation and System Certification 54
- 1.5 Formal Verification and Formal Methods 55
  - 1.5.1 Introduction to Formal Methods 55
  - 1.5.2 Model-Checking Theory 57

1.5.3 Timed Automata 62
1.5.4 Hybrid Automata and others methodologies 64
1.5.5 Abstraction and The Great Debate 66
1.6 Towards Integrating Formal Verification and Simulation for V&V 67
1.6.1 Integration of Multiple Formal Verification Tools 68
1.6.2 Combining Formal Verification and Simulation 70
1.6.3 Formal Verification of Simulation Models 72
1.7 Conclusion 73

2 A Combined Formalism: DEv-PROMELA 75
2.1 Introduction to Combined Methods 75
2.2 General Approach 76
2.2.1 Abstract Approach 77
2.2.2 Proofs and discussion 78
2.2.3 Approach using Model-Driven Engineering 80
2.3 Introduction to DEv-PROMELA 82
2.3.1 PROMELA Overview 82
2.3.2 PROMELA in Details 84
2.3.3 Building DEv-PROMELA: Syntax 94
2.3.4 Meaning of DEv-PROMELA : Semantics 103
2.4 Relations and Morphisms 105
2.4.1 Relations between DEv-PROMELA and DEVS 105
2.4.2 Relation between DEv-PROMELA and PROMELA 109
2.5 DEv-PROMELA and Simulation Formalisms Hierarchy 113
2.6 Verification, Simulation, Interoperability and Limits 114
2.6.1 Model-checking and Static Verification 114
2.6.2 Simulation and Dynamic Verification 115
2.6.3 Interoperability 117
2.6.4 Comparison with other PROMELA timed extensions and Limits 118
2.7 Conclusion 123

3 Modelling, Verification and Validation with DEv-PROMELA 125
3.1 Introduction 125
3.2 Framework Entities and Intuitive Relationships 126
3.3 Modelling and Verification of Software 127
3.3.1 Verification techniques 127
3.3.2 Validation techniques 128
3.4 Modelling and Verification of Simulation Models 130
3.5 Integrated Verification and Validation Environment (IVVE) 132
3.6 Conclusion 134

4 Applications : Modelling, Verification and Validation of ... 137

4.1 ... Mutual Exclusion Protocols 137
4.2 ... A Video Game Software : PACMAN 143
4.2.1 Requirement Phase 143
4.2.2 Requirement Analysis 145
4.2.3 High Level Design 145
4.2.4 Low Level Design 147
4.2.5 Verification using Model Checking 150
4.2.6 Verification using Simulation 151
4.2.7 Coding 154
4.2.8 Validation using Simulation 154
4.2.9 What about V&V of simulation model ? 154
4.3 ... A Manufacture Chain : ST Microelectronics' Case Study 155
4.3.1 The Problem 155
4.3.2 Results 158
4.4 Conclusion 159

Conclusion **163**

Bibliography **164**

List of Figures

1.1 Schema of Complexity [ZKP00]. 29
1.2 Homomorphism relation [ZKP00]. 33
1.3 Space of Timed Models. 37
1.4 Hierarchy of discrete-event formalisms. 41
1.5 Schema of V&V Process from [AQH15]. 47
1.6 Waterfall model representation from [DA12]. 48
1.7 V-model representation from [DA12]. 49
1.8 V&V in Software Development Life Cycle proposed by Desai and Abhishek [DA12]. 51
1.9 The Sargent Circle for V&V of Simulation Model [Sar91]. 53
1.10 Simplified V&V proposed in [Pet10]. 54
1.11 Model-checking schema from [BK08]. 56
1.12 Simplified Semantics of LTL operators [BK08]. 60
1.13 A conceptual model of the challenges involved in using automatic verification tools. [Owe07]. 69
1.14 Approach of combining PROMELA specifications with TSM. 71

2.1 Schema of Combined Model Verification Space. The grey arrows means the increasing space by adding data. 76
2.2 Example of mapping UML diagram to ER diagram [LWK10] - Source metamodel, target metamodel, source model (abstract syntax), target model (abstract syntax), source model (concrete syntax), and target model (concrete syntax) 80
2.3 PROMELA Metamodel proposed by McUmber and Cheng [MC01]. 81
2.4 PROMELA Assignment. 85
2.5 PROMELA Assertion. 85
2.6 PROMELA Send statement generated statespace. 86
2.7 PROMELA Select. 87
2.8 PROMELA Loop. 88
2.9 State space generated by the unless structure. 88
2.10 State space generated by the proctype structure. 90

2.11 DEv-PROMELA Assignment. 97
2.12 DEv-PROMELA Event Channel. 98
2.13 DEv-PROMELA Rendez-vous Handshake (receive). [clt:τ →emit:valuea] a?3; 99
2.14 PROMELA selection construct generated statespace. 100
2.15 Generated statespace by the Algorithm 4. 102
2.16 Representation of the relation between DEv-PROMELA, PROMELA and DEVS. 113
2.17 DEv-PROMELA LTL Verification using Simulation. 115
2.18 Modular system using DEv-PROMELA. 118
2.19 Time representation in DT-PROMELA [BD98a]. 120
2.20 Combining Model-Checking and Simulation. 124

3.1 Combined V&V Entites. 126
3.2 V&V in Software Development Life Cycle. 127
3.3 V&V of Simulation Models. 130
3.4 V&V of Simulation Models using DEv-PROMELA. 131
3.5 Combined V&V Environment Architecture. 132
3.6 The DEv-PROMELA Studio Environment. 134

4.1 Example of an invalid path generated by the Program 8 for the property (1). Pi_j indicates the current line executed by the process P_i. 141
4.2 MS4 Me Environment. 142
4.3 DNL model of a Process. 142
4.4 Example of Pacman. 144
4.5 Components Model of Pacman. 146
4.6 Representation of the PROMELA model for Pacman. 153
4.7 Example of Pacman UML Diagram - Red classes are real software classes, while white classes are simulator classes. 155
4.8 The Manufacture Chain. Each square represent a step or a process/-operation. 156
4.9 Example of the operation 1180. 156

List of Tables

2.1 A list of PROMELA basic datatypes. 88
2.2 Comparison between PROMELA timed extensions. 122

4.1 Fisher's Mutual Exclusion Protocol in different timed extensions of PROMELA [NJJ08]. 139
4.2 Results of Verification of concurrent access property. 143
4.3 Comparison between Results of Verification using Model Checking and Simulation. 152
4.4 Results of chain checking. 159

In memory of Prof. Norbert GIAMBIASI.
In memory of my grandfather Amirdine YACOUB.

Introduction

Making reliable systems and software has always been a big challenge in any research fields for decade, and this challenge is even more true in the 21st century. Our world has become connected, and involves billions of systems that communicate with each other at each second of our life. Whether in transport, economics, health, manufacturing, schools, or even business, automation is everywhere. *Simple* systems, if we can qualify them as "simple", have even been replaced by more elaborated solutions, like System of Systems (SoS) concepts [Ack71]. These new technologies are so elaborated that they can take important decisions in microseconds. For instance, the avionics of the Airbus A380 is able to achieve from 10^5 to 10^6 functionalities in few microseconds, in order to pilot the aircraft without any human interference [Iti07; But10; Bie+12]. We can also talk about software that allows companies to earn daily billions of dollars in any stock exchange.

However, because our life strongly depends on them, the slightest mistake is not permitted and it is important to ensure that these huge interconnected systems successfully fulfill their tasks. Therefore, understanding the exact behaviour and the consequences of the several interactions between all of these complex systems is more than important. However, if Verification and Validation procedures [Sha16] have evolved with the complexity of systems, and substantially allowed the improvement of quality, reliability and safety of new products, they are always heavy and costly to implement in the development cycle [KCB02]. A lot of the existing procedures are time- and effort-consuming, and some of existing tools in fact are not able to fully capture the real behaviour, but just focus on a part of the system under study.

At first glance, we can easily divide these approaches in two categories [Dil98]. The first one concerns mathematical studies and formal verification. These approaches try to represent exactly the reality with a plenty of equations. Proofs on these equations then guarantee that the represented systems work as intended, however they need a strong knowledge and strong mathematical background.

The second one concerns empirical techniques. They are based on experiments and on computer simulations. If they are easy to implement and to understand, they face to two major problems: repeatability and consistency. Indeed, simulations and experiments depend more on the human capabilities than automatic proofs, and by extension they are more error-prone, even if many solutions have been developed in order to reduce this risk. To these considerations, another one, which increases the difficult of applying these methodologies, must be added: automation. Because manufacturing needs quick development and deployment, proofs and simulations have also been computerized and automated. However, if making a model that really represents the reality is hard, automatic proofs on these models are in many cases likely impossible. The true fact is that many of these problems are already proved undecidable, especially on timed systems, or need too much time to get a correct answer. As a result, tests and real experiments are more used than automatic proofs, while these latter are applied only on critical systems. Moreover, even in these cases, automatic proofs are applied on a reduced model with a strong hypothesis: if the correctness of the checked property is verified on the model, it will also probably be true on the real system. If we can prove this hypothesis in some cases, experience shows that it doest not guarantee the effectiveness of the result on the real system.

From these findings, research has been oriented in three directions: improving modelling and simulation methodologies, finding new algorithms that reduce the size of models without losing informations in order to perform automatic proofs, and proving that the computerized models used in simulations are trully representations of the real systems that they are supposed to stand for. If these three approaches have similarities, they seem to evolve independently each other. For instance, formal methods use simulation only as a supportive method, while some work recommends to use only simulation to perform validation. Concerning the third approach, it focuses only on the correctness of the computerized simulation model against the specifications of the abstract model or of the simulator, and not trully against the real system. If we look at deeper the literature, we can also see a new tend that tries to combine mathematical and empirical approaches especially in circuit designs. These methods try to select the less time-consuming approach in order to validate models. If the existence of these approaches shows that simulation can be integrated with formal verification, they bring two crucial question: Are formal verification and simulation trully checking the same stuffs ? What exactly the differences between formal verification and simulation results ?

The work presented in this thesis is not revolutionary, and is not intended to make deep changes in the Verification and Validation procedures. It is a part of this new tend that tries to combine formal verification and simulation. The main goal is to introduce an approach that makes easier deep integration between

model-checking and discrete-event simulation. Literature of the two last decades has many attempts to reduce the gap between these two families, but they focus on special cases or see the one or the other just as a supportive method. We tried to propose a more generic approach that really allows combining simulation and formal verification, especially by introducing the DEVS formalism [Zei76; ZKP00] into model-checking tools. Then, we tried to apply our methodology on a real example of software, and we tried to propose a way to use our methodology in a development cycle.

The Chapter 1 focuses on the existing work in the literature about Verification and Validation, Formal Verification and Simulation. Indeed, it is important to well understand the different notions and problems behind the design and the implementation of systems and software. The existing solutions are introduced and analyzed in order to understand the benefits and the limits of them.

The Chapter 2 introduces our approach of integration of a simulation formalism into formal methods. We develop in this chapter a new extension of a *verifiable specifications language* called *DEv-PROMELA* as an illustration of our methodology. The syntactic changes and the semantics mapping are addressed. We also talk about performing model-checking and simulation on DEv-PROMELA models, and how modifying DEv-PROMELA in order to use it with other simulation formalisms.

The Chapter 3 shows the effects of our approach on Verification and Validation procedures. We focus on two aspects: verification and validation of simulation models on the one hand, and verification and validation of software on the other hand. Indeed, because DEv-PROMELA has a kind of reflective property, we can see two levels of verification and validation which can have their importance when checking software.

The Chapter 4 addresses three cases of study of Verification and Validation of software.

Finally, the conclusion recaps the important points of the work and discusses about the weakness and the limits of such an approach. It also presents some future work to do in order to make it more robust and more effective.

Chapter 1 State of the Art

The only true wisdom is in knowing
you know nothing.
Socrates

Any fool can know. The point is to understand.
Albert Einstein

1.1 Introduction

Nowadays, we cannot deny that systems, software or hardware, have become more and more complex. They involve more and more interactions and the amount of exchanged data is constantly increasing. Moreover, development methodologies have enormously changed from small team performing small tasks to large operations involving hundreds or thousands of people. With the increasing of complexity of systems and software, and the growing size of teams, development processes and development life cycles have deeply changed [Sha16] in order to take into account these new considerations. Moreover, the need of reliability and quality, especially in critical systems, makes more important evaluation procedures. As a consequence, Verification and Validation (V&V) procedures have also deeply changed from informal processes performed by the engineer himself during the development to a separate activity performed during the overall development life cycle [And86].

Furthermore, over the 20 past years, plenty of new V&V concepts, methods and tools have been proposed to make safer systems and to make more efficient V&V. Some research fields, like Formal Verification (FV) or Simulation, have developed, in the same time, specific techniques related to particular systems.

This sometimes leads to divergent interpretations of a same problem, while it is well-accepted that all these activities are complementary.

The objective of this chapter is thus to clearly define key concepts behind the V&V procedures. Complementary tasks like Certification and Accreditation are also tackled. Once the definitions of the main notions are given, we make a survey of the existing methods and tools proposed by the Formal Verification and Simulation research fields. We will also be interested in existing approaches that combine all these methods.

1.2 Definitions of Primal Concepts

Before entering the real subject, it is important to give clear definitions which will be used in this thesis. Indeed, as stated by Gaudel [Gau11], literature about Verification and Validation are facing up to multiple concepts and approaches, due to advances in these domains. The main problem is that terminology and vocabulary are often misused, and sometimes appear as in opposite while they mark out the same thing. Sometimes, terms are really so vague that we do not know exactly what we are doing or talking about. We do not pretend that the definitions that we will give in this section are the most appropriate, but they will make more clear the work introduced in this thesis.

System The notion of *system* is touched on in the almost all research works and all the scientific domains. Regardless of the domain, physics, economics, sciences, etc. all people talk about *systems*. But what is exactly a *system* ? If we refer to the definition given in [Gau11], a system is:

Definition 1: System

A dynamic entity of the real world, and which can be observed only thanks to some limited interface or procedure.

A system is therefore a complex thing whose the real behaviour cannot be exactly captured. The only way to understand it is to execute it, namely giving some inputs, and then observing the outputs. Zeigler, Kim, and Praehofer [ZKP00] talk about "a black-box" and define the behaviour of the system as "the relationship between input time histories and output time histories". As a consequence, the real behaviour of a system can never be entirely and exactly defined. In fact, only its expected behaviour can be known through some *specifications*, *models* or *programs*. Then testing a system is just equivalent to stimulate it on a finite set of chosen inputs. This definition includes software, for which unexpected behaviours can also occur and lead to errors, defects/bugs, faults or failures.

Program The second question is: what is exactly a *program* ? Indeed, software can be programmed and behaviours of programs are reputed to be well-known. The same author [Gau11] says that a program is:

Definition 2: Program

A piece of text written in a well-defined language. In some case, it is annotated by assertions: pre- and postconditions, invariants, that are formulas in another well-suited language.

A program is therefore like a text which can be read and that have an understandable meaning. Then, we can formally reason on a program either by using the rules of the operational semantics of the programming language, or by using a formal system that considers annotated programs. So, the behaviour of a program is fully known and can be proved. If the language is a formal language, with a formal semantics, we talk about *formalism*. However, in computer sciences, it is well-known that a model is obtained thanks to a modelling language or a formalism. Then, what is the difference between a model and a program ?

Model The term of *model* is certainly the most difficult to define, as its meaning depends on the domain. A first definition given by Gaudel [Gau11] says that:

Definition 3: Model

Models or specifications are something which omit the details that appear in the program, and used for system description, design and analysis.

In computer sciences, there is a lot of types of models depending on the kind of requirements, but it is generally a state-transition structure that represents the behaviour of a program.

In physics, a model may be a differential equation. In biology, it is often an homogeneous population of mice or frogs...

while Petty says in [Pet09; Pet10] : "In general terms, a model is a representation of something else, e.g., a fashion model representing how a garment might look on a prospective customer".

If we refer to the definition of Minsky [Min65], a model is

Definition 4: Model

To an observer B, an object A* is a model of an object A to the extent that B can use A* to answer questions that interest him about A.

If we look at the definition given by the mathematics,

Definition 5: Model

A model is a description of a system using mathematical concepts and language. Furthurmore, a model of a theory is a structure (e.g. an interpretation) that satisfies the sentences of that theory.

A model cannot therefore represent all the reality [Rot89]. A model is then an abstraction, a simplification, a cheap representation of something and in which interesting properties are verified. These properties are called *specifications* or *requirements*.

Specifications Petty [Pet09] says that "the requirements specify which aspects of the object of interest must be modeled, and for those to be included, how accurate the model must be. The requirements are driven by the intended application". Gaudel [Gau11] notes that there exists "other sorts of high-level descriptions, that could be also called models, are based on logical formulas: set of axioms, pre- and postconditions, predicate transformers, etc.". These models are called *formal specifications* [Lam00].

In other words, a *specification* is

Definition 6: Specification

A description of an expected behaviour of a model. A collection of properties some system under study should satisfy, at some level of abstracton.

Using these definitions, we can deduce that a model of a program is more abstract than the model of the system. Specifications are models for model, models are models for program, and programs are models for system. Consequently, we can deduce that any object which can be manipulated and whose the behaviour can be fully described **is a model**, including software source code.

This statement implicitely makes a relation between models. Indeed, if a program is a model of system, and a model is a representation which has less details than the program, we can deduce the model is something like generic or general. In fact, a model lacks details that are not interesting for its intended use.

Abstraction This relation between models is called abstraction [CC77] and defined by Zeigler, Kim, and Praehofer [ZKP00] as

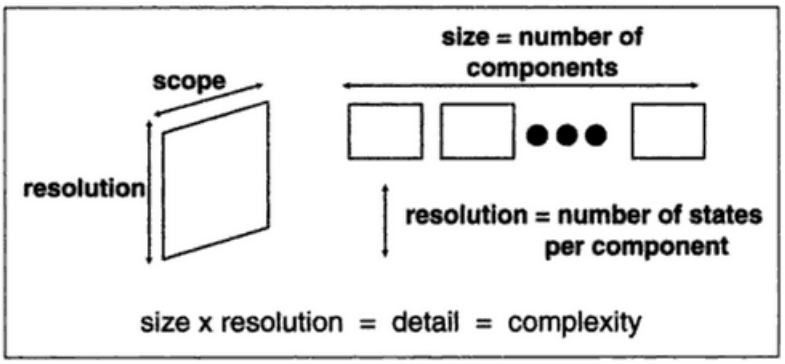
Definition 7: Abstraction

A method, process or algorithm that extracts, from a complex reality, set of entities and relationships. Applied to a model, abstraction reduces its complexity while preserving its validity. It is a *valid simplification*.

Therefore, abstraction is way to represent a *low-level* model with fewer details. The reverse process, from a *high-level* model to a *low-level* model with more details is called *refinement* [WWZ10]. We thus talk about *level of abstraction* and *accuracy*. Abstraction and Refinement are essential because they are morphisms that preserves certain properties between models. The simplication makes the study of a model more easier but reduces its accuracy, while the refinement makes that the model is closer than the represented object, but increases the difficulty of its study.

Complexity of a Model Then, to each model, it is possible to associate a *complexity*. Zeigler, Kim, and Praehofer [ZKP00] define the complexity of a model as a product (figure 1.1) of

Figure 1.1: Schema of Complexity [ZKP00].



1. size : the number of components (submodels) of a model;
2. resolution : the size of each component in term of states;

Then, the authors define three types of complexity:

- the *analytic* complexity which is strictely the definition given above;
- the *simulation* complexity which measures the resources needed to execute the model (memory size, time, etc.);
- the *exploratory* complexity which characterizes the resources needed to explore a statespace.

We can understand that the objective of modelling is therefore the reduction of the complexity by keeping as much as possible accuracy and details.

1.3 Theory of Modelling and Simulation

All these concepts underlie an important theory: the Theory of Modelling and Simulation (M&S) [Zei76; ZKP00] is a well-known theory whose the objective is to make uniform the concepts of *modelling* and *simulation*, which are extensively used in many disciplines like medicine, physics, etc. It also defines a global and universal framework and methodology for modelling and simulation, and which is not dependent on the domain of application.

1.3.1 M&S as activies: Why Modelling ? Why Simulate ?

The previous definitions suggest that it is impossible for human people to understand the reality without making representations of objects of their environment. Therefore, modelling is, as said by Rothenberg [Rot89], just the ability to think, imagine and communicate using signs and languages. Modelling is a way to deal with the reality without its complexity, a way to experiment things that we are not able to test in the real world. Suggested, all cognitive operations that we do are relative to a model and never the system (except the real and physical manipulations). This implies that we are able to understand and analyze models and not systems. Rothenberg defines three attributes that characterize the modelling process, and which summarize the definitions given before:

- Reference: the "of something";
- Purpose: the intended use of the model;
- Cost-effectiveness: it is better to use the model than the reference for the intended purpose.

The simulation activty is therefore the reproduction of the behaviour of an existing system in order to understand it. In fact, modelling cannot exist without simulation because modelling is the activity of constructing a representation of something in order to understand it, while simulation is the fact of using this model in order to understand what it represents. This means that the notion of modelling encompasses the concept of simulation. Consequently, we can even consider *model analyzing* as a simulation. In order to make clear these notions, Sokolowski and Banks [SB11] reserve the term of modelling for "model building", and simulation for:

- a method for implementing model over time;
- a method for testing, analyzing and training;
- a method for extracting information from a model by observing its behaviour;

- imitation.

Then, [BC86; Ban98; Mar97] say that simulation would be used when:

- it is impossible or extremely expensive to observe certain processes in the reality, or to interact directly with them;
- the real system has some level of complexity, interaction or interdependence between various components, or pure size that makes it difficult to grasp in its entirety;
- there is no simple analytic model or it is impossible or extremely expensive to validate the mathematical model describing the system.

In fact, the existance of simulation implies the incapacity of even understanding a model, which is always too complex to be analyzed. Indeed, modelling is the process of simplifying the reality in order to get a grasp of it. If simulation, in the sense of executing, is needed, then that means the model is always too hard. Then, why not making another model of this model ? This is the start point of the model-checking *best practices* which considered the model must be as simple as possible. In fact, simplification involves loosing the meaning of certain parts of a model for its intended use.

If simulating helps to understand, it has also major drawbacks:

- often simulations are time- and data-consuming, and costly;
- simulation is a reproduction of the behaviour of a system through a model, meaning that if the model is not correct (invalid model or erroneous assumptions), the result is not guaranteed, like in any model-based methodology;
- if the simulation is done through another system (like a simulation software), errors can come from defects of this system;
- simulation results can be difficult to interpret.

Fortunately, literature provides a plenty of works for making *correct* models and simulations.

1.3.2 System Specification Hierarchy and Morphisms

The underlying basis of the Theory of Modelling and Simulation [Zei76] is relatively simple. It assumes that basically a system behaves over time, meaning that inputs and ouputs evolve according to a duration. As a consequence, the model would have to use time as a basics of what it is representing. From this postulate, Zeigler introduces a system specification hierarchy based on the Klir system hierarchy [Kli85] which helps the designer to describe the system under study, in order to produce a model that represents its *dynamics*:

- At the level 0, the description gives the variables to observe, the inputs used in order to stimulate the system, and how to observe their dynamics over time;
- At the level 1, the description consists on a time-indexed input/ouput pairs;
- At the level 2, the initial state is known, and each input produces a unique output;
- At the level 3, the state structure is highlighted;
- At the level 4, the model describes the interaction between components (which can be themselves subsystems).

However, just decribing system at these levels is equivalent to make several models of the system under study. Then, if one of these models statifies a specification, it is important to ensure that the others models also satisfy it. For that, it is important to make a relationship between the levels. This is done by a simple association, by construction. Furthermore, because a model is just a representation of a system at one of the level of specifications, it is also important to be able to establish a correspondance between different models at the same level of specifications. Zeigler defines the concept of *morphisms* for that.

Definition 8: Morphism

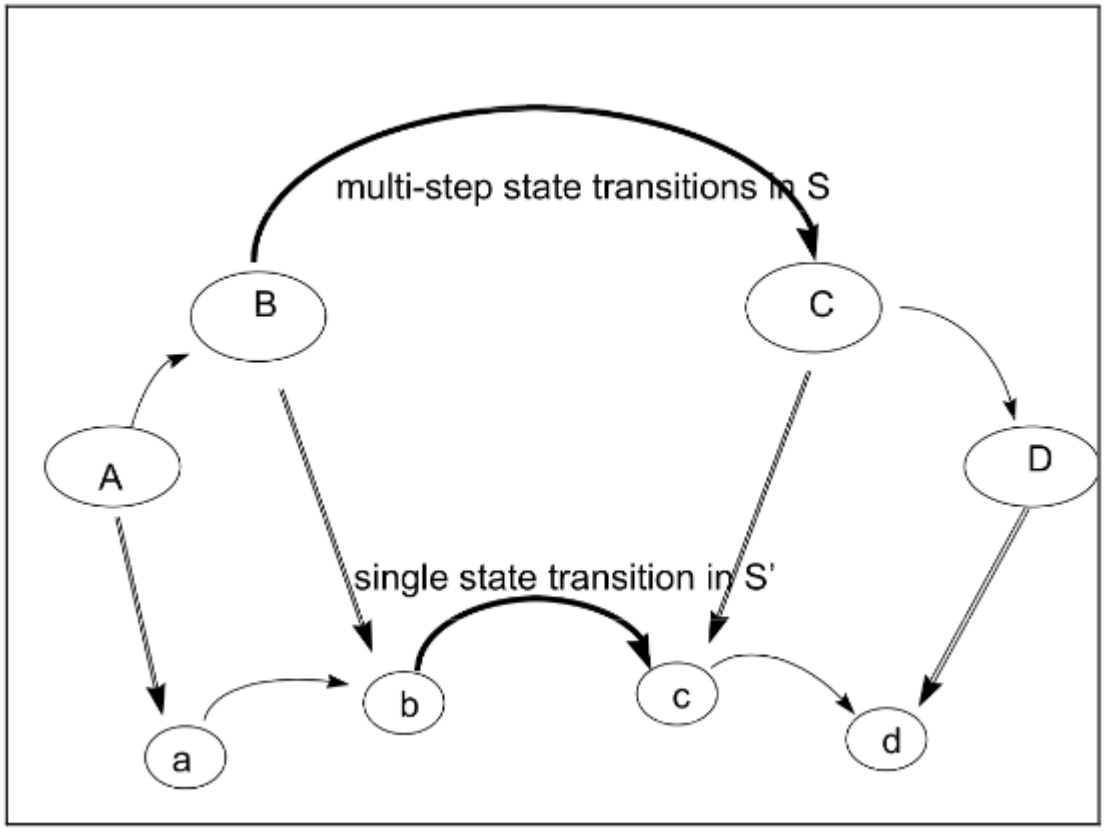
A morphism is a relation between two system descriptions that places in correspondance elements at the same level of specifications.

These two concepts imply that if a morphism exists between two models at an upper level, then there exists a morphism at a lower level that holds the properties satisfied at the upper level. In other world, a model, at any level of specifications, is a representation of a system if it is possible to establish a morphism between the real system (or what we know about it) and the model. Then, as long as we can make a relation between models, we can consider the model as a good representation of the system, meaning that it holds all the properties of the modelled system. In other words, we can say that this is a model in the mathematical meaning (definition 5). Furthermore, the concept of morphism establishes a relation between two different systems. This seems obvious, but this statement gives an important result: given two models, if we can prove the existence of any relationship between them and that the morphical properties given in [Zei76] are preserved, and if these models are expressed at same level of specifications, all the properties satisfied by one of the models are satisfied by the other.

Especially, [Zei76] defines a specific morphism called *homormorphism*. Two state-transition systems are homomorphic if we can place in correspondance

theirs states, and if for each path on the first transition system, there exists a path, in the second one, that links the corresponding states (figure 1.2).

Figure 1.2: Homomorphism relation [ZKP00].



That means two homomorphic state-transition models hold the same properties and are valid representations of one or several aspects of the system under study.

1.3.3 Entities

Now we have given elements for turning systems into models, we talk about the main relationships that support the Framework of Modelling and Simulation. Zeigler [Zei76] and Petty [Pet09] define four entities linked by two mains relationships.

The system or simuland this is the object that we want to model, with its environment. Thus, the simuland is composed by the data gathered from the observations and the experiments done on the real system, or the hypothesis done about its behaviour. Zeigler, Kim, and Praehofer [ZKP00] introduce the

notion of experimental frame, which is a specification of the conditions under which the system is observed. In other words, the experimental frame is the reason of modelling the object. Petty [Pet10] calls that *requirements*, and this is exactly the definition of the experimental frame. Therefore, an experimental frame is just a specification. A system can be viewed from many experimental frames, and that is the experimental frame which *validates* the model through the *validity* relations, whose one is important for this thesis: the structural validity.

The model This is what we are talking about the beginning of this chapter. Zeigler [Zei76] defines it as a system specification at any of level discussed before, while Petty [Pet10] distinguishes two notions: the *conceptual* model and the *executable* model. The first one is the model as a representation of the simuland. Liu, Yu, Zhang, et al. [Liu+11] suggest that the conceptual model is composed by

- The simulation context which is a set of requirements;
- The simulation concept which is a set of constraints from the simulation environment;
- The simulation elements which are the model itself.

Nance [Nan81] identifies the core elements as:

1. A model is a set of objects and their relationships, equivalent to the level 4 of the specification hierarchy;
2. An object is a set of attributes to which values are assigned;
3. Attributes describe an aspect of an object, or an aspect of a relation;
4. Values are something that gives a meaning to the attributes (like numerical values);

The executable model is therefore a model which can be executed, meaning a computer program. Sargent [Sar91] calls that the *computerized* model or the *simulation*[a] model.

[a] Carson [Car04] considers a simulation model as "a representation that incorporates time and the changes that occur over time". We don't retain this definition which is the definition of the conceptual model in general.

The simulator Zeigler, Kim, and Praehofer [ZKP00] calls *simulator* an agent, a system which can read and execute the instructions given by a conceptual model, and which generates its behaviour. In general, a simulator can be viewed as a generic computer or software, or even the previous simulation model. Indeed, we just have seen that a simulation model is a computer program which can be executed. However, identifying the simulator to the simulation model can lead to some misunderstanding in the next of this thesis. That is why we use the following definitions:

Definition 9: Simulator

On the one hand, we call *simulator* the software, represented by its source code, which executes the model, independantly of how it is generated (meaning a generic simulator or a model-specific simulator).

Definition 10: Simulation Model

On the other hand, the simulation model is any model of a simulator.

A simulation relation is defined between the simulator and the model: the simulator correctness. A simulator is correct if it is guaranteed that it reproduces the behaviour of the model, meaning it generates the correct outputs given inputs. These all definitions allow us to establish a stronger relation: if there exists a morphism between a simulator and a simulation model, and if there exists a morphism at level 2 of the previous hierarchy between the simulation model and the conceptual model, then the simulator is correct.

It is important to note that if a model is correct, it can be also seen as a valid simplification, in other word as the result of an abstraction process.

Simulation A simulation is therefore just the execution of a model over time in order to generate its behaviour by acting on its inputs and its parameters [Zei76; Pet10], according to the experimental frames. We can thus easily understand why simulation is an empirical methodology, which strongly depends on the observations done on the real system. The second most important thing in simulation is *time*. While *executing* consists on launching the real system and using it in a physical real time (which can be measured with a real clock), simulation consists on stimulating a model using a logical time which is somehow embedded in the model. However, if time is not taken into account in the simulation, or if the goal is only to generate the statespace, we talk about *execution* of a model or *animation* of specifications [Bic+97; MS10].

All these notions are important because it means that it is possible to simulate a simulator, while executing a simulator is making a simulation of a conceptual model. This difference is the base of all of the work that we will introduced in the next chapter.

1.3.4 Timed Models

Because simulation models include time in their definition, it is important to be able to represent this attribute. Modelling time involves defining certain concepts [Nan81; Ban+10]:

- An instant is a value of system time at which the value of at least one attribute of an object can be assigned (altered).
- An interval is the duration between two successive instants.
- A span is the contiguous succession of one or more intervals.
- The state of an object is the enumeration of all attribute values of that object at a particular instant.
- An event is a change in object state, occurring at an instant. An event is fully determined by its occurence, in other word by a function of time.
- An activity is the state of an object over an interval, meaning between two events.
- An object activity is the state of an object between two events describing successive state changes for that object.
- A process is the succession of states of an object over a span (or the contiguous succession of one or more object activities).

Then, litterature defines four main paradigms for modelling time.

Untimed/Timeless Model This kind of designs does not explicitely model the time, meaning that the evolution of variables is not expressed in a function of time. That is the case for example of the famous Moore and Mealy machines. Time is implicit in the way that the model is generally interested in the possible execution paths. Then, the order of the executed transitions implicitely models the time. Untimed model are appropriate for modelling timed-constant system or simplifying model. It is certainly the less expensive representation of time and the easier to deal with.

Discrete-Time Model Discrete-time models are the most intuitive from all the four paradigms. They model time as a tick that occurs at a fixed rate. Thus, the interval between two tick is always constant, allowing a stepwise execution of the model like in untimed models. Time is then a multiple of ticks, which gives an explicit time measure. This kind of time representation involves an explosion of the complexity of the model, because this latter is evaluated even when no change overcomes.

Continuous Model In continous models, time is considered as continuous. Then, the current state, meaning the values of the variables at time t, is expressed via a differential equation. Simulating a continuous model generally needs a costly integrator or needs to discretize the space in order to derivate the time function.

Discrete-Event Model There are models of systems whose the state changes at various time instants, depending on instant occurrences of events. Between two events, the model is constant, whereas it changes its state only when an event occurs. This event can be the result of an autonomous behaviour or a reaction to an incoming event. It is certainly the best compromise between efficiency and accuracy.

Figure 1.3: Space of Timed Models.

All the stakes of timed modelling is to know if all time modelling are equivalent (figure 1.3). Indeed, if there is the case, all the computational aspects can be reduced in the simpliest model (the untimed model). The second representation expresses the notion of abstraction, while the third representation suggests that untimed models and timed models finally represent different things. A fourth representation can also be imagined: the existance of an intersection between timed and untimed models.

1.3.5 The DEVS Formalism

A first answer to this question is given by the Discrete-Event System Specifications (DEVS) formalism, introduced by Zeigler [Zei76]. The DEVS formalism can be seen as an generalization of the Moore Machine formalism by associating each state with a lifespan. DEVS was proved to be unique and universal, and allows modelling of a large variety of systems thanks to its multicomponent and multiformalism capabilities. The Classic DEVS relies on the following notions:

- Each state is associated with a real number called lifespan. This real number can take its value on $[0; +\infty]$. When the lifetime of a state has expired, the system emits an output and changes its current state according to the transition table;
- When an input is consumed, the state of the system changes according to the transition table, regardless of the current lifetime of the current state;
- As a result of the previous point, transitions can be characterized as internal or external transitions. Internal transitions model autonomous behaviours while external transitions correspond to reactions to any external events;
- Events are well-dated and can be ordered;
- There is no non-deterministic behaviour. If two events occur at the same time, thus either they are equivalent events ($e_1 = e_2$) or they are prioritized;
- The state, input and output trajectories are piecewise segments; the distribution of events can follow any non-linear function, unlike for discrete-time systems in which the time is determined by a linear function of periods;

DEVS Atomic Model More formally, a DEVS model is a coupling of DEVS atomic models. A DEVS atomic model is the smallest simulable unit defined by

$$A = (X, Y, S, \delta_{int}, \delta_{ext}, \lambda, ta)$$

where:

- X is the set of input values;
- Y is the set of output values;
- S is the set of states;
- $\delta_{int} : S \to S$ is the internal transition function;
- $\delta_{ext} : Q \times X \to S$ is the external transition function;

- $\lambda : S \to Y$ is the output function;
- $ta : S \to \mathbb{R}^+$ is the time advance function;
- $Q = \{(s, e) \mid s \in S,\ e \in [0, ta(s)]\}$ is the total state set; e is the time elapsed since the last transition.

DEVS Coupled Model Then, a DEVS coupled model is defined by

$$M = (X, Y, M, EIC, EOC, IC, Select)$$

where:

- X is the set of input values;
- Y is the set of output values;
- M is the set of components (atomic or coupled models);
- EIC is the external input coupling that connects external inputs to component inputs;
- EOC is the external output coupling that connects component outputs to external outputs;
- IC is the internal coupling that connects component outputs to component inputs (without direct feedback loops);
- Select is the tie-breaking function that chooses the next event from the set of simultaneous events.

Semantics of DEVS The meaning of a DEVS is given by its abstract simulator and can easily be depicted as follow. At any time t, the system is in a state s. If no external event occurs, the system stays in s for time $ta(s)$. If the lifetime expires, meaning the elapsed time e from the last event is equal to $ta(s)$, the system outputs the value $\lambda(s)$ and changes to the state $\delta_{int}(s)$. If an external event x occurs before the expiration time, meaning that the system is in a state $q = (s, e)$ with $e \leq ta(s)$, then the system changes its state to $\delta_{ext}(q, x)$. The event can transit into the coupled model using the previously defined coupling: an external event coming in the system is transmitted to the components using EIC, while an output generated by a component transits using EOC or IC.

Closure under coupling One of the best properties of DEVS is the closure under coupling. Given a coupled model, Zeigler [Zei76] shows that it is possible to obtain an atomic DEVS which behaves exactly as the coupled model. This property therefore ensures that a DEVS hierarchical model is always simulable, if the DEVS is really the expression of a system.

Legitimacy This property sets the condition under a DEVS structure is indeed a system. It is shown that this property is verified if always the time advances during the simulation [ZKP00].

Simulation model of DEVS The power of DEVS comes also from the separation between the conceptual model and the simulation model. This is a key notion because it makes independant the model of the system under study and the implementation of the simulator. However, the simulator of DEVS is obtained thanks to two mapping:

1. Associate to each atomic model a simulator which makes operations on states and transitions of the model;

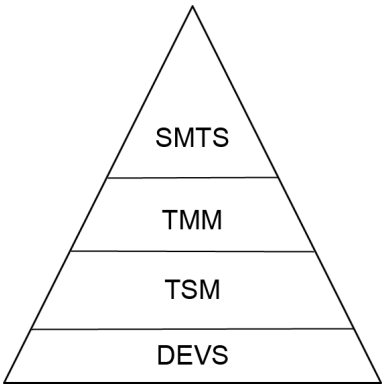
2. Associate a coordinator to each coupled model which computes the time advance function, propagates events and synchronizes sub-models.

These mapping have for consequence that the structure of the simulation model is morphic to the structure of the conceptual model (at the level 4), while each atomic simulation model is homomorphic to its corresponding atomic conceptual model. Moreover, this hierarchical DEVS has the advantage to allow formalisms interoperability, and modular, multicomponent, multifacetted and multiformalism modelling [Zei84; ZKP00].

1.3.6 Hierarchy of Simulation Formalisms

The literature about DEVS Modelling and Simulation is too large to be exhaustively tackled in this work. Many extension of DEVS were developed to model particular aspects of systems like Generalized DEVS [GC06], Real-Time DEVS, RTA-DEVS [SW10], Fuzzy DEVS, etc. Each extension encapsulates the others, increasing the DEVS modelling capabilities. Some of these extensions have been classed in hierarchies [Wai09; Gia09; Hwa11; Hwa14] according to their expressiveness. We choose to use the hierarchy proposed in [Gia09] which can be represented as a hierarchy of top-down pyramids (Figure 1.4), in which the pyramid at the top represents the less expressive and most restrictive formalism, and the pyramid with the lowest base represents the most expressive and less restrictive formalism.

Figure 1.4: Hierarchy of discrete-event formalisms.



Sequential Machine With Transitory States (SMTS) SMTS allows representing reactive systems which will react to events only when they are in a *steady state*. For instance, an old analogic elevator which cannot be stopped when going up can be modelled as a SMTS system. Formally, a SMTS model M is defined as a 7-uplet:

$$M =< X, Y, S, \delta_i, \delta_e, \lambda >$$

where

X is a finite set of input events;

Y is a finite set of output events;

$S = S_T \cup S_S$ where $S_T \cap S_S = \emptyset$, S_S is the set of steady states (i.e. the model remains in these states until an event occurs), and S_T is the set of transitory states;

$\delta_i : S_T \rightarrow S$ is the internal transition function which describes the autonomous behaviour of the model;

$\delta_e : S_S \times X \rightarrow S$ is the external transition function which describes the reactive behaviour of the model;

$\lambda : S_T \rightarrow Y$ which defines the output event generated by the model when it transits from a transitory state to the next state.

We must also denote that the notion of timed events is not defined in SMTS.

Temporal Moore Machine (TMM) TMM can be viewed as an extension of SMTS, with the introduction of two concepts (timed event and state lifespan). TMM is very useful to describe systems which will perform tasks when they receive an event, and which will then immediately return to a passive state. Thus,

formally a TMM model M is an 8-uplet:

$$M =< X, Y, S, \delta_i, \delta_e, ta, \lambda >$$

where

X is a finite set of input events;

Y is a finite set of output events;

$ta : S \to \mathbb{R}^+ \cup \infty$ defines the lifetime of each state;

$S = S_T \cup S_S$ where $S_T \cap S_S = \emptyset$, S_S is the set of steady states (i.e. $s \in S_S \Rightarrow ta(s) = +\infty$), and S_T is the set of transitory states (i.e. $s \in S_T \Rightarrow ta(s) \in [0; +inf[$);

$\delta_i : S_T \to S_S$ is the internal transition function;

$\delta_e : S_S \times X \to S_T$ is the external transition function;

$\lambda : S_T \to Y$ is the output function defined only for transitory states.

We can denote the restriction on δ_i and δ_e which enforces that a steady state is followed by a transitory state, and a transitory state is followed by a steady state. In fact, in semantics of TMM and SMTS, this is not a restriction. Because the notion of lifespan and timed event were introduced, a succession of transitory states in SMTS is compressed in one transitory state with a finite lifetime in TMM.

TMM also supports coupling which allows modular building of models. This notion is important because it introduces the capability to model complex systems by components which could interact with each other.

Temporal Sequential Machine (TSM) TSM is a distension of TMM by allowing events to occur in transitory states. TSM allows the modelling of a new elevator which can stop at each level even if a call button is pressed during an elevation phase. A TSM model M is an 8-uplet:

$$M =< X, Y, S, \delta_i, \delta_e, ta, \lambda >$$

where

X is a finite set of input events;

Y is a finite set of output events;

$ta : S \to \mathbb{R}^+ \cup \infty$ defines the lifetime of each state;

$S = S_T \cup S_S$ where $S_T \cap S_S = \emptyset$, S_S is the set of steady states (i.e. $s \in S_S \Rightarrow ta(s) = +\infty$), and S_T is the set of transitory states (i.e. $s \in S_T \Rightarrow ta(s) \in [0; +inf[$);

$\delta_i : S_T \rightarrow S$ is the internal transition function;

$\delta_e : S \times X \rightarrow S$ is the external transition function;

$\lambda : S_T \rightarrow Y$ is the output function defined only for transitory states.

We can note that restriction of the succession of input/ouput events was removed, allowing TSM without autonomous cycle.

Using such a hierarchy helps designers to focus on interesting models and involves implicit requirements. At least, the structure imposed by each formalism must be respected by the model (otherwise, there is not a model !). This notion is important for model validity and ensures that the model is a good representation of the system. We also saw that models are used to understand the reality, and to validate hypothesis about an existing or a future system. This process is called *Verification and Validation*. However, we said that the only things we can extract from a model concern the model itself. Then, a question can be asked: are informations extracted from a simulation really validating assumptions on the real system ?

1.4 Overview of Verification and Validation

Verification and *Validation* became two common words in many industries and institutions : Software and Computer Systems, Food and Drug, Health Care, Traffic and Transport, Model, Civil Engineering, Economy, etc. The literature about these two activities is also furnished, many works attempt to verify and validate something. However, the first main question that is important to answer is : what are exactly Verification and Validation ? Depending on the point of view, many definitions of Verification and Validation can be found in the litterature. Moreover, as Tran [Tra99] stated, terms *Verification* and *Validation* are often misused or used to refer the same thing, while they denotes two different processes. It is thus important to clearly distinguish the differences between Verification and Validation. We will study them through three points of view: software development, systems, and modelling and simulation.

1.4.1 Verification and Validation in Project, Quality and Risk Management

The two base definitions of Verification and Validation are given in the Project Management Body Of Knowledge (PMBOK Guide [Ins04]).

Definition 11: Verification

The evaluation of whether or not a product, service, or system complies with a regulation, requirement, specification, or imposed condition. It is often an internal process. Contrast with *validation*.

Verification is thus an activity that checks that a system meets its development constraints and specifications. In other words, Verification consists on ensuring that the final resulting system fits to the initial model.

Boehm [Boe81; Boe91] summarizes this definition by saying about Verification : "*Are we building the system right ?*" This means that the goal of Verification is thus not to check if the product meets the needs of the final customers, but only to check if the product is well made. For example, typically verification techniques in machinery concerns [SK97] design qualification (is the equipment well acquired ?), installation qualification (is the equipment well installed ?), operation qualification (is the equipement working ?). However, if these three procedures are performed from the final user point of view, meaning these tests consist on verifying that the equipment suits to the uses of the final customer, they fall into the Validation process.

The PMBOK [Ins04] defines Validation as:

Definition 12: Validation

The assurance that a product, service, or system meets the needs of the customer and other identified stakeholders. It often involves acceptance and suitability with external customers. Contrast with *verification*.

According to this definition, Validation is an external activity that checks that a system meets the real operational needs of the final user. Validation thus focuses on the intended uses of the product. Boehm [Boe81] says about it: "*Are whe build the right product ?*". Unlike the Verification in which requirements are defined by the designers, Validation requirements are expressed by the customers. Validation checks for example that the characteristics of a future product will meet the safety requirements defined in the legislative rules (prospective validation), or that an existing product meets the initial expectations (retrospective validation). Validation also concerns full-scale validation (using a real sample), partial validation (tests concern only critical aspects), cross-validation (if sample is generated by using two or more methods, it is validated against each method), revalidation (validation is periodically performed), concurrent validation (validation are duplicated and carried out by a lot of validators).

If we consider only these definitions and these activities, we can clearly understand why Verification and Validation are often confounded. Indeed, the only apparent difference between these two activities is that Verification is done from the developers' point of view, while Validation is done from the users' point of view. They are both related to requirements, and at this time, differences between design and final specifications are not clearly defined. Moreover, if we consider V&V activities themselves as products, it appears that we can verify and validate V&V procedures. Verifying a verification/validation procedure would mean that we check that this procedure is well built — i.e. it has all the verification/validation steps. However, these steps clearly depend on their intended uses. Validating would mean that we check that any modifications performed on the verification/validation procedures will lead to modifications on the final product; and, that modifications must ensure the resulting product always fulfills its design requirements. In other words, validating a verification/validation procedure is related to the same properties than verifying a verification/validation procedure. Then, how te be sure we are verifying or validating ?

In fact, the difference between V&V is mainly the "stage of the product" in which they are applied. On the one hand, Validation is mainly performed only on the *release candidate* or the *release* version of the product, and mainly concerns tests on [Gre96]:

- Selectivity/Specificity: is the product has the characteristics needed by the customer ?
- Accuracy : is the product answering the needs with precision ?
- Repeatability : are the tests repeatable ?
- Reproducibility : are the tests reproducible ?
- System suitability : are the product able to fit to changes of its conditions of use ?

On the other hand, Verification is performed at each step of the development in order to check that the "used tools" are well used for building the product.

The most important thing is that V&V refers here to the real system.

1.4.2 Verification and Validation in Software Engineering

This difference between Verification and Validation can be easily understood if we look at the definitions given by the Software Engineering discipline. According to IEEE Standard [IEE90; IEE12; Abr+04] and the Capacity Maturity Model [Pau+93], Verification and Validation are :

Definition 13: Software Verification

The process of evaluating software to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase.

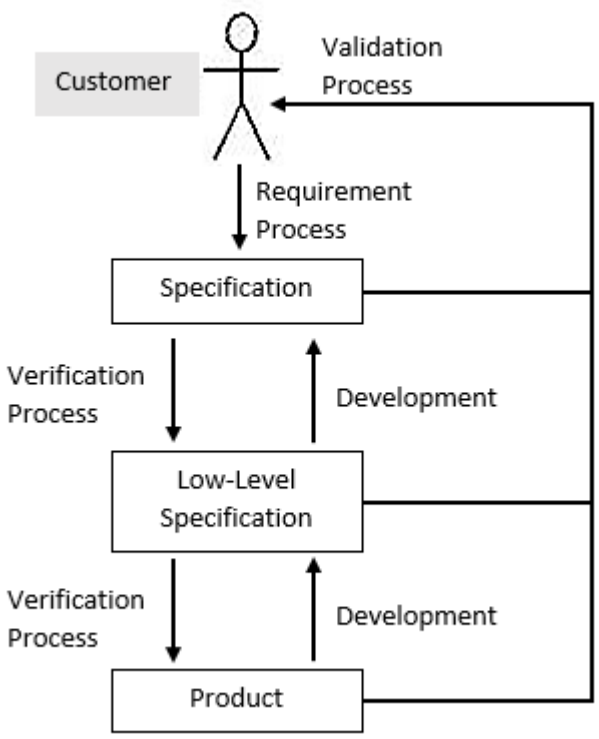
Definition 14: Software Validation

The process of evaluating software during or at the end of the development process to determine whether it satisfies specified requirements.

Tran [Tra99] and Desai and Abhishek [DA12] then use these definitions to clearly defines Verification and Validation. According to these authors, Verification consists on checking that at the end of each phase of development whether the software behaviour conforms to the input defined at the start of the phase. Verification does not care to know if input – i.e. the requirement specifications – are correct and complete, but only assesses whether the system acts as it was defined for or not. As a consequent, Verification cannot detect errors resulting from incorrect specifications, and it is not its purpose. Consequently, errors may propagate without detection through next stages.

On the opposite, Validation checks, at the end of development (figure 1.5) whether software acts as expected by the final user. Then, Validation tries to detect errors that come from incomplete or incorrect specifications, by checking the correctness of the input to a phase, and whether incorrect requirements have introduce defects in the system. Therefore, validation allows pointing out errors in inputs/ouputs while verification only shows that the software doesn't act as intended for a given set of specifications. Through this definition, reader can see appearing in background the system specification hierarchy defined in the previous section. Validating is then strongly connected to the experimental frame, while verifiying is strongly connected to morphisms.

Figure 1.5: Schema of V&V Process from [AQH15].



As for V&V in Management, the author arrives to the conclusion that Verification answers to the question *Are we building software right ?* while Validation answers to the question *Are we building right software ?*. Then, the objectives of Software Verification and Validation can be grouped in five general activities, as said in [Col88] and in another version [DK12]:

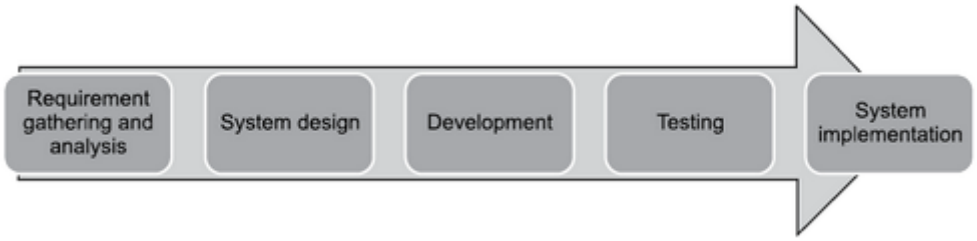
- checking correctness: the fact that the software is fault free;
- checking consistency: Is the software consistent within itself and with other products ?
- checking necessity: Is everything in the software necessary ?
- checking sufficiency: Is the software complete ?
- checking performance: Is the software fulfilling its performance requirements ?

Wallace and Fujii [WF89] goes further by saying that Software V&V allows designers to discover earlier errors and avoid bad bugfixes. Moreover, Software

V&V give a better comprehension of the quality of the system under development, and a better view of the system performance. Then, we can easily understand that the goal of Verification and Validation is the discovering of errors as soon as possible during the development. But how V&V can be useful at earlier stage of development if validation is done at the end of the development, and noted by Ryser and Glinz [RG99] as a problem ?

A first answer is given by looking at techniques used in Software Verification and Validation, and even before if we look at the models of Software Development Life Cycle (SDLC). The old "Testing" phase in the Waterfall SDLC model (figure 1.6) was replaced by many testing step in the modern V-model (figure 1.7).

Figure 1.6: Waterfall model representation from [DA12].

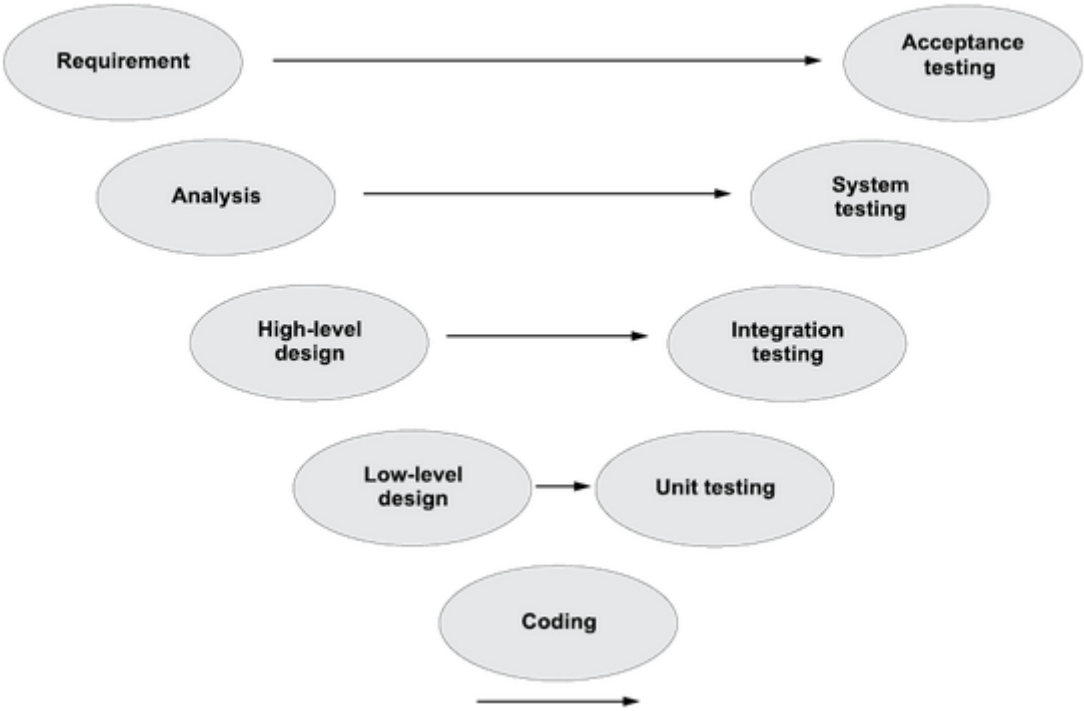


These testing phase were replaced by more elaborated Verification and Validation techniques. Because the base of the techniques are tests, Adrion, Branstad, and Cherniavsky [ABC82] talk about *Verification, Validation and Testing of Software*. Then, Tran [Tra99], Desai and Abhishek [DA12], and Wallace and Fujii [WF89] divide software verification techniques into two categories: *static testing* and *dynamic testing*.

According to the authors, static testing corresponds to strategies that perform a static analysis of the system representation in order to detect faults and errors. By this, we mean that requirements and source code are compared without executing this latter. Static testing is itself subdivided in two categories:

- **consistency techniques** that verify correctness of syntax, parameters matching, correct requirements and specification translations;
- and **measurement testing** in which some measures are done on the source code to get its globale quality (formal or informal review, peer review, inspection, technical review, etc.).

Figure 1.7: V-model representation from [DA12].



On the other hand, dynamic testing execute the source code to detect defects and errors. In this case, some scenarios are played by the software. Outputs are retrieved and compared with the expected outputs. These tests can be performed using white- or black-box testing. White-box testing are using to check the consistency and the correctness of the source code, while black-box is used to check functionalities. Always according to the authors, dynamic testing falls into three sub-categories:

- **structural testing** tests all the internal functions of the software (its structure);
- **functional testing**, including emulation, which tests functions of the system as defined within the specifications;
- **random testing** which chooses randomly test cases among all possibles scenarios. That's the case of exhaustive testing for instance, even it is in fact almost impossible to do.

From these definitions, we can make three interesting observations. Firstly, there is a clear sharing between verification about the syntax and the structure of the software on the one hand, and the verification of its dynamics/behaviour

on the other hand. Moreover, while static testing seems to be performed by formal techniques, dynamic testing relies on execution. Secondly, all these verifications techniques are performed directly on the software. We mean that it is directly the real source code which is tested. Gaudel [Gau11] talks about *program proving*, which is possible as software is a set of programs. The third thing is that verification does not rely only on a mathematical layer, but it is also relative to empirical tests.

Now, if we look at the software validation techniques, among fault injection and dependability analysis, the authors talk about *formal methods* and *logic simulation*. If simulation can be assimilated to a test, Tran [Tra99] clearly says:

> "Formal methods is not only a verification technique but also a validation technique. Formal methods means the use of mathematical and logical techniques to express, investigate, and analyze the specification, design, documentation, and behavior of both hardware and software."

More surprising, these two last techniques involves models (we talk about this term in the next section) and not the real software, whereas the definition of Validation involves the evaluation of the final product, generally by the customer itself as we have seen. Then, if software validation is done through a model, this means necessarily that the model must also been verified and validated. Otherwise, the model cannot be used for Verfication and Validation purposes. Then, what about V&V of models ?

1.4.3 Verification and Validation of Simulation Models

If the limit between Verification and Validation seems to be blured in the domain of Software Engineering, the Modelling and Simulation (M&S) domain does not make it more clear. Kleijnen [Kle95] admits that the "Terminology in the area of verification and validation is not standard", and adopts two definitions:

Definition 15: Simulation Model Verification

Verification is determining that a simulation computer program performs as intended, i.e., debugging the computer program...

Definition 16: Simulation Model Validation

Validation is concerned with determining whether the conceptual simulation model (as opposed to the computer program) is an accurate representation of the system under study".

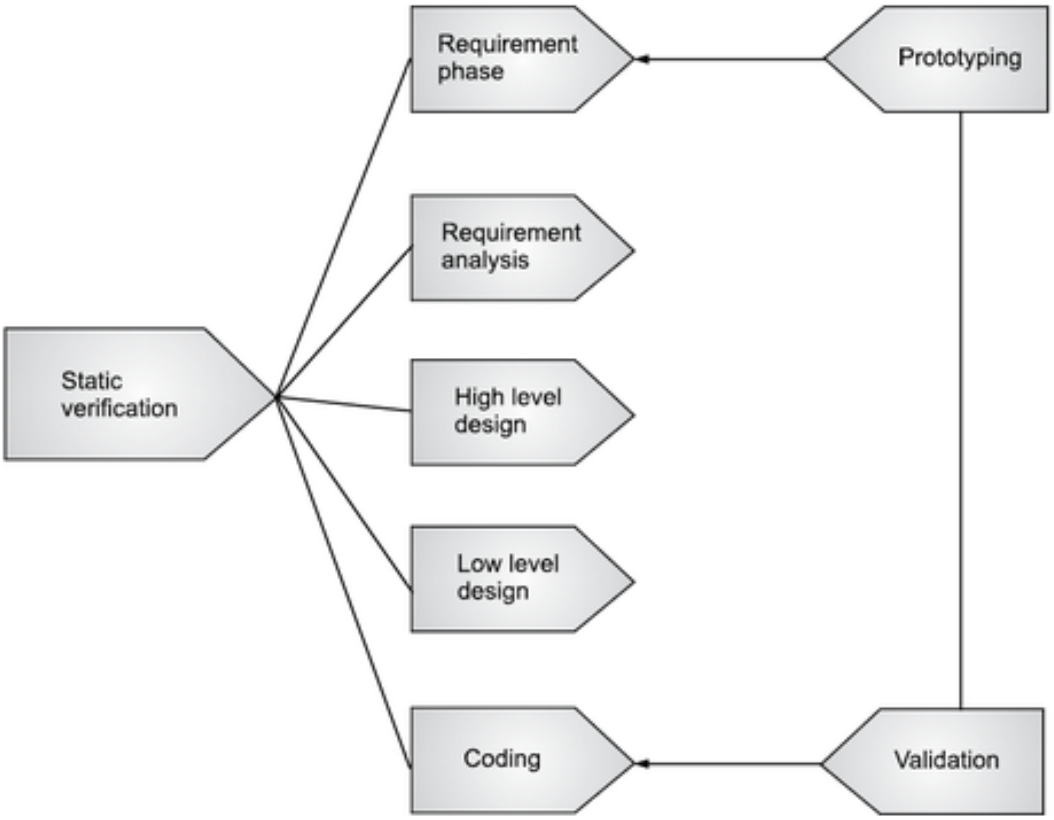


Figure 1.8: V&V in Software Development Life Cycle proposed by Desai and Abhishek [DA12].

If debugging is not an activity of verification in the sense of Software Engineering [DA12], this definition gives details about the context of V&V of Simulation Models. This definition becomes more obscure if we take the definitions given by Sargent [Sar91], who recalls the definitions given by Schlesinger et al.:

Definition 17: Model Verification

Ensuring that the computer program of the computerized model and its implementation are correct.

Definition 18: Model Validation

Substantiation that a computerized model within its domain of applicability possesses a satisfactory range of accuracy consistent with the intended application of the model.

If there is no ambiguity concerning Verification, Validation is more equivocal.

Indeed, while Verification concerns the computerized model (i.e. the real simulation software) in both definitions, Validation seems to concern both conceptual model and computerized model. Moreover, if simulation is used as a validation technique for software, the question is : what are we really verifying and validating when we perform a V&V of simulation models ? Are we really verifying the software, the conceptual model or are we evaluating the simulator (i.e. are we validating the simulation computerized model against the conceptual model ?). Then, if the verification is about the simulation model, which is a transformation of the conceptual model, don't we need to verify also the conceptual model ?

All these blurred questions find a first answer is given by Gaudel [Gau11]:

> "A problematic issue is the gap from the model to the program and the system. What is checked is that the model satisfies a property. It is not a guarantee that the program or the system do so but in special circumstances such as: the program is derived via some certified translation from the model, or conversely, the model is extracted from the program."

Therefore, there is another level of V&V, called Model Verification and Validation [Bal07]:

Definition 19: Model Verification

Model Verification is substantiating that the model is transformed from one form into another, as intended, with sufficient accuracy. Model verification deals with building the model right. This includes tranformations between models.

Definition 20: Model Validation

Model Validation is substantiating that the model, within its domain of applicability, behaves with satisfactory accuracy consistent with the study objectives. Model validation deals with building the right model.

Definition 21: Model Testing

Model Testing is ascertaining whether inaccuracies or errors exist in the model. In model testing, the model is subjected to test data or test cases to determine if it functions properly.

We then see appearing many levels of Verification and Validation, which are well represented in the Sargent Circle [Sar91] (figure 1.9). A relation between Verification and Validation is also implicitely defined. Indeed, a model is verified against requirements, then the morphisms that allows translating a source

model to a target model is also verified, because all the requirements properties are hold by the verified model. Then, validation of the target model involves also validation of the source model. Finally, Operational Validation achieves the overall validation process. That is why the simulation specification model is verified against a conceptual model, and that the simulation model is verified against the simulator specifications. Conceptual Validation and Operational Validation ensures that both the conceptual model and specifications are validated.

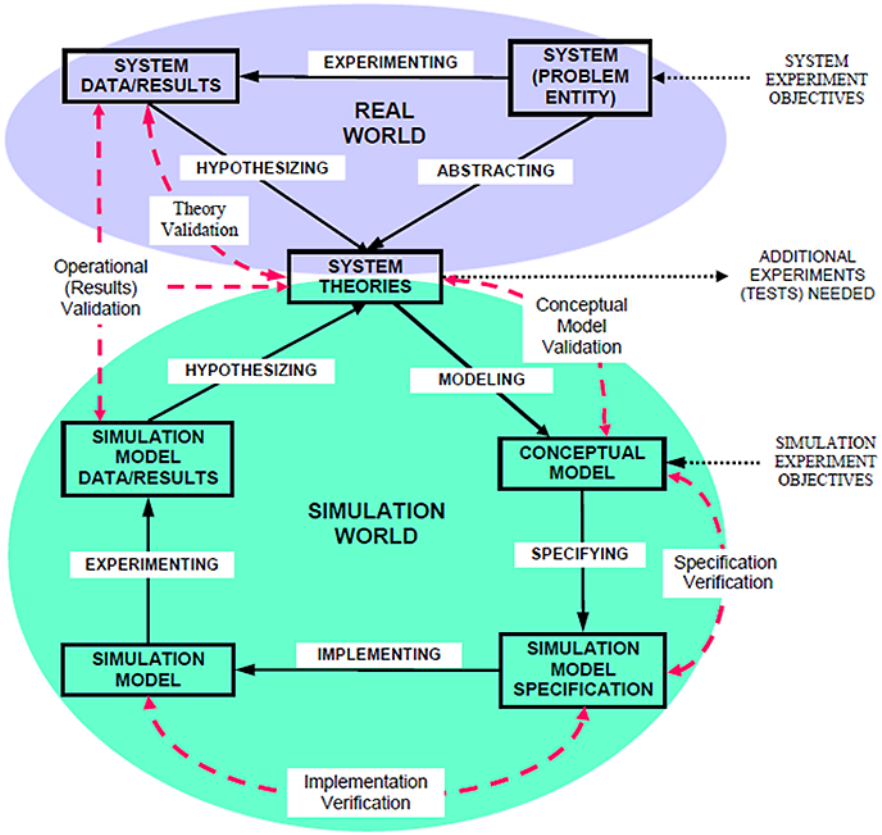


Figure 1.9: The Sargent Circle for V&V of Simulation Model [Sar91].

If the system entity is a program, a verification step is added between the *system theories* and the *conceptual model*. If we complete this model with those one proposed in [Pet10] (figure 1.10), meaning by adding a verification step between requirements and conceptual model, then we can deduce that: if the conceptual model is a correct model of the software, and if the conceptual model verifies the requirements, then the software also verifies the specifications.

Simulation Verification is then the fact to verify that the computerized model (the simulator) conforms to the simulation model specifications, while the *Simulation Validation* is the fact that the simulator simulates correctly the conceptual model. A validated simulator can be then used to validate the real system.

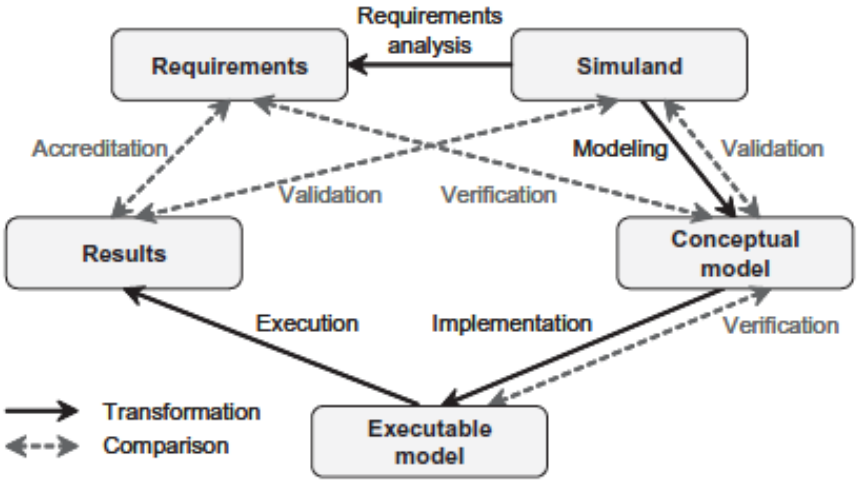


Figure 1.10: Simplified V&V proposed in [Pet10].

Concerning the techniques, Sargent [Sar01] preconizes to use tests, comparisons between models, animations, historical validation, traces, etc., which are classical Software V&V techniques. Kuhn, Craigen, and Saaltink [KCS03] proposes the introduction of formal methods for the conceptual model validity and for the implementation verification. This is interesting to see that formal verification approaches are again proposed for validation purposes.

1.4.4 Model Accreditation and System Certification

The last relation between Model and System concerns *accreditation* and *certification*. Tran [Tra99] defines the certification as

Definition 22: Certification

A written guarantee that a system or compnent complies with its specified requirements and is acceptable for operational use.

A certified system is then a verified and validated system. In the same manner, M&S domain defines the notion of accreditation [Pet10]:

Definition 23: Accreditation

Accreditation is the official certification that a model is acceptable for use for a specific purpose.

Accreditation concerns both conceptual model and simulation model. From that, we can deduce a fact: a simulator must be certified before the model it represents can be accredited.

1.5 Formal Verification and Formal Methods

1.5.1 Introduction to Formal Methods

In the last section, as in introduction, we talk about Formal Verification (FV) and Formal Methods (FM). Formal Verification is a *verification* methodology that "dates back to the origin of computer sciences" [CC10], and whose objective is to check whether a system is correct against some formal specifications. FV methods are based on rigorous and mathematical proof techniques: Formal Methods (FMs) are a set of formal notation and tools allowing strict and rigourous description of the system under study, with formal semantics and an automatic proof mechanism [BH95].

Formal Methods are divided into two families:

- Automated theorem proving methods show that a set of statements of a system can be deducted from another set of statements. Formally, we consider Γ, a set of logical properties describing the system (we called them axioms and hypothesis), and ϕ a set of specifications (that we called conjectures). Theorem proving methods try to find a proof that $\Gamma \vdash \phi$, in other words, that we can syntactically deduce specifications from properties of the system.

- Model Checking methods show that a system satisfies a set of properties. Formally, we consider M, a model (in the mathematical sense) of the system, and ϕ, a set of logical properties. Model Checking methods check whether $M \models \phi$: all models M syntactically and semantically satisfy ϕ. In fact, because the system is generally modelled by a finite automaton, model checking tools systematically explore the entire state space of the system model, inducting to the well-known state space explosion problem, which is extensively treated in the literature [Cla08; HR05; BK08].

FMs are powerful methods to check the correctness of a system. However, it is well-known that these techniques could become very heavy, time and effort consuming because they require advanced mathematical skills and knowledge [Hei98], and are not very practical in large and complex systems. Although formal methods research is focusing on efficiency and scalability, formal methods are faced with the complexity of systems and the of data domains [Zer+13].

We will here discuss only the case of Model-Checking. Indeed, even it is a pure verification method at its origin, the literature seen before seems to consider it as a validation technique. Therefore, as both Verification and Validation method, it is interesting to see what is really behind model-checking.

Model-Checking was introduced by Clarke and Emerson [CE82] and Queille and Sifakis [QS82]. It is a *model-based* verification techniques that explores systematically the all system statespace, and whose the most elaborated algorithms can handle statespace with more than 10^{476} [BK08] states for specific problem. The process of Model-Checking (figure 1.11) is similar to the M&S Framework seen previously.

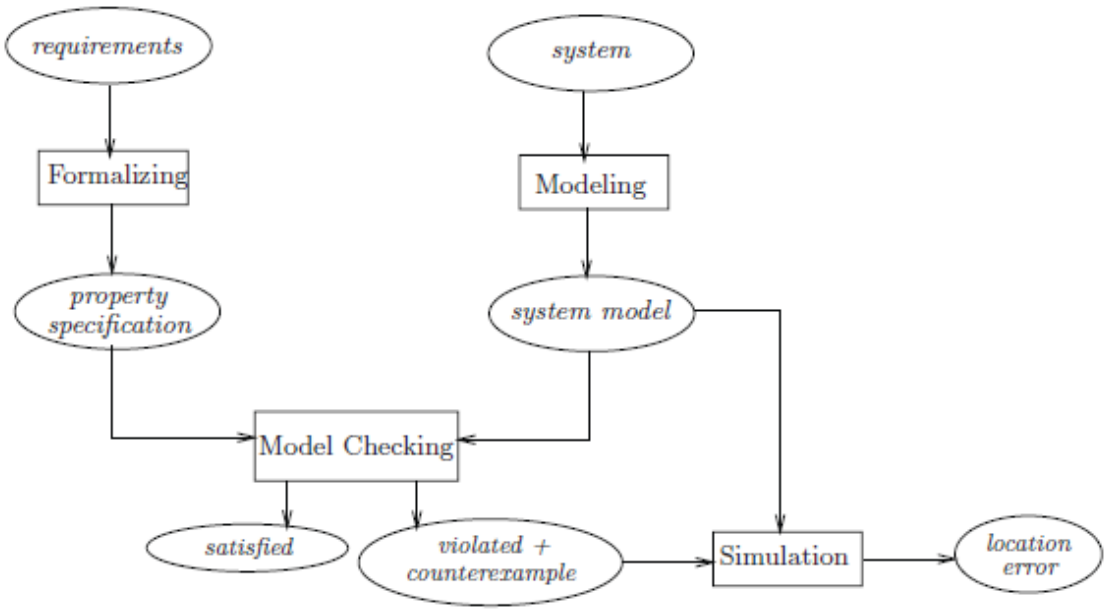


Figure 1.11: Model-checking schema from [BK08].

Given a system model and requirements, the model-checking tries to apply the requirement to each state. If one state does not satisify the property, a counter-example is returned by the checker. While model-checking is presented as an exhaustive verfication method, it does not suffer from the empirical aspects of simulation and tests which focus on the most probable defects. However, model-checking suffers from several drawbacks [BK08]:

- it is not suited for data-intensive applications;
- it is subject to decidability issue. For example, infinite-state systems like G-DEVS cannot be checked [HG05];
- Verification concerns the model and not the system. Like for simulation, complementary verification and tests are needed;
- Like for simulation, it checks only stated requirements. Therefore, there is no guarantee of completeness.
- It suffers from the state-space explosion problem;

- It requires some expertise in finding appropriate abstractions to obtain smaller system models and to state properties in the logical formalism used;
- The model-checker itself could have errors;
- It does not allow checking generalizations: in general, checking systems with an arbitrary number of components, or parameterized systems, cannot be treated. Model checking can, however, suggest results for arbitrary parameters that may be verified using proof assistants.

Thus, it is possible to see many common points with the Modelling and Simulation drawbacks. If both suffers from the same problems, Model-checking is more restrictive than Simulation.

1.5.2 Model-Checking Theory

The idea behind the model-checking theory is trivial. Given a model, the idea is to explore the total statespace. Model-checking is not relative to execution like simulation, that is why non-deterministic models are not a problem, and even are necessar. Indeed, the strong assumption is that, because all the statespace will be explored, all the scenarios will be covered. This is can be achieved only if we can go through any of the concurrent transitions at each instant.

The verification model From a formal point of view, the model M is a non-deterministic finite automaton

$$A = (Q, \Sigma, \delta, Q_0, F)$$

where

Q is a finite set of states;

Σ is an alphabet;

$\delta : Q \times \Sigma \to 2^Q$ is a transition function;

$Q_0 \subset Q$ is the set of initial states;

$F \subset Q$ is the set of final (accept) states.

with a Buchi [Büc66] acceptance criterion.

Definition 24: Run

We call *run* any ordered set of pairs of $\{(q_0, \sigma_0, q_1), (q_1, \sigma_1, q_2), ...\}$ such that $\forall i, (i \geq 0) \implies q_{i+1} = \delta(q_i, \sigma_i)$.

Definition 25: Accepted word and accepted language

Given a finite word $w = \sigma_0\sigma_1...\sigma_{n-1}$, $w \in \Sigma^*$ is an *accepted* word by A if there exists a run $\{(q_0, \sigma_0, q_1), (q_1, \sigma_1, q_2), ..., (q_{n-1}, \sigma_{n-1}, q_n)\}$ such that:

$q_0 \in Q_0$

$q_n \in F$ (the last state of the run is a final state).

The set of accepted words is called the *accepted language* of A and is denoted $\mathcal{L}(A)$.

Definition 26: ω-acceptance (Buchi acceptance)

Given A a non-deterministic finite automaton, and w an infinite word on Σ^*. w is accepted if there exists an accepting infinite ω-run σ such that

$\sigma = \{(q_0, \sigma_0, q_1), (q_1, \sigma_1, q_2), ...\} \in \Sigma^\omega$

$\exists q_f, q_f \in F \wedge q_f \in \sigma^\omega$ where σ^ω is the set of states that appear infinitely often in σ.

This means a run is ω-accepted only iff an accepting state is visited infinitely often in the run.

Theorem 1: Langage Emptiness is equivalent to Reachability

Given A a non-deterministic Buchi Automaton. Therefore, $\mathcal{L}(A) \neq \emptyset$ iff $\exists q_0 \in Q_0$ and $q_f \in F$ such that $q_f \in Reach(q_0)$.

Proof - Theorem 1. $\mathcal{L}(A) \neq \emptyset$ means that there is at least one accepted infinite word by A, i.e. it exits at least one accepting ω-run. Then, there exists at least one $q_f \in F$ which is reachable from an initial state $q_0 \in Q_0$. □

Definition 27: Synchronous Product of Automata

Given two automata $A = \{(S_A, s_{0_A}, L_A, T_A, F_A)\}$ and $B = \{(S_B, s_{0_B}, L_B, T_B, F_B)\}$, the synchronous product $A \otimes B$ is defined as:

$S = S'_A \times S_B$, where S'_A is a set of states in which an ϵ self-loop is attached to every states of S_A that have no successor [Hol03];

$s_0 = (s_{0_A}, s_{0_B})$;

$L = L'_A \times L_B$;

$T = \{(t_a, t_b), t_a \in T'_A, t_b \in T_B\}$ such that $t = (t_a, t_b)$ if

$$\frac{t_a = q_a \xrightarrow{A} q_{a'} \wedge t_b = q_b \xrightarrow{A} q_{b'}}{(q_a, q_b) \xrightarrow{A} (q_{a'}, q_{b'})};$$

$F = F_A \times F_B$; $f = (f_a, f_b) \in F \implies f_a \in F_A \vee f_b \in F_B$.

By construction,

$$\mathcal{L}(A \otimes B) = \mathcal{L}(A) \cap \mathcal{L}(B).$$

Linear Temporal Logic Temporal Logic is the branch of logic which allows reasoning both casual and temporal relation of properties [Pnu77; HR05]. There are a plenty of temporal logics but Baier and Katoen [BK08] classify them in three categories:

- Linear Temporal Logic (LTL) is based on a linear-time perspective, meaning at each moment in time, only one moment sucessor is possible. Then, LTL models time as a sequence of states, extending infinitely into the future;
- Computation Tree Logic (CTL) [CE82] is a branching-time model, meaning time is view as a tree structure, and may split into alternative paths;
- Timed-CTL is an alternative version of CTL with quantified time.

We present here only the LTL syntax and semantics, while it is the most used and implemented in modern model-checkers [SAK06; Cla08]. We also focus only on the most important operators of the LTL formulae, whose a simplified semantics is given in figure 1.12.

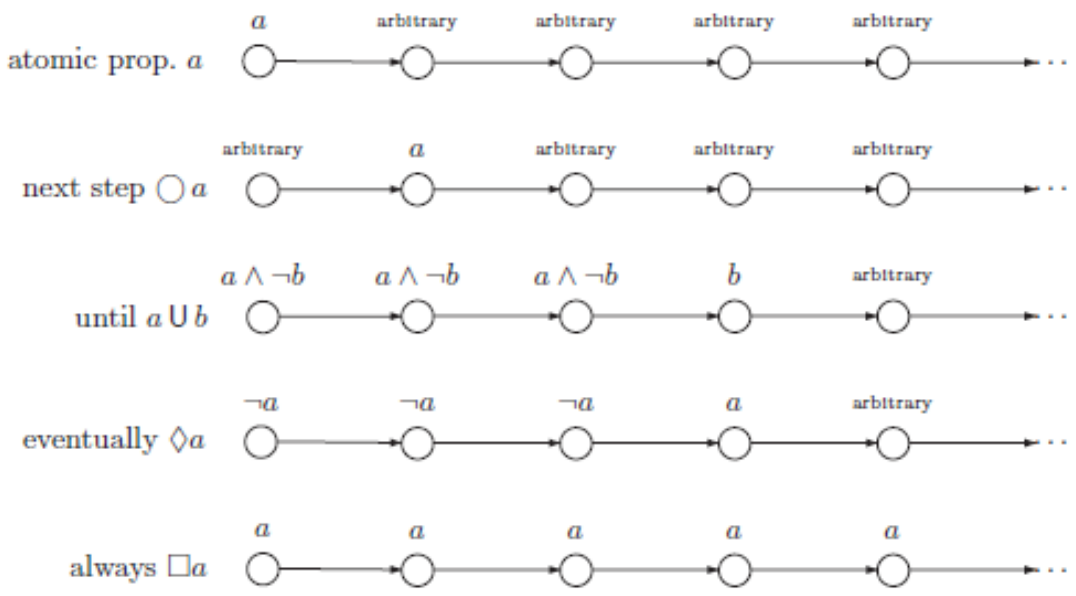


Figure 1.12: Simplified Semantics of LTL operators [BK08].

Definition 28: Until

Given two boolean formulae p and q, and an ω-run σ, we say "p until q" and we note $p\mathrm{U}q$ (or pUq) if

(weak until) $\sigma[i] \models p\mathrm{U}q \Leftrightarrow \sigma_i \models q \vee (\sigma_i \models p \wedge \sigma[i+1] \models (p\mathrm{U}q))$

(strong until) $\sigma[i] \models pUq \Leftrightarrow \sigma[i] \models p\mathrm{U}q \wedge \exists j, j \geq i, \sigma_j \models q$

The weak until definition says that a sub-formula p becomes false as soon as a sub-formula q becomes true, but never requires that q becomes true. The strong until adds this requirement.

Definition 29: Always

Given a boolean formula p, and an ω-run σ, we say "always p" and we note $\Box p$ if

$$\sigma \models \Box p \Leftrightarrow (p \ \mathrm{U} \ false)$$

This definition captures the fact that p is invariantly true.

Definition 30: Eventually

Given a boolean formula p, and an ω-run σ, we say "eventually p" and we note $\diamond p$ if

$$\sigma \models \diamond p \Leftrightarrow (true \ U \ p)$$

This definition captures the fact that p becomes true at least one time throughout a run.

Definition 31: Next

Given a boolean formula p, and an ω-run σ, we say "next p" and we note $\mathrm{X}\ p$ if

$$\sigma[i] \models \mathrm{X}\ p \Leftrightarrow \sigma_{i+1} \models p$$

This definition simply states the immediate next state in a run holds p.

Types of property which can be expressed in LTL Using the LTL, it is then possible to express two categories of properties [Lam77]:

- A safety property asserts that nothing bad happens. The most simple safety property consists on checking that all states variables satisfies conditions. Especially deadlocks are detected through safety properties;
- A liveness property asserts that something good eventually happens. Especially progress property guarantees that certain actions will eventually happen.

Interpretation of LTL formulae and Verification It was shown [BK08] that verifying a LTL formula can be interpreted as finding accepting paths in a Kripke Structure [Kri63], which is a transition systems labelled by atomic propositions. Indeed, a LTL formula defines a set of infinite words that satisfies it. Then, as a consequence, it is equivalent to find a path in the reachability graph of a transition system. Pnueli [Pnu77] shows also that a LTL formula can be translated into a Buchi automaton. Then, in order to verify that a system holds a LTL formula, model-checkers proceed as follow:

1. The LTL claim p is translated into a LTL never claim (the opposite of the property which must be verified).
2. The LTL never claim is transformed into a Buchi automaton.
3. If an automaton M holds the property p, then it never holds the corresponding never claim. This means that the intersection between the language defined by M and the language generated by p is empty. However, as given in definition 27,
$$\mathcal{L}(A \otimes B) = \mathcal{L}(A) \cap \mathcal{L}(B)$$
Then, the synchronized product between the two automata is performed.
4. Theorem 1 states that the emptiness problem is equivalent to the reachability problem. Moreover, as said before, verifying a LTL formula can be

interpreted as finding accepting paths in a Kripke Structure. Then, the reachability graph of the synchronized product is generated.

5. If the reachability graph is not empty, then the never claim is hold by the modelled system, meaning that the initial property is not verified.

Efficiency of model-checking hence depends on the efficiency of the algorithm [SAK06; Fra+10] used in order to compute the reachability graph. Many approaches were given using Binary Decision Diagram, symbolic verification, partial order reductions, etc [Cla08]. We won't discuss about the efficiency of algorithm, but we can clearly see why model-checking is subject to decidability problem. Indeed, a minimal condition for using model-checking is that the initial model is finite, or at least countable. Otherwise, it is impossible to compute the reachability graph. Therefore, infinite models can be infinite checked only if finitary abstractions can be found, unlike simulation for which infinite systems are not a problem.

1.5.3 Timed Automata

As stated in [BK08], the basic model-checking is only interesting in how reactive systems evolve from one state to another, without taking into account timing aspects. For example, there is no way to describe the fact that a system will stay in a state for a time. Then, these time-critical systems cannot be modelled. The authors also state that

> "Correctness in time-critical systems not only depends on the logical result of the computation but also on the time at which the results are produced."

This statement will be debated in a next section. Nevertheless, we present theories elaborated to face this problem. The first and the more simple technique was to consider that operations are done in a tick. By this way, time is modelled as discrete-time, and clocks are modelled as loops. One loop corresponds to a time pulse. In this manner, LTL and CTL can be used for checking properties. However, we can easily imaginate that this method leads to very huge statespace.

Another approach called Timed Automata (TA) was developped by Alur and Dill [AD94]. Timed Automata are transition systems in which transitions are constrained by a real-valued clock variables. These clocks can be only inspected and reset to zero. Formally, a timed automaton TA is

$$TA = (S, Act, Clock, T, Inv, S_0, AP, L)$$

where

L is a finite set of states;

Act is a finite set of actions;

$Clock$ is a finite set of clocks;

l_0 is a an initial states;

$T \subseteq S \times \mathcal{C}(Clock) \times Act \times 2^{Clock} \times S$ is a finite set of transitions. $\mathcal{C}(Clock)$ represents a set of constraints on clocks;

AP is a set of atomic propositions;

L is a labelling function;

Inv is an invariant that associates an invariant to each states.

Then, the semantics of TA is defined by a Timed Transition System (TTS)

$$T = (S, S_0, \rightarrow, \Sigma)$$

where

$S = L \times \mathbb{R}^{Clock}$ is a set of states;

$s_0 = (l_0, v_0)$ is a initial state; v_0 is the clock valuation function and $v_0(x) = 0$ for each $x \in Clock$;

$\rightarrow \subseteq S \times (\Sigma \cup \mathbb{R}_+) \times S$ is a relation between states:

1. $\rightarrow$ is a action transition $(l, v) \xrightarrow{a} (l', v')$ iff $\exists t \in T, t = l \xrightarrow{g,a,r} l'$ such that $v \models g$ (v satisifies the clock constraint), $v' = [r \leftarrow 0]v$ (the clock r are resetted), $v' \models Inv(l')$ (the invariant of the next location is always respected);
2. $\rightarrow$ is a delay transition $(l, v) \xrightarrow{d} (l, v + d)$ iff $v + d \models Inv(l)$. Delay transitions increase all the clock by the same amount of time.

Model-Checking of Timed Automata Laroussinie et al. recall in [NM13] the decision procedure for checking reachability in TA, proposed by Alur and Dill [AD94]. Alur states that the problem of TA is that the number of states can be uncoutable, due to the extended states (s, v) where v is a clock interpretation. However, if it is possible to build a finite automaton that mimics the behaviour of the TA, then the verification becomes possible. As the problem comes from the clock valuation, Alur proposes to build an equivalence relation between v and v', thanks to an intuitive observation: given two valuation v and v' and two extended states (s, v) and (s, v'), if v and v' have the same integer parts on all clock values, and if all the clocks values can be ordered in the same manner according to the fractional part, then all runs starting from the two extended states are similar. This allows defining an equivalence relation noted $v \sim v'$ which is hold iff:

1. $\forall c \in Clock, (\lfloor v(x) \rfloor = \lfloor v'(x) \rfloor)$ or $v(x) \geq c_x \wedge v'(x) \geq c_x$, where c_x is the largest interger c such that $x \leq c$ or $x \geq c$ in a subformula of some clock constraints appearing in $\rightarrow$; this constraint imposes that both valuation have the same integer parts (or both have the same order with the bounding constraint);

2. $\forall x, y \in Clock$ with $v(x) \leq c_x$ and $v(y) \leq c_y$, $frac(v(x)) \leq frac(v(y)) \Leftrightarrow frac(v'(x)) \leq frac(v'(y))$; all the clocks values can be ordered using their fractional part;

3. $\forall x \in Clock$ with $v(x) \leq c_x$, $frac(v(x)) = 0 \Leftrightarrow frac(v'(x)) = 0$; this defines the condition for having the same integer parts.

Using this equivalence, Alur shows that the number of clock regions (a class of clock interpretations induced by $\sim$) is finite and bounded. As a consequence, it is possible to build a finite *region* automaton which behaves exactly as the timed automaton. Then, the model-checking can be applied for verifying temporal properties.

Clock constraints and limits Clock constraints in Timed Automata are expressed using the following grammar:

$$\delta \ := \ x \leq c | c \leq x | \neg\delta | \delta_1 \wedge \delta_2$$

where c is an integer constants or any linear combinaisons of constants. Moreover, actions are only limited to accessing and resetting clocks, meaning it is impossible to define state variables that explicitely depend on time. Alur shows that without these constraints (for example by allowing linear combinaisons of clocks), the problem of emptiness of timed automata becomes undecidable. In fact, the model is not more timed regular, and it becomes impossible to build an untimed automaton with the same behaviour.

As a consequence, if we want model datas that have a time-dependant evolution, it must be done in a manner similar to those used in discrete-time modelling. Then, the size of the statespace dramatically increases. Finally, Timed Automata are only able to model systems with delayed transitions.

1.5.4 Hybrid Automata and others methodologies

In order to solve this problem, Alur, Courcoubetis, Halbwachs, et al. [Alu+95] and Henzinger [Hen96] propose to develop a class a model called *hybrid* automata, in which states variables evolve using differential equations.

Formally, an hybrid automaton HY is

$$HY = (S, T, \Sigma, X, Init, Inv, Flow, Jump)$$

where

S is a finite set of states;

$T \subseteq Loc \times Loc$ a finite set of discrete transitions;

Σ a set of events, with a labelling function $lab : T \to \Sigma$;

X a finite set of real-valued variables.

$Init$, a function that gives the initial values of variables;

Inv, an invariant function that gives the condition of variables at each state;

$Flow$, a function that describes of variables of each state;

$Jump$ a function that gives the guards that enable the transition and how the variables are updated before transiting.

Given such an automaton, Alur showed the problem of emptiness is undecidable except for a subclass called *Linear Hybrid Automata* (LHA), in which variables evolve following constant differential equation. The demonstration of this resultat is not recall here, but it illustrates the impossibility of formally checking certain classes of timed systems. Otherwise, for the LHA, the methodology consists on using convex polyhedra to abstract the LHA into a finite-state automaton. This automaton is then checked using the classical model checking theory introduced before.

Litterature on model-checking introduces many other types of abstraction including Abstract State Machine [SBS01; Bör05; Rei12], Probabilistic and Stochastic Model Checking [KNP07; Kat10; KP12], etc. but all of them suffer the same problem of abstraction, explosion statespace problem or decidability.

Probabilistic Model Checking is efficient for the analysis of systems which exhibit stochastic behaviour, and which can be modelled by Markov Chains. For instance, PRISM Model Checker [Hin+06; KNP07] supports the verification of both Discrete-Time Markov Chains (DTMC) and Continous-Time Markov Chains (CTMC). DTMC express probabilistic choices, in the sense that the designer expresses the probability of performing an action. In these models, time is modelled as discrete steps. CTMC model probabilistic choice and the continuous time, in the sense that designer can models the rate of performing a transition from one state to another. Then, stochastic model checking achieves the reachability analysis like traditional model checking, and computes likelihood of the occurrence of certain events during the execution of a system. But as stated in [SV13a], "*this approach suffers the well known state-space explosion problem, i.e. it does not scale well when the system complexity grows*". That is why Statistical Model Checking (SMC) [LDB10] has been studied. Indeed, SMC uses

discrete-event simulations in order to approximate the behaviour of a probabilistic system, and use hypothesis testing to infer whether the samples provide a statistical evidence for the satisfaction or violation of the specification. However, the main drawback is that SMC does not provide exact results.

1.5.5 Abstraction and The Great Debate

From the previous section, we can deduce that efficiency and results of model-checking strongly depends on the capability of making abstraction of models. *Abstraction-Refinement* [HL98; GS05; Gru06] is then the main process in the model-checking activity. Even, model-checking and theorem proving are both performed in a top-down manner, by refining an abstract model to the implementation, or in a bottom-up manner, by extracting a model for an implementation, using a certified translation. The main question is: how ensuring that two models express the same thing ? Is the certified extraction enough ?

Informally, refinement can be defined as adding details to a model, while abstraction is the reverse process, in which some elements of the source model are identify to a unique element of the abstracted model. Implementation relations are then used for comparing models, meaning that transition systems from an abstract system specification and a detailed system specification are compared [BK08]. Then, if the second one is a correct refinement (implementation) of the first one, that means the abstracted model has enough relevant details for the analysis of a given property. Grumberg [Gru06] identifies three abstraction types whose one is discussed here: *data abstraction* [Lon93]. It consists on, for each variable, choosing an abstract domain that is smaller than the original domain. Then, an abstraction function maps concrete data to abstract data, and therefore induces a mapping function from concrete states to abstract states. In the most of cases, this guarantees the finiteness of the checked model. Data abstraction is supposed to ensure that the abstracted model behaves like the concrete model. This leads to time abstraction like argues in [Hol03]. The author says that

> "Verification of system properties is based on the fundamental assumption that correctness should be independent of performance".

However, explicit time is not just relative to performance. Moreover, the existence of formalism like Timed Automata or Hybrid Automata shows that in practice, untimed model are not equivalent to timed models. Indeed, variables generally evolves according to the time like shown in the M&S Theory. We agree that in certain cases, time-invariant properties can be abstracted, but generalizing this fact to all variables leads to unaccurate models. Then, this question is relative to know : which is the relation between timed and untimed model that we have represented in figure 1.3.

The other debates concern states vs event approaches and open vs closed systems. For the first one, as model-checking enumerates states and does not interest in events, adopting a state approach makes easier the formalizing of temporal properties. Indeed, they are built up from simple boolean conditions holding by states. The nature of model-checking then makes more intuitive the use of state-based models. Concerning the second debate, closed system enforces designer to embbed all the sources of input in the model. If this is helpful and enforces the designer to focus explicitely on hidden assumptions, the drawback is it is likely impossible to built a modular and hierarchical systems like in M&S theory. Then, it can blur and make more difficult the modelling of huge systems with several inputs sources. Moreover, designers sometime want to hide the details of input generation, in order to focus only on relevant details of the model. Therefore, closed models are not appropriate for modelling systems with many interactions between components.

The last thing is : why model-checking is considered as a validation method ? Model-checking is a verification method but which is interested in behavioural properties. Indeed, it is possible to check safety and liveness properties, but both of them can express structural and behavioural properties. For the second one, model-checking is just look for ordered actions, in opposite in simulation which stimulates the model according to a time-advance function. As model-checking is therefore able to check behavioural properties, it can check final user requirements. If the model is verified (and validation), then model-checking can be used for validating the initial system by checking properties on the model.

Knowing all these drawbacks, some work try to limit them by combining model-checking and simulation approaches.

1.6 Towards Integrating Formal Verification and Simulation for V&V

Gaudel [Gau11] states that:

> "It is now quite well acceptedthat activities such as model-checking, proof-supported refinement, program proving, system testing, etc, are complementary[...]: for instance it is different to perform concurrently two of the activities mentioned above, drawing global conclusions at the end, and to transpose one method developed for one of these activities to another one in order to improve it."

For instance, the author talks about techniques which have been combined and which blur the line between static and dynamic verification methods, like symbolic execution, concolic testing or runtime verification. At a higher level, formal verification techniques have also been combined, and attempts on combining formal verification and simulation exist.

This section will try to make a short survey of how techniques are combined for achieving efficient Verification and Validation.

1.6.1 Integration of Multiple Formal Verification Tools

The first category of combination between tools is the integration of multiple formal methods. For instance, Kindler, Rubin, and Wagner [KRW06] propose to integrate multiple models into components which are connected each other. Each component can be verified or simulated using a defined technique (for example: Petri nets, Timed Automata..). The authors also talk about transformations that allow translating a model which can be understood by a method into another model understandable by another method, using what they call the *triple graph grammar*, and which is based on Model-To-Model techniques [SK03]. However, the author do not say how they guarantee the correctness of the transformations, nor the benefits on changing methods without changing level of abstraction.

In another work, Owen [Owe07] shows the inconsistency of the results obtained by different model-checkers and different model-checking algorithms when checking a same property on a same model. The most impressive result is the the disagreement between NuSMV [Cim+00] and CadenceSMV [McM00] which are both symbolic model-checkers. Then, Owen proposes to combine model-checkers to improve accuracy, performance and robustness. To achieve this, the specification is encoded into the SCR Toolset and translated into different specification languages which can be used with different checkers [Hei+05].

Owen gives the figure 1.13 to summarize its proposed approach, and argues that

> The SCR Toolset's translators used in our experiments are complementary, for example, in the sense that the output for SMV is a smaller model than the output for SPIN, and so can be verified more efficiently; yet the output for SPIN is a more complete representation of the specification.

If this statement is true as explicit model-checking performs less abstraction than symbolic model-checking, it can be discussed if we take into account the fact that all model-checkers finally reduce the model to a finite state-machine. As stated before, it is generally not enough to increase accuracy.

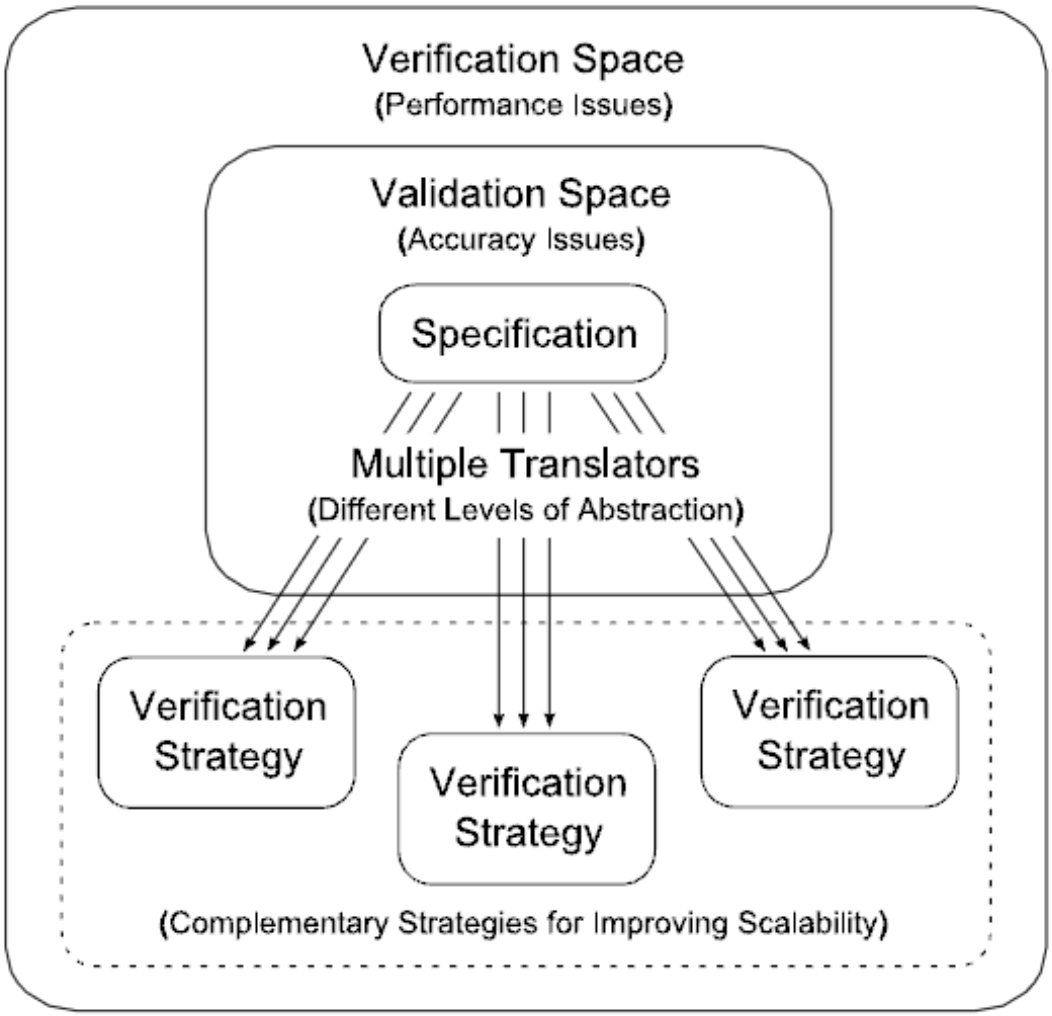


Figure 1.13: A conceptual model of the challenges involved in using automatic verification tools. [Owe07].

In a similar way, He [He01] states also that formal methods based on a single fundation have limits, and are just suitable to represent specific aspects of software systems. To overcome this problem, He proposes to study the relationship between formal methods and builds a new formalism called PZ-Nets which integrates Petri nets and the Z formal notation. In this way, he makes a new formal verification approach in which a unified formal model is used for specifying different aspects of a system (structure, control flow, data types, and functionality) and for specifying different types of systems (sequential, concurrent, and distributed systems).

1.6.2 Combining Formal Verification and Simulation

While Godefroid [God13] says that "model checking can be combined with testing to define a dynamic form of software model checking based on systematic testing", confirming that static and dynamic approaches can be both used for Verification and Validation, Goldberg [Gol08] admits that simulation and formal verification are complementary. The author states that if formal verification proves that a model holds property for all point, the main problem is its unscability. On the other, simulation probes the search space at a subset of points, and "works surprisingly well even though the set of test points (further referred to as the test set) comprises a negligible part of the search space". Then, she proposes a way to build a sufficient test set for SAT-circuit and shows how simulation can be used for verifying sequential circuit. In the similar way, Stuart, Brockmeyer, Mok, et al. [Stu+01] defines a new analysis methodology that he calls *Simulation-verification*. In this approach, simulation is used for generating a computation prefix that restricts the reachability graph on which formal verification is performed.

Li, Szygenda, and Thornton [LST05] state that modern formal verification methods overcomes the weakness of non-exhaustive simulation, and can be used for validating designs. The authors propose an architecture for validation of integrated circuit design. In this architecture, the model is partitioned and analyzed in order to select the most efficient techniques (formal validation or simulation-based validation). After performing the validation, a coverage analysis is done and reintegrated in the validation loop. In a similar approach, Abdulhameed, Hammad, Mountassir, et al. [Abd+14] proposes to validate SysML models by using Timed Automata and SystemC simulator. Using the Model-Driven Engineering approach, the authors translate the SysML specifications into a SystemC model which is simulated for validating non-functional properties. The SystemC model is then translated into a Timed Automata and checked against the requirements translated from the SysML model. The cons of this approach is that the translation from SystemC to Timed Automata can be done under restrictive conditions [Poc+11]. The authors don't provide way to ensure the translation is possible. In [YHF14b; YHF14a], we propose to combine PROMELA and TSM by extracting the behaviour of a verification model and completing the specifications with timed informations (figure 1.14). The resulting model was then encoding in TSM and simulated. However, the cons of this approach is the same than the previous work: we have no guarantee that there is a relation between abstractions.

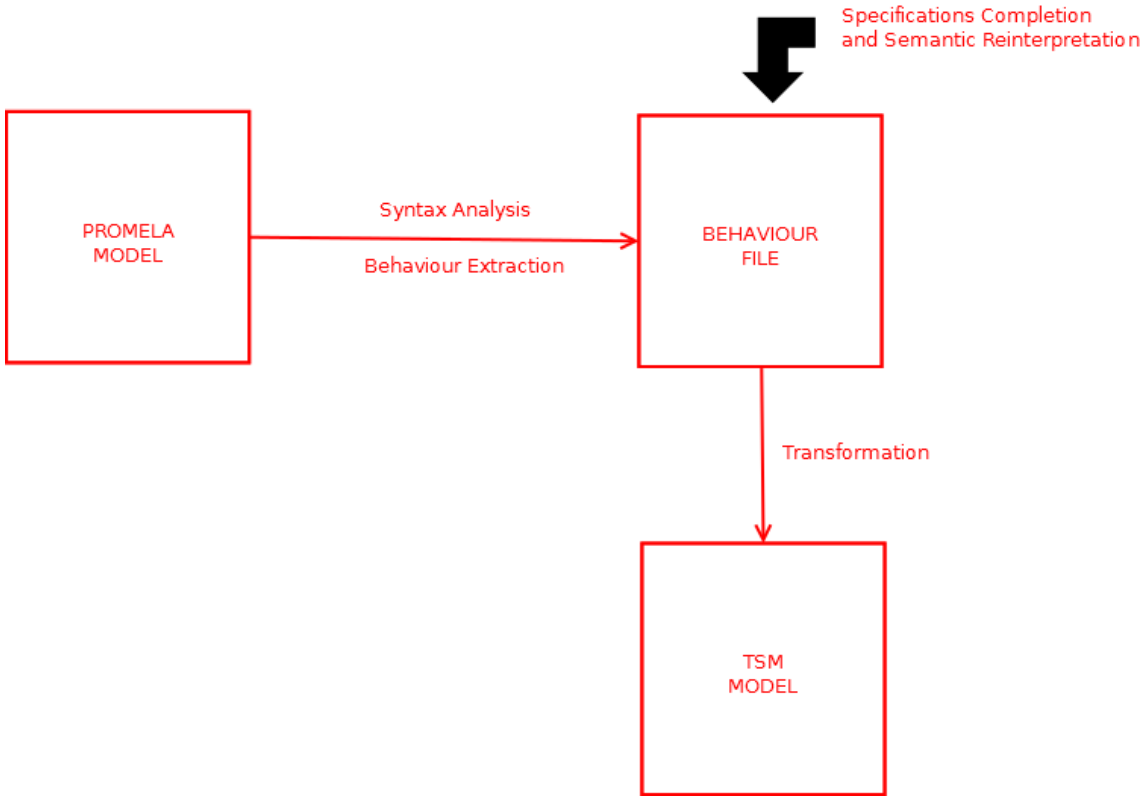


Figure 1.14: Approach of combining PROMELA specifications with TSM.

In a very recent approach which can be seen as another application of the work introduced in this thesis and confirms our results, Aliyu, Maïga, and Traoré[AMT16; AT16a; AT16b] choose to achieve the combination of formal methods and simulation by using Model-Driven Engineering methodology. They create the High Level Language for System Specification (HiLLS) which integrates system-theoretic concepts from DEVS and software engineering concepts from Object-Z. As a result, HiLLS have the semantics of DEVS and is able to model both concepts described in DEVS and Object-Z. HiLLS model is then derived into a DEVS model and Object-Z model using a MDE approach, while we choose in our work to use a direct mapping between automata.

In the opposite direction, Kunzli, Poletti, Benini, et al. [Kun+06] proposes to combine formal analysis methods and simulation to speed-up simulation, in the case of System-Level Performance Analysis. By defining an interface between simulator (in this case, SystemC) and formal method (in this case, Real-Time Calculus [CKT03]) which converts data output from one method to the other, the run-time of simulation is divided by two.

In another work, Girard and Pappas [GP06] show how simulation can be used for verification of metric transition systems using a finite number of simulations. However, this approach does not work for non-metric systems (i.e. without an observation map).

Savicks [Sav16] integrates formal methods and simulation by defining an operational semantics for Event-B models. The Event-B components is then coupled to others simulation models, and the master model is co-simulated in the master environment. This is achieved using an open standard for tool-independent model exchange and cosimulation called Functional Mock-Up Interface. In this case, interoperability is done through the tools and not through the conceptual models.

The last discussed methodology is the Assertion-Based Verification (ABV) [FKL04]. This approach, appreciated in circuit design, consists on putting assertions in the model when this latter is written. Assertions are logical properties that must be verified by the model during the execution. When executed or simulated, assertions are monitored and trigger errors if there are not verified. Assertions can also be used in formal verification, as they are evaluated without participating to the behaviour of the model.

1.6.3 Formal Verification of Simulation Models

The last way to consider combination of formal verification and simulation models in through the V&V of simulation models. As an active research domain, a lot of works were done for increasing the confidence put into simulation models. Beyond the V&V Framework [WL07], many of attempts consist on translating DEVS models into verifiable models like Timed Automata [DG05; DG07; Ino+16] (only in the case of the DEVS model is a TCDEVS model), by reducing the model to a finite and deterministic model [HZ06; HZ09] and applying the model-checking algorithms, by translating DEVS models into TLA+ [Cri07]. Trojet, Frydman, and Hamri [TFH09; Tro10] propose to encode some structural properties of DEVS model into Z specifications for verifying properties like determinism and completeness. Maiga, Bright Ighoroje, and Kaba Traoré [MBK12] transform the DEVS simulation model into Z, CSP and CTL models, depending on the level of abstraction. However, whatever these techniques, it was shown that they can be applied only on subsets of DEVS, while Generalized DEVS cannot be verified [HG05].

In [BJ14; ZN15; ZNS16], the authors discuss the use of the morphism concepts proposed by M&S theory for transforming DEVS models into models more suitable for analysis by model checking, symbolic extraction of test cases, etc. They give an example of how mapping DEVS into a PROMELA model. Simulation is

used for exploring the system's parameter space and identifying boundaries beyond which any particular proof fails to hold, or for checking scenarios that are outside the scope of model-checking, while model-checking is used for checking the property in a smaller statespace. The authors state, like what we state in our work, that the extended conditions space is greater than the idealized conditions space. This confirms the approach that we will introduce in the next chapter.

1.7 Conclusion

We have given in this chapter some key definitions concerning models, systems and software. Modelling is the central activity of Verification and Validation of Systems, whatever the domain. We have also seen that all activities of V&V, other than testing directly the real system, is equivalent to checking only the model. Then, if we can prove there is a relation between the system and the model, then we can infer the validity of the system. This relation is proved through a model verification process. Then, a V&V process involves many kind of levels of verification and validation: the simulator, the simulation model, the conceptual model and the real system. They involve also independantly static and dynamic techniques, proving that both approaches are complementary.

Especially in software engineering, model-checking is both a verification and validation technique. However, model-checking reasons on the model, and results can be transposed to the software only if the model extraction was proved. Model-checking as simulation supposes also that the checker was verified and validated. Moreover, we have shown that model-checking suffers from decidability issues and state-space explosion as soon as timed-variables are introduced in the model. There, to ensure the model can be checked – i.e. it is a finite model – data and events must be abstracted. As a consequence, model-checking is not really appropriate for event-driven architecture, and nor for data-driven architecture.

Then, Formal Methods and Simulation have the same objectives. We showed also that what seems in opposite are complementary. Indeed, what formal methods cannot detect can be detected by simulation while what takes time to be checked using simulation can be quickly verified using formal methods.

Our idea is then to integrate simulation and model-checking in a same framework, and use them in combination in order to improve V&V processes.

Chapter 2 A Combined Formalism: DEv-PROMELA

Alone we can do so little; together we can do so much.
Hellen Keller

The strength of the team is each individual member. The strength of each member is the team.
Phil Jackson

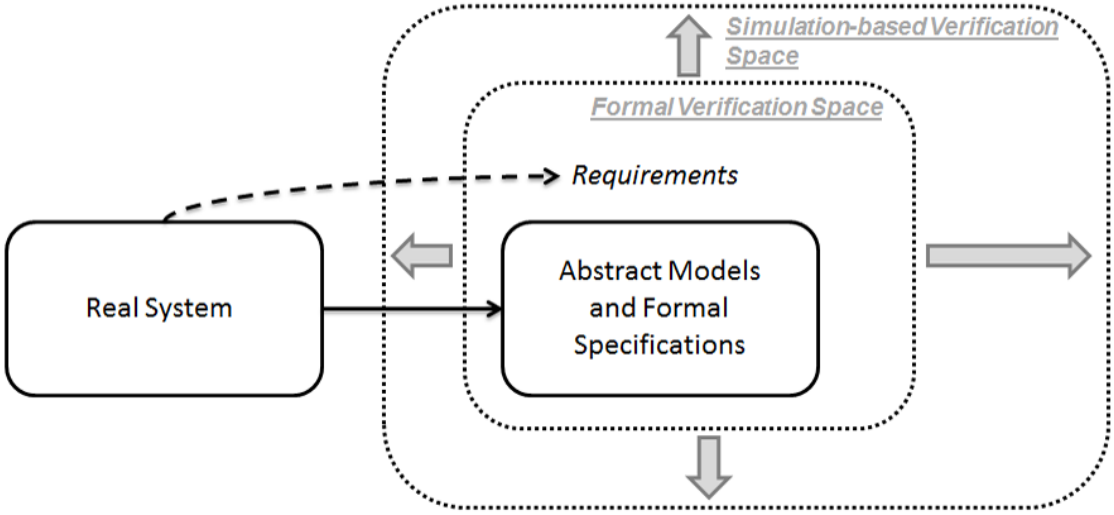
2.1 Introduction to Combined Methods

As seen in the previous chapter, goals of simulation and verification are sensibly the same. There are also existing Verification and Validation methodologies that combine formal verification and simulation. The approach that we propose in this work looks for filling weaknesses of model checking by using discrete-event simulation.

We have seen in this previous chapter that abstraction guarantees that the abstracted model is greater than the concrete model [Gru06], in the meaning that it represents more things because it is more generic. However, the practical abstract model takes less memories than the concrete model thanks to symbolic representations, meaning that concrete model is physically greater than the abstract model. If we take this sentence literally, we are facing a "contradiction". And this is that contradiction which we will exploit. Our idea is that it is possible to use the fact that the simulation-space including data is greater than the verification-space, whereas the abstract model is more general. In fact, the accuracy is loss when data abstraction is applied to the model, and that is

data that makes harder formal verification. Therefore, reintroducing data in the simulation-space (figure 2.1) will allow modelling of time-dependent systems. By this, we do not mean just applying time constraints on transitions, but really model time-dependent states and variables. The verification space then can be seen as the union of a restricted formal verification space for time-invariant properties, and a simulation-based verification space for time-dependent properties.

Figure 2.1: Schema of Combined Model Verification Space. The grey arrows means the increasing space by adding data.



This is achieved by creating a new specification language that will support model checking and discrete-event simulation thanks to morphisms. Then, modellers can use models expressed in this new formalism in order to perform verfication and validation of properties by using model-checking and dicrete-event simulation.

As a realization of our approach, we develop an extension of the Process Meta Language (PROMELA) called Discrete-Event PROMELA (DEv-PROMELA). As we will see, the DEv-PROMELA language can be reduced using a simulation formalisms hierarchy, that gives more control to the modellers.

Note that in this chapter the term *simulation* refers to ***discrete-event*** *simulation*.

2.2 General Approach

2.2.1 Abstract Approach

The approach that we propose [YHF16a] consists on building a specification language upon an existing specification language by introducing discrete-event concepts. We can summurize the methodology in the following steps:

1. We choose the verification formalism that we want to improve among the existing specification formalisms. We call this formalism the *source formalism*. As verification formalism, in the case of model-checking, we mean the final language (like PROMELA or UPAAL Timed Automata, etc..) used by the model-checking tools to model systems. Indeed, as seen in the previous section, model-checking approaches always use an underlying transition systems as models. The resulting transition systems depends on the capabilities of the specification language and the algorithm used by the model-checking tools. For conveniance, we call the initial model *the source model*.

2. We choose the simulation formalism that we want to use to improve the *source formalism*. We call this the *target formalism*. The *target formalism* must be chosen so that it represents the system under study at best. However, it must at least have an operational semantics [Hen90; Plo81; Plo04; Sco77; Ten76].

3. We determine from the *target formalism* what discrete-event concepts are missing in the *source formalism*. At least, these notions are: event, state lifespan and types of transition (internal or external).

4. We introduce new syntactical elements into *the source formalism* in order to model the previous concepts. If needed, we also add a abstract *real* datatype to represent infinite and unbounded values.

5. We define a new operational semantics for the *source formalism* and based on the *target formalism*. The initial structure of the underlying automaton is not changed.

6. We use the new specification language for modelling systems. For conveniance, we call the model obtained *the target model*.

Then, we can be sure that we are able to easily define a morphism which translates models expressed in the new formalism into models expressed in the source formalism and which conserve all their structural properties. Indeed, step 5 ensures that the automaton underlying the new formalism is built upon the automaton underlying the source formalism. Moreover, we are also sure that there exists a simulation model which has the same behaviour than the behaviour of the model expressed in the new formalism. Then, our new model can be verified and validated using both model-checking and discrete-event simulation.

2.2.2 Proofs and discussion

This section discusses about the validity of the general approach. Two important questions must be argued. The first concerns structural preservation and semantics alignment.

These two properties can be approached by talking about the *simulation preorder* [Par81].

Definition 32: Simulation Preorder (Similarity) [Par81]

Given a labelled transition system $(S, \Lambda, \rightarrow)$, and a relation $R \subseteq S \times S$. R is a simulation preorder iff

$$\forall (p,q) \in R, \forall \alpha \in \Lambda, \forall p' \in S, p \xrightarrow{\alpha} p' \implies \exists q' \in S, q \xrightarrow{\alpha} q', (p', q') \in R\,.$$

We say that q simulates p or p and q are similar, and we denote $p \leq q$. If p simulates q and q simulates p, then p and q are said bisimilar.

Moreover, as we saw in the Chapter 1, the model underlying model-checking theory is Kripke Structure, which can be seen as Moore Machine. Our proposed methodology suggests that the target formalism must at least have a Structural Operational Semantics, meaning it is possible to generate an automaton. Then, by extension, if the model generated by target formalism simulates the model generated by the source formalism, this means:

- all the structural and invariant properties of the initial model are preserved on the obtained model, but the reverse is not true (1);
- the final model contained at least the behaviour of the initial model, but the reverse is not true (2).

Proof - Property 1. Considering two state-transition systems $T' = (S', \Lambda', \rightarrow')$ and $T'' = (S'', \Lambda'', \rightarrow'')$, and a transition system $T = T' \sqcup T''$. Then, the definition 32 involves that T'' simulates T' if there exists a simulation order $R \in \texttt{STATE}\,(T) \times \texttt{STATE}\,(T)$ in which all states $p \in S'$ has a corresponding state $q \in S''$. Then, there is two cases:

1. The target model (i.e. the automaton representing its semantics) is obtained by introducing no modification into the structure of the source model (i.e. the automaton representing its semantics), for example just by only adding state lifespan. In this case, all states $q \in S''$ simulate at least one state $p \in S'$. In this case, T' is exactly an abstraction of T'' [Zei84; ZKP00; Gru06].
2. The target model is obtained by adding one or several states or transitions. In this case, this means that a new abstraction (i.e. a refined source model)

which takes into account these new states/transitions can be found. Because abstraction guarantees structure preservation [Gru06], the property (1) remains valid. The refined abstraction source model can be obtained by introducing states/transitions which have no effect on the state variables.

□

Proof - Property 2. This property is a direct consequence of the property 1. Indeed, if there exists one path between two states p and p' in the source model, there exists at least one path between the corresponding states q and q' in the target model. Consequently, the state space generated by the source model is at least contained in the generated state space of the target model [KW16]. □

In the case in which the target formalism is DEVS, [Gia09] shows that any sequential machine [Brz03] can be easily rewritten in a DEVS atomic model. This strong result implies that any formalism which can be checked using model-checking can be rewritten using a DEVS model. Ensuring the simulation preorder between the models generated by the two formalisms is only needed.

Furthermore, if the models are bisimilar, thus this mean that any properties verified on the source model is true on the target model, and all properties verified on the target model are true on the source model. This property is usefull in the case of Verification and Validation of Simulation Models.

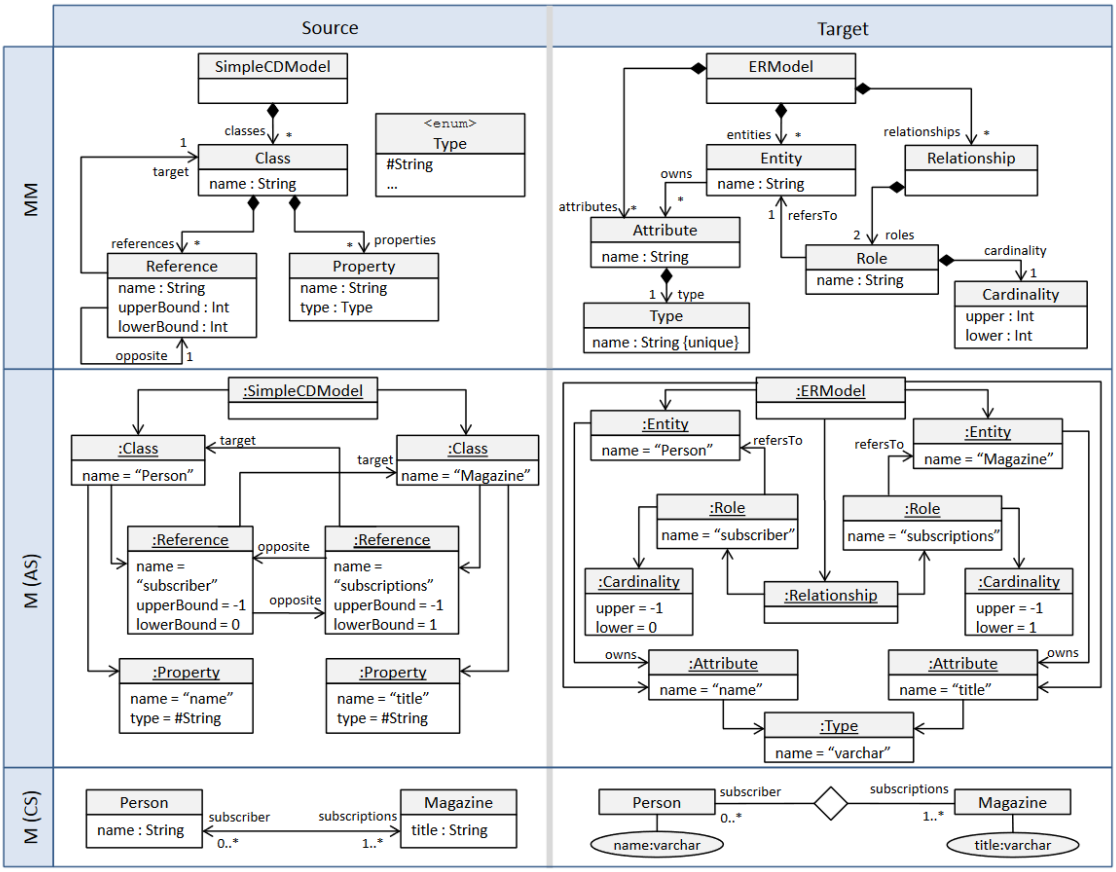
The second question is about the point 3 of the methodology. Are the concepts of event, state lifespace and types of transition enough to define a discrete-event model ? If we refer to Zeigler [Zei76], and as we have already seen in the Chapter 1, a discrete-event model is constant between two occurences of events. Events are well-defined by their respective date. If the dynamics of the model is defined by an automaton, then it remains to define a time-advance function in order to describe events occurences and the reaction of the model according to these events. Like in the DEVS formalism, the time-advance function can easily be defined by associating a lifespan to each state. This lifespan will describe when the automaton can autonomously behave. Then, for each transition, we must define if it corresponds to an autonomous behaviour or a reaction to an input event. This *type* of transition is then just a way to define this behaviour.

This semantics alignement can be done using the mapping and homomorphism concepts introduced by Zeigler, Kim, and Praehofer [ZKP00]. If the source and the target formalism have an operational semantics, the proof of alignment consists only on mapping states and transitions from one model to the other.

2.2.3 Approach using Model-Driven Engineering

When we talk translation of models, we think about Model-Driven Engineering (MDE). This approach provides a natural way to perform transformation of models. Indeed, Model-To-Model (M2M) transformation is the most important concept of MDE [SK03]. Kleppe, Warmer, and Bast [KWB03] define the transformation as "an automatic generation of a target model from a source model, according to a transformation definition. A transformation definition is a set of transformation rules that together describe how a model in the source language can be transformed into a model in the target language. A transformation rule is a description of how one or more constructs in the source language can be transformed into one or more constructs in the target language". M2M thus allows defining conceptual or syntactic transformations between models.

Figure 2.2: Example of mapping UML diagram to ER diagram [LWK10] - Source metamodel, target metamodel, source model (abstract syntax), target model (abstract syntax), source model (concrete syntax), and target model (concrete syntax)



Using M2M has two advantages: it allows to directly translate one syntactic

language to another, meaning it can be used for translating directly the formal specifications into the simulation model (or into the conceptual model then into the simulation model). If the metamodel (model of models) is well-specified, automatic translation can also be checked using formal proof. The other advantage is that the metamodel can express both syntax and semantics.

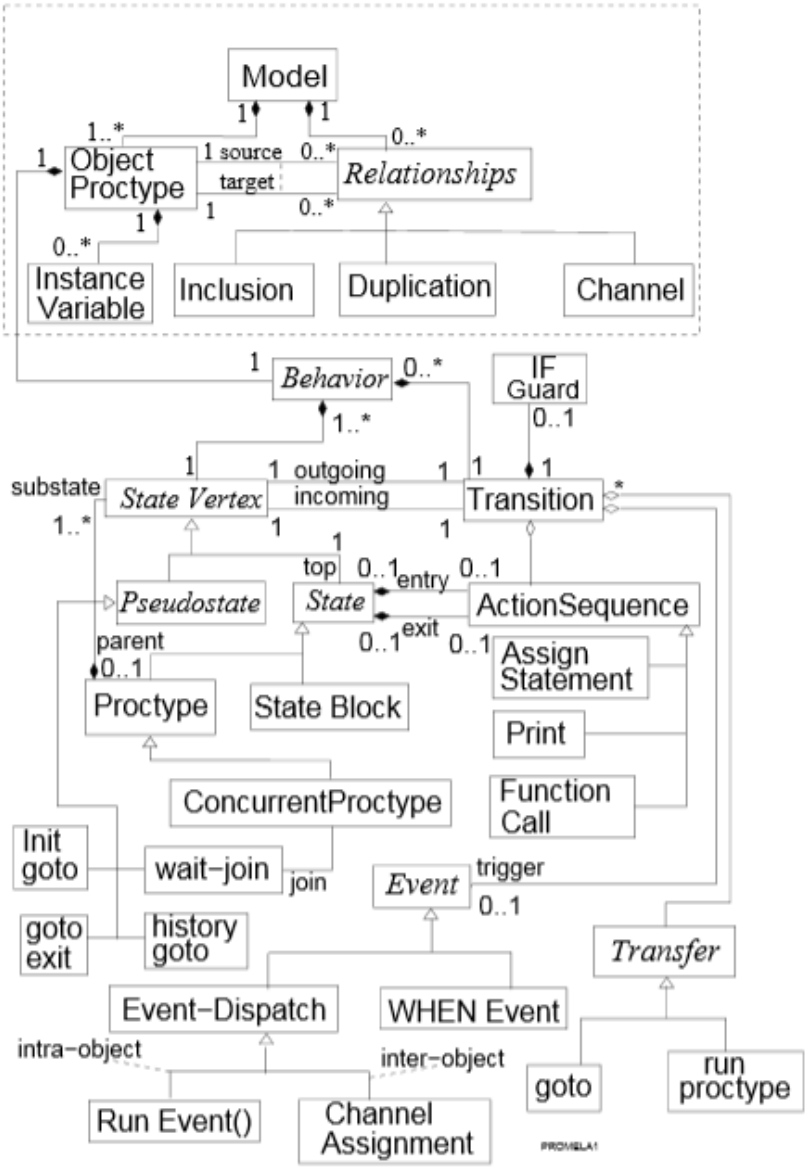


Figure 2.3: PROMELA Metamodel proposed by McUmber and Cheng [MC01].

However, more the language is complex, more the metamodel can be difficult to generate. In some cases, like for PROMELA (figure 2.3), the metamodel

brings a lot of useless informations (in this case states and transitions are hidding by syntactic elements). Indeed, while the underlying model generated by the model-checking algorithm is an automaton, it is easier and less error-prone to directly align the transition models themselves. Moreover, if using M2M makes easier the automatic translation, the proof of simulation preorder can be harder.

2.3 Introduction to DEv-PROMELA

Discrete-Event PROMELA (DEv-PROMELA) [Yac+15] is an extension of the Process Meta Language (PROMELA) [Hol91; Hol97; HNS00; Hol03] for the verification of asynchronous processes. DEv-PROMELA was developed using the methodology introduced in the previous section. It allows the verification and validation of discrete-event systems using both model-checking and discrete-event simulation. DEv-PROMELA was built using both PROMELA structure and the DEVS abstract simulator semantics. We have done manually the semantics alignment, but note the MDE approach could be applied, since there exists works on how transforming State-Transition models to DEVS models [GSB13]. Before introducing the syntactic and semantics of DEv-PROMELA, we need to make recall about PROMELA and its interpreter SPIN.

2.3.1 PROMELA Overview

PROMELA is a specification language, suited for modelling concurrent processes. It allows the verification of concurrent protocols, involving synchronous or asynchronous communication between processes. Based on Djikstra's Guarded Command Language, its syntax is close to any imperative language like the C-language, making their use very easy, compared with other formal methods. Indeed, PROMELA was developed to make easier the translation from any implementation to a verification model, and reduce the risk of translation errors. Even in its semantics, PROMELA is close to the C language. The difference is in the level of abstraction: PROMELA is interesting in only interactions between components/processes and not in computations. As a result, while a C software is deterministic, a PROMELA model is not. A PROMELA specification is thus a set of two separate parts: the system specification, on the one hand, which describes the behaviour of the model, and on the other hand, the properties to verify on the model.

System Specifications A PROMELA system is a *finite* set of components: *instances* of asynchronous *processes*. These ones can communicate each other thanks to different mechanisms as *buffered messages*, *shared global variables* or *rendez-vous handshakes*. Each instance of each process is modelled by a *finite* set of guarded or labeled command called *statements*. A statement is said non-blocked if the state of the system allows its execution, otherwise it is said

blocked. Then, one execution of the specifications, at any time t_i, corresponds to the execution of one among all of non-blocked statements, without any assumption about duration of the statement execution.

ALGORITHM 1: A simple example of PROMELA program.

```
 1: int z = 1;
 2:
 3: active proctype A {
 4:     int x = 2, y = 2;
 5:     if
 6:     :: ( x == 2 ) → x = 3;
 7:     :: ( y == 2 ) → y = 4;
 8:     fi;
 9: }
10:
11: active proctype B {
12:     int x = 2, y;
13:     do
14:     :: ( y == 2 ) → x = 2;
15:     :: ( x == 2 ) → y = 4;
16:     od;
17: }
18:
19: ltl {[](z == 1);}
```

Instructions are divided into two categories: statements that modify the system state and control-flow instructions. Statements relative to state changes are assignments and *message* exchange instructions. *Assignment* statements involve local and global variables, whereas communication statements involve buffered channels. It is important to note that, if assignements are always considered as *enabled* statements (i.e. they can be always executed), the instructions relative to channels can be *blocked* if the associated buffered channel is empty or full. *Control-flow* statements are classical conditionnal and loop instructions. These ones allow selection of the next statement among different branches regarding a guard. Because PROMELA processes are *non-deterministic*, if several guards are satisfied, a random one is selected. If none of them is satsfied, the control-flow structure is blocked. PROMELA also provides a *timeout* instruction (usable as a guard) which is enabled if all instructions are blocked in the whole system.

Datas in PROMELA are represented by local and shared variables. Local variables are those which are relative to only the process which they belong to, whereas global variables are shared by all processes. A variable is characterized by its value and its type, or any finite combination (structures) or finite arrays of these types. Each PROMELA type represents a finite set of values.

Properties Specifications SPIN supports the verification of Linear Temporal Logic (LTL) properties on the PROMELA models. LTL properties are converted into a *never-claim* process (comparable to any *normal* processes) which don't "participate" to the behaviour of the system. The goal of a never-claim process is only to guarantee the system satisfies the property which is encoded in it. In this sense, a never-claim process acts as a *monitor*. To get the final verified system, all the processes (including the monitors) are combined into a huge model on which formal verification is performed. Then, the formal verification of PROMELA specifications intuitively corresponds to the checking of all executable paths against a given property, without any assumptions of duration. It results that the next state of a PROMELA model does not depend on the time elapsed in a previous state.

2.3.2 PROMELA in Details

Now we have introduced PROMELA, we can see how it works in details. This part analyzes the syntactic elements of PROMELA and their semantics given in [Hol03]. It is interesting to note that the semantics of PROMELA is given by the SPIN interpreter. Natarajan and Holzmann [NH97] tries to give some outlines to define a formal operational semantics independant from the implementation. Because this semantics was redefined in an operational model in the chapter 7 of [Hol03], we use it as reference. For each element of the language, we give its syntax and its semantics. The whole semantics of PROMELA is then tackled.

As said in the previous subsection, PROMELA is a finite set of *processes*, whose each of them is a finite set of *statements*. PROMELA defines six basic statements: assignments, assertions, print statements, send and receive statements, and expressions. Each of these statement can be evaluated (as any C-statement) with two values (**blocked** and **non-blocked**) and acts on variables. Blocked statements correspond to a control state without transition (in the generated statespace), while a non-blocked statement corresponds to two control states linked by a transition. Each transition is labelled by the statement that corresponds to the action performed on the state variables.

Skip `skip` is the most basic statement. It is just a non-blocked transition between two states, and which performs no operation.

Assignment The assignment statement is just the fact to set the value of a given variable (local, global shared). Syntactically, it is defined as :

`x = value;`

where `x` is a variable. This statement is always evaluated as non-blocked, meaning it can be performed regardless of the state of the system. Semantically, it

corresponds just to a transition between two states 0 and 1. The value of x after triggering the transition is *value* (figure 2.4).

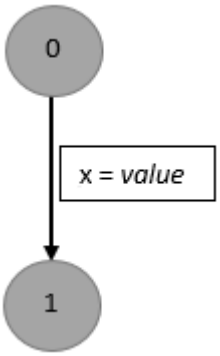


Figure 2.4: PROMELA Assignment.

Assertion The assertion statement is a simple way to state simple safety properties. This statement is always executable like assignment, and has only one side effect: change the control state of the process that executes it.

```
assert(condition);
```

Assertion works like in C: if the argument is evaluated to false, an assertion violation is returned by the checker. However, assertions has no effect in the semantics of the modelled system. That is why it is just a transition between two states 0 and 1, and that set a special flag to true or false (figure 2.5).

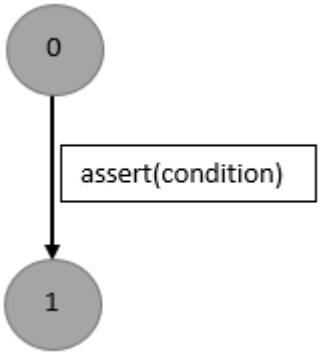


Figure 2.5: PROMELA Assertion.

Send statement The `send` statement is used in order to send message through a buffered channel. Channels are global sets of messages. A message is a set of global variables. Channels can be used for sharing data in a synchronous (rendez-vous handshake) or asynchronous manner.

```
chan a = [1] of int;
...
a!3; // Send 3 through a
```

The expression is evaluated to blocked if the channel is full. As a consequence, the automaton stops and will perform its next action as soon as the channel is free. Otherwise, the send statement is non-blocked. Then, the statespace generated by a send statement depends on the emptiness of the channel. It can be one state without transition, or a transition between two states (figure 2.6).

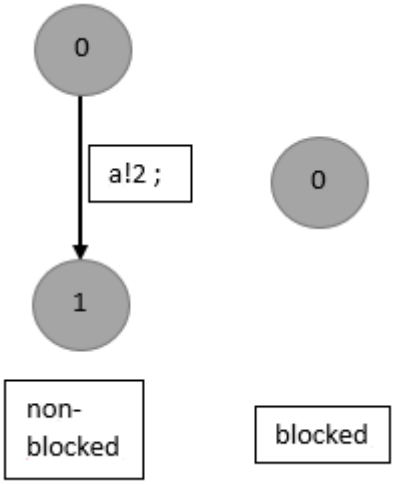


Figure 2.6: PROMELA Send statement generated statespace.

For *rendez-vous handshake*, the send statement is blocked until a corresponding statement is called (and conversely). In the case in which the rendez-vous is accepted, the send statement is called and the receive statement is immediately executed (meaning the next state of the system is the result of the both two operations).

Receive statement The `receive` statement is exactly the opposite of the `send` statement.

```
chan a = [1] of int;
...
a?3; // Receive 3 from a
```

The statement is evaluated to blocked if the channel is empty, or if the retrieved value from the channel does not match to the expected value. Otherwise, the statement is evaluated to non-blocked.

Expression Expression corresponds to any logic expressions or arithmetic computations. If the expression is a boolean condition, it is blocked until the value become true. If the expression is a computation, it is non-blocked.

Selection Selection is a control-flow construct that allows defining the underlying automaton. Selection is composed by a set of guards called options, which are expressions, and has a unique start and stop state. Each option corresponds to a path whose the first transition outgoes from the start state. The end of selection structure leads to the control state of the next construct (figure 2.7).

```
if
::  a == 2;
::  a == 3 -> a + 1;
fi;
```

Selection construct allows evaluation of each guard. Because a guard is an expression or a basic statement, only the executable guards can be executed. If several guards can be executed, one is chosen in a non-deterministic manner. If no guard can be executed, the system is blocked.

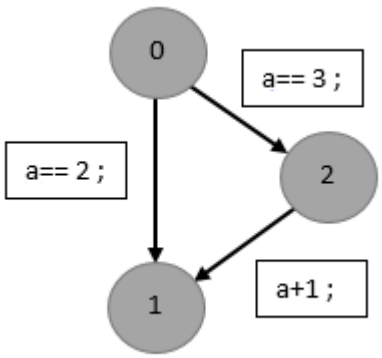


Figure 2.7: PROMELA Select.

Note that there is a special statement called `else` whose the semantics is different from its classical semantics in programming language. `else` is true if **all** the other guards are non-executable.

Loop Loop is similar to selection construct. The difference with the selection is that the path returns to the contral state before the construct (figure 2.8).

```
do
::  a == 2;
::  a == 3 -> a + 1;
od;
```

The `break` statement is the only one that is evaluated to non-blocked and which go to the next control state of the next structure.

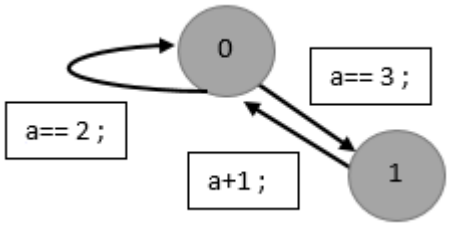


Figure 2.8: PROMELA Loop.

Unless The unless construction is a way to allow defining priority between statements. The left part is called the *main sequence*, while the right part is called the *escape sequence*.

```
a = b unless b = c
```

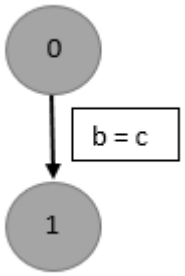


Figure 2.9: State space generated by the unless structure.

Main and escape sequences are finite block of statements. However, the executability of the main sequence depends on the non-executability of the whole escpace sequence. In other word, the main sequence is executed unless one of the statement of the escape sequence becomes executable (figure 2.9). If the main sequence is executed and one of the statement of the escape sequence becomes executable, the remainder of the escape sequence is executed and never returns to the main sequence.

Table 2.1: A list of PROMELA basic datatypes.

Type	Size (bits)	Value Range
`bit`, `bool`	1	$[0; 1]$
`byte`	8	$[0; 255]$
`mtype` (constants)	8	$[0; 255]$
`short`	16	$[-2^{15}; 2^{15} - 1]$
`int`	32	$[-2^{31}; 2^{31} - 1]$

Processes A process is a labelled finite set of *local* variables and sequential statements. Each variable is defined by a name and a type (see Table 2.1). The type defines a finite domain of data. Syntactically, a process is defined in a `proctype` block which can take one or several formal parameters.

```
proctype A (int a) {
    int b = 0; // Define a b variable
    b = a;
    a = 2;
}
```

Semantically, a process is denoted as

$$P =< pid, lvars, lstates, initial, curstate, trans >$$

where:

- pid is a positive value which identify the process;
- $lvars$ is a set of variables $\{(name, scope, domain, inival, curval)\}$;
- $lstates \subseteq INT$, which defines the identifiers of the local states of the process; $lstates$ hold no information;
- $initial$ is the initial state of the process; when $curstate = initial$, $\forall v \in lvars, v.curval = v.inival$;
- $curstate$ is the current state of the process.
- $trans$ is a the finite set of transitions $\{(tr\_id, source, target, cond, effect, prty, rv)\}$.

If this operational model suggests that a PROMELA process is a Moore Machine, it can be simplified and viewed as a simple finite state machine $A = (S, T, s_0, F_A)$ in which:

1. $S \in \{(id, v_0, v_1, .., v_n) \in D \times D \times ... \times D\}$ where D is a domain defined by the type of the variable i ($i \in [0..n]$) and id is an identifier of the state. This means the state is directly defined by the value of its local variables; then, each state holds informations about the local variables;
2. $T \subseteq S \times L \times S$, the finite set of transitions where L is the set of statements syntactically defined. Formally, a statement l can be seen as a function $l : S \rightarrow S$ which is a valuation function. Thus, $(s_0, l, s_1) \in T$ and noted $s_0 \xrightarrow{l} s_1$ if there exists a statement l in the specifications for which $l(s_0) = s_1$. T defines thus a partial transition function;
3. s_0 is the initial state;

4. F_A is the set of final states.

Thus, the automaton of a process is sequentially built by make transitions between the control states generated by each instruction. The final state is special state for which the ingoing transitions are labelled by the special "end" instruction. For instance, the previous example is defined by:

$$S = \{(0, i, 0), (1, i, i), (2, 2, i), (F, 2, i)\}$$

$$T = \{s_0 \overset{b=a}{\rightarrow} s_1, s_1 \overset{a=2}{\rightarrow} s_2, s_2 \overset{end}{\rightarrow} s_F\}$$

$$s_0 = (0, i, 0)$$

$$F = \{(F, 2, i)\}$$

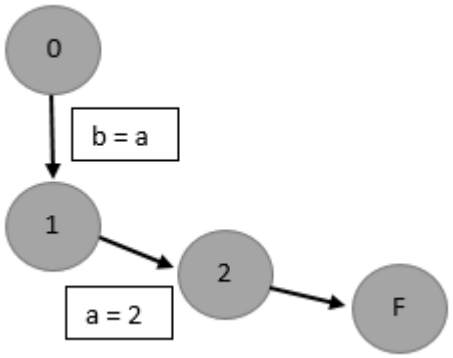


Figure 2.10: State space generated by the proctype structure.

Program A PROMELA program is a finite set of *global* variables, processes blocks and LTL blocks. Global variables are like local variables, but are defined outside from any processes. Their scope is global. The operational model defines a program as a set of system states

$$Pr = \{(gvars, procs, chans, exclusive, handshake, timeout, else, stutter)\}$$

defined as followed:

- $gvars$ is a set of variables (variables are defined as before);
- $procs$ is a set of processes;
- $chans$ is a set of channels;
- $timeout$ and $else$, two boolean that enforces the corresponding PROMELA statements.

ALGORITHM 2: PROMELA Semantics Engine.

```
1: while((E = executable()) ≠ {})
2: {
3:    for some {p, t} from E
4:    {
5:       s' = apply(t.effect, s)
6:       if(handshake == 0)
7:       {
8:          p.curstate = t.target
9:          s = s'
10:      }else
11:      { // try to complete an rv handshake
12:         E' = executable()
13:         // if E' is {}, s remains unchanged
14:
15:         for some {p', t'} from E'
16:         {
17:            p.curstate = t.target
18:            s = apply(t'.effect, s')
19:            p'.curstate = t'.target
20:         }
21:         handshake = 0
22:      }
23:   }
24: }
25: while(stutter){s = s} /* the 'stutter' extension */
```

These system states are interpreted by the operational engine of SPIN (Algorithm 2). The algorithm randomly takes an *executable* statement from all the executable statements in the program (i.e. all the processes). It executes this stamement (l.5) and applies its effect to the current global state, possibly changes the values of the variables, channels,... and gets the next system state. If no synchronization is needed (l.6), the global current state is set to the next state. Otherwise, we verify that another process can match the synchronization. If it is the case, the current state of both processes are updated, otherwise the algorithm chooses another executable transition. A transition is said executable if its condition is satisfied and it is not a synchronization request.

PROMELA introduces also a `timeout` statement which becomes true et can be executed only if there is no more executable instruction in the program. Timeout is useful to enforce the execution of a program which is in a deadlock situation.

Intuitively, the semantics previously defined corresponds to the asynchronous product between all processes.

Definition 33: Asynchronous Product of Automata

Given n automata $A_i = \{(S, s_0, L, T, F)\}$, the asynchronous product $A = \prod_{i=0}^{n-1} A_i$ is defined as:

$S = \times_{i=0}^{n-1} A_i.S$, the Cartesian product of each automaton set of states;

$s_0 = (A_0.s_0, A_1.s_0, ..., A_{n-1}.s_0)$;

$L = \cup_{i=0}^{n-1} A_i.L$;

$T = \{t = (s', l, s'')\}$ where $(s', s'') \in S \times S$ and $l \in L$. t exists if l labels a transition state change the valuation of any components of s' to any components of s'';

$F \subset S$ whose at least one components of each elements is a final state of one of the A_i.

Then, we can consider a PROMELA Program M as a finite state machine

$$P = (Q, T, s_0, F)$$

where

$Q = \{q_i = (id, l_1, ..., l_m, g_1, ..., g_n, c_1, ..., c_o) \in \mathbb{N} \times_{i=1}^{m} L_i \times_{j=1}^{n} G_j \times_{k=1}^{o} C_k\}$, the finite set of states; a state is characterized by the values of each local and shared variables, and channels (the all sets L_i, G_j and C_k);

$T \subseteq S \times S$, the set of transitions. By definition, T is *left-total*, meaning $\forall s \in S, \exists s' \in S$ such as $(s, s') \in T$;

$s_0 = (s_{0_1}, ..., s_{0_n})$ where $\forall i \in [1; n], s_{0_i}$ is the initial state of the process p_i;

$F \subset Q$, the finite set of final states of the program;

Denote $r = (s_m, s_n) \in S^2$. We note q_{m_i}, the state of the process p_i in s_m, and q_{n_i}, the state of the process p_i in s_n. Thus, $r \in T \implies \exists t \in T_i \setminus t = (q_{m_i}, l_i, q_{n_i})$, where T_i is the set of transitions of the process p_i and $l_i \in L_i$ (here, L_i is the set of statements of the process p_i). By this, we mean r is a transition of a PROMELA program only if there exists a transition t that changes the state of one of the processes composing the program, or changes the value of a global variable/channel. For the second part, l does not affect the state of a process but the state of the program. This means that we can also see a PROMELA program as a kind of hierarchical automaton [Har87; MLS97; AY98; Alu03] in which superstates are

$$Q = \{q_i = (a_1, ..., a_m, g_1, ..., g_n, c_1, ..., c_o) \in \times_{i=1}^{m} A_i \times_{j=1}^{n} G_j \times_{k=1}^{o} C_k\}$$

where A_i is the automaton of a PROMELA process.

LTL Properties LTL properties are `ltl` blocks of logical instructions. They use a particular set of binary operators:

[] the always operator;

<> the eventually operator;

! the negation operator;

U the strong until operator;

V the dual of U;

-> the implication operator;

<-> the equialence operator.

LTL properties are encoded into never claim automata which are built in the same manner than any processes. The automaton is executed with the other processes in the PROMELA program. In fact, these never claim automata corresponds to Büchi automata and are multiplied in a synchronous manner with the system automaton. That is why there is no semantics interpretation of ltl elements. Then, we can consider never claim automata as processes which have no effect on the behaviour of the system, and that act just as monitors.

```
ltl { <> (b == 3) } // Eventually b == 3

is equivalent to

never {
    accept: !(b != 3);
    if
    :: assert(!(b==3)) -> goto accept;
    fi;
}
```

2.3.3 Building DEv-PROMELA: Syntax

Now we have made recalls about the PROMELA syntax and semantics, we can use our proposed methodology to build our DEv-PROMELA formalism. We use the DEVS formalism we have defined in Chapter 1 as *target* formalism. The first step is to identify missing discrete-event concepts in the *source* formalism. We can quickly see that PROMELA and DEVS have both structure and semantics based on automaton. However, PROMELA does not use the following concepts:

- Events: the notion is missing because PROMELA makse the choice to focus on states for verification purpose. If an event occurs, PROMELA makes the assumption that the consequences of the event will necesserally appear in the state. Thus, if we want to record an occurence of an event, we just need to check that the implied modifications exist in the executions. Non-occuring events are thus the result of non-executable transitions. Concerning the concept of handshake and rendez-vous, it is represented by the channels synchronization mechanisms (send and receive statements).

- Time: as a direct consequence of the previous point, PROMELA makes the choice to focus only on untimed models. Thus, it is not possible in PROMELA to represent states that explicitely depend on time (i.e. depending on the elapsed time in the previous state).

- Transitions: Because there is no notion of events, there is no notion of autonomous behaviour or reactive behaviour. Indeed, we can interprete PROMELA Programs as closed systems which have only an autonomous behaviour. The internal behaviour of each process can be reactive (through the channel handshake), but they normally evolve independently from each other.

- Finiteness: As a restriction of formal verification, PROMELA enforces finiteness of the underlying model, even they can have an infinite behaviour. Then, states, transitions and data sets are finite, while DEVS allows infinite models.

Now we have identified the missing notions, we can define syntactic and semantics changes in order to allow modelling them.

A new datatype Firstly, a new abstract datatype **real** is introduced. It allows the representation of infinite and unbouded real values. This is especially needed for the modelling of time, but it can be used for modelling data. As any other PROMELA datatype, real can be used for defining local and global variables, like any scalar variables:

```
real i, j, k;
```

Real variables can also be used in structures and in arrays, without restriction. Because real is an abstract datatype, its semantics depends on the context. In simulation, real will be interpreted as a floating value, while it will be restricted to the integer domain in formal verification (in order to enforce the finiteness).

Events Events are just defined by integer constant values. `#define` or `mtype` can be used as follow:

```
#define evta 1 // Define evta
mtype = { evta, evtb, evtc } // Define evta, evtb, evtc
```

Statements PROMELA statements define the actions which are done when the system changes its state. DEv-PROMELA extends statements by prefixing each of them with *an event descriptor*. Event descriptors describe the delay between the execution of any previous statement and the prefixed one, or describe an event which will trigger the execution of this statement. By doing this, we define the concept of events and allows modelling of autonomous/reactive behaviours. Event descriptors are defined as follows in the Backus-Naur Form:

```
<event stmnt> ::= "[" <timed trans> "]" <stmnt> | <stmnt>
<timed trans> ::= <clt expr> | <evt expr> | <clt expr> <op> <evt
expr>
<clt expr> ::= "clt:" <real expr> "->emit:" <evt val>
<evt expr> ::= "evt:" <evt val> [ <op> <evt expr> ]
<op> ::= "|"
<evt val> ::= <mtype> | "silent"
<real expr> ::= <real> | "infinity" | /* Any C-function returning a
real value */
```

`clt` descriptors are called *autonomous descriptors* and describe the autonomous behaviour which the model will have after an amount of time. When it occurs, a message is generated by the model. `evt` descriptors are called *reactive descriptors* and correspond to reactions to an input message.

Consider the following examples:

1. `[clt`: $3.0 \rightarrow$ `emit:newa] a = a + 10;`
2. `[evt:newb] b = a - b;`
3. `[clt:lifespan(c)` $\rightarrow$ `emit:newc | evt:newd | evt:newe] c = c * d;`

(1) means that the execution of `a = a + 10` is performed 3.0 units of time after the execution of a previous statement. Before executing this statement, an event `newa` is emitted.
(2) means that the statement will be triggered only if the event "newb" is received.
(3) means that the statement `c = c * d` will be triggered either if the elapsed time between the execution of a previous statement and this one is equal to the value lifespan(c) (in this case, "newc" will be outputed) or, if the the event "newd" or "newe" occurs.

The third example shows one of the main characteristics of DEv-PROMELA: a statement can be executed in different manners, with at most one explicit timed descriptor (defined by `clt` command) and with at most one descriptor per event. Note that the `clt` command is optional. If it is not defined, we consider the elapsed time before the execution of the statement is equal to ∞. For convenience, if there is no event descriptor (no `clt` command nor `evt` command), the statement is interpreted as if it is prefixed by

`[clt`: $0.0 \rightarrow$ `emit:silent]`.

The statement is executed without any delay by emitting the default **silent** event. The silent event is a predefined event which does not cause any explicit change in the system.

Variables DEv-PROMELA supports global variables but their semantics is changed. Global variables (and channels) are in fact duplicated in each local process, and are considered as local variables. This ensures the principle of encapsulation of the DEVS theory.

Skip The only difference between the old and new `skip` is the event descriptor. Skip statement remains always executable. The event is sent before applying the skip action.

$[\texttt{clt:}\tau \rightarrow \texttt{emit:silent}]$ ***skip***;

Assignment As other statements, assignments are prefixed by an event descriptor. Evaluation of the new value and assignments are done in the same time. This corresponds to two states with one transition per event descriptor.

```
[clt:τ →emit:silent | evt:evta] x = a+b;
```

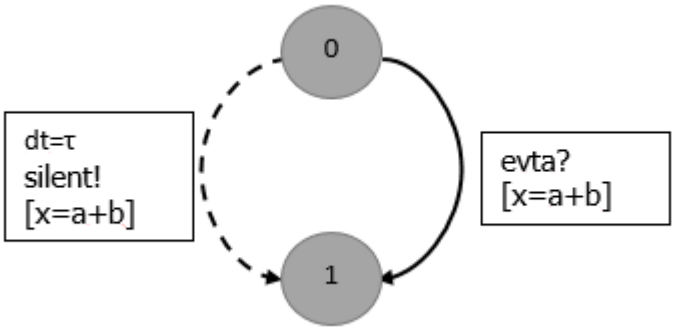


Figure 2.11: DEv-PROMELA Assignment.

Assignement structure is also changed for global variables. When the value of a global variable is changed, an event is emitted to inform the other processes of this changes. Synchronization mecanisms that change the local value of a global variable can be considered as `skip` statement in the corresponding PROMELA model.

Send and Receive statement As we have seen, `send` and `receive` statements are used in order to allow communications between processes in a synchronous or in an asynchronous manner. The communication between processes in DEv-PROMELA can be done using events, but events cannot exactly simulate the asynchronous communication. However, *Event Channels* allow delayed synchronous and asynchronous communications. The underlying structure changes from the PROMELA structure seen previously.

```
chan a = [1] of int;
...
[clt:τ →emit:valuea] a!3; // Send 3 through a
[evt:valuea] a?3; // Receive 3 through a
```

Communications with event channels work in three steps (figure 2.12):

1. if the current state is a state in which channel is full (resp. empty) and the next statement is `send` (resp. `receive`), the state waits an event that informs the channel is usable;

2. the message is put in (resp. retrieve from) the channel according to event specification; the silent event is emit;

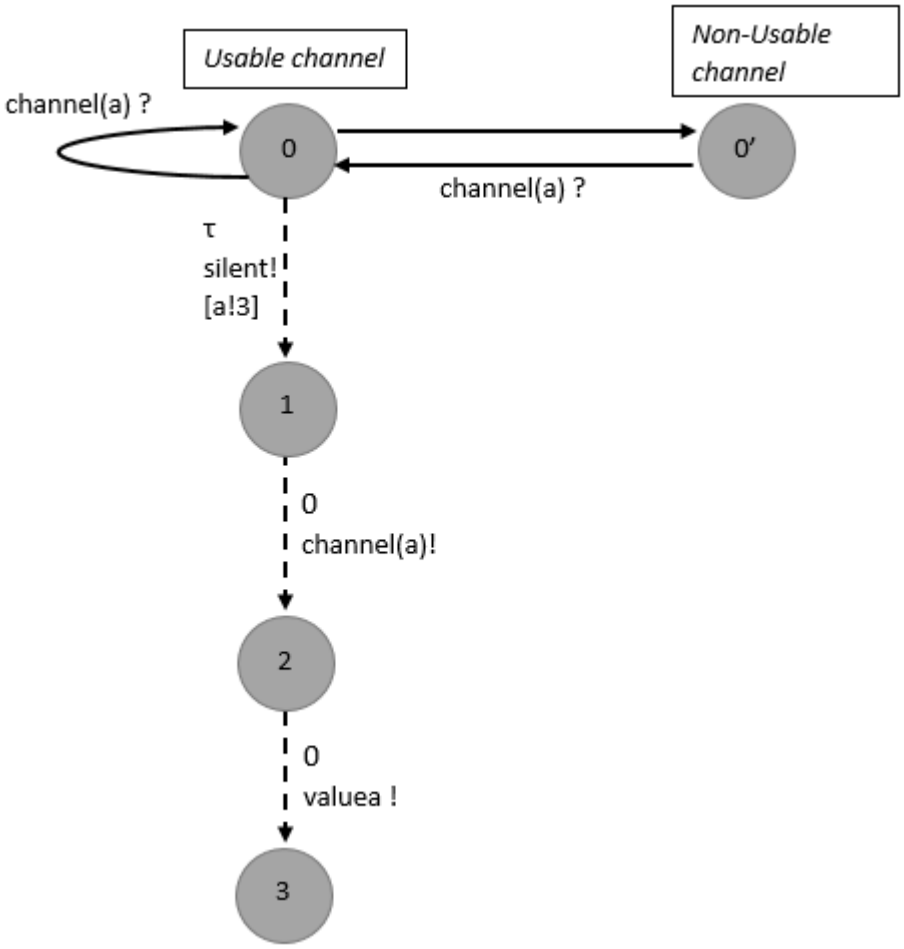


Figure 2.12: DEv-PROMELA Event Channel.

3. an event is immediately emitted in order to inform other processes of the change of the channel;

4. the real event (those specified in the model) is emitted.

This behaviour allows avoiding of retrieving values from an inconsistent channel. Indeed, because the event associated to a transition is emitted before processing the action, directly emitting the message could lead to inconsistent situation (trying to read an empty channel for instance).

Another question concerns the validity of this structure. The answer can be given only by describing the semantics of DEv-PROMELA. However, we can give some trivial elements for now. From a strict point of view, the behaviour of the DEv-PROMELA send/receive statement is not exactly the same than the PROMELA send/receive statement. Indeed, communication through a channel is a transition between two system states in PROMELA (because channels are global). However, DEv-PROMELA expresses the blocked transition by an infinite-

state. That is the reason why we can have two initial states. Secondly, the first transition (0→1) is used for updating the value of the channel. When it is done, the two next transition are immediately executed without changing the local state (i.e. the values of the variables). Thus, these transitions exist only for synchronozing purpose. Then, the equivalent PROMELA model is:

```
chan a = [1] of int;
...
atomic{
a!3; // Send 3 through a
skip;
skip;
}
```

Concerning *rendez-vous handshake*, DEv-PROMELA simulates it by adding an event that resizes the channel size to 1 when the receive statement is called. By this way, the send statement can be executed, and immediately after the receive statement is then executed. At the end of the receive statement, an autonomous transition resizes the size of the channel to 0.

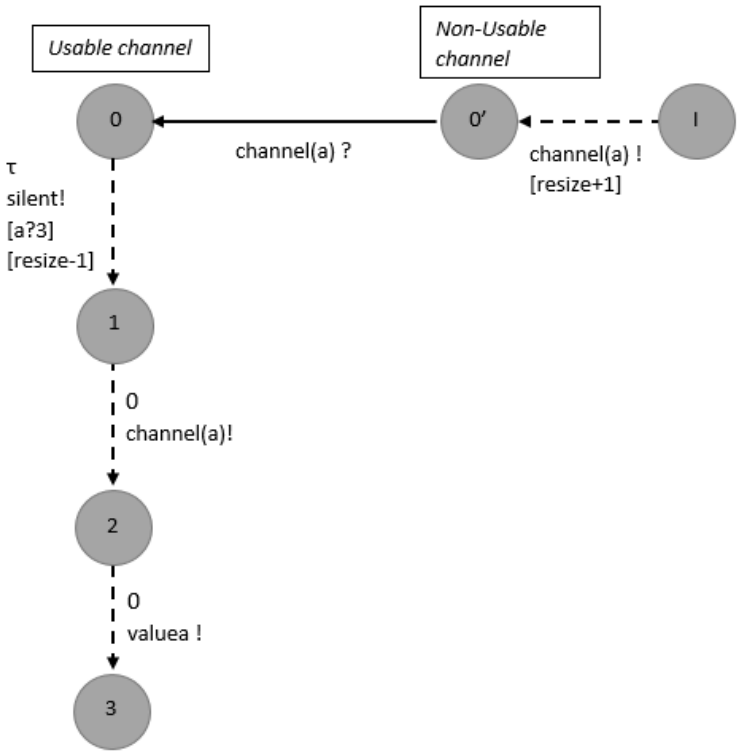


Figure 2.13: DEv-PROMELA Rendez-vous Handshake (receive).
[clt:τ →emit:valuea] a?3;

Selection construct The DEv-PROMELA Selection construct is like the PROMELA selection a way to define the structure of the underlying automaton. Each statement of each option can be prefixed by an event descriptor (Algorithm 4). However, the meaning of the DEv-PROMELA selection construct is not exactly the same than the one of the PROMELA selection.

PROMELA chooses the executed path according to the executable guards and allows non-deterministic behaviours. Take a look at the Algorithm 3. The meaning of this program is that if the statement `x==2` is executable, the path l.2 can be chosen by the program. In the same manner, if the statement `x==3` becomes executable, the path l.3 can be also chosen. Then, the statepace generated by the Algorithm 3 is ambiguous (figure 2.14).

ALGORITHM 3: PROMELA conditional structure.

```
1:  if
2:  :: ( x == 2 ) → x = 3;
3:  :: ( y == 2 ) → y = 4;
4:  fi;
```

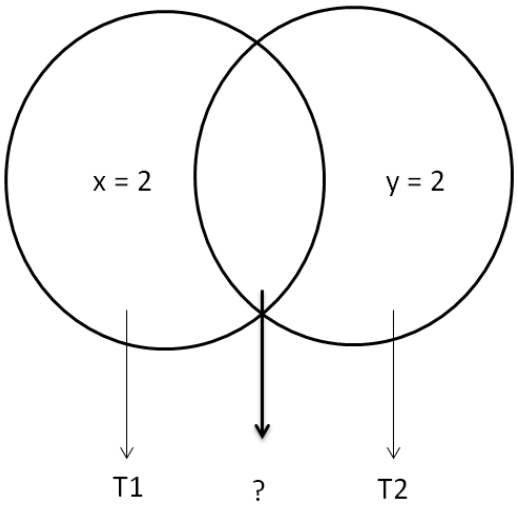


Figure 2.14: PROMELA selection construct generated statespace.

In the DEv-PROMELA model, such a construct defines transitions between a set of *equivalent* states. By this, we mean that the guard (the first statement of each option) clearly identifies the current state. Thus, the guards are evaluted according to three criteria in that order:

1. the logic of the first guard; for example, `x==3` means all states in which x is equal to 3, while `x=1` means *all the states* (because `x=1` can be performed on any states);

2. the lifespan; for example, Algorithm 4 means that if y is equal to 2, then the l.3 will be ever executed;

3. if multiple guards can be fulfilled in the same time, only the first declared guard can be executed.

ALGORITHM 4: DEv-PROMELA conditional structure.

```
1: if
2: :: [clt: 3.0 → emit:silent] ( x == 2 ) → [clt: 0.0 → emit:newx] x = 3;
3: :: [clt: 1.3 → emit:silent] ( y == 2 ) → y = 4;
4: fi;
```

In this way, DEv-PROMELA enforces a deterministic behaviour needed by any discrete-event simulation models. If at first glance this can be viewed as a restriction, it is not really the case. Indeed, this enforces designer to fully define the behaviour of the modelled system, and reduces the risk of error (even if purposes of abstraction is effectively to not take care of the determinism).

Note that the event descriptor can be placed before the control structure. In this case, this means that all evaluations of each guard will occur after the same delay.

Repetition construct The repetition construct works as the selection construct. The difference is that the outgoing transition returns to the begin of the loop.

ALGORITHM 5: DEv-PROMELA loop structure.

```
1: do
2: :: [clt: 3.0 → emit:newx] ( x == 3 ) → y++;
3: :: [clt: 1.3 → emit:newy] ( y == 2 ) → break;
4: od;
```

Consider Algorithm 5. If the program is in a state $(x = 3, y = 1)$, the first transition will be triggered after 3.0 units of time, leading to a new state $(x = 3, y = 2)$. The next instruction `y == 2` is then executed after a delay of 1.3 units of time.

Process priority Priority between processes is another thing that we need to be able to define. Indeed, if two events occur at the same time, we need to know what event must be processed in first. For that, DEv-PROMELA defines a process descriptor using the following grammar:

```
<proctype decl> ::= "[" priority "=" <int> "]" <proctype> .
```

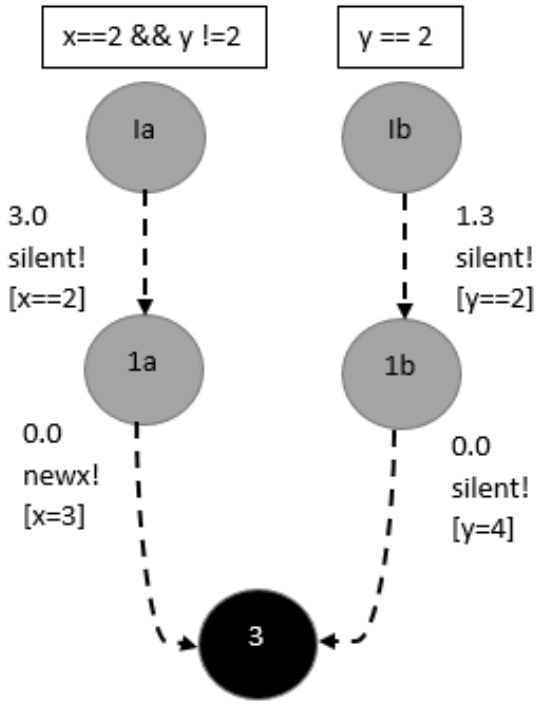


Figure 2.15: Generated statespace by the Algorithm 4.

Note that PROMELA includes a concept of process priority with a different meaning. PROMELA process priority enforces the execution of the process with the highest priority, unless it is blocked. In DEv-PROMELA, priority is handled by event order and process priority resolves conflict between two events that occur at the same time.

Clock and timeout The last syntactic element concerns time handling. PROMELA defines a **timeout** keyword used as an escape for a blocked system (i.e. a system for which there is no more enabled statement), for example when a system has no valid option to progress through a selection construct. Thus, `timeout` models, in an abstracted way, the fact that the system is able to handle deadlocks. In DEv-PROMELA, such a case means that the system is not well specified, or the model is incorrect. Deadlock must be handled through events. For this reason, `timeout` is not allowed in DEv-PROMELA.

Each process is also associated to a virtual local clock which measures the elapsed time since last event. DEv-PROMELA allows transitions depending on the elpased time. Two convenient instructions, **getElapsedTime** and **getCurrentDate**, allow access to the clock valuation.

2.3.4 Meaning of DEv-PROMELA : Semantics

After modifying the syntax in order to allow modelling of DEVS concepts, we introduce the semantics of DEVS abstract simulator, and we align it with the semantics of PROMELA.

Semantics of a DEv-PROMELA process A DEv-PROMELA process P with a set of statements L is an automaton $T = (S_\tau, E, \delta_i, \delta_e, s_0, F)$ where

- $S_\tau = \{s_i = (t_s, i, l_1, ..., l_m, \in \mathbb{N} \times \prod_{i=1}^{m} L_i \times \prod_{j=1}^{n} G_j \times \prod_{k=1}^{o} C_k)\}$ is the set of states. i is the identifier of the state related to the statement l which defines it; the sets L_i (resp. G_j) are the sets of values of each local (resp. global) variable l_i (resp. g_j).
- E is the set of events; E contains at least the silent event denoted ϵ;
- $\delta_i : Q_f \rightarrow Q_0 \times E$ is the internal transition partial function; δ_i is partial because it can be not defined for all $q \in Q_f$ (especially when $ta(s) = \infty$);
- $\delta_e : Q \times E \rightarrow Q$ is the external transition partial function; δ_e is partial because it can be not defined (especially when $e = \epsilon$).
- s_0 is the initial state;
- F is the set of final states.

Moreover, we define:

- $ta : \left\{ \begin{array}{l} S_\tau \rightarrow \mathbb{R}^+ \\ ta(s) \mapsto t_s \end{array} \right.$ is the state lifetime function; the lifetime of each state is given by the delay before executing the next statement in the specifications;
- $Q = \{q = (s, dt), \forall s \in S_\tau\}$ such that $0 \leq dt \leq ta(s)$ is the set of total states; dt denotes the time elapsed in the state s;
- $Q_0 = \{q = (s, 0), \forall s \in S_\tau\} \subset Q$;
- $Q_f = \{q = (s, ta(s)), \forall s \in S_\tau\} \subset Q$.

Consider a DEv-PROMELA process P in a state s at time t, and the next statement l with its event descriptor. We can admit the process P is in fact in a state $q = (s, t)$ (if t denotes the elapsed time since the last event). If l denotes an internal transition and if $t = ta(s)$, then the statement l is enabled. The event associated with the transition is emitted to all the other processes composing the program, before the transition is triggered, and the next event for the process P is defined by :

$$d_{e'} = getCurrentDate + ta(s')$$

with $((s', 0), e') = \delta_i(s)$. If l denotes an external transition on an event e, then the transition is triggered only if the process receives the event e. In this case, denote t the date of the event e. The next state is given by $q' = \delta_e(q, e)$ with $q' = (s', 0)$. If δ_e is not syntactically defined for (s, e), then the next state is given by $q' = (s, dt)$ and $ta(s) = t_s$. This behaviour corresponds to the semantics given by the DEVS abstract simulator.

The automaton underlying a DEv-PROMELA process is thus built by making transitions between control states (represented by each statement). The automonous event descriptors define the lifespan of the initial control state, while the reactive descriptors define external transitions between the initial control state and the end control state.

Semantics of a DEv-PROMELA program A DEv-PROMELA program P_r is a transition system $T = (S, \Lambda, \rightarrow)$ where

- $S = S_{\tau_0} \times ... \times S_{\tau_n}$ is the cartesian product of the set of states of each process;
- Λ is the union of the sets of all statements;
- $\rightarrow$ the set of transitions. $t \in \rightarrow$ and $t = s \rightarrow s'$ if:
  1. given two states $s = (s_{p_i}, s_{q_j}, ...)$ and $s' = (s'_{p_i}, s'_{q_j}, ...)$. Then, $s \xrightarrow{l} s'$ with $l \in \Lambda$ if there exists an internal transition from s_{p_i} to s'_{p_i}, or from s_{q_j} to s'_{q_j}, ... and if it does not exist any other internal transition which can be triggered before the date t. In other words, the next event of Pr is the minimum value of all the next events of each process and external events. If two events can occur at the same date, the first proceeded event is those generated by the highest priority process;
  2. given two states $s = (s_{p_i}, s_{q_j}, ...)$ and $s' = (s'_{p_i}, s'_{q_j}, ...)$. Then, $s \xrightarrow{l} s'$ with $l \in \Lambda$ if there exists an external transition from s_{p_i} to s'_{p_i}, or from s_{q_j} to s'_{q_j}, ... and if it does not exist any other external transition which can be triggered before the date t. If two events can occur at the same date, the first proceeded event is those generated by the highest priority process;
  3. if $(s'_{p_i}, e) = \delta_{i_p}(s_{pi})$, then $s' = (s'_{p_i}, \delta_{e_q}(s_{q_j}, e), ...)$. Thus, s' is the state of the program after processing all internal and external transitions at the date t. Moreover, this property ensures that a process can not proceed its own generated event.

In the case in which we consider that global variables and channels are handled at the system level like in PROMELA and don't need synchronization, the semantics of DEv-PROMELA Program slightly changes. In this case, a DEv-PROMELA program P_r is a transition system

$$T = (S, \Lambda, \rightarrow)$$

where

- S is the cartesian product of the set of states of each process, the set of global variables and channels that compose the program;
- Λ is the set of all statements, including statements changing the value of global variables and channels.
- $\rightarrow$ the set of transitions;

These two definitions are strictely identical. Indeed, the second definition generates a transition system that simulates the first one.

2.4 Relations and Morphisms

The main goal of DEv-PROMELA is to provide another way to enhance modelling, verification and validation of discrete-event systems by using simulation and model checking. To do that, we must demonstrate that a DEv-PROMELA model can be simulated and verified, namely we have the relationship described in figure 2.16.

2.4.1 Relations between DEv-PROMELA and DEVS

The first relation can be easily demonstrated by showing that, for a given DEv-PROMELA model, there exists a DEVS model that simulates it.

Proposition 1: DEv-PROMELA Process Simulation

A DEv-PROMELA process P is a DEVS atomic model A, and A simulates P.

This demonstration is relatively easy, thanks to the construction of DEv-PROMELA. Consider a DEv-PROMELA process $P = (S_\tau, E, \delta_i, \delta_e, s_0, F)$ and a DEVS atomic model $A = (X, Y, S, \delta_{int}, \delta_{ext}, \lambda, ta)$. P will define the same system as A if and only if :

1. $S_\tau = S$, both models have the same state space;
2. $X \subseteq E$ and $Y \subseteq E$; the sets of inputs and outputs are subsets of the set of events;
3. given s and s' two states such that $s' = \delta_{int}(s)$ and $y = \lambda(s)$; then, there exists s_τ and s'_τ such that $((s'_\tau, 0), y) = \delta_i((s_\tau, ta(s_\tau)))$ and $ta(s) = ta(s_\tau)$;
4. given s and s' two states, and x an input such that $\delta_{ext}(s, x) = s'$; then there exists s_τ and s'_τ such that $s'_\tau = \delta_e((s_\tau, x)$;

Proof. Considering a DEVS atomic model built upon a DEv-PROMELA process model, in which $X = E$, $Y = E$, and $S = S_\tau$. We define δ_{int} and λ such that:

- if $ta(s) \neq \infty$ and $\delta_i(q) = (q', e)$ such that $q = (s, ta(s))$ and $q' = (s', 0)$, then $\delta_{int}(s) = s'$ and $\lambda(s) = e$.
- if $ta(s) = \infty$, $\delta_{int}(s) = s$ and $\lambda(s) = \varnothing$. s is a passive state, then this transition will never be enabled.

We define δ_{ext} as follows: if $\delta_e(q, e) = q'$ with $q = (s, dt)$ and $q' = (s', dt')$, then $\delta_{ext}(q, e) = s'$. If $q = q'$ and $ta(s) \neq \infty$, then $ta(s') = ta(s) - dt$ and $q' = (s', 0)$. This condition ensures that δ_{ext} is defined for all $(q, e) \in Q \times X$ and time is preserved.

Then, we can show that A simulates P. Considering the transition system $< S', \Lambda, \rightarrow >$ where

- $S' = S_\tau \cup S$;
- $\Lambda = (X \cup E) \times (Y \cup E)$;
- $\rightarrow = Im(\delta_i) \cup Im(\delta_{int}) \cup Im(\delta_e) \cup Im(\delta_{ext})$;

where $\cup$ denotes the disjoint union operator. Therefore, A simulates P if there is a simulation $R = S' \times S'$ such that for all $(p, q) \in R$ and $l = (x, y) \in \Lambda$, if

$$p \xrightarrow{l} p'$$

then

$$q \xrightarrow{l} q'$$

However, $p \xrightarrow{l} p'$ only if

1. $(p', y) = \delta_i(p)$, meaning that p' is reached by an internal transition that outputs y. By construction, we know that there exists $(q, q') \in \rightarrow$ such that $q' = \delta_{int}(q)$ and $y = \lambda(q)$. Moreover, $ta(p) = ta(q)$ and $ta(p') = ta(q')$ by construction.
2. $p' = \delta_{int}(p)$ and $y = \lambda(p)$, meaning that p' is reached by an internal transition that outputs y. By construction, we know that there exists $(q, q') \in \rightarrow$ such that $(q', y) = \delta_i(q)$. Moreover, $ta(p) = ta(q)$ and $ta(p') = ta(q')$ by construction.
3. $p' = \delta_e(p, x)$, meaning that p' is reached by an external transition that consumes x. By construction, we know that there exists $(q, q') \in \rightarrow$ such that $q' = \delta_{ext}(q, x)$. Moreover, $ta(p) = ta(q)$ and $ta(p') = ta(q')$ by construction.
4. $p' = \delta_{ext}(p, x)$, meaning that p' is reached by an external transition that consumes x. By construction, we know that there exists $(q, q') \in \rightarrow$ such that $q' = \delta_e(q, x)$. Moreover, $ta(p) = ta(q)$ and $ta(p') = ta(q')$ by construction.

Thus, A simulates P. Symetrically, we can show that for all $(p', q') \in R$, if

$$q \xrightarrow{l} q'$$

then

$$p \xrightarrow{l} p'$$

Thus, P simulates A, meaning P and A are bisimilar. □

We can then build a DEVS atomic model that simulates exactly the behaviour of a DEv-PROMELA process.

Proposition 2: DEv-PROMELA Program Simulation

A DEv-PROMELA program P_r is a DEVS coupled model C (or a DEVS atomic model A).

Proof. Given a DEv-PROMELA program P_r with n processes P_1 to P_n. Given $EVENT$, a convenience function such that $EVENT(P_i)$ is the set of events of P_i. Then, we can define a DEVS atomic model $A = (X, Y, S, \delta_{int}, \delta_{ext}, \lambda, ta)$ which simulates the DEv-PROMELA program:

1. $S_\tau = \cup_{i=1}^{n} S_i \times \prod_{j=1}^{n} G_j \times \prod_{k=1}^{o} C_k$, where S_i are the sets of the states of each process, G_j the sets of the values of the global variables, and C_k the sets of the values of the channels;

2. $X \subseteq \cup^n EVENT(P_n)$ and $Y \subseteq \cup^n EVENT(P_n)$; the sets of inputs and outputs are subsets of the set of events;

3. $\delta_{int} : S_\tau \to S_\tau$. Given $s = (s_1, s_2, ...)$ and $s' = (s'_1, ...)$ and $s'' = (..., s'_2, ...)$ in S_τ. Then:
   - if $\delta_{i_1}(s_1, ta(s_1)) = (s'_1, 0, e)$ where δ_{i_1} is the internal transition function of the process 1, then $\delta_i(s) = s'$;
   - if $\delta_{int}(s) = s'$ and $\lambda_1(s_1) = e$, and $\delta_{e_2}(s_2, dt, e) = (s'_2, dt')$, then $\delta_{int}(s') = s''$ and $ta(s') = 0$; this case describes the internal coupling of DEv-PROMELA processes ;

4. $ta : S_\tau \to \mathbb{R}$. $s = (s_1, s_2, ...)$ and $s' = (s'_1, ...)$, then:
   - if s is the initial state, $ta(s) = min(ta(s_1), ..., ta(s_n))$;
   - if $s' = \delta_{int}(s)$, then $ta(s') = min(ta(s'_1), ta(s_2) - ta(s_1), ..., ta(s_n) - ta(s_1))$;
   - if $s' = \delta_{ext}(q, e)$ and $q = (s, dt)$, then $ta(s') = min(ta(s'_1), ta(s_2) - dt, ..., ta(s_n) - dt))$;

5. $\delta_{ext} : Q \times X \rightarrow S_\tau$; Given $s = (s_1, s_2, ...)$ and $s' = (s'_1, ...)$ and $s'' = (..., s'_2, ...)$ in S_τ. If $\delta_{e_1}(s_1, dt, x) = (s'_1, dt')$, then $\delta_{ext}(s, e) = s'$;

6. given $s = (s_1, ...)$, $\lambda(s) = y$ if $\lambda(s_1) = y$;

We must show that A simulates P_r. We denote by $STATESPACE(P_r)$ the total statespace of the DEv-PROMELA program P_r. If A simulates P_r, this means that for each $(p, p') \in STATESPACE(P_r)$ such that $p \rightarrow p'$, there exists $(q, q') \in S_\tau$ such that there exists an internal or an external transition to go from q to q'. But, $p \rightarrow p'$ if:

1. $p = (s_{p_i}, ..)$, $p' = (s_{p'_i}, ..)$ and $\rightarrow = (\delta_{i_p}, ...)$ such that $(s_{p'_i}, y) = \delta_i(s_{p_i})$; by construction, we know that there exists $\delta_{int}(q) = q'$ and $y = \lambda(q)$ that corresponds to the transition $\delta_i(s_{p_i})$. Moreover, $ta(p) = ta(q)$ and $ta(p') = ta(q')$. Indeed, the next event in P_r is generated by the minimum value of all the future events. And by definition, $ta(q') = min(ta(s'_1), ta(s_2) - ta(s_1), ..., ta(s_n) - ta(s_1))$.

2. $p = (s_{p_i}, ..)$, $p' = (s_{p'_i}, ..)$ and $\rightarrow = (\delta_{e_p}, ...)$ such that $(s_{p'_i}) = \delta_e(s_{p_i}, x)$ where x is an internal event generated by any other process of the system. This transition is enabled before the internal transition that has emitted the event is triggered. However, by construction, we know that there exists $\delta_{int}(q) = q'$ and $\varnothing = \lambda(q)$ that corresponds to the transition $\delta_e(s_{p_i})$, and $ta(q) = 0$.

3. $p = (s_{p_i}, ..)$, $p' = (s_{p'_i}, ..)$ and $\rightarrow = (\delta_{e_p}, ...)$ such that $(s_{p'_i}) = \delta_e(s_{p_i}, x)$ where x is an external event received by the system. However, by construction, we know that there exists $\delta_{ext}(q, x) = q'$ that corresponds to the transition $\delta_e(s_{p_i}, x)$.

Thus, A simulates P_r. □

We can then define a DEVS atomic model which simulates exactly the behaviour of a DEv-PROMELA program. It is interesting to note that in the case of a DEv-PROMELA program without global variables and channels (or using the definition 1 of the DEv-PROMELA Program), we can build a DEVS coupled model that simulates the DEv-PROMELA specifications, but encoding each process by an atomic DEVS model. In that case, the property of closure under coupling gives exactly the DEVS atomic model described above, and in this case, the coupled model is similar to the DEv-PROMELA program.

Proposition 3: Legetimate Property

A DEv-PROMELA program P_r is legitimate if the DEVS equivalent model is legitimate.

Because a DEv-PROMELA program can be simulated by a DEVS model, we can deduce all the properties of the program from the DEVS model. Particularly, a

DEv-PROMELA program is legitimate if the DEVS equivalent model is legitimate. For example, if the DEVS model goes into an infinite loop of internal events where time does not advance beyond a certain point, we can deduce that the DEv-PROMELA program has the same behaviour.

2.4.2 Relation between DEv-PROMELA and PROMELA

Model-checking on a DEv-PROMELA model is possible only if we can at least find an equivalent PROMELA model, meaning the structure expressed by the DEv-PROMELA model is at least included in the PROMELA model. For that, we must prove that there exists at least one PROMELA model which is an abstraction of the given DEv-PROMELA model. We can do that by using the pre-order simulation relationship between models.

Proposition 4: DEv-PROMELA Structural Preservation Property

Given a DEv-PROMELA process model P, there exists a PROMELA process model P' that preserves the structural properties of P.

Proof. The proof is done by construction. Consider a DEv-PROMELA process model P. We get a PROMELA process model P' by removing all the event descriptors and abstracting data from P. P and P' are two state-transition systems, whose respective entire statespace is denoted S and S'. Thus, P' preserves the structure of P if and only if

$$\forall (s, s') \in S \times S, \exists (t, t') \in S' \times S', \delta_i(s) = (s', e) \vee \delta_e(s, e) = s' \Rightarrow t \xrightarrow{l} t'$$

where l is a statement. Look at each type of statement defined previously.

Assignment A DEv-PROMELA assignment is a statement l with an event descriptor ev that defines one or several transitions between two states s and s'. A PROMELA assignment is a statement l that defines only one transition between two states t and t'. Then, if P' is obtained by removing the event descriptor ev, there exists a $(t, t') \in S' \times S'$ such that $t \xrightarrow{l} t'$.

Selection and repetition constructs A DEv-PROMELA selection (repetition) construct defines transitions between subsets of S. Given $S_a \subset S$ and $S_b \subset S$ such that S_a verifies a guard, and S_b is the subset of end states related to the selected option. This means

$$\forall s_a \in S_a, \exists s_b \in S_b, \delta_i(s_a) = (s_b, e) \vee \delta_e(s_a) = (s_b, e)$$

by executing the option. Then, if P' is obtained by removing the event descriptor, each couple (s_a, s_b) can be mapped to a (t_a, t_b) such that $t_a \xrightarrow{l} t_b$. Moreover, we

can be sure there exists at least one such couple because PROMELA allows non-deterministic behaviours.

Channels As viewed previously, the mechanism of channels is exactly preserved in DEv-PROMELA. Sending and receiving operations link two states s and s'. Only the lifespan and the meaning of transitions are changed. Thus, removing the event descriptor preserves the link between the states.

Therefore, we can find a PROMELA process model that preserves the structure of the DEv-PROMELA model. Note that we are talking about the automaton underlying the DEv-PROMELA model, not the generated statespace. □

Definition 34: Symbolic DEv-PROMELA

We call *Symbolic DEv-PROMELA* a DEv-PROMELA model in which events are ocurring in symbolic time. Then, we call *instance* of a DEv-PROMELA model P any parametrization of the lifespans of states and events (events occurences are not expressed in symbolic time).

Consequently, the Symbolic DEv-PROMELA statespace corresponds to all the possible scenarios and ordering of events of a DEv-PROMELA specifications.

DEv-PROMELA model allows modelling systems whose next states depend on the time elapsed in the current state. This behaviour can obviously, for example, lead to deadlock in passive state. Because PROMELA does an abstraction of time, these kinds of behaviours cannot be captured or modelled. However, a PROMELA model will be a good abstraction if it covers at least all the parametrizations of the DEv-PROMELA model without passive state.

Proposition 5: DEv-PROMELA Process Verification

Given a DEv-PROMELA process model P, and a PROMELA process model P' obtained by removing all the event descriptors. Then, all instances of P simulates P'.

This proposition can also be interpreted in the following way: there exists a PROMELA process model P for which the Symbolic DEv-PROMELA model corresponding to the specifications simulates the former.

Proof. As demonstrated, by construction, P' preserves all the structural properties of P. The set of instances of P (meaning the Symbolic DEv-PROMELA model) contains all the possible orders of events. Moreover, because P is a process, only a change in conditional/loop structures can lead to different behaviours between instances, because non-determinism can occur only in these structures. However, the PROMELA model P' contains all the possible paths

for these structures. As a consequence, P' is simulated by all the autonomous instances of P.

More formally, considering a Symbolic DEv-PROMELA process $P = (S_\tau, E, \delta_i, \delta_e, s_0, F)$ and the PROMELA process $A = (S, T, s_0, F_A)$. By construction, we have:

$\forall (p, p') \in S \times S, \forall l \setminus p \xrightarrow{l} p'$, then $\exists (q, q') \in S_\tau \times S_\tau \setminus \delta_i(q) = (q', e) \vee \delta_e(q, e) = q'$

Moreover, q' is obtained by applying the action l to q. This property is true thanks to the abstraction process. Then, there exists a simulation preorder between the states of P and P' in which some states of P simulates all states of P'. Then, the Symbolic DEv-PROMELA simulates the PROMELA abstraction. □

Proposition 6: DEv-PROMELA Program Verification

A PROMELA program P'_r obtained from a DEv-PROMELA program P_r by removing all event descriptors preserves the structural properties of P_r. Moreover, all instances of $P_{r'}$ simulates P_r.

Proof. A global state of a DEv-PROMELA program is the cartesian product of the set of states of each process. Thus, at a time t, the next event (and the next statement) is selected by taking the minimum value of the date of the next event of each process. This means that the statespace represented by all autonomous instances contains all the possible permutations between statements of several processes. This is exactly the reachability graph of the PROMELA model P'_r.

Thus, there exists a PROMELA model that is an abstraction of a DEv-PROMELA model. This model is obtained by only removing the event descriptors from the source model. Moreover, we can say that the DEv-PROMELA model simulates the PROMELA model.

More formally, denote P a DEv-PROMELA process with L the set of statements. As previously introduced, P is an automaton $A = (S_\tau, E, \delta_i, \delta_e, s_0, F_A)$. Denote P', a PROMELA process represented by a finite state machine $B = (S, T, s'_0, F_B)$. If P' is a structural equivalent to P, that means there exists a morphism M that transforms P to P', such as:

- $\forall s'_i = (l_1, ..., l_n, g_1, ..., g_m) \in S$,
  $\exists s_i \in S_\tau,\ s_i = (t_s, l_1, ..., l_n, g_1, ..., g_m)$; we denote $S \subset S_\tau$ by abstraction and $\phi : S_\tau \to S$, the abstraction function; (1)

- $\forall t_s \in \mathbb{R}^+$, $s_i = (t_s, l_1, ..., l_n, g_1, ..., g_m) \in S_\tau \implies$
  $\exists!\ s'_i \in S$ such as $s'_i = \phi(s_i) = (l_1, ..., l_n, g_1, ..., g_m)$; (2)

- $\forall t = (s'_i, l, s'_j) \in T,\ \exists (s_i, s_j, e) \in S_\tau \times S_\tau \times E$ such as $\delta_i(s_i) = (s_j, e)$ or $\delta_e(s_i, e) = s_j$, and $\phi(s_i) = s'_i$ and $\phi(s_j) = s'_j$; (3)

- $\forall(s_i, s_j, e) \in S_\tau \times S_\tau \times E,\ \delta_i(s_i) = (s_j, e)$
  or $\delta_e(s_i, e) = s_j \implies \exists(s'_i, s'_j) \in S \times S \setminus \exists t = (s_i, l, s_j) \in T$ and $\phi(s_i) = s'_i$ and $\phi(s_j) = s'_j$; (4)

- $s'_0 = \phi(s_0)$; (5)

- $F' = \phi(F)$. (6)

(1) and (2) mean that we can always make a projection of any state of S_τ to a corresponding state of S. Moreover, all states s of S_τ sharing the same memory state are projected to an unique state s' of S with the same memory representation. ϕ is thus a *surjective function*.
(3) and (4) mean that if there exists a transition function, either internal or external, from any state s_i to any state s_j, and if s'_i (resp. s'_j) is a projection of s_i (resp. s_j) by ϕ, then there exists a transition from s'_i to s'_j.
(5) and (6) means that initial and final states are preserved by abstraction.

Suppose M is constructed by only removing the event descriptors of P. By this, we mean that we delete the concepts of event, state lifetime and the characterization of each transition, in other words the notion of internal and external transition. M is thus an time-abstraction function such as (1) and (2) are verified (all timed states are projected to their untimed equivalent depending only on their memory state). Because we don't remove any statement, (3) and (4) are trivially verified. Indeed, even if a DEv-PROMELA (internal or external) transition is defined between two states $q_i = (s_i, dt_i)$ and $q_j = (s_j, dt_j)$ in Q, a such transition exists only if there is a relationship between s_i and s_j. However, a such relationship is defined syntactically (transitions are defined by the statements). Because we don't remove any statements, a such relationship is always existing, and even for the branching structures. For the same reason, (5) and (6) are also true.

Because a PROMELA program is a asynchronous product of each automaton that composes it, the graph of DEv-PROMELA program P is included in the one of P'. Indeed, the graph of P' is composed by the all possible permutation between statements, whereas the graph of P is only composed by the permutation of ordered events. That means, given two events e_1 and e_2 associated to two statement l_1 and l_2, if $date(e_1) < date(e_2))$, the graph of P will semantically not take into account the path where l_1 is executed before l_2 (the path may exists, but may be not valid). Thus, the morphism M defined previously is also valid for the entire program. As a consequent, the reachability graph of the Symbolic DEv-PROMELA can be abstracted to the reachability graph of the PROMELA model. Then, we can easily deduce that the symbolic DEv-PROMELA model simulates the PROMELA model. □

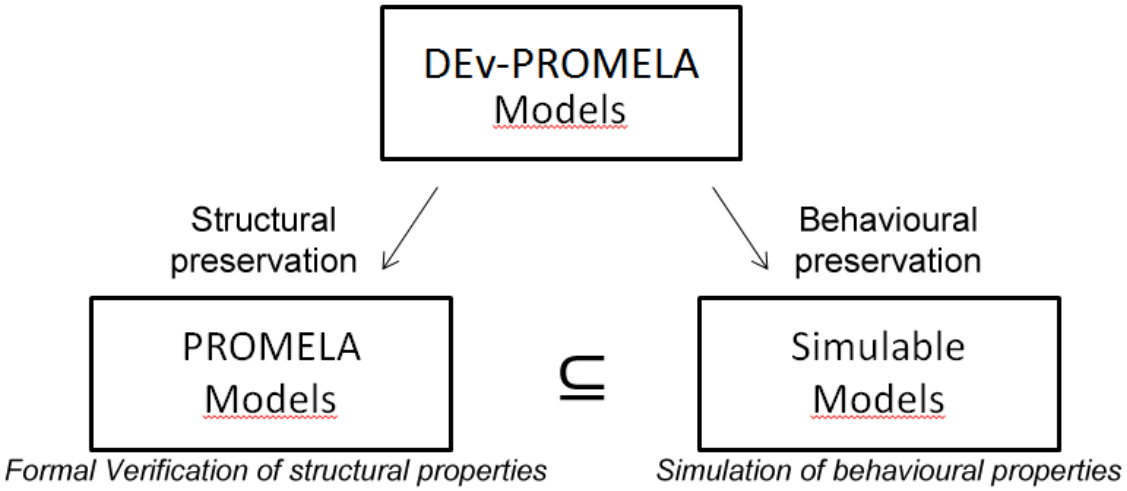


Figure 2.16: Representation of the relation between DEv-PROMELA, PROMELA and DEVS.

2.5 DEv-PROMELA and Simulation Formalisms Hierarchy

Before talking about of verification and simulation using DEv-PROMELA, we can talk about one aspect of DEv-PROMELA: the use of the simulation formalisms hierarchy that we have described in Chapter 1. This is useful if modellers want to restrict the behaviour of the model. Because DEv-PROMELA embbeds the semantics of Classic DEVS, it is possible to model all subclasses of Classic DEVS in the hierarchy. As example, we use the hierarchy proposed by Giambiasi [Gia09].

Sequential Machine with Transitory States (SMTS) SMTS can be modelled using the SMTS-PROMELA formalism, which is a DEv-PROMELA model with these syntax restrictions:

1. A SMTS-PROMELA model have only one process, no channel and no global variable.
2. An event descriptor can emit an output.
3. Only one type of descriptor (autonomous or event) is permitted by statement.
4. Lifetime of transitory states is 0.

The first constraint ensures the model is atomic (a SMTS is not modular). The second constraint allows modelling of the ouput function. The third constraint allows differentiation between steady and transitory states.

Temporal Moore Machine (TMM) TMM-PROMELA enforces these restrictions:

1. A TMM program can not involve channels or global variables;
2. Only one type of descriptor (autonomous or event) is permitted by statement.
3. An event descriptor can be followed only by an autonomous descriptor, and vice-verse.

The first constraint ensures that the synchronozing actions for channels and global variables do not break the TMM semantics. The second ensures that a steady state is followed by a transitory state and vice-verse. Because a TMM-PROMELA is a TMM, it is possible to compose a TMM program by modelling each process with a TMM atomic model. The closure under coupling described in [Gia09] ensures the TMM-PROMELA program is a TMM.

Temporal Sequential Machine (TSM) TSM-PROMELA specifications are DEv-PROMELA specifications with :

1. A TSM program can not involve channels or global variables;
2. An pure event descriptor must be followed by an autonomous descriptor;
3. Clock are reset.

Classic DEVS Finally, the DEv-PROMELA model is a Classic DEVS model without restriction. Using the hierarchy allows modeller to be sure to respect some structural properties. By translating the syntax of DEv-PROMELA into formal specifications, these structure properties can be verified using a theorem proving on the language (syntax proof) for example.

2.6 Verification, Simulation, Interoperability and Limits

2.6.1 Model-checking and Static Verification

Now we have proved the relations between DEv-PROMELA, PROMELA and DEVS, the question is what can be verified using DEv-PROMELA ?

Considering the definition 32 and that the PROMELA model is an abstraction of the DEv-PROMELA model, we can deduce that if a structural property is not fulfilled by the former model, then it will be not fulfilled by the second one. However, this property does not ensure structural equivalence in the generated statespace (i.e the behaviour of the models). By this, we mean that the absence of structural errors in the PROMELA model does not ensure the absence of error

in the DEv-PROMELA generated statespace (because of data abstraction). Structural deadlocks and static errors then can be found thanks to model-checking. However, behaviours and time-dependant properties must be verified/validated using both model-checking and simulation.

Furthermore, we have a proved there is a bisimilarity relationship between DEv-PROMELA models and their equivalent DEVS models. This means that DEv-PROMELA offers a way to formally verify invariant properties and structural deadlocks on a subclass of DEVS problems (those we can translate into a DEv-PROMELA specifications). Moreover, non-determinism cannot occur because the DEv-PROMELA syntax and semantics enforces determinism. DEv-PROMELA is thus a way to model a subclass of discrete-event systems.

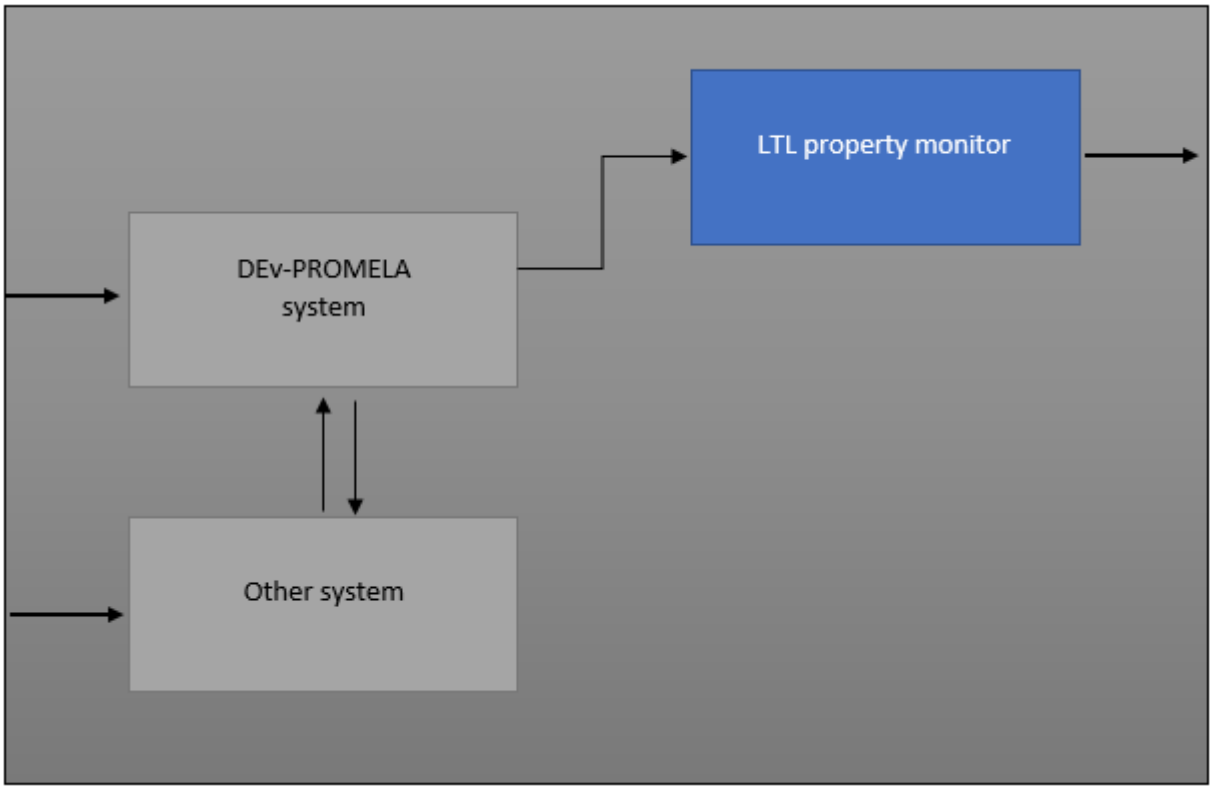


Figure 2.17: DEv-PROMELA LTL Verification using Simulation.

2.6.2 Simulation and Dynamic Verification

In the previous section, we show that a DEv-PROMELA model can be simulated as a DEVS model. Another way to consider the dynamics of a DEv-PROMELA program is to define a simulator. As for the DEVS simulator, the DEv-PROMELA simulator employs two time variables *tl* and *tn*.

ALGORITHM 6: DEv-PROMELA Simulator Algorithm.

```
 1: DEv-PROMELA-simulator
 2:   variables
 3:     tl          // time of the last event
 4:     tn          // time of the next event (init value: ∞)
 5:     model       // associated model
 6:   when receive i-message(i,t) at time t
 7:     tl = t - e
 8:     for each next imminent instruction s sorted by process priority
 9:       tn_old = tn
10:       tn = min(tl + ta(s), tn)
11:   when receive *-message(*, t) at time t
12:     if t != tn then error: bad synchronization
13:     y = λ(s);
14:     for each next instruction s' belonged to other processes than p(s)
15:       s' = δe(s', t - tl, y)
16:     send y-message (y, t) to parent coordinator
17:     s = δi(s)
18:     tl = t
19:     for each next imminent instruction s sorted by process priority
20:       tn_old = tn
21:       tn = min(tl + ta(s), tn)
22:   when receive x-message(x, t) at time t with input value x
23:     if not (tl ≤ t ≤ tn) then error: bad synchronization
24:     for each next imminent instruction s sorted by process priority
25:       s = δe(s, t - tl, x)
26:     tl = t
27:     for each next imminent instruction s sorted by process priority
28:       tn_old = tn
29:       tn = min(tl + ta(s), tn)
30: end DEv-PROMELA-simulator
```

The simulator of DEv-PROMELA is simple as shown in Algorithm 6. At each step, the simulator evaluates the next imminent instruction among all of the next instructions of each processes. The next event is the event with the minimum delay. An internal state transition message (*,t) causes the sending of an output message which is consummed be other processes. Then, the transition is executed and the date of the next event is recomputed among all imminent instructions.

When an input message (x,t) is received by the simulator at time t, all processes consummed the event. The date of the next event is then computed by taking the minimum delay among all imminent instructions.

Because simulation takes into account elapsed time, it can be used for verifying and validating behavioural properties like variables evolution. This can be done by encoding the LTL properties into a DEVS atomic model (we recall that a LTL property can be encoded through an automaton) which will act as a monitor (figure 2.17). This monitor registers all the fluctuations of interesting variables and changes its state according to the LTL semantics. If a property is violated, it emits an output event to the outside.

2.6.3 Interoperability

The most important and interesting functionnality of DEv-PROMELA is this property of equivalence between DEv-PROMELA and DEVS conceptual models. Indeed, this property allows DEv-PROMELA to benefit all the advantages of DEVS, including the independance between the conceptual model and the simulator on the one hand, and the DEVS Bus [ZKP00] on the other hand.

For the first, this means that DEv-PROMELA model can be implemented using any DEVS simulator. This allows a kind of interoperability between tools, which is an important thing even for the validation of simulators. Moreoever, that also means that a DEv-PROMELA model can be validated as a closed system, but also as an opened system. Indeed, it is possible to model the entire environment of the system and couple it with the DEv-PROMELA model. In this way, modelling interactions between components of the environment and the DEv-PROMELA model is easier than expressing all constraints in a formal language. The verification model can be as simple as possible, respecting the philosophy of formal verification, and validated under complex conditions.

This fact is pictured by the DEVS bus. The DEVS bus is a concept that allows multiple discrete-event formalism to interoperate each other. In other words, it allows multicomponent and multiformalism modelling and simulation. As a discrete-event formalism, DEv-PROMELA can be integrated in a such environment (figure 2.18). This is useful for example to model computations using

any of other DEVS formalisms, and pass the data to the DEv-PROMELA model which will not focus on computational aspects of the system. Verification of the DEv-PROMELA part is then done without remodelling the component. That allows modeller to choose where he wants to apply formal verification, and have more control on the complexity of the model. Then, we can easily see a DEv-PROMELA model as an autonomous component, or a subpart of an algorithm whose we want check only a part.

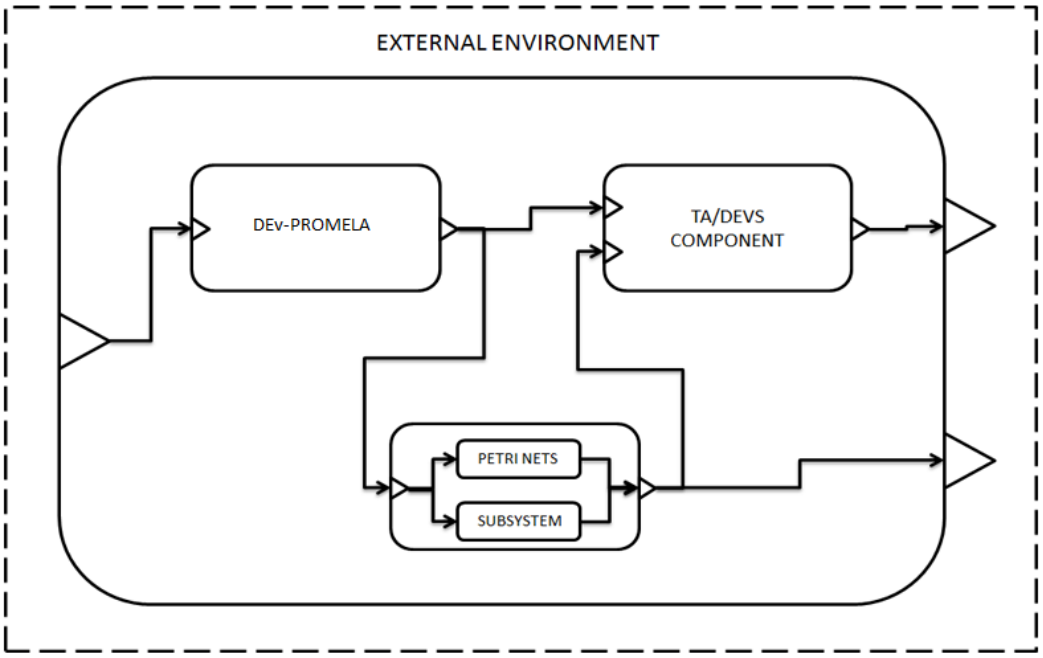


Figure 2.18: Modular system using DEv-PROMELA.

2.6.4 Comparison with other PROMELA timed extensions and Limits

The most effective way to appreciate the benefits and the limits of DEv-PROMELA is to make a comparison between DEv-PROMELA and existing PROMELA timed extensions. A summary is done in Table 2.2.

Real-Time PROMELA Real-Time PROMELA (RT-PROMELA) [TC96] was the first extension introduced to model real-time systems in PROMELA. The motivation was to introduce "*quantitative aspect of time*" into formal specifications, while "traditional formalisms dealed only with *qualitative aspect* of time, that is, the order of certain system events". Even this statement is not fully true from the point of view of discrete-event modelling as we will see in the next section, *quantitative* aspect of time is effectively important as said previously.

RT-PROMELA considers time as *dense*. Like Timed Automata (TA) [AD94], RT-PROMELA is not interested in the real order of events, but considers that a finite

number of successive events can occur between two defined moments. More precisely, this means RT-PROMELA is only interested in intervals of time in which events can occur, not in the date and the full order of the events. Taking into account the semantics of PROMELA, RT-PROMELA introduces two new concepts: *clock* and *timed statements*.

As in TA theory, a clock measures the time which linearly elapses. By this, many clocks can be used and be compared in guards of timed statements. These latter are those which depends on time. Because statements represent transitions between states, a timed statement is thus an action which is fired when a certain quantity of time was elapsed. This can be interpreted as two manner:

1. either a system must stay in a state during at least or at most d units of time;
2. or an action is fired before or after d units of time;

As a result, a RT-PROMELA program is thus a Timed Transition System (TTS) $S_\tau =< Q_\tau, \rightarrow_\tau >$ where:

- $Q_\tau = \{(q, v)\}$ with $q \in Q$ and v is a clock valuation (i.e. the tuple composed by the value of each clock)
- $\rightarrow_\tau \subseteq Q_\tau \times Q_\tau$ is the set of timed transitions.

Denote $q_{\tau i} = (q, v)$ and $q_{\tau j} = (q', v')$ in Q_τ. Given a timed statement $l = (st, R, \mu)$ labelling a transition $t = (q_{\tau i}, q_{\tau j}) \in \rightarrow_\tau$, t is enabled if $v \in \mu$, in other words if v satifies all timed constraints of l, and if $(q, st, q') \in T$ is enabled. RT-PROMELA also defines *time transitions* between (q, v) and $(q, v + \delta)$ with $\delta \in \mathbb{R}^+$.

If this is enough to explain relations between set of actions, RT-PROMELA does not explicitly enforce an order between *events*. Because the operator $==$ is defined, date of events can be precisely defined (on $\mathbb{N}$) with conjunction of process priority. As a consequent, a DES can be modelled with a RT-PROMELA model with some tricks: If all clock constraints are encoded with the operator $==$, lifespan can be encoded in the same manner as in DEv-PROMELA (because the semantics of clock valuation is given by $s = (q, v)$ with v, the values of all clocks for the state q). Events are then encoded using channels, and priority using PROMELA priority concept, to resolve the problem of non-determinism. So, what is the difference between RT-PROMELA and DEv-PROMELA ?

First, the RT-PROMELA clocks linearly evolve like in TA and can be compared only with integer values, restricting their possible valuation. This means, the following kind of properties cannot be expressed in RT-PROMELA:

$$next(e) = e * e$$

If we want to model such time evolution, modellers need to use a polynomial function to approximate the time curve, resulting a new state space explosion.

Second, RT-PROMELA allows non-deterministic behaviours which can lead to invalid paths in the sense of discrete-event simulation, while DEv-PROMELA enforces determinism and reduces the complexity of the model (if the model is deterministic, non-existing paths can be not generated).

Third, DEv-PROMELA allows modelling states that depends on the elapsed time. Datas in DEv-PROMELA can be function of time, while it is more difficult (but not impossible) to express that in RT-PROMELA without generating a new combinatorial explosion.

Last, DEv-PROMELA proceeds formal verification on the simpliest verification model, in the philosophy of formal verification. Timed properties and behavioural properties are verified and validated by simulation. As a resultat, the global speed of the model-checking is speed up and the complexity of the formal verification is reduced (because DEv-PROMELA avoids the double state space explosion induced by the introduction of time in the model-checking).

Discrete-Time PROMELA Discrete-Time PROMELA (DT-PROMELA) [BD98a; BD98b] also relies on the notion of time bounds. Time is sliced into intervals of fixed size indexed by natural numbers (called *ticks*), and events are framed into each slices. By this way, events belonged to two different frames can be ordered with a good quantitative approximation of the time that elapses between them. But inside a same frame, events have only a qualitative relation (Figure 2.19). Like RT-PROMELA, DT-PROMELA introduces *timers* which define the value of the current tick. In this sense, while RT-PROMELA uses a dense-time representation with a linear progression, DT-PROMELA assumes a discrete-time representation.

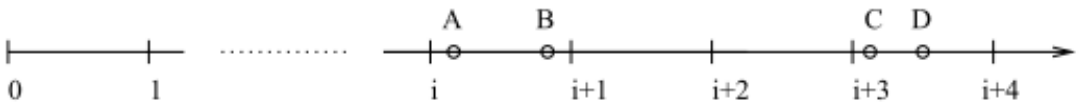


Figure 2.19: Time representation in DT-PROMELA [BD98a].

Formally, a DT-PROMELA program is a PROMELA one, for which each action is guarded by an integer value that represents the time. This value is decreased (or increased) by a monitoring process, enabling then transitions which depend on it. While in DEv-PROMELA time evolves along to events, DT-PROMELA provides a discrete representation of time, in which timed is "simulated" by increasing a tick. As a consequent, events cannot be fully ordered and systems modelled with DT-PROMELA are not event-based. However, the fact that DT-PROMELA is based on a discrete-time representation inevitably increases the statespace because variables are evaluated at each tick.

Timed PROMELA A third extension called Timed PROMELA (T-PROMELA) [NJJ08] also relies on time bounds. Similary to RT-PROMELA, each statement is bounded by an upper or a lower value, but with the difference that T-PROMELA statements do not depend on clocks. An instruction can be simply delayed or executed at exactly after d units of time. By this way, a T-PROMELA can be seen as a RT-PROMELA with one clock. It results that the semantics of a T-PROMELA is likely the same than the semantics of Timed Automata with Discrete Data (TADD). Verification of properties on T-PROMELA is done by performing verification on its TADD equivalent. From a strict theorical aspect, T-PROMELA suffers the same problem than RT-PROMELA.

Extensions of PROMELA for Hybrid Systems Finally, we shortly discuss about extensions of PROMELA for hybrid systems. "*A hybrid system is a system that evolves following a continuous dynamic, which may instantaneously change when certain internal or external events occur*" [GP14]. Hybrid automata [Hen96] have been developed in order to model hybrid systems and some verification techniques have been studied for particular classes of hybrid systems. Mainly, Alur et al. [Alu+93] show that the reachability problem is undecidable for hybrid automata and propose techniques for verifying safety properties of piecewise-linear hybrid automata.

Bosnacki [BD98b] propose an extension of PROMELA called HyPROMELA for modelling and verification of discrete-time rectangular automata. A discrete-time rectangular automaton is a rectangular automaton in which *implicit time transitions that represent the time flow and the evolution of the variables* is performed with a fixed duration of one time unit. In this way, an underlying discrete-time automaton of the rectangular automaton can be obtained and used for verifying properties.

In [GP14], the authors propose an non-intrusive methodology to extend SPIN and PROMELA for the verification and the analysis of some decidable classes of hybrid automata like linear hybrid automata, whose continuous variables evolve following constant differential equations, and rectangular hybrid automata. The verification is achieved by making a convex polyhedra abstraction of continuous behaviour of the system, as in Hybrid-PROMELA [SCR06]. However, the proposed methodologies always face to the state space explosion due to the continuous components.

Limits of DEv-PROMELA Finally, DEv-PROMELA seems to be a good compromise for modelling and verification of discrete-event models. However, DEv-PROMELA is suffering two major drawbacks. The most evident concerns the behavioural verification using simulation. How can be sure the verification and validation are really finished ? This problem is inherited from the empirical aspect of simulation. This problem is the same than completeness of tests in verification and validation of software. Efficiency of tests depends on the coverage

Table 2.2: Comparison between PROMELA timed extensions.

DEv-PROMELA	PROMELA	RT-PROMELA	DT-PROMELA
Supports Formal Verification with SPIN for non-timed properties	Supports Formal Verification with SPIN Model-Checker	Supports Formal Verification with RT-SPIN	Supports Formal Verification with DT-SPIN
Supports Discrete-Event Simulation using any Classic DEVS Simulator	Simulation is likely execution of the model	-	-
Supports time modelling through discrete-event behaviour	No explicit time	Linear clocks	Discrete-Time
Infinite sets can be modelled (only real for the moment)	Finite Sets	Finite Sets	Finite Sets
Models can be combined with any discrete-event models thanks to the DEVS Bus	-	-	-
Verification can be not exhaustive	Exhaustive verification	Exhaustive verification	Exhaustive verification

of the scenarios played. If it is possible to simulate all the statespace, the process can be long. A possible solution would be to integrate coverage tests of scenarios in the process, but that is not the object of this work.

The second limitation concerns the kind of systems that DEv-PROMELA can handle. In fact, DEv-PROMELA is able to check properties on a subclass of Classic DEVS problems, meaning all the Parallel DEVS models can not be expressed in DEv-PROMELA. This is because of the semantics of PROMELA. This is not

really a problem at this point because we could apply our general approach to improve DEv-PROMELA or use another verifiable specifications language which would be more appropriate to represent the systems under study. Indeed, like stated by Zeigler, Kim, and Praehofer [ZKP00], "Formalisms are proposed, and sometimes accepted, because they provide convenient means to express models for particular classes of systems and problems".

2.7 Conclusion

In this chapter, we have proposed a generic approach in order to integrate simulation formalisms into formal specifications languages. This approach is decomposed in three global steps and consists on adding to the verifiable formalisms some discrete-event concepts. These concepts can be identified using the formal definitions of the language, or using an MDE approach (which allows automations of translation). The second step consists on aligning the semantics between the two formalisms and defining two morphisms that respects a simulation preorder relations between models. Finally, the model described in the new specifications language can be translated (figure 2.20) into the initial formal specifications for formal checking of structural properties (especially invariants and deadlocks), or into the simulation language for verification and validation of behavioural properties (properties depending on time). The most advantage is that we move out the complexity problem to the simulation, and performs simple formal verification on the model. The global result of verification and validation is then given by using both methodologies: if an error is found during the model-checking, we can be sure the model is incorrect, while only both formal verification and simulation can detect behavioural problems.

Then, we have introduced a new formalism called DEv-PROMELA which is the result of combining Classic DEVS and PROMELA using our approach. DEv-PROMELA gathers all the advantages of DEVS with a formal verification capability. As an event-based formalism, it can also be seen as a PROMELA extension that implements an hybrid event-state approach for modelling processes. We show how we can use DEv-PROMELA in a verification and simulation environment using multiple formalisms and multiple components.

We succinctly compared DEv-PROMELA and other timed extensions, and present the two majors drawbacks of PROMELA: the empirical aspect of simulation and the limit of the class of problems. Also, a more precise study of the generated statespace is needed to reduce the complexity of simulation. Indeed, since a DEv-PROMELA model is equivalent to a DEVS model, reduction techniques could be applied.

The question is now: how using efficiently a combined V&V approach in a development cycle. This will be the topic of the next chapter.

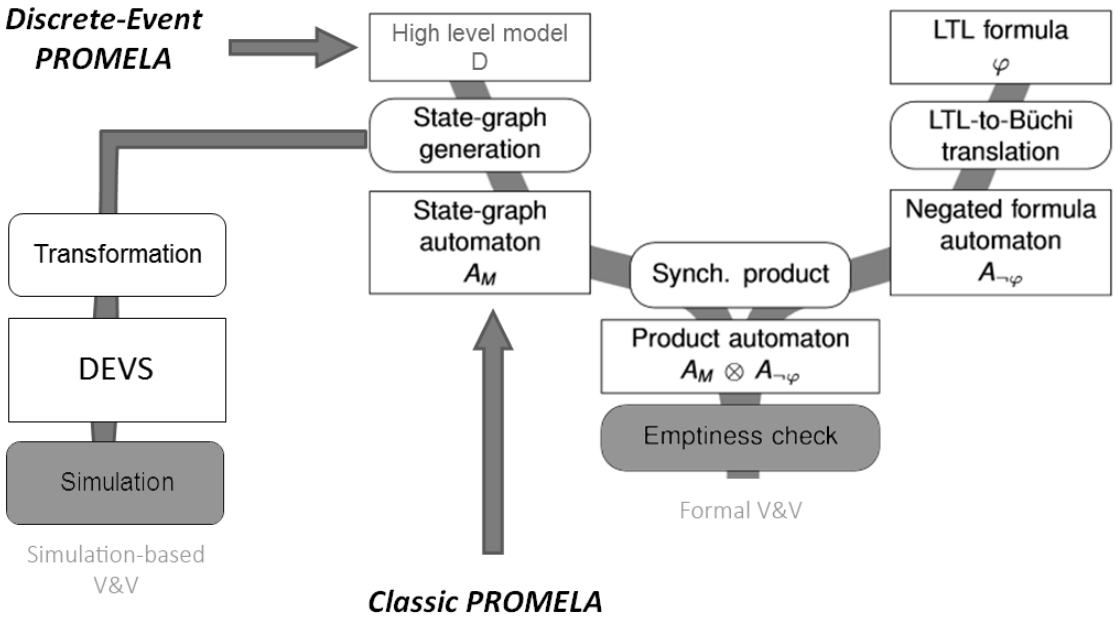


Figure 2.20: Combining Model-Checking and Simulation.

Chapter 3 Modelling, Verification and Validation with DEv-PROMELA

Design is an art, creative artist must have a beautiful mind.
Bhupesh B. Patil

An algorithm must be seen to be believed.
Donald Knuth

3.1 Introduction

In the previous chapter, we have defined a new specifications language DEv-PROMELA which can be used for modelling, verification and validation of discrete-event systems. We also touched on the interoperability between tools and formalisms. However, we must tackle the topic: How can we really use DEv-PROMELA in the Verification and Validation process of Simulation Models ? of Software ? What is the impact of such a methodology in the Software Development Life Cycle ?

We will answer these questions in this short chapter. Firstly, we will interest in how using DEv-PROMELA for the verification and validation of software, and especially event-driven software implemented using the event-driven programming paradigm (even if it works also for any software that have timed constraints). This will allows us to define the notion of level of V&V which will help to understand what exactly designers are verifying and validating. Then, we will see what is the impact of DEv-PROMELA in the V&V of simulation models.

3.2 Framework Entities and Intuitive Relationships

The framework of Combined V&V of Software can be easily depicted as in figure 3.1. Given a software source code (which is a program, and so a model) and requirements (or conceptual model), a DEv-PROMELA model is extracted or generated. If the source code of the software is generated from the DEv-PROMELA model, a syntax proof is performed to ensure that the translation is correct. Then, in order to perform combined Verification and Validation of the Software Model, the DEv-PROMELA model is translated into a PROMELA model and into a DEVS conceptual model. Syntax verifications are performed in these step in order to ensure correctness of the translation. The PROMELA is then verified using the SPIN model-checker, achieving the formal part of the combined V&V.

The DEVS conceptual is implemented into a simulation model. Classic V&V techniques of simulation models are applied (or a combined approach can be used). Then, the simulation-based part of the combined V&V is performed using the simulation model. If all the morphisms and translations are correct (including the certification of the simulator), then properties verified using the combined approach are guaranteed on the software. During the process, all properties which the model-checker was not able to verify were checked using simulation and vice-verse.

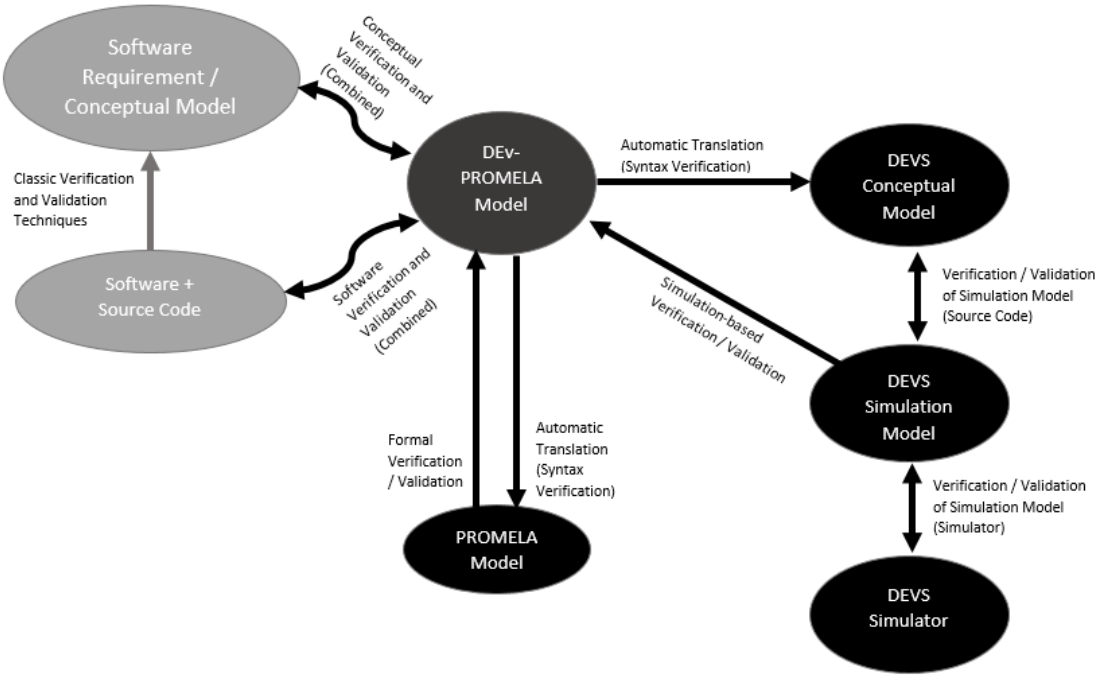


Figure 3.1: Combined V&V Entites.

3.3 Modelling and Verification of Software

As we seen in the Chapter 1, Software V&V essentially consists on performing static and dynamic tests on the software at each phase of the SDLC, while validation was performing at the first phase and at the coding phase. We also saw that prototyping could help validation and it was possible to compare results of the prototype and the final software. We extend this cycle by introducing new activities at each phase as shown in figure 3.2.

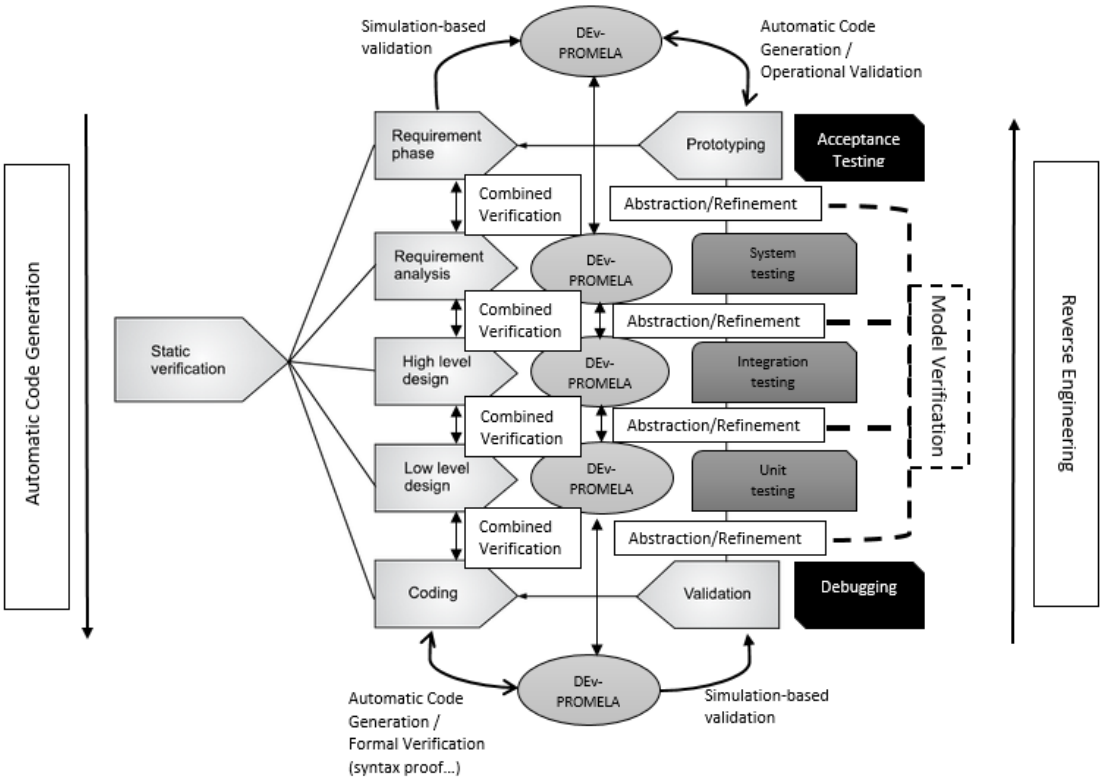


Figure 3.2: V&V in Software Development Life Cycle.

3.3.1 Verification techniques

Requirement phase During this phase, a DEv-PROMELA model can be made as supportive model for the prototype, or can be the prototype. In the first case, the DEv-PROMELA model is simulated to generate outputs and to give a first idea of the behaviour. The outputs of the models can be compared with the outputs of the prototype in order to determine if the constraints and assumptions made during the requirement phase are conformed to the needs of the user (validation). In the second case, simulation models are already software that simulates the behaviour of the final product. Because DEv-PROMELA simulator

can be syntactically built from the conceptual model, prototyping is faster and easier. Moreover, this is in this step that test datas are generated, and test scenarios elaborated. Then, in summary, the DEv-PROMELA model at this step is just coupling input/output functions.

Requirement analysis In this step, informal requirements are formalized, and system test cases are written. The DEv-PROLEMA model at this step is refined from the DEv-PROMELA model at the previous step. Combined Formal Verification and Simulation is used for verifying that the conceptual model represented by the DEv-PROMELA model holds the requirement of the previous phase (for example, bounds valuations, mathematical function definition).

High level design At this level, the DEv-PROMELA model is then refined to make appearing coupling processes. Behaviour and communications between black boxes are verified. Simulation-based verification helps to detect missing input/output.

Low level design At the this level, the DEv-PROMELA model is refined to make appearing state-transition structure in the white box. Formal Verification can help to verify the structure of the state-machine while simulation-based verification helps to verify if the behaviour is always conformed to the requirements defined above.

Coding At this level, the DEv-PROMELA model is close to the implementation, and remains a conceptual model of the source code, even if the corresponding simulation model is used as base for the final code (especially if the *event-driven programming* [Dab+02] or *time-driven programming* [PT96] paradigms are used). The final source code can be checked using formal syntax proof, while the conceptual model is verified using model-checking for invariant properties. Simulation-based verification completes the formal verification by checking the behaviour of the model.

All these steps can be completed by using traditional static techniques that we have introduced in Chapter 1.

3.3.2 Validation techniques

For Validation, we take the right side of the 'V' from bottom to up.

Coding Validation is proceeded after debugging, by executing the software in its real environment. Furthermore, because we have a simulation model of the code, it is possible for the customer to validate the behaviour of the software in a logic time. The ouputs of the simulation model and the software must be the

same. Note that the simulator must have been certified. Model-checking is used as a complementary formal validation technique.

Low level design As for the validation step, unit testing are completed with simulation-based validation. In fact, some tests can be performed on the simulation model. If both simulation model and software have different behaviour, then designers can deduce that something were wrong during the development. At this level, tests are focusing on atomic models. Model-checking is used as a complementary formal validation technique.

High level design Integration testing are completed by tests on the coupled models (the components of the software). Scenarios are played also on the simulation model to make a double validation of the conceptual model and the software.

Requirement analysis Systems testing and simulation-based validation validates the behaviour of the simulation model and of the software in its real environment.

Requirement phase Acceptance tests are performed on the software and used as scenarios of the simulation model.

It is interesting to note that model-checking on the DEv-PROMELA model can also be used to check some properties during the validation phase. In this case, validation is performed combining a simulation-based validation, tests and formal validation. Moreover, our proposed workflow implies the use of a conceptual model and a simulation. Indeed, each step of Verification and Validation uses a DEv-PROMELA model which is a model of the software, in addition of the traditional tests on the software and requirements. However, this is possible only if the simulation model is conforms to the conceptual model embbeded in the DEv-PROMELA model. This means the model must be verified between each step and that the simulation model (which is a computerized simulator) must also be verified and validated.

As a consequent, our proposed workflow have at least two levels of Verification and Validation:

1. The software level which consists on really verifying and validating the software using tests and conceptual models (DEv-PROMELA, UML..);
2. The simulation level which consists on verifying and validating the simulation model used in the higher level;

3. The model level which consists on validating the conceptual model used in the higher levels. This level is likely included in the top level, during the requirement phase, using the prototype and the simulation model.

The next section will talk about the second level of V&V: Modelling and Verification of Discrete-Event Simulation Models.

3.4 Modelling and Verification of Simulation Models

A simulation model is software that implements the behaviour of a conceptual model. Thus, like any software, it must be verified and validated before used. Figure 3.3 recalls the step of the V&V of simulation models seen in Chapter 1.

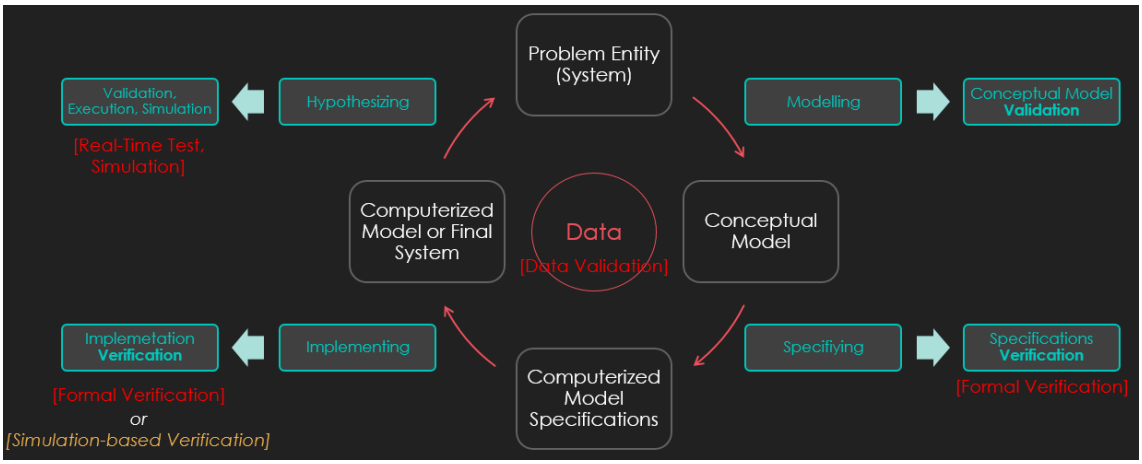


Figure 3.3: V&V of Simulation Models.

There are five critical points:

- Data Validation ensures that the data used to perform V&V are correct and corresponds to the modelled system;
- Conceptual Model Validation that ensures the conceptual model represents the modelled system;
- Specification Verification which checks that the computerized model specifications (the software) fulfills the requirement;
- Implementation Verification which ensures the simulation model (the software) meets the requirements of the specifications;
- Validation which ensures that simulator is well built and works as expected using the validated data.

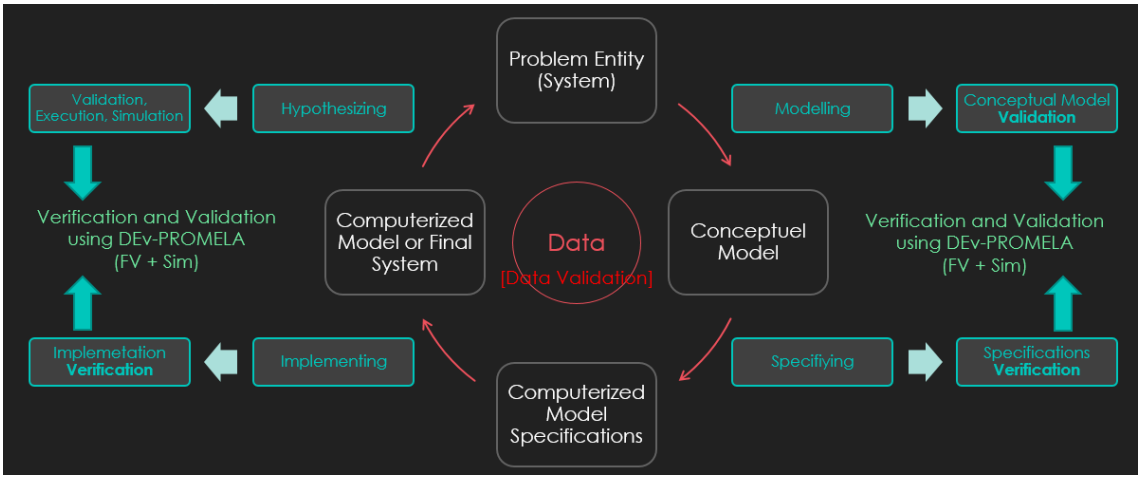


Figure 3.4: V&V of Simulation Models using DEv-PROMELA.

In our workflow, we use also DEv-PROMELA for V&V of Simulation Models (figure 3.4). In the case of Software V&V, the problem entity is the software which will be developed. Data and conceptual model are created during the Requirement Phase using DEv-PROMELA. Then, the Conceptual Model is validated using the prototype. The specifications of the computerized model are verified using traditional methods (as introduced in Chapter 1), but also using DEv-PROMELA formal verification and simulation. Readers can ask a question: "How can I verify and validate a conceptual model using both model-checking and simulation, whereas I am building the simulator that helps me to perform the combined V&V introduced in this work ?"

The answer is simple: by using another existing simulator. Yes, the purpose of this part is building a simulator corresponding to the conceptual model and which will help to perform the combined V&V. However, if another simulator already exists, it can be used in order to verify and validate the simulator we are building using the combined V&V approach.

Implementation verification can be done using theorem proving or also using the combined method introduced in this work. Indeed, DEv-PROMELA is a syntactic model, then MDE can be used in order to generate the source code of the simulator. In this same manner, syntax proof can be used for ensuring that the translation was correctly done. Finally, Validation of the simulator is performed by simulating data. Also, in this step, simulation-based validation can be performed.

This workflow shows an important property: a DEv-PROMELA simulator is a discrete-event software and can be itself verify and validated using our combined methodology. This creates between models strong relations that make

that the simulator is likely the final software. As a consequence, if a DEv-PROMELA simulator is built, verified and validated, writting the conceptual model of any discrete-event software in DEv-PROMELA (or any formalism that combined discrete-event simulation and model-checking) will potentially gives a clean base for the development of the real software.

The architecture of the implementation of the DEv-PROMELA simulator is not discussed in this work, but can be inspired by the event-driven programming paradigm and the object-oriented implementation of DEVS proposed by Zeigler, Kim, and Praehofer [ZKP00].

3.5 Integrated Verification and Validation Environment (IVVE)

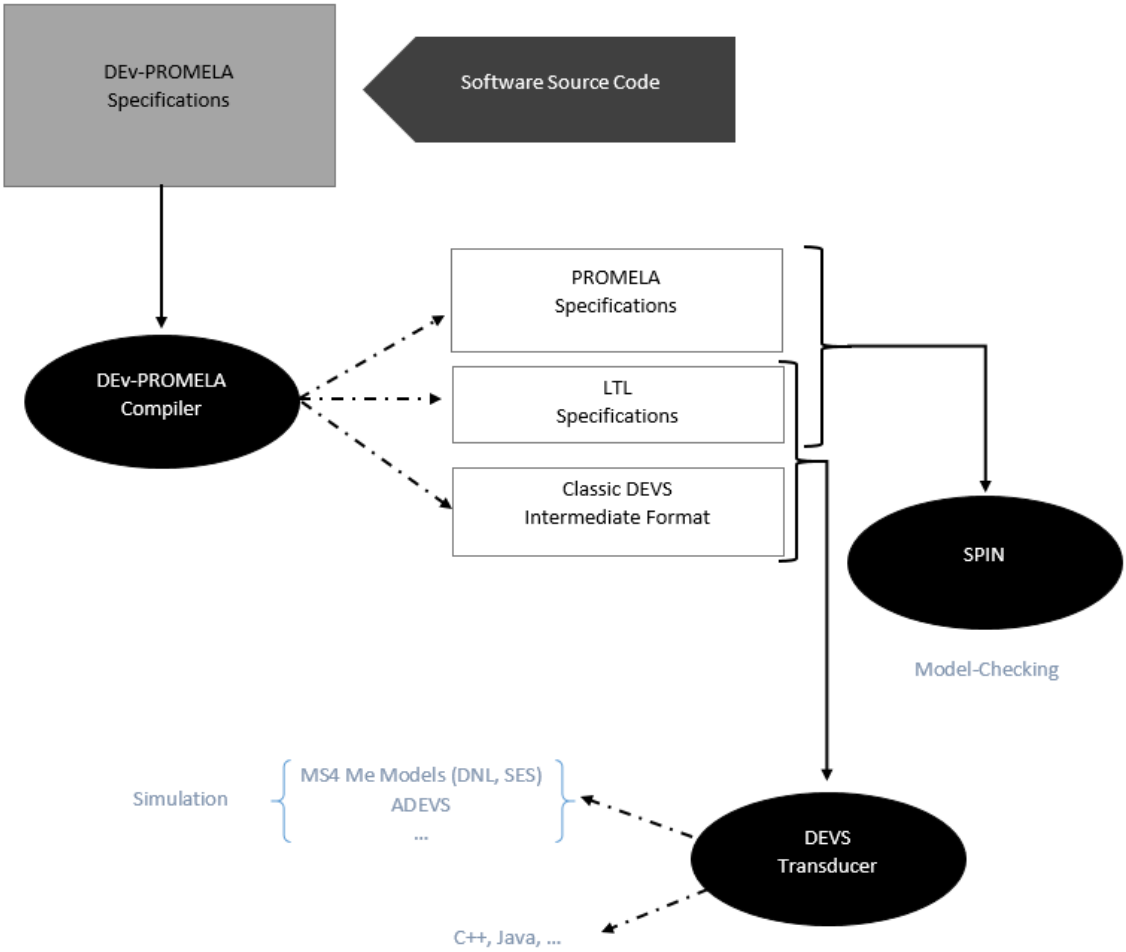


Figure 3.5: Combined V&V Environment Architecture.

In order to understand exactly how works the combined Verification and Validation, we describe the architecture of the environment of Combined V&V (figure 3.5).

First, a DEv-PROMELA model can be extracted from the Software Source Code (in case of Reverse Engineering). The DEv-PROMELA Specifications consists on a text file which is processed by a DEv-PROMELA compiler. This compiler makes a lexical, syntatical and semantical analyzing of the specifications and produces three outputs:

1. the PROMELA specifications corresponding to the equivalent structural model as introduced in Chapter 2;
2. the LTL specifications to verify on the model;
3. A Classic DEVS conceptual model in an intermediate format.

Both the PROMELA model and LTL specifications are sent to the SPIN model-checker which will perform the classic verification. The LTL specifications and the DEVS Intermediate Format are passed to a DEVS transducer that produces the simulation model for the target simulator. The LTL specifications are encoded into DEVS monitor and coupled with the DEVS model. When verification is performed, outputs from the simulation and the model-checking are compared. If both results are different, the environment warns the designer.

The advantage of a such architecture is that it is possible to reuse existing tools without reimplement all the algorithms. Moreover, if the source or the target language changes, only the corresponding components needs to be changed. For example, if we want to use TMS-PROMELA, only replacing the DEv-PROMELA compiler by a TMS-PROMELA compiler is needed.

As an example of environment, we are developing an integrated verification and validation environment for DEv-PROMELA (figure 3.6). This environment embbeds the SPIN model checker, a DEv-PROMELA simulator and code generator which generates the simulation model in a C++ implementation which can be used as base for software implementation. The C++ implementation can be itself verified and validated using traditional V&V approaches, while the V&V of the software (or its conceptual model) is checking using both techniques: it is simulated and formally verified.

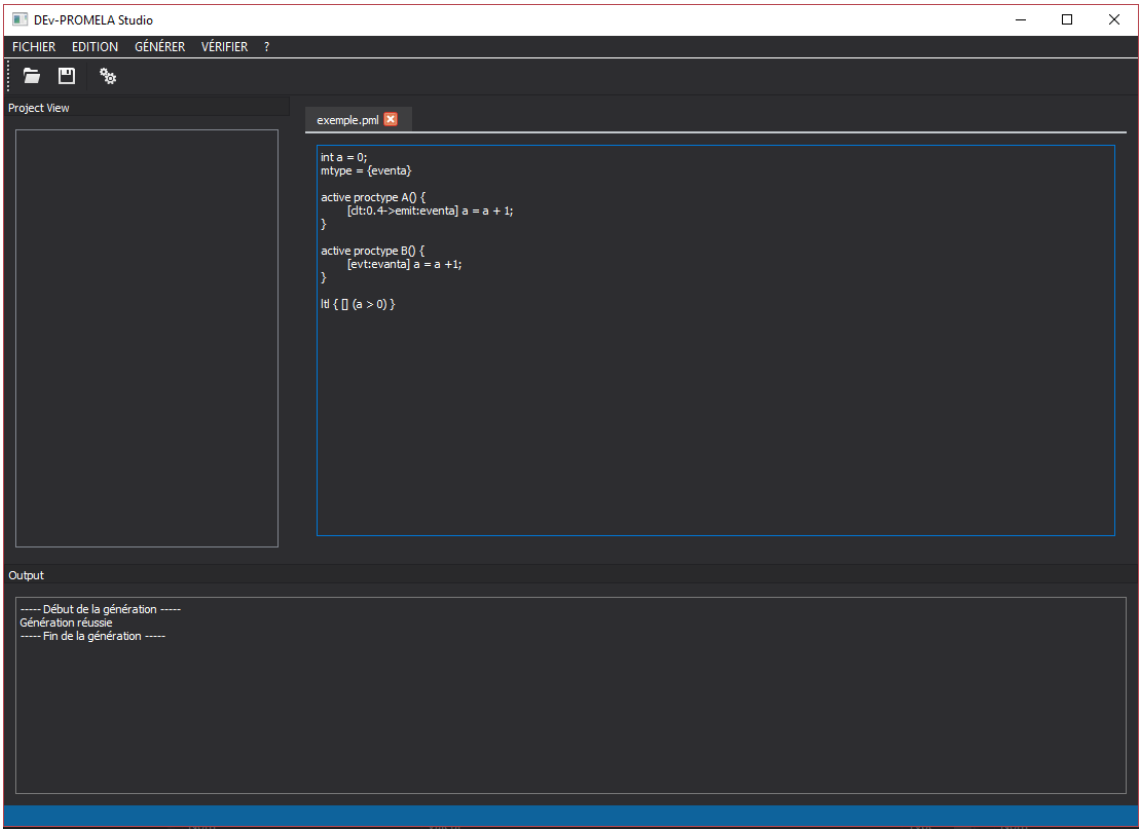


Figure 3.6: The DEv-PROMELA Studio Environment.

3.6 Conclusion

As a conclusion, we saw in this chapter how using DEv-PROMELA in the Verification and Validation procedures. Concerning Verification and Validation of Software, and especially Discrete-Event Software, a DEv-PROMELA model can be used as a conceptual model at each step of the development in order to help designers to ensure the correctness of their design. Combined Verification uses Formal Verification and Simulation on the DEv-PROMELA simulation model. However, because a simulation model is also a computerized model, V&V of software will be correct only if the DEv-PROMELA model is verified and validated. It thus appears a second level of V&V, which is Verification and Validation of Simulation Models. Then, by defining a process for V&V of Simulation models, we see that it is finally the same thing than verify the real modelled software. A kind of reflectivity similar to the work in [Wan06] appears, meaning that a simulation model can be verified using our proposed methodology or using the existing techniques. Moreover, because DEv-PROMELA is a syntactic languague, proofs of correctness between the DEv-PROMELA conceptual model and the DEv-PROMELA simulation model are easier, using formal proofs.

We also introduce an environment architecture for the combined Verification and Validation. This architecture reuses existing simulation and model-checking tools in order to perform the verification using model-checking and simulation, and the validation using model-checking and simulation. Thanks to the properties of DEv-PROMELA, both techniques can be applied to verify structural (static) / behavioural (dynamic) properties on the model without any remodelling.

The next chapter gives concrete examples of case studies, in which our method was applied.

Chapter 4 Applications : Modelling, Verification and Validation of ...

Any physical theory is always provisional, in the sense that it is only a hypothesis: you can never prove it. No matter how many times the results of experiments agree with some theory, you can never be sure that the next time the result will not contradict the theory.
Stephen Hawking

It doesn't matter how beautiful your theory is ...
If it doesn't agree with experiment, it's wrong.
Richard Feynman

4.1 ... Mutual Exclusion Protocols

The first example concerns the well-known Fischer's Mutual Exclusion Protocol [AL94] which is regulary cited in the model-checking litterature. The algorithm (given in Algorithm 7) is easy: firstly, the process p checks whether another process either is already or wants to enter the critical section (line 4). If it is the case, the process stays in an active wait. Then, the process p declares its willing to enter the critical section before entering a sleep mode. When it weaks up, it checks whether another process is entered the critical section during its sleeping. If it is the case, the protocol restarts from the beginning, else the process can enter the critical section.

Fisher's protocol can be seen as a timed system, and also a timed event system. Table 4.1, Program 8 and Program 9 give all variants of this algorithm using each timed extension of PROMELA presented in this work.

ALGORITHM 7: Fischer's Mutual Exclusion Protocol (1984).

```
1: while true do
2: begin
3:    /* non-critical section */
4:    L: if id ≠ 0 then goto L;
5:    id := i;
6:    pause(delay);
7:    if id ≠ i then goto L;
8:    /* critical section */
9:    id := 0;
10: end
```

ALGORITHM 8: Fischer's Mutual Exclusion Protocol in Classic PROMELA.

```
1: int pid = 0;
2:
3: active proctype P ( int id ) {
4: do ::
5: /* non-critical section */
6: wait_L:
7:    if
8:    :: pid != 0 → goto wait_L;
9:    :: else → skip;
10:   fi;
11:   pid = id;
12:   if
13:   :: pid != id → goto wait_L;
14:   :: else → skip;
15:   fi;
16:   /* critical section */
17:   pid = 0;
18: od;
19: }
20:
```

Let's doing the analysis of each version. The Classic PROMELA version is an untimed specification of the Fisher's algorithm. The notion of time explicited line 6 with the `pause` can't be modelled. Thus, the property

"*Two processes cannot be in critical section in same time.*" $(\Box\ \neg(c_i \wedge c_j))$ (1)

is as expected invalid on this model. Indeed, in this case, all possible permutations between n processes are verified without any notions of priority, generating the path given in Figure 4.1. However, this model well verifies the property

"*Is there cases in which two processes cannot be in critical section in same time ?*"
$(\Diamond\ \neg(c_i \wedge c_j))$ (2)

RT-PROMELA	DT-PROMELA	Timed PROMELA

clock y[5];
proctype P(byte id) {
do::
reset{y[id]} X == 0 →
when{y[id] < deltaB}
reset{y[id]} X = id+1 →
atomic{when{y[id] > deltaC}
X == id+1; in_crit++; } →
atomic{X = 0; in_crit--;}
od
}
	proctype P(byte id) {
	timer y, y1;
	do::
	udelay(y);
	X == 0 →
	atomic{
	bdelay(y, deltaB, y1)} →
	X = id+1;
	}
	atomic{delay(y,deltaC);
	udelay(y)} →
	atomic{
	X == id+1;
	in_crit++;
	} → udelay(y);
	atomic{X = 0; in_crit--;}
	od
	}
		proctype P(byte id) {
		do::
		X == 0; →
		if
		:: true → X = id+1;
		:: [timeout deltaB] → skip;
		fi;
		[wait deltaC] X == id+1 →
		in_crit++; →
		atomic { in_crit--; X = 0; }
		}

Table 4.1: Fisher's Mutual Exclusion Protocol in different timed extensions of PROMELA [NJJ08].

ensuring that there are cases in which the Fisher's algorithm is able to guarantee the property of non-concurrency in critical section.

Now, take the RT-PROMELA or the T-PROMELA version: time representation of the sleeping phase is given as a lower bounding value. Firstly, the first condition can be checked at any time. Then, all next instructions are timed. Affectation of X must overcome in a finite delay (expressed by the upper bound $deltaB$), and the pause takes at least *delay* units of time (modelled by the lower bound $deltaC$). That means that the second test is not effective at $t = deltaC$, but at some time after this delay. By this, we mean that a process can be indefinitely blocked, which is not something expressed by the algorithm (pause necessarily means to wake). Verification is thus made on irrelevant paths and implies an explosion of the statespace which would be avoidable. But, take the case in which the condition on the lower bound $deltaC$ is expressed as an equality instead of an inequality. In that way, the infinite behaviour is not expressed anymore. And the result of the verification is the same: the Fischer's algorithm guarantees the property (1) iff $deltaB < deltaC$ (otherwise, for instance if $deltaB = 5$ and $deltaC = 1$, the invalid path given in Figure 4.1 can be generated). However, when $deltaB >= deltaC$, RT-PROMELA is able only to ensure the correctness of the property (2). RT-PROMELA allowed thus getting more precision about the conditions under which the Fischer's protocol was valid, but without any

ALGORITHM 9: Fischer's Mutual Exclusion Protocol in DEv-PROMELA.

```
 1: int pid = 0;
 2: mtype = { changepid }
 3:
 4: [ priority = id ]
 5: active proctype P ( int id ) {
 6: real delay = 2.0 * id;
 7: do ::
 8: /* non-critical section */
 9: wait_L:
10:    if
11:    :: [clt:0.1 → emit:silent] pid != 0 → goto wait_L;
12:    :: [clt:0.1 → emit:silent] else → skip;
13:    fi;
14:    [clt:0.1 → emit:changepid] pid = id;
15:    if
16:    :: [clt:delay → emit:silent | evt: changepid]
17:       pid != id → goto wait_L;
18:    :: [clt:delay → emit:silent] else → skip;
19:    fi;
20:    /* critical section */
21:    [clt:0.1 → emit:changepid] pid = 0;
22: od;
23: }
24:
```

precision on the invalid cases.

Concerning the DT-PROMELA version, the specification focus on the second test can overcome in $[delay; delay + 1]$. There is also no notion of ordered events in this interval. In this case, if we take any process P with the same delay, DT-PROMELA generates the same statespace as the Classic PROMELA one (because all of the processes can execute the second condition at $[delay; delay + 1]$, so the resulting graph is composed by all of the possible permutations of instructions).

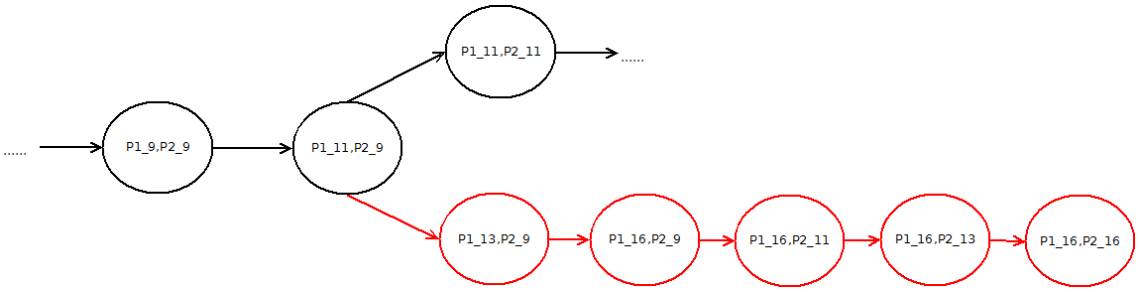


Figure 4.1: Example of an invalid path generated by the Program 8 for the property (1). Pi_j indicates the current line executed by the process P_i.

DEv-PROMELA offers a discrete-event interpretation of the Fischer's protocol. Because all instructions correspond to autonomous behaviours, we consider they depends only on the current state. Because of priority, we know that each process will have the hand in turn. When a process is sleeping, it is blocked until the delay is exactly expired. In other words, for a process p, the next event will be at $getCurrentDate() + delay$ after the execution of line 14. But if the global value of `pid` is changed, we know that we must return to the beginning. This corresponds to the receiving of an event `changepid`. If two processes wakes up at the same time, *priority* will decide of the next proceed event. By this, the execution of this DEv-PROMELA program is deterministic, and only one path of the total graph will be possible for the same configuration. Thanks to structural preservation, the DEv-PROMELA model guarantees the correctness of the property (2) (thanks to the formal verification on the Classic PROMELA equivalent program), but performance and real behaviour analysis are made through simulation. For instance, the DEv-PROMELA model ensures that the Fischer's algorithm verifies the property (1) if all processes are symmetrical (i.e their execution occurs in the same way), regardless the relation between $deltaB$ and $deltaC$. Simulation-based verification also provides a great advantage: checking of irrelevant paths is avoided. For instance, while the path given in Figure 4.1 was verified in Classic PROMELA and RT-PROMELA, it simply does not exist in DEv-PROMELA.

For this example, we verify the property

"Is always at most one process in critical section in same time ?"

using only model-checking (with RT-SPIN) and using DEv-PROMELA (using SPIN for Model Checking and MS4 Me environment – figure 4.2 – for simulation [Yac+15]). For that, the DEv-PROMELA specifications is encoded in a DNL Model (figure 4.3), while a PROMELA abstract model is extracted from the DEv-PROMELA model. Results are given in Table 4.2. We can easily see that timed verification generates a statespace nineteen times bigger for a simple algorithm, while memory usage is multiplied by two. The results show also that as expected SPIN was unable to prove the non-violation of the property. Indeed, in

that case, the property was not a structural property but a behavioural property. Simulation is thus needed to confirm the model-checking result.

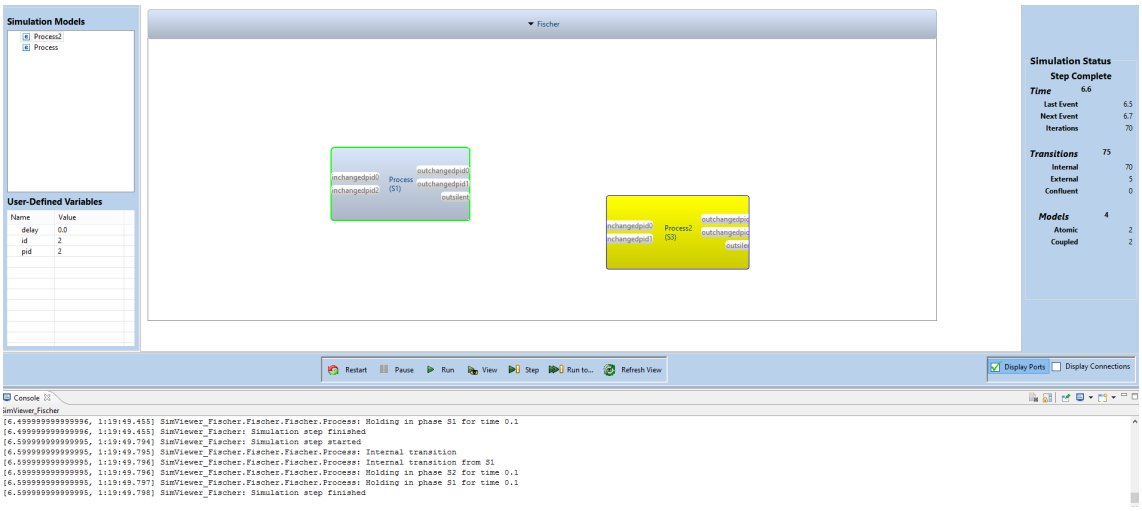


Figure 4.2: MS4 Me Environment.

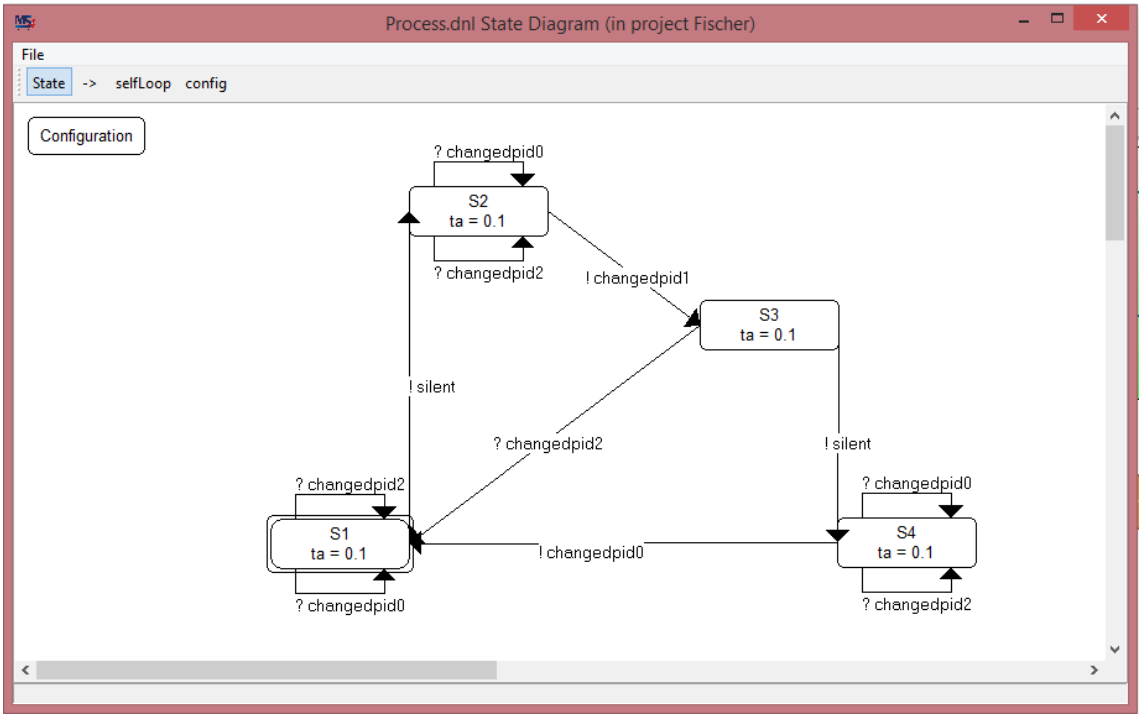


Figure 4.3: DNL model of a Process.

Table 4.2: Results of Verification of concurrent access property.

	RT-PROMELA	DEv-PROMELA
State Space	4632	159 (PROMELA)
Returning Path	1001	128 (PROMELA)
Visited Transitions	5633	287 (PROMELA)
Memory	428 Ko	289 Ko
Results	Violation not found	Violation found (model-checking) - Simulation confirms the inexistance of cases in which the property is not verified (153 played scenarios)

4.2 ... A Video Game Software : PACMAN

Pacman is a well-known and well-developed video game, released on many platforms since 1980. Its source code has been experienced for many years, and a great number of bugs have been fixed, but it is a good example to illustrate the use of DEv-PROMELA in the software development process, as Pacman was used to illustrate the benefits of formal specifications [Gor15] or graph transformations for the evaluation of the playability of a game through Simulation [SV08; SV13b]. This example was already tackled in [YHF16b]. However, we will illustrate here how using the Software V&V Approach introduced in Chapter 3. Thus, we will follow the workflow step-by-step and explain what we are exactly doing at each step.

Note that a DEv-PROMELA model in this section is any models that contains DEv-PROMELA specifications. We mean that a multicomponent multiformalism simulation model that contains a DEv-PROMELA component is considered as a DEv-PROMELA model.

4.2.1 Requirement Phase

Take a Pacman game with relatively simple specifications expressed in an informal language:

- A human player plays a character called Pacman (in yellow in the figure 4.4);
- Ghosts are controlled by a computer artificial intelligence (AI);

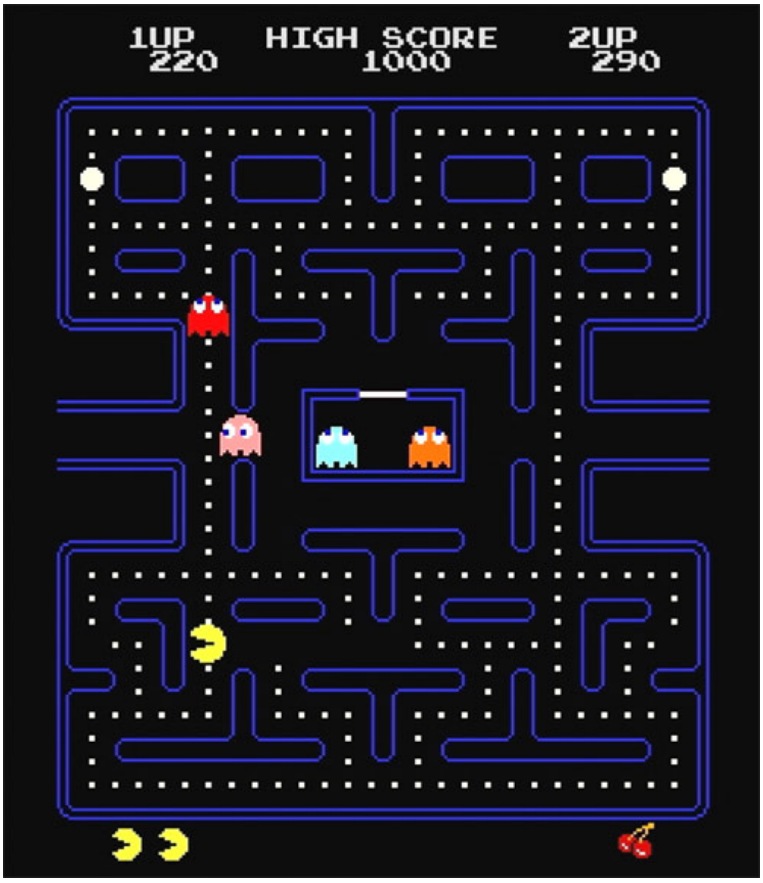


Figure 4.4: Example of Pacman.

- Pacman can move in the labyrinth in vertical and horizontal direction, or stay in place; Pacman cannot walk through the walls;
- Pacman's speed depends on the duration of pressure on the controller;
- Pacman eats balls; each ball gives points to the player;
- If a ghost eats Pacman (i.e. Pacman and a ghost are in the same place), the player loses a life; if the player loses their three lives, the game is over.

Acceptance tests and playable scenarios can then be:

1. Test that the pacman can move when the player controls it. For that, push the pad controller to up, down, left and right and look at what happens.
2. Test that the ghost can move only when the AI take a decision.
3. Push the controller in up-left and test the pacman cannot move in diagonal direction.

4. Test what happens when Pacman goes on a ball.

5. etc.

We note also that the next state of a video game is rarely independent from the time elapsed in a previous state. A game is reactive and for the sake of realism, the time must be taken into account. Traditionally, a game is implemented using a game loop which runs until the end game condition is reached. At each loop, entities like pacman and ghost are updated according to the time elapsed in the loop (the time needed in order to compute a frame). However, taking into account the time in a traditional model-checking process could lead to an inefficient verification. That is why a verification model will generally simply ignore the time. However, discrete-event models represent good abstraction (especially when a game is implemented using a event-driven architecture). For this reason, we can use DEv-PROMELA.

At this step, a first prototype of the game can be developed. This prototype can be a simulation model built from a DEv-PROMELA specifications or any other simulation model, or a fast prototype (figure 4.4) built or not from a DEv-PROMELA specification. The DEv-PROMELA model can be also built by modelling the prototype, the simulator or other. Validation is then performs:

- By executing the prototype. That helps to know the real bounds, datas, etc.
- By simulating the simulation model and comparing results with the prototype (if the prototype is not the simulator itself).

4.2.2 Requirement Analysis

Specifications are formalized at this step, and LTL properties are defined. For instance:

> `If p is the pacman at position x,y and b a ball at position x,y then: never p and b.`
>
> `If p is the pacman at position x,y and g a ghost at position x,y and l, the life of the player, then :` $p \wedge g \implies l\text{-}1$.

The DEv-PROMELA model is refined by introducing the LTL properties, while static verification are performed on the document to ensure they respect the requirements defined in the project management.

4.2.3 High Level Design

At this step, the first game models are designed and focus on interaction between the components of the game. UML Components Diagrams are used and development paradigm are chosen. Traditionnaly, video games are rarely independent

from time, and event-driven paradigm or time-driven paradigm are chosen. Indeed, the components of a game respond to event coming from the players. If event-driven programming paradigm is chosen, then making a DEv-PROMELA model from the UML components diagram is easier. For instance, we could have the structure illustrated in figure 4.5.

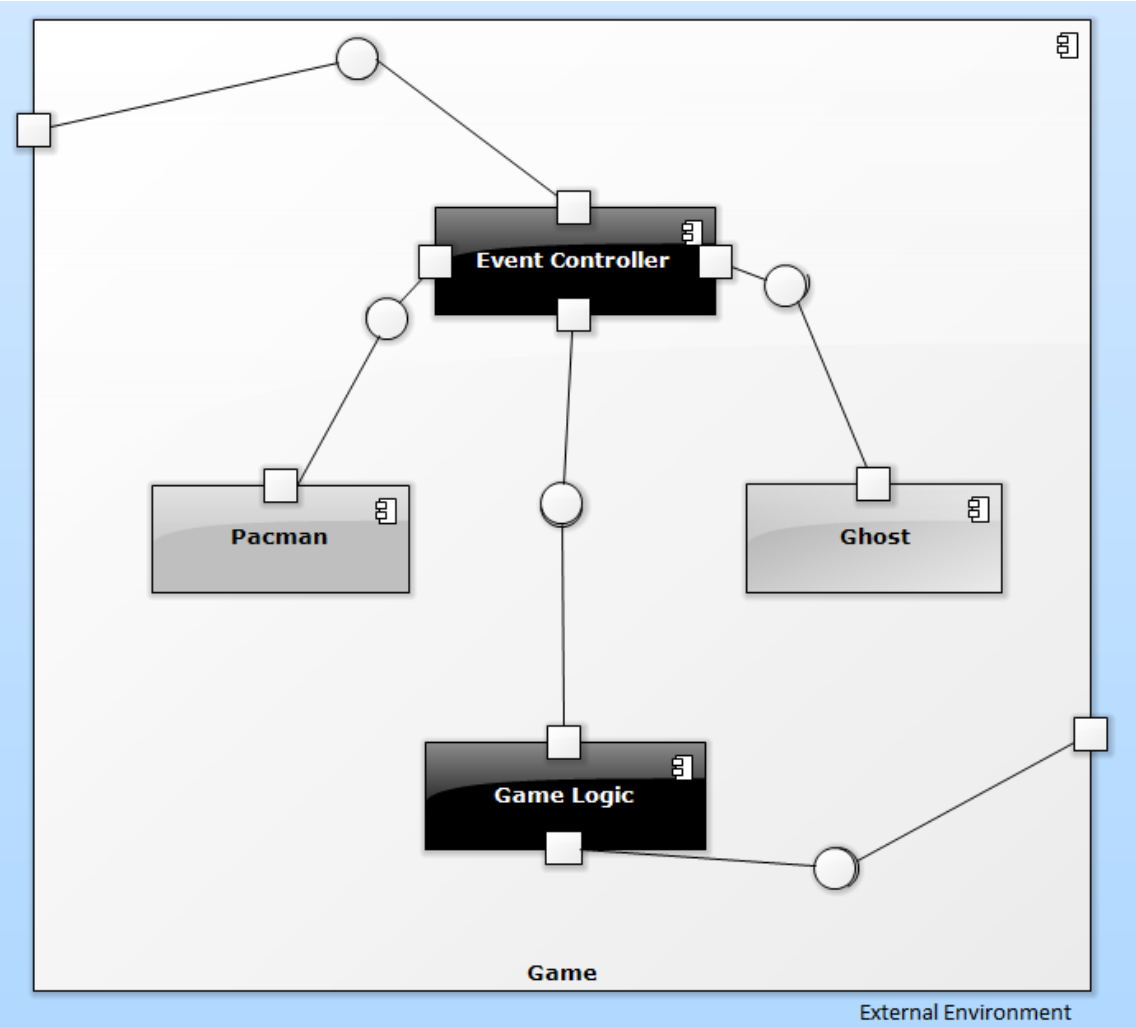


Figure 4.5: Components Model of Pacman.

Then, the DEv-PROMELA model is refined to make appearing the component structure (one process per box), and defining input/output events between boxes. The model can be structurally verified using model-checking and simulation against the requirements defined in the previous phase, and compared with the UML model, using for example syntactic proof. New LTL properties are also introduced, representing the constraints on the input/output event couples.

4.2.4 Low Level Design

Internal FSM structures are defined at this step and the boxes are filled. Pacman and ghosts can be seen as asynchronous processes because they can move independently. Score, lives and positions can be modelled by number. For convenience purposes, we will focus on an example with one pacman and one ghost, as given in Algorithms 10 and 11. The DEv-PROMELA model can also be completed with UML diagrams. LTL propeties on bounds and limits are added for the possible invariants.

Modelling the Pacman Algoritm 10 shows the DEv-PROMELA model of the Pacman. Pacman is seen as a process. Each 0.1 unit of time (line 9), this process checks if the game is finished. If it is not the case, it waits an event from the player (the all `PAC_*` event), and updates its position according to the time elpased between the first check and the occurence of the given event. If the pacman and the ghost are in the same position, the game is over. Furthermore, the discrete-event nature of the game can be seen through two elements. Firstly, we can consider that each instruction is performed after a delay, which corresponds to the time needed by the processor before performing the next instruction. In other words, this is the time before that the video game changes its state to another state by performing the next instruction. Thus, this delay corresponds to the lifespan of the current state. This is modelled by the constant 0.1 (line 9) for instance. The designer can change this constant by any value, or any variable or any function which can be evaluated as a real. When the lifespan of its current state expires, a process can emit an event to the others processes in order to notify them its change. This is done through the `emit` instruction.

Secondly, if a process can emit an event, it can also consume it. Thus, a process must react to an event which comes from another process or from the outside, for instance when the player press a button of its controller. An example of this type of external event is given in line 11. The assignment is done only if the event `PAC_LEFT` is received. Furthermore, DEv-PROMELA allows to combine a `clt` clause and with an `evt` clause. In this case, if the external event is received before the end of the delay, the internal transition is preempted by the external transition. DEv-PROMELA also allows mixing clauses in the selection constructs.

Modelling the Ghost Ghost is just the symetrical of Pacman as shown in Algoritm 11. Each 0.1 unit of time (line 9), this process checks if the game is finished. If it is not the case, it waits an event from the computer (all the `GHS_*` events), and updates its position according to the time elpased between the first check and the occurence of the event. If the pacman and the ghost are in the same position, the game is over.

ALGORITHM 10: DEv-PROMELA model of Pacman Process.

```
 1: real pacman_x, pacman_y, opacman_x, opacman_y;
 2: bool end_game = false;
 3:
 4: [ priority = 1 ]
 5: active proctype PACMAN()
 6: {
 7:     real t = 1;
 8:     do
 9:     :: [clt:0.1→emit:silent] end_game == false →
10:            if
11:            :: [evt:PAC_LEFT] t = getElapsedTime;
12:                if
13:               :: atomic { (end_game == false && pacman_x < 5 && pacman_x
   > 0) →
14:                  opacman_x = pacman_x;
15:                  opacman_y = pacman_y;
16:                  pacman_x = pacman_x - t*1
17:                 }
18:                :: else → skip;
19:                fi;
20:            :: [evt:PAC_RIGHT] t = getElapsedTime;
21:                if
22:               :: atomic { (end_game == false && pacman_x < 5 && pacman_x
   > 0) →
23:                  opacman_x = pacman_x;
24:                  opacman_y = pacman_y;
25:                  pacman_x = pacman_x + t*1
26:                 }
27:                :: else → skip;
28:                fi;
29:                // Stuff and other code ;
30:            fi;
31:            if
32:            :: atomic { (pacman_x == ghost_x) && (pacman_y == ghost_y) →
33:                end_game = true
34:              };
35:            :: else → skip;
36:            fi;
37:     :: end_game == true → break;
38:     od;
39: }
```

ALGORITHM 11: DEv-PROMELA model of Ghost Process.

```
 1: real ghost_x, ghost_y, oghost_x, oghost_y;
 2:
 3: [ priority = 2 ]
 4: active proctype GHOST()
 5: {
 6:     real t = 1;
 7:     do
 8:     :: [clt:0.1→emit:silent] end_game == false →
 9:           if
10:           :: [evt:GHS_LEFT] t = getElapsedTime;
11:                if
12:               :: atomic { (end_game == false && ghost_x < 5 && ghost_x >
   0) →
13:                  oghost_x = ghost_x;
14:                  oghost_y = ghost_y;
15:                  ghost_x = ghost_x - t*1
16:                 }
17:                :: else → skip;
18:                fi;
19:           :: [evt:GHS_RIGHT] t = getElapsedTime;
20:                if
21:               :: atomic { (end_game == false && ghost_x < 5 && ghost_x >
   0) →
22:                  oghost_x = ghost_x;
23:                  oghost_y = ghost_y;
24:                  ghost_x = ghost_x + t*1
25:                 }
26:                :: else → skip;
27:                fi;
28:                // Stuff and other code ;
29:           fi;
30:           if
31:           :: atomic { (pacman_x == ghost_x) && (pacman_y == ghost_y) →
32:               end_game = true
33:              };
34:           :: else → skip;
35:           fi;
36:     :: end_game == true → break;
37:     od;
38: }
```

4.2.5 Verification using Model Checking

The pacman and the ghost processes check the state of the game after 0.1 units of time. If the game is running, then they wait for an event to move. In this model, if the pacman and the ghost are in the same position, the game is over. Now that the DEv-PROMELA model is designed, we can use it to verify some properties:

1. Is eventually the game over ?
2. Is it possible that the pacman goes out of the limit of the board ?
3. Is it possible that the pacman and the ghost intersect without the game being over ?
4. Can the life of the player be a negative number ?
5. Can any process always progress ?
6. Is there any unreachable state ?
7. Can the score be a negative number ?

The PROMELA model is obtained by removing the event descriptor and making an abstraction of real values, using the rules described before. Then, the three properties are encoded into LTL:

1. (◇ `end_game`) : "Eventually the game is over". If this property is verified by the model-checking, that means there is a great chance that the DEv-PROMELA model also ends at time t;
2. (□(`p_x >= 0 && p_x <= 5 && g_x >= 0 && g_x <= 5 && p_y >= 0 && p_y <= 5 && g_y >= 0 && g_y <= 5`)) : "Always the positions of pacman and ghost are inside the limit of the labyrinth";
3. (□ !(`end_game && ((og_x < op_x && g_x > p_x) || (og_x > op_x && g_x < p_x) || (og_y < op_y && g_y > p_y) || (og_y > op_y && g_y < p_y))`)) : "It is never possible that the game is over and the distance between the previous pacman and ghost positions is greater than the distance between the new positions".

Then, the model-checker will explore all the statespace, meaning all the executable paths will be checked against these properties. Because each executable path in the PROMELA model is an abstraction of a DEv-PROMELA model, we know that if a property is not true in the PROMELA model, it will not be true in the DEv-PROMELA model. Concerning the first properties, the SPIN model checker says that the model satisfies the property. That means the model has no

structural deadlock and the game can be ended. The second property is always verified, meaning the ghost and the pacman will always stay in the labyrinth. The third property is also verified by the model according to SPIN, meaning the pacman and the ghost cannot cross each other without the game being over. The other properties are also verified by the model.

4.2.6 Verification using Simulation

The DEv-PROMELA model is then encoded into a DEVS atomic model, using the morphism described in the Chapter 2. The state of the DEVS model is composed of a tuple of the values of local and global variables of both processes, and the line of the next executed instruction. The initial state is for example

$$\begin{gathered} s_0 = (t_p = 1, \\ t_g = 1, \\ l_p = 8, \\ l_g = 8, \\ px, py, opx, opy, \\ gx, gy, ogx, ogy, \\ end\_game = false) \end{gathered}$$

Then, line 9 of each process describes an internal transition after 0.1 unit of time. This indicates that both internal events will occur at the date $t = 0.1$. Thus, we can deduce that

$$ta(s_0) = 0.1$$

and because `PACMAN` has priority over `GHOST`, we can define δ_{int}, s_1 and λ such that

$$\begin{gathered} \lambda(s_0) = \epsilon, \delta_{int}(s_0) = s_1 \\ s_1 = (t_p = 1, \\ t_g = 1, \\ l_p = 9, \\ l_g = 8, \\ px, py, opx, opy, \\ gx, gy, ogx, ogy, \\ end\_game = false) \end{gathered}$$

And $ta(s_1) = min(\infty, 0.1 - 0.1) = 0$. Indeed, the date of the next event in the process PACMAN is ∞ (the next executable instruction depends only on an external event), and the date of the next event in the process GHOST is given by

Table 4.3: Comparison between Results of Verification using Model Checking and Simulation.

	Formal Verification	Simulation
Property 1	At least one path	At least one scenario
Property 2	Impossible	Possible
Property 3	Impossible	Possible
Property 4	Always positive	Positive
Property 5	Always progress	Not in all cases
Property 6	No unreachable state	Unreachable states can exist
Property 7	Always positive	Positive

substracting the elapsed time from the duration given in line 8. Thus,

$$\lambda(s_1) = \epsilon, \delta_{int}(s_1) = s_2$$

and

$$\begin{gathered} s_2 = (t_p = 1, \\ t_g = 1, \\ l_p = 9, \\ l_g = 9, \\ px, py, opx, opy, \\ gx, gy, ogx, ogy, \\ end\_game = false) \end{gathered}$$

And $ta(s_2) = min(\infty, \infty) = \infty$. Because the next event depends only on an external event, we have:

$$\begin{aligned} \delta_{ext}(s2, PAC\_LEFT) &= s3 \\ \delta_{ext}(s2, PAC\_RIGHT) &= s4 \\ \delta_{ext}(s2, GHS\_LEFT) &= s5 \\ \delta_{ext}(s2, GHS\_RIGHT) &= s6 \end{aligned}$$

Following the same reasoning, we build the DEVS atomic model corresponding to the DEv-PROMELA model and we simulate it.

External events are injected into the model during the simulation to allow pac-man and ghost to be moved. The lifespans are also changed to see the difference between different instances of the model. Multiple simulations are done and outputs are then verified to check if the properties are verified. The results are different, as shown in Table 4.3.

The first property is true because we found at least one scenario for which the

game finishes. The second and third properties are not verified. Indeed, the DEv-PROMELA model, in simulation mode, takes into account the time elapsed in the loop to describe the moves of the entity, while in model-checking mode, moves can only evolve along integer values. Intuitively, the model-checking model acts as if the labyrinth were a grid and entites can only move from a case to an adjacent case, as shown in Figure 4.6.

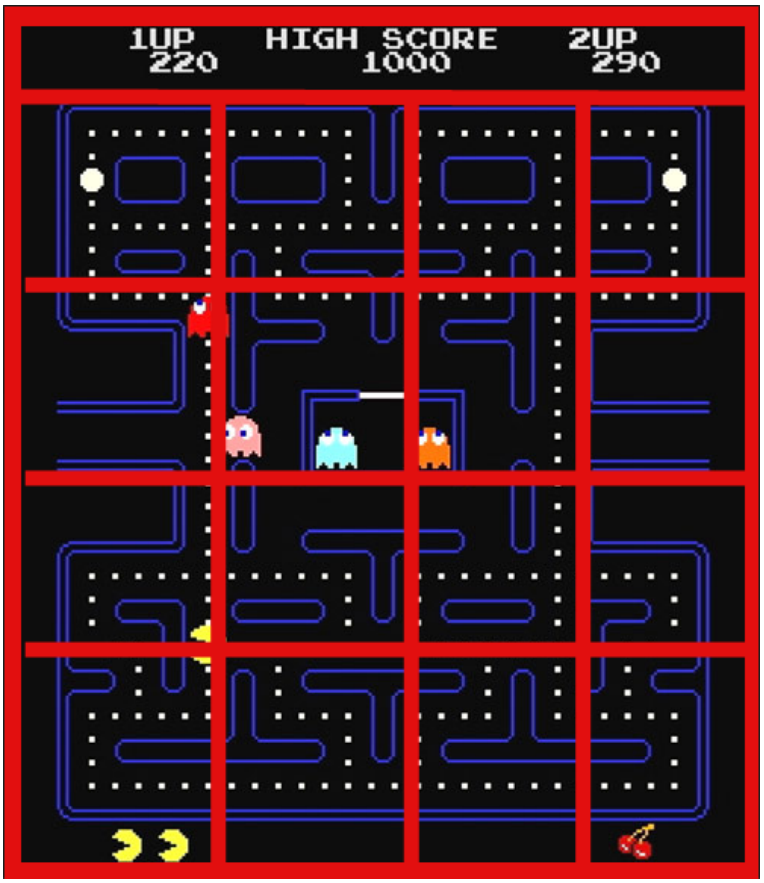


Figure 4.6: Representation of the PROMELA model for Pacman.

Yet, in simulation mode, moves can evolve along real values, using any distribution. The condition that limits the position in l.12 and 22 is not enough. Nevertheless, the designer must not necessarily conclude that the model is incorrect: in the case when the implementation uses integers to represent positions, the property is satisfied. The same reasoning can be done about property 3. Replacing this condition by a comparison between the previous positions and the new positions ensures the correctness of the model in all cases. In this previous example, we see that verification using simulation partially invalidates the results of model checking and gives another interpretation of the model checking results. Finally, generating the source code from the DEv-PROMELA model increases the confidence we can put into the game about the correctness of these

three properties.

Properties 4 and 7 are a time-independent property, thus simulation confirms the model-checking results.

Concerning properties 5 and 6, the simulation gives another interpretation. Indeed, model-checking confirms there is no deadlock induced by the structure, but if the external events `GHS_*` and `PAC_*` are never received, the system is blocked. In this case, the possible deadlock can come from the absence of event. In this example, this assumption is trivial and easy to understand, but in an extended model involving a lot of communication, this kind of problem can be tedious to understand. The other advantage of DEv-PROMELA is the capability to model communication between processes without necessarily using channels and explicit synchronization. At each step, each transition can emit or consume a message. Thus, channels can be reserved to model specific communication protocols.

4.2.7 Coding

At this step, the simulation model can be used as based for the final code of the software. In this case, software verification can be done through formal proofs. The code is completed with other modelled elements (figure 4.7), while the DEv-PROMELA model can be refined by adding some computing aspects. A more generic simulation model can be also built by integrating the DEv-PROMELA model into a multiformalism model. A model can be extracted from the code and verified/validated using a Combined Approach, and compared with the results given by the existing DEv-PROMELA model (if this model have not been used for generating the source code).

4.2.8 Validation using Simulation

Now we have the source code, validation is performed by both executing the software, tests, simulation and model-checking. When going up along the right side of the "V", tests results are compared with both execution and simulation. If one test failed, designers can deduce that something was wrong.

4.2.9 What about V&V of simulation model ?

Indeed, readers can see that we have use a conceptual model to perform V&V during all the workflow. However, building the conceptual model using the methodology ensures its correctness. Then, if the simulator used was already verified and validated, and if there is a proof that it is possible to construct a correct simulation model from the DEv-PROMELA specification, then it is possible to use both formal verification/validation and simulation-based verification/validation on the conceptual model to prove properties on the real software.

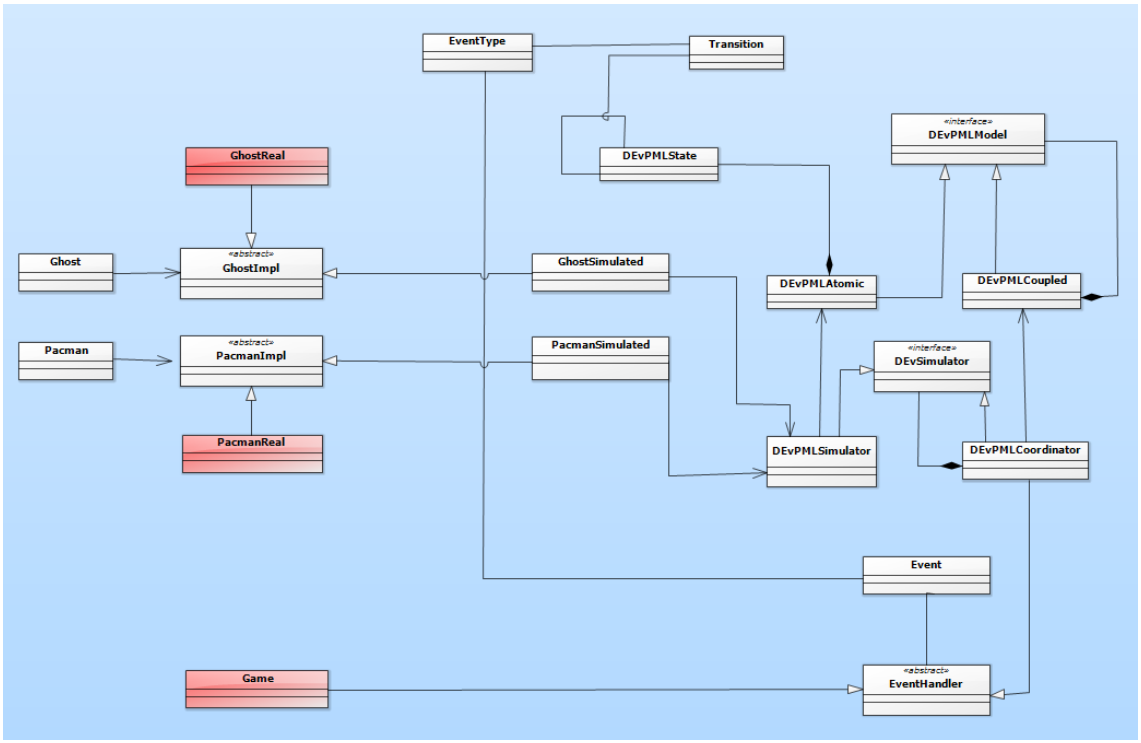


Figure 4.7: Example of Pacman UML Diagram - Red classes are real software classes, while white classes are simulator classes.

Note that in this example we have used a top-down approach, meaning the code could be automatically generated from the DEv-PROMELA model. If the DEv-PROMELA simulator is implemented in an object-oriented manner, getting the final game from the simulation model is relatively easy. However, this is possible only because we have chosen an event-driven paradigm. A bottom-up approach, meaning building the model from the source code, is also possible. The verification and validation are thus performed by making successive abstractions.

4.3 ... A Manufacture Chain : ST Microelectronics' Case Study

4.3.1 The Problem

Manufacture problems were classical and already intensively studied in the litterature [DY00; LMN12; TER14]. This problem consists on a set of thousand processes which are executed on several batches of products. At each stage of the production, a controller performs some operations which take an amount of time.

When an operation is finished, a signal is sent to the controller which leads the batch to the next operation. Operations are just encoded with Bash-like

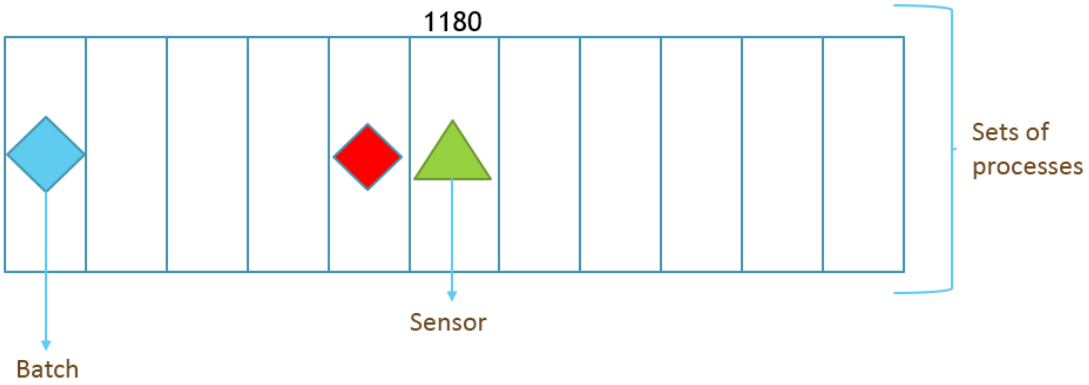


Figure 4.8: The Manufacture Chain. Each square represent a step or a process/-operation.

instructions, like shown in Figure 4.9. In order to show the potentiel of our proposed methodology, we model a chain with only two operations and one sensor, successively using PROMELA, Real-Time PROMELA [TC96] and DEv-PROMELA. Then, we check if there is any unreachable state and that, in any cases, the attribute `batch_attr1615` is equal to 0 at end of the manufacturing process when attribute 1001 is equal to 0. While the PROMELA and RT-PROMELA models are checked using model-checking, the DEv-PROMELA model is verified using model-checking and discrete-event simulation.

```
REWORK
BEGIN
START
LOG EVENT PRPMARKLAS FOR LASER
IF ATTRIBUTE (1001) OF LOT EQ LAS01 THEN GO TO 8511
LOG EVENT PRMCHECWID FOR MEASURE
IF ATTRIBUTE (1615) OF LOT NE 0 THEN GO TO 9000
LOG EVENT PRMD0LAS01D FOR MEASURE
LOG EVENT MSLD0LAS01D FOR LASER
SET ATTRIBUTE (1615) OF LOT TO 0 NODISPLAY
END
```

Figure 4.9: Example of the operation 1180.

The translation from the Bash instruction to PROMELA is very easy. Each operation is encoded using a PROMELA process.

Batches' attributes are modelled using integers variables `batch_*`. The sensor is modelled using a process that randomly changes the values of attributes.

ALGORITHM 12: PROMELA model of the operation 1180.

```
 1: active proctype op1180 () {
 2:    event!PRPMARKLAS;
 3:    event?code; // Log Event PRPMARKLAS
 4:    if
 5:    :: batch_attr1001 == 0 → goto STEP8511;
 6:    :: else → skip;
 7:    fi;
 8:    event!PRMCHECWID;
 9:    event?code; // Log Event PRMCHECWID
10: STEP8511: if
11:    :: batch_attr1615 ! = 0 → goto STEP9000;
12:    :: else → skip;
13:    fi;
14:    event!PRMD0LAS01D;
15:    event?code; // Log Event PRMD0LAS01D
16:    event!MSLD0LAS01D;
17:    event?code; // Log Event MSLD0LAS01D
18: STEP9000: batch_attr1615 = 0;
19: ...
20: }
```

Communications between sensors and processes are modelled using channels. With PROMELA, quantitative delays are totally abstracted, because they cannot be modelled, while RT-PROMELA allows modelling of time using clock, as shown in Algorithm 14.

The DEv-PROMELA model (Algorithm 13) is slightly different. Because DEv-PROMELA is based on DEVS, the event mechanism is directly expressed by the transitions, and each event are well-dated. Moreover, even the sensor can be also modelled using DEv-PROMELA, we decided to use a classic DEVS model to represent it. Then, the model-checking is only applied of the model of operations. Then, by coupling the DEv-PROMELA model and the DEVS model of the sensor, we check by simulation the potential deadlocks, while the safety property concerning the attribute 1615 is always checked using model-checking. Results are presented in Table 4.4.

Two subsequent models are generated from the DEv-PROMELA models. The formal verification model is generated using Classic PROMELA by removing the event descriptors, and using the mapping previously defined. The obtained untimed model represents all the possible event permutations, but does not include the time. This level of abstraction is enough to verify structural properties or untimed properties like the one that we want to verify: "Is the attribute 1615 equal to 0 at the end of the manufacturing process ?".

The second model is a DEVS abstract model independant of the target simula-

ALGORITHM 13: DEv-PROMELA model of the operation 1180.

```
1:  active proctype op1180 () {
2:      [clt : 0.28 → emit : PRPMARKLAS]
3:      skip; // Log Event PRPMARKLAS
4:      if
5:      :: [evt : laser] batch_attr1001 == 0 → goto STEP8511;
6:      :: [evt : laser] else → skip;
7:      fi;
8:      [clt : 0.27 → emit : PRMCHECWID];
9:      skip; // Log Event PRMCHECWID
10: STEP8511: if
11: ...
12: }
```

ALGORITHM 14: RT-PROMELA model of the operation 1180.

```
1:  int batch_attr1615;
2:  int batch_attr1001;
3:  chan event = [1] of byte;
4:  byte code;
5:
6:  active proctype op1180 () {
7:      when {y ≤ 28 } reset { y } event!PRPMARKLAS;
8:      event?code; // Log Event PRPMARKLAS
9:      if
10:     :: batch_attr1001 == 0 → goto STEP8511;
11:     :: else → skip;
12:     fi;
13:     event!PRMCHECWID;
14: ...
15: }
```

tor. This model has the same semantics than those of the DEv-PROMELA model. This DEVS abstract model is then implemented in a DEVS simulation model (in this case, in the MS4 Me Environment [Seo+13]) and simulated to check timed properties.

4.3.2 Results

As expected, clocks in RT-PROMELA generate a huge statespace compared with PROMELA and DEv-PROMELA. Indeed, both model generated by PROMELA and DEv-PROMELA for formal verification are untimed models. In the three cases, model-checking was able to detect that the event `PRMCHECWID` is never generated by the model. But, the main difference in this case is about the deadlock found

Table 4.4: Results of chain checking.

	States	Transitions	Time	Memory	Results
PROMELA	315	558	0.2	128653	4 unreachable states, no deadlock
RT-PROMELA	945	1697	0.5	256686	4 unreachable states, no deadlock
DEv-PROMELA	128	353	0.09	100302	2 unreachable states, no deadlock found by model checking, deadlocks found in simulation

by the checkers. In the case of PROMELA and RT-PROMELA, no deadlock were detected in the model, while DEv-PROMELA allows us to detect some deadlocks. In fact, the problem comes from the model of the sensor. The level of abstraction of time used in PROMELA and RT-PROMELA does not allow modelling cases in which the sensor does not send the `laser` event. But, the discrete-event simulation shows that there exists some cases in which this event is never generated. In these cases, the model stays locked, and the operation cannot progress. It is important to note that if simulation could detect the deadlock, it was mainly because scenarios were well-chosen. Indeed, simulation-based verification depends on the played scenarios.

The other important aspect of DEv-PROMELA is that we can combine different discrete-event simulation formalisms for the simulation-based verification. Indeed, as we said previously, even we could model the sensor using DEv-PROMELA, it is possible to use Petri Nets or any others discrete-event simulation formalisms to model the sensor and combine it with the DEv-PROMELA model (thanks to the DEVS Bus concept). However, in this case, model-checking is like a supportive method to the simulation-based verification.

4.4 Conclusion

We have applied DEv-PROMELA in various application, going from the simple mutual exclusion algorithm to a manufacture controller system. In all the cases, formal verification was completed with simulation to detect errors that an untimed model-checking could not detect alone. In all cases, the cost of combined verification and validation seems cheaper than a timed model-checking. However, all the examples show that the results strongly depend on the played scenarios and test cases, and also on requirements. Our V&V approach is thus designed as a complementary technique which must be used only in addition of the existing traditionnal V&V procedures.

The case of the Pacman Video Game shows also in which condition the use of DEv-PROMELA is more natural. If the software is naturally an event-driven program, the use of discrete-event models is easier. In other cases, the DEv-PROMELA focuses only on behavioural aspects which depend on time.

Conclusion

As a conclusion, the work presented in this thesis tries to introduce a new approach for the modelling, verification and validation of discrete-event systems, by combining model-checking and discrete-event simulation. Even these approaches seem to be in opposite, we see that model-checking and simulation have some common points and objective. For example, even if model-checking is a verification method, it is used as a validation technique, while some simulation-based verification techniques were developed. Then, the objective is to improve the confidence put in the results of untimed model-checking by completing verification with simulation. In that way, we can reduce the explosion problem due to the introduction of time in verification models and allows verifying some properties on models which normally cannot be verified. For that, we propose to restrict formal verification on structural and time-invariant properties and use simulation to verify behavioural properties. From that point, we propose to build a new specification language from a verification language by:

1. Introducing discrete-event concepts into the verification formalism;
2. Defining syntactical changes to allow modelling these concepts;
3. Defining a new semantics based on the simulation formalism;
4. Defining morphisms between
   - the new formalism and the verification formalism. The morphism must preserve structural properties;
   - the new formalism and simulation formalism. The morphism must preserve behavioural properties.

Using a simulation pre-order relation, we prove that structural properties verified on the verification model will be true on the combined model, and behavioural properties verified on the simulation model will be true on the combined model. The great benefit is that no more remodelling is needed because the combined model becomes verifiable and simulable (in the discrete-event simulation sense).

As a result of this approach, we build a new specifications language called DEv-PROMELA which is the crossing of PROMELA and Classic DEVS. DEv-PROMELA then appears as a natural way to model discrete-event systems in a syntactic fashion. This point is important because it makes easier and safer transformations from the conceptual model to the simulation model. Moreover, DEv-PROMELA gathers all the advantage of DEVS, including the multicomponent and multiformalism capabilities in simulation mode. Furthermore, a subclass of DEVS can be expressed in DEv-PROMELA, meaning that a subclass of DEVS simulation models can be verified using our combined methodology.

To going further, we also proposed a modified Software V&V Cycle in which a DEv-PROMELA model (or any other combined models) is developed at each step of the Software Development Life Cycle, in addition of the traditional V&V techniques. The DEv-PROMELA model is then used in order to perform complementary verification and validation at each step, and the simulation model can be used as base of the implementation. We find there the advantages of the refinement approach of formal specifications. We also show that there is a strong relation between the conceptual model, the simulation and the software. Indeed, because the software model and the simulation model are both built from the same conceptual model, validation of properties on the one will validate the same property on the other.

We have also shown how this approach allows integration of existing tools in the same environment for the combined V&V. As there exists a morphism between the combined model, the verification model and the simulation model, a transducer can be developped to transform the combined specifications into a model usable by a formal method, or into a model usable by an existing simulator.

Finally, we applied our approach on many examples, from the classic mutual exclusion algorithm, to a video game. This has allowed showing the limits of this approach: the empirical aspect of simulation. Indeed, all the methodology relies on the capacity of the designer to predict test cases and scenarios. If existing techniques can help, there is no guarantee that the all entire statespace is verified. Even, it is possible on a simulation model (i.e. the finite computerized model, but it would take too much time), this is not possible on the symbolic infinite representation.

Future works then must concern how improving and reducing this dependancy to test cases or how using test coverage to generate scenario for simulations. We must also study the real complexity of the generated simulation models. Indeed, we have not perform any performance analysis in this work, but it will

be interested to know if it is possible to reduce the simulation models. More works are also needed to allow modelling of other DEVS subclasses (like Parallel DEVS), and to study a combining approach using theorem proving and discrete-event simulation. Indeed, we do not provide in this work material for integration theorem proving and syntax check between each translations. Using approaches like Z-integration into simulation model are maybe promising to achieve these verifications. Model-Driven Engineering also seems to be a promising way for achieving more integration between all these V&V techniques and procedures.

Bibliography

[AL94] Martin Abadi and Leslie Lamport. “An Old-fashioned Recipe for Real Time”. In: *ACM Trans. Program. Lang. Syst.* 16.5 (Sept. 1994), pp. 1543–1571 (cit. on p. 137).

[Abd+14] Abbas Abdulhameed, Ahmed Hammad, Hassan Mountassir, et al. “An Approach Combining Simulation and Verification for SysML using SystemC and Uppaal”. In: *CAL 2014, 8ème conférence francophone sur les architectures logicielles*. 2014 (cit. on p. 70).

[Abr+04] Alain Abran, JW Moore, P Bourque, et al. “Software Engineering Body of Knowledge”. In: *IEEE Computer Society, Angela Burgess* (2004) (cit. on p. 45).

[Ack71] Russell L Ackoff. “Towards a system of systems concepts”. In: *Management science* 17.11 (1971), pp. 661–671 (cit. on p. 21).

[ABC82] W. Richards Adrion, Martha A. Branstad, and John C. Cherniavsky. “Validation, Verification, and Testing of Computer Software”. In: *ACM Comput. Surv.* 14.2 (1982), pp. 159–192. ISSN: 0360-0300 (cit. on p. 48).

[AQH15] W. Ahmad, U. Qamar, and S. Hassan. “Analyzing different validation and verification techniques for safety critical software systems”. In: *Software Engineering and Service Science (ICSESS), 2015 6th IEEE International Conference on*. Sept. 2015, pp. 367–370 (cit. on p. 47).

[AMT16] Hamzat Olanrewaju Aliyu, Oumar Maïga, and Mamadou Kaba Traoré. “The high level language for system specification: A model-driven approach to systems engineering”. In: *International Journal of Modeling, Simulation, and Scientific Computing* 07.01 (2016) (cit. on p. 71).

[AT16a] H.O. Aliyu and M.K. Traore. “Toward an integrated framework for the simulation, formal analysis and enactment of discrete events systems models”. In: 2016, pp. 3090–3091 (cit. on p. 71).

[AT16b] H.O. Aliyu and M.K. Traoré. “Integrated framework for model-driven systems engineering: A research roadmap”. In: 2016 (cit. on p. 71).

[Alu03] Rajeev Alur. “Formal Analysis of Hierarchical State Machines”. In: *Verification: Theory and Practice: Essays Dedicated to Zohar Manna on the Occasion of His 64th Birthday*. Ed. by Nachum Dershowitz. Springer Berlin Heidelberg, 2003, pp. 42–66 (cit. on p. 93).

[Alu+95] Rajeev Alur, Costas Courcoubetis, Nicolas Halbwachs, et al. “The algorithmic analysis of hybrid systems”. In: *Theoretical computer science* 138.1 (1995), pp. 3–34 (cit. on p. 64).

[Alu+93] Rajeev Alur, Costas Courcoubetis, Thomas A. Henzinger, et al. “Hybrid Systems”. In: ed. by Robert L. Grossman, Anil Nerode, Anders P. Ravn, et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993. Chap. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems, pp. 209–229 (cit. on p. 121).

[AD94] Rajeev Alur and David L. Dill. “A Theory of Timed Automata”. In: *Theoretical Computer Science* 126 (1994), pp. 183–235 (cit. on pp. 62, 63, 118).

[AY98] Rajeev Alur and Mihalis Yannakakis. “Model Checking of Hierarchical State Machines”. In: *Proceedings of the 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. SIGSOFT ’98/FSE-6. Lake Buena Vista, Florida, USA: ACM, 1998, pp. 175–188. ISBN: 1-58113-108-9 (cit. on p. 93).

[And86] Stephen J. Andriole. *Software Validation, Verification, Testing and Documentation: A Source Book*. Princeton, NJ, USA: Petrocelli Books, Inc., 1986. ISBN: 0894332694 (cit. on p. 25).

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008 (cit. on pp. 55, 56, 59–62, 66).

[Bal07] Osman Balci. “Verification, Validation, and Testing”. In: *Handbook of Simulation*. John Wiley & Sons, Inc., 2007, pp. 335–393 (cit. on p. 52).

[Ban98] Jerry Banks. *Handbook of simulation: principles, methodology, advances, applications, and practice*. John Wiley & Sons, 1998 (cit. on p. 31).

[BC86] Jerry Banks and John S. Carson II. “Introduction to Discrete-event Simulation”. In: *Proceedings of the 18th Conference on Winter Simulation*. WSC ’86. Washington, D.C., USA: ACM, 1986, pp. 17–23. ISBN: 0-911801-11-1 (cit. on p. 31).

[Ban+10] Jerry Banks, John S. Carson, Barry L. Nelson, et al. *Discrete-event system simulation*. 5th ed. Prentice Hall, 2010. ISBN: 0-13-606212-1 (cit. on p. 36).

[Bic+97] Juan Bicarregui, Jeremy Dick, Brian Matthews, et al. “Making the most of formal specification through animation, testing and proof”. In: *Science of Computer Programming* 29.1 (1997), pp. 53–78 (cit. on p. 35).

[Bie+12] Pierre Bieber, F Boniol, MARC Boyer, et al. “New Challenges for Future Avionic Architectures.” In: *AerospaceLab* 4 (2012), p–1 (cit. on p. 21).

[Boe81] Barry W. Boehm. *Software Engineering Economics*. 1st. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1981 (cit. on p. 44).

[Boe91] Barry W. Boehm. “Software risk management: principles and practices”. In: *IEEE Software* 8.1 (Jan. 1991), pp. 32–41 (cit. on p. 44).

[Bör05] Egon Börger. “Abstract State Machines: a unifying view of models of computation and of system design frameworks”. In: *Annals of Pure and Applied Logic* 133.1 (2005), pp. 149–171 (cit. on p. 65).

[BD98a] Dragan Bosnacki and Dennis Dams. “Discrete-time Promela and Spin”. English. In: *Formal Techniques in Real-Time and Fault-Tolerant Systems*. Ed. by AndersP. Ravn and Hans Rischel. Vol. 1486. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1998, pp. 307–310. ISBN: 978-3-540-65003-4. DOI: 10.1007/BFb0055359. URL: http://dx.doi.org/10.1007/BFb0055359 (cit. on p. 120).

[BD98b] Dragan Bosnacki and Dennis Dams. “Integrating Real Time Into Spin: A Prototype Implementation”. English. In: *Formal Description Techniques and Protocol Specification, Testing and Verification*. Ed. by Stan Budkowski, Ana Cavalli, and Elie Najm. Vol. 6. The International Federation for Information Processing. Springer US, 1998, pp. 423–438. ISBN: 978-1-4757-5262-5. DOI: 10.1007/978-0-387-35394-4_26. URL: http://dx.doi.org/10.1007/978-0-387-35394-4_26 (cit. on pp. 120, 121).

[BH95] Jonathan P. Bowen and Michael G. Hinchey. *Applications of Formal Methods*. Prentice Hall PTR, 1995 (cit. on p. 55).

[BJ14] Zeigler B.P. and Nutaro J. “Combining DEVS and Model-Checking: Using System Morphisms for Integrating Simulation and Analysis in Model Engineering”. In: *Proceedings of the 26th European Modeling and Simulation Symposium*. 2014, pp. 350–356 (cit. on p. 72).

[Brz03] Janusz A. Brzozowski. “Sequential Machine”. In: *Encyclopedia of Computer Science*. John Wiley and Sons Ltd., 2003, pp. 1564–1569 (cit. on p. 79).

[Büc66] J. Richard Büchi. “Symposium on Decision Problems: On a Decision Method in Restricted Second Order Arithmetic”. In: *Logic, Methodology and Philosophy of ScienceProceeding of the 1960 International Congress*. Ed. by Patrick Suppes Ernest Nagel and Alfred Tarski. Vol. 44. Elsevier, 1966, pp. 1–11 (cit. on p. 57).

[But10] Henning Butz. “Open integrated modular avionic (ima): State of the art and future development road map at airbus deutschland”. In: *Signal* 10 (2010), p. 1000 (cit. on p. 21).

[Car04] John S. Carson II. “Introduction to Modeling and Simulation”. In: *Proceedings of the 36th Conference on Winter Simulation*. WSC ’04. Washington, D.C.: Winter Simulation Conference, 2004, pp. 9–16. ISBN: 0-7803-8786-4 (cit. on p. 34).

[CKT03] Samarjit Chakraborty, Simon Kunzli, and Lothar Thiele. “A General Framework for Analysing System Properties in Platform-Based Embedded System Designs”. In: *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 1*. DATE ’03. IEEE Computer Society, 2003 (cit. on p. 71).

[Cim+00] Alessandro Cimatti, Edmund Clarke, Fausto Giunchiglia, et al. “NUSMV: a new symbolic model checker”. In: *International Journal on Software Tools for Technology Transfer* 2.4 (2000), pp. 410–425 (cit. on p. 68).

[Cla08] Edmund M. Clarke. “The Birth of Model Checking”. In: *25 Years of Model Checking: History, Achievements, Perspectives*. Springer Berlin Heidelberg, 2008, pp. 1–26 (cit. on pp. 55, 59, 62).

[CE82] Edmund M. Clarke and E. Allen Emerson. “Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic”. In: *Logic of Programs, Workshop*. Springer-Verlag, 1982, pp. 52–71 (cit. on pp. 56, 59).

[Col88] James S. Collofello. *Introduction to Software Verification and Validation*. SEI Curriculum Module SEI-CM-13-1.1. University of Arizona, 1988 (cit. on p. 47).

[CC77] Patrick Cousot and Radhia Cousot. “Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints”. In: *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL ’77. Los Angeles, California: ACM, 1977, pp. 238–252 (cit. on p. 28).

[CC10] Patrick Cousot and Radhia Cousot. “A gentle introduction to formal verification of computer systems by abstract interpretation.” In: *Logics and Languages for Reliability and Security*. IOS Press, 2010, pp. 1–29 (cit. on p. 55).

[Cri07] Maximiliano Cristia. “A TLA+ encoding of DEVS models”. In: *Proceedings of International Modeling and Simulation Multiconference (Buenos Aires, Argentina*. 2007, pp. 17–22 (cit. on p. 72).

[Dab+02] Frank Dabek, Nickolai Zeldovich, Frans Kaashoek, et al. “Event-driven Programming for Robust Software”. In: *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop*. EW 10. Saint-Emilion, France: ACM, 2002, pp. 186–189 (cit. on p. 128).

[DG05] H. Dacharry and N. Giambiasi. “Formal Verification with Timed Automata and DEVS Models: a case study”. In: *Proceedings of Argentine Symposium on Software Engineering*. 2005, pp. 251–265 (cit. on p. 72).

[DG07] Hernán P. Dacharry and Norbert Giambiasi. “A Formal Verification Approach for DEVS”. In: *Proceedings of the 2007 Summer Computer Simulation Conference*. SCSC ’07. San Diego, California: Society for Computer Simulation International, 2007, pp. 312–319. ISBN: 1-56555-316-0 (cit. on p. 72).

[DA12] Sandeep Desai and Srivastava Abhishek. *Software Testing: A Practical Approach*. India: Phi Learning Private Limited, 2012. ISBN: 8120345347, 9788120345348 (cit. on pp. 46, 48, 49, 51).

[DK12] Omprakash Deshmukh and Mandakini Kaushik. “An Overview of Software Verification &Validation and Selection Proces”. In: *International Journal of Computer Trends and Technology (IJCTT)* 4.2 (2012), pp. 177–182 (cit. on p. 47).

[DY00] Richard B Detty and Jon C Yingling. “Quantifying benefits of conversion to lean manufacturing with discrete event simulation: a case study”. In: *International Journal of Production Research* 38.2 (2000), pp. 429–445 (cit. on p. 155).

[Dil98] D. L. Dill. “What’s between simulation and formal verification?” In: *Design Automation Conference, 1998. Proceedings*. June 1998, pp. 328–329 (cit. on p. 21).

[FKL04] Harry D Foster, Adam C Krolnik, and David J Lacey. *Assertion-based design*. Springer Science & Business Media, 2004 (cit. on p. 72).

[Fra+10] Marc Frappier, Benoît Fraikin, Romain Chossart, et al. “Comparison of Model Checking Tools for Information Systems”. In: *Formal Methods and Software Engineering: 12th International Conference on Formal Engineering Methods, ICFEM 2010, Shanghai, China, November 17-19, 2010. Proceedings*. Ed. by Jin Song Dong and Huibiao Zhu. Springer Berlin Heidelberg, 2010, pp. 581–596 (cit. on p. 62).

[GP14] María-del-Mar Gallardo and Laura Panizo. “Extending model checkers for hybrid system verification: the case study of SPIN”. In: *Software Testing, Verification and Reliability* 24.6 (2014), pp. 438–471. ISSN: 1099-1689. DOI: 10.1002/stvr.1505. URL: http://dx.doi.org/10.1002/stvr.1505 (cit. on p. 121).

[GSB13] Stephane Garredu, Jean François Santucci, and Paul-Antoine Bisgambiglia. “From State-Transition Models to DEVS Models - Improving DEVS external interoperability using MetaDEVS: a MDE approach”. In: *Simultech 2013*. Reykjavik, Iceland, 2013, pp. 186–196. URL: https://hal.archives-ouvertes.fr/hal-01293513 (cit. on p. 82).

[Gau11] Marie-Claude Gaudel. “Checking Models, Proving Programs, and Testing Systems”. In: *Tests and Proofs: 5th International Conference, TAP 2011, Zurich, Switzerland, June 30 – July 1, 2011. Proceedings*. Ed. by Martin Gogolla and Burkhart Wolff. Springer Berlin Heidelberg, 2011, pp. 1–13 (cit. on pp. 26–28, 50, 52, 67).

[Gia09] Norbert Giambiasi. “From Sequential Machines to DEVS Formalism”. In: *Proceedings of the 2009 Summer Computer Simulation Conference*. SCSC ’09. Istanbul, Turkey: Society for Modeling; Simulation International, 2009, pp. 216–222 (cit. on pp. 40, 79, 113, 114).

[GC06] Norbert Giambiasi and Jean-Claude Carmona. “Generalized discrete event abstraction of continuous systems: {GDEVS} formalism”. In: *Simulation Modelling Practice and Theory* 14.1 (2006), pp. 47–70. ISSN: 1569-190X (cit. on p. 40).

[GP06] Antoine Girard and George J. Pappas. “Verification Using Simulation”. In: *Hybrid Systems: Computation and Control: 9th International Workshop, HSCC 2006, Santa Barbara, CA, USA, March 29-31, 2006. Proceedings*. Ed. by Joao P. Hespanha and Ashish Tiwari. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 272–286 (cit. on p. 72).

[God13] Patrice Godefroid. *Combining Model Checking and Testing*. 2013. URL: http://research.microsoft.com/apps/pubs/default.aspx?id=200544 (cit. on p. 70).

[Gol08] Eugene Goldberg. “On Bridging Simulation and Formal Verification”. In: *Verification, Model Checking, and Abstract Interpretation: 9th International Conference, VMCAI 2008, San Francisco, USA, January 7-9, 2008. Proceedings*. Ed. by Francesco Logozzo, Doron A. Peled, and Lenore D. Zuck. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 127–141 (cit. on p. 70).

[Gor15] Ross Gore. *SpringSim 2015 - Conceptual Modeling with Alloy*. 2015. URL: https://github.com/rossgore/alloy-tutorial (cit. on p. 143).

[Gre96] J. Mark Green. "Peer Reviewed: A Practical Guide to Analytical Method Validation". In: *Analytical Chemistry* 68.9 (1996), 305A–309A (cit. on p. 45).

[Gru06] Orna Grumberg. "Abstraction and Refinement in Model Checking". In: *Proceedings of the 4th International Conference on Formal Methods for Components and Objects*. FMCO'05. Amsterdam, The Netherlands: Springer-Verlag, 2006, pp. 219–242 (cit. on pp. 66, 75, 78, 79).

[GS05] Anubhav Gupta and Ofer Strichman. "Abstraction Refinement for Bounded Model Checking". In: *Computer Aided Verification: 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005. Proceedings*. Ed. by Kousha Etessami and Sriram K. Rajamani. Springer Berlin Heidelberg, 2005, pp. 112–124 (cit. on p. 66).

[Har87] David Harel. "Statecharts: A Visual Formalism for Complex Systems". In: *Sci. Comput. Program.* 8.3 (June 1987), pp. 231–274. ISSN: 0167-6423 (cit. on p. 93).

[He01] Xudong He. "{PZ} nets — a formal method integrating Petri nets with Z". In: *Information and Software Technology* 43.1 (2001), pp. 1–18 (cit. on p. 69).

[Hei+05] C. Heitmeyer, M. Archer, R. Bharadwaj, et al. "Tools for constructing requirements specifications: The SCR toolset at the age of ten". In: *International Journal of Computer Systems Science and Engineering* (2005) (cit. on p. 68).

[Hei98] Constance Heitmeyer. "On the need for practical formal methods". In: *Proceedings of 5th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*. Springer Berlin Heidelberg, 1998, pp. 18–26 (cit. on p. 55).

[Hen90] Matthew Hennessy. *The Semantics of Programming Languages: An Elementary Introduction Using Structural Operational Semantics*. New York, NY, USA: John Wiley & Sons, Inc., 1990 (cit. on p. 77).

[Hen96] T. A. Henzinger. "The theory of hybrid automata". In: *Logic in Computer Science, 1996. LICS '96. Proceedings., Eleventh Annual IEEE Symposium on*. July 1996, pp. 278–292 (cit. on pp. 64, 121).

[HG05] A. Hernandez and N. Giambiasi. "State Reachability for DEVS models". In: *Proceedings of Argentine Symposium on Software Engineering*. 2005, pp. 267–277 (cit. on pp. 56, 72).

[Hin+06] Andrew Hinton, Marta Kwiatkowska, Gethin Norman, et al. “Tools and Algorithms for the Construction and Analysis of Systems: 12th International Conference, TACAS 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 25 - April 2, 2006. Proceedings”. In: ed. by Holger Hermanns and Jens Palsberg. Springer Berlin Heidelberg, 2006. Chap. PRISM: A Tool for Automatic Verification of Probabilistic Systems, pp. 441–444 (cit. on p. 65).

[Hol03] Gerard Holzmann. *The SPIN Model Checker : Primer and Reference Manual*. Addison-Wesley Professional, 2003 (cit. on pp. 59, 66, 82, 84).

[Hol91] Gerard J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, Inc., 1991 (cit. on p. 82).

[Hol97] Gerard J Holzmann. “The model checker SPIN”. In: *IEEE Transactions on software engineering* 23.5 (1997), pp. 279–295 (cit. on p. 82).

[HNS00] Gerard Holzmann, Eli Najm, and Ahmed Serhrouchni. “SPIN model checking: an introduction”. In: *International Journal on Software Tools for Technology Transfer* 2.4 (2000), pp. 321–327 (cit. on p. 82).

[HL98] Y.-W. Hsieh and S. P. Levitan. “Model Abstraction for Formal Verification”. In: *Proceedings of the Conference on Design, Automation and Test in Europe*. DATE ’98. Le Palais des Congrés de Paris, France: IEEE Computer Society, 1998, pp. 140–147. ISBN: 0-8186-8359-7 (cit. on p. 66).

[HR05] M. Huth and M. Ryan. *Logic in computer science: modelling and reasoning about systems*. Cambridge University Press, 2005 (cit. on pp. 55, 59).

[HZ09] M. H. Hwang and B. P. Zeigler. “Reachability Graph of Finite and Deterministic DEVS Networks”. In: *IEEE Transactions on Automation Science and Engineering* 6.3 (2009), pp. 468–478 (cit. on p. 72).

[Hwa11] Moon Ho Hwang. “Taxonomy of DEVS Subclasses for Standardization”. In: *Proceedings of the 2011 Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium*. TMS-DEVS ’11. Boston, Massachusetts: Society for Computer Simulation International, 2011, pp. 152–159 (cit. on p. 40).

[Hwa14] Moon Ho Hwang. “Taxonomy of DEVS Variants”. In: *Proceedings of the Symposium on Theory of Modeling & Simulation - DEVS Integrative*. DEVS ’14. Tampa, Florida: Society for Computer Simulation International, 2014, 22:1–22:6 (cit. on p. 40).

[HZ06] Moon Ho Hwang and Bernard P Zeigler. “A reachable graph of finite and deterministic DEVS networks”. In: *SIMULATION SERIES* 38.1 (2006), p. 48 (cit. on p. 72).

[IEE90] IEEE. “IEEE Standard Glossary of Software Engineering Terminology”. In: *IEEE Std 610.12-1990* (Dec. 1990), pp. 1–84 (cit. on p. 45).

[IEE12] IEEE. “IEEE Standard for System and Software Verification and Validation”. In: *IEEE Std 1012-2012 (Revision of IEEE Std 1012-2004)* (May 2012), pp. 1–223 (cit. on p. 45).

[Ino+16] A Inostrosa-Psijas, Veronica Gil-Costa, Gabriel A. Wainer, et al. “Formal Verification of DEVS Simulation: Web Search Engine Model Case Study”. In: *Proceedings of 2016 Summer Computer Simulation Conference (SCSC)*. SummerSim ’16. Society for Computer Simulation International, July 2016 (cit. on p. 72).

[Ins04] Project Management Institute. *A Guide To The Project Management Body Of Knowledge (PMBOK Guides)*. Project Management Institute, 2004 (cit. on pp. 43, 44).

[Iti07] Jean-Bernard Itier. “A380 integrated modular avionics”. In: *Proceedings of the ARTIST2 meeting on integrated modular avionics*. Vol. 1. 2. 2007, pp. 72–75 (cit. on p. 21).

[Kat10] Joost-Pieter Katoen. “Advances in probabilistic model checking”. In: *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer. 2010, pp. 25–25 (cit. on p. 65).

[KRW06] Ekkart Kindler, Vladimir Rubin, and Robert Wagner. “Component Tools: Integrating Petri Nets with Other Formal Methods”. In: *Petri Nets and Other Models of Concurrency - ICATPN 2006: 27th International Conference on Applications and Theory of Petri Nets and Other Models of Concurrency, Turku, Finland, June 26-30, 2006. Proceedings*. Ed. by Susanna Donatelli and P. S. Thiagarajan. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 37–56 (cit. on p. 68).

[Kle95] Jack P.C. Kleijnen. “Verification and validation of simulation models”. In: *European Journal of Operational Research* 82.1 (1995), pp. 145–162 (cit. on p. 50).

[KWB03] Anneke G. Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003 (cit. on p. 80).

[Kli85] George Klir. *Architecture of systems problem solving*. Springer Science & Business Media, 1985 (cit. on p. 31).

[Kri63] Saul A. Kripke. “Semantical Analysis of Modal Logic I Normal Modal Propositional Calculi”. In: *Mathematical Logic Quarterly* 9.5-6 (1963), pp. 67–96 (cit. on p. 61).

[KCS03] D Richard Kuhn, Dan Craigen, and Mark Saaltink. “Practical application of formal methods in modeling and simulation”. In: *Summer Computer Simulation Conference*. 2003, pp. 726–731 (cit. on p. 54).

[KCB02] Richard D Kuhn, Ramaswamy Chandramouli, and Ricky W Butler. “Cost effective use of formal methods in verification and validation”. In: *Foundations'02 Workshop on Verification and Validation* (2002) (cit. on p. 21).

[KW16] Matthias Kunze and Mathias Weske. *Behavioural Models: From Modelling Finite Automata to Analysing Business Processes*. Springer International Publishing, 2016 (cit. on p. 79).

[Kun+06] S. Kunzli, F. Poletti, L. Benini, et al. “Combining Simulation and Formal Methods for System-Level Performance Analysis”. In: *Proceedings of the Design Automation Test in Europe Conference*. Vol. 1. 2006, pp. 1–6 (cit. on p. 71).

[KNP07] Marta Kwiatkowska, Gethin Norman, and David Parker. “Stochastic model checking”. In: *International School on Formal Methods for the Design of Computer, Communication and Software Systems*. Springer. 2007, pp. 220–270 (cit. on p. 65).

[KP12] Marta Kwiatkowska and David Parker. “Advances in Probabilistic Model Checking”. In: *Proc. 2011 Marktoberdorf Summer School: Tools for Analysis and Verification of Software Safety and Security*. Ed. by O. Grumberg, T. Nipkow, and J. Esparza. IOS Press, 2012. URL: https://hal.inria.fr/hal-00664777 (cit. on p. 65).

[Lam77] Leslie Lamport. “Proving the correctness of multiprocess programs”. In: *IEEE transactions on software engineering* 2 (1977), pp. 125–143 (cit. on p. 61).

[Lam00] Axel van Lamsweerde. “Formal Specification: A Roadmap”. In: *Proceedings of the Conference on The Future of Software Engineering*. ICSE '00. Limerick, Ireland: ACM, 2000, pp. 147–159. ISBN: 1-58113-253-0 (cit. on p. 28).

[LWK10] Philip Langer, Manuel Wimmer, and Gerti Kappel. “Model-to-Model Transformations By Demonstration”. In: *Theory and Practice of Model Transformations: Third International Conference, ICMT 2010, Malaga, Spain, June 28-July 2, 2010. Proceedings*. Ed. by Laurence Tratt and Martin Gogolla. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 153–167 (cit. on p. 80).

[LDB10] Axel Legay, Benoît Delahaye, and Saddek Bensalem. “Runtime Verification: First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings”. In: ed. by Howard Barringer, Ylies Falcone, Bernd Finkbeiner, et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010. Chap. Statistical Model Checking: An Overview, pp. 122–135 (cit. on p. 65).

[LST05] Lun Li, Stephen A Szygenda, and Mitchell A Thornton. “Combining simulation and formal verification for integrated circuit design validation”. In: *Proceedings of the 9th World Multi-Conference on Systemics, Cybernetics and Informatics (WMSCI)*. 2005, pp. 92–97 (cit. on p. 70).

[Liu+11] Junjie Liu, Yongli Yu, Liu Zhang, et al. “An Overview of Conceptual Model for Simulation and Its Validation”. In: *Procedia Engineering* 24 (2011), pp. 152–158 (cit. on p. 34).

[Lon93] David E Long. “Model checking, abstraction, and compositional verification”. PhD thesis. Citeseer, 1993 (cit. on p. 66).

[LMN12] Francesco Longo, Marina Massei, and Letizia Nicoletti. “An Application OF Modeling And Simulation to Support Industrial Plants Design”. In: *International Journal of Modeling, Simulation, and Scientific Computing* 03.01 (2012), p. 1240001 (cit. on p. 155).

[MBK12] Oumar Maiga, Ufuoma Bright Ighoroje, and Mamadou Kaba Traoré. “INTEGRATION DES METHODES FORMELLES DANS LA SPECIFICATION, LA VERIFICATION ET LA VALIDATION DE MODELES DE SIMULATION A EVENEMENTS DISCRETS”. In: *9th International Conference on Modeling, Optimization & SIMulation*. Bordeaux, France, June 2012 (cit. on p. 72).

[Mar97] Anu Maria. “Introduction to Modeling and Simulation”. In: *Proceedings of the 29th Conference on Winter Simulation*. WSC ’97. Atlanta, Georgia, USA: IEEE Computer Society, 1997, pp. 7–13 (cit. on p. 31).

[McM00] Ken McMillan. *The SMV System*. 2000. URL: www.kenmcmil.com/tutorial.ps (cit. on p. 68).

[MC01] William E McUmber and Betty HC Cheng. “A general framework for formalizing UML with formal languages”. In: *Proceedings of the 23rd international conference on Software engineering*. IEEE Computer Society. 2001, pp. 433–442 (cit. on p. 81).

[MS10] Dominique Méry and Neeraj Kumar Singh. “Real-Time Animation for Formal Specification”. In: *Complex Systems Design & Management: Proceedings of the First International Conference on Complex System Design & Management CSDM 2010*. Ed. by Marc Aiguier,

Francis Bretaudeau, and Daniel Krob. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 49–60 (cit. on p. 35).

[MLS97] Erich Mikk, Yassine Lakhnechi, and Michael Siegel. "Hierarchical automata as model for statecharts". In: *Advances in Computing Science — ASIAN'97: Third Asian Computing Science Conference Kathmandu, Nepal, December 9–11, 1997 Proceedings*. Ed. by R. K. Shyamasundar and K. Ueda. Springer Berlin Heidelberg, 1997, pp. 181–196 (cit. on p. 93).

[Min65] Marvin Minsky. "Matter, mind and models". In: (1965) (cit. on p. 27).

[NJJ08] Wojciech Nabialek, Agata Janowska, and Paweł Janowski. "Translation of Timed Promela to Timed Automata with Discrete Data". In: *Fundam. Inf.* 85.1-4 (Jan. 2008), pp. 409–424. ISSN: 0169-2968. URL: http://dl.acm.org/citation.cfm?id=1516165.1516193 (cit. on pp. 121, 139).

[Nan81] Richard E. Nance. "The Time and State Relationships in Simulation Modeling". In: *Commun. ACM* 24.4 (Apr. 1981), pp. 173–179. ISSN: 0001-0782 (cit. on pp. 34, 36).

[NH97] V Natarajan and Gerard J Holzmann. "Outline for an operational semantics of promela". In: *The SPIN Verification Systems. DIMACS Series in Discrete Mathematics and Theoretical Computer Science. AMS* 32 (1997), pp. 133–152 (cit. on p. 84).

[NM13] Nicolas Navet and Stephan Merz. *Modeling and verification of real-time systems*. John Wiley & Sons, 2013 (cit. on p. 63).

[Owe07] David R. Owen. "Combining Complementary Formal Verification Strategies to Improve Performance and Accuracy". AAI3298734. PhD thesis. Morgantown, WV, USA, 2007. ISBN: 978-0-549-44379-7 (cit. on pp. 68, 69).

[PT96] F. De Paoli and F. Tisato. "On the complementary nature of event-driven and time-driven models". In: *Control Engineering Practice* 4.6 (1996), pp. 847–854 (cit. on p. 128).

[Par81] David Park. "Concurrency and automata on infinite sequences". In: *Theoretical Computer Science: 5th GI-Conference Karlsruhe, March 23–25, 1981*. Ed. by Peter Deussen. Berlin, Heidelberg: Springer Berlin Heidelberg, 1981, pp. 167–183 (cit. on p. 78).

[Pau+93] Mark C Paulk, Bill Curtis, Mary Beth Chrissis, et al. "Capability maturity model, version 1.1". In: *IEEE software* 10.4 (1993), pp. 18–27 (cit. on p. 45).

[Pet09] Mikel D Petty. "Verification and validation". In: *Principles of modeling and simulation: A multidisciplinary approach* (2009), pp. 121–149 (cit. on pp. 27, 28, 33).

[Pet10] Mikel D. Petty. “Verification, Validation, and Accreditation”. In: John Wiley & Sons, Inc., 2010, pp. 325–372 (cit. on pp. 27, 34, 35, 53, 54).

[Plo81] Gordon D. Plotkin. *A structural approach to operational semantics*. Technical report. Aarbus University, 1981 (cit. on p. 77).

[Plo04] Gordon D. Plotkin. “A structural approach to operational semantics”. In: *The Journal of Logic and Algebraic Programming* 60 (2004), pp. 17–139 (cit. on p. 77).

[Pnu77] Amir Pnueli. “The Temporal Logic of Programs”. In: *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*. SFCS ’77. Washington, DC, USA: IEEE Computer Society, 1977, pp. 46–57 (cit. on pp. 59, 61).

[Poc+11] Marcel Pockrandt, Paula Herber, Holger Gross, et al. “Optimized Transformation and Verification of SystemC Methods”. In: *Pre-Proceedings of the 12th International Workshop on Automated Verification of Critical Systems (AVoCS 2012)*. Vol. 2. 2011, pp. 15–61 (cit. on p. 70).

[QS82] Jean-Pierre Queille and Joseph Sifakis. “Specification and Verification of Concurrent Systems in CESAR”. In: *Proceedings of the 5th Colloquium on International Symposium on Programming*. Springer-Verlag, 1982, pp. 337–351 (cit. on p. 56).

[Rei12] Wolfgang Reisig. “The expressive power of abstract-state machines”. In: *Computing and Informatics* 22.3-4 (2012), pp. 209–219 (cit. on p. 65).

[Rot89] J. Rothenberg. “Artificial Intelligence, Simulation &Amp; Modeling”. In: ed. by Lawrence E. Widman, Kenneth A. Loparo, and Norman R. Nielsen. New York, NY, USA: John Wiley & Sons, Inc., 1989. Chap. The Nature of Modeling, pp. 75–92. ISBN: 0-471-60599-9 (cit. on pp. 28, 30).

[RG99] Johannes Ryser and Martin Glinz. “A practical approach to validating and testing software systems using scenarios”. In: *QWE’99, 3 rd International Software Quality Week Europe*. 1999 (cit. on p. 48).

[SW10] Hesham Saadawi and Gabriel Wainer. “Rational Time-advance DEVS (RTA-DEVS)”. In: *Proceedings of the 2010 Spring Simulation Multiconference*. SpringSim ’10. Orlando, Florida: Society for Computer Simulation International, 2010, 143:1–143:8. ISBN: 978-1-4503-0069-8 (cit. on p. 40).

[Sar91] Robert G. Sargent. “Simulation Model Verification and Validation”. In: *Proceedings of the 23rd Conference on Winter Simulation*. WSC ’91. Phoenix, Arizona, USA: IEEE Computer Society, 1991, pp. 37–47. ISBN: 0-7803-0181-1. URL: http://dl.acm.org/citation.cfm?id=304238.304253 (cit. on pp. 34, 51–53).

[Sar01] Robert G. Sargent. “Some approaches and paradigms for verifying and validating simulation models”. In: *Simulation Conference, 2001. Proceedings of the Winter*. Vol. 1. 2001, pp. 106–114 (cit. on p. 54).

[Sav16] Vitaly Savicks. “Integrating Formal Verification and Simulation of Hybrid Systems”. PhD thesis. University of Southampton, 2016 (cit. on p. 72).

[Sco77] Dana S. Scott. “Logic and Programming Languages”. In: *Commun. ACM* 20.9 (Sept. 1977), pp. 634–641 (cit. on p. 77).

[SV13a] Stefano Sebastio and Andrea Vandin. “MultiVeStA: Statistical Model Checking for Discrete Event Simulators”. In: *Proceedings of the 7th International Conference on Performance Evaluation Methodologies and Tools*. ValueTools ’13. Torino, Italy: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2013, pp. 310–315. ISBN: 978-1-936968-48-0 (cit. on p. 65).

[SK03] Shane Sendall and Wojtek Kozaczynski. “Model Transformation: The Heart and Soul of Model-Driven Software Development”. In: *IEEE Softw.* 20.5 (2003), pp. 42–45. ISSN: 0740-7459 (cit. on pp. 68, 80).

[Seo+13] Chungman Seo, Bernard P. Zeigler, Robert Coop, et al. “DEVS Modeling and Simulation Methodology with MS4 Me Software Tool”. In: *Proceedings of the Symposium on Theory of Modeling & Simulation - DEVS Integrative M&S Symposium*. San Diego, California: Society for Computer Simulation International, 2013, 33:1–33:8 (cit. on p. 158).

[Sha16] Unnati S. Shah. “An Excursion to Software Development Life Cycle Models: An Old to Ever-growing Models”. In: *SIGSOFT Softw. Eng. Notes* 41.1 (Feb. 2016), pp. 1–6 (cit. on pp. 21, 25).

[SB11] John A Sokolowski and Catherine M Banks. *Principles of modeling and simulation: a multidisciplinary approach*. John Wiley & Sons, 2011 (cit. on p. 30).

[SCR06] Hosung Song, Kevin J. Compton, and William C. Rounds. “SPHIN: A model checker for reconfigurable hybrid systems based on {SPIN}”. In: *Electronic Notes in Theoretical Computer Science* 145 (2006). Proceedings of the 5th International Workshop on Automated Verification of Critical Systems (AVoCS 2005)Automated Verification of Critical Systems 2005, pp. 167–183 (cit. on p. 121).

[SBS01] Robert F. Stark, E. Borger, and Joachim Schmid. *Java and the Java Virtual Machine: Definition, Verification, Validation with Cdrom*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2001. ISBN: 3540420886 (cit. on p. 65).

[SAK06] Elisabeth A Strunk, M Anthony Aiello, and John C Knight. "A survey of tools for model checking and model-based development". In: *University of Virginia* (2006) (cit. on pp. 59, 62).

[Stu+01] Douglas A. Stuart, Monica Brockmeyer, Aloyius K. Mok, et al. "Simulation-Verification: Biting at the State Explosion Problem". In: *IEEE Trans. Softw. Eng.* 27.7 (July 2001), pp. 599–617 (cit. on p. 70).

[SK97] Michael E Swartz and Ira S Krull. *Analytical method development and validation*. CRC Press, 1997 (cit. on p. 44).

[SV08] Eugene Syriani and Hans Vangheluwe. "Programmed Graph Rewriting with Time for Simulation-Based Design". In: *Proceedings of the 1st International Conference on Theory and Practice of Model Transformations*. Springer Berlin Heidelberg, 2008, pp. 91–106 (cit. on p. 143).

[SV13b] Eugene Syriani and Hans Vangheluwe. "A modular timed graph transformation language for simulation-based design". In: *Software & Systems Modeling* 12.2 (2013), pp. 387–414 (cit. on p. 143).

[TER14] R. Tajini, S.L. Elhaq, and A. Rachid. "Modelling methodology for the simulation of the manufacturing systems". In: *International Journal of Simulation and Process Modelling* 9.4 (2014), pp. 285–305. DOI: 10.1504/IJSPM.2014.066372 (cit. on p. 155).

[Ten76] R. D. Tennent. "The Denotational Semantics of Programming Languages". In: *Commun. ACM* 19.8 (Aug. 1976), pp. 437–453 (cit. on p. 77).

[Tra99] Eushiuan Tran. *Verification/Validation/Certification*. Carnegie Mellon University, 1999 (cit. on pp. 43, 46, 48, 50, 54).

[TC96] Stavros Tripakis and Costas Courcoubetis. "Extending Promela and Spin for Real Time". In: *Proceedings of the Second International Workshop on Tools and Algorithms for Construction and Analysis of Systems*. TACAs '96. Springer-Verlag, 1996, pp. 329–348 (cit. on pp. 118, 156).

[TFH09] Mohamed Wassim Trojet, Claudia Frydman, and Maâmar El-Amine Hamri. "Practical application of lightweight Z in DEVS framework". In: *Proceedings of the 2009 Spring Simulation Multiconference*. Society for Computer Simulation International. 2009, p. 154 (cit. on p. 72).

[Tro10] M.W. Trojet. “Approche de vérification formelle des modèles DEVS à base du langage Z”. PhD thesis. 2010. URL: http://books.google.fr/books?id=M9AbtwAACAAJ (cit. on p. 72).

[Wai09] Gabriel A. Wainer. *Discrete Event Simulation and Modeling: Theory and Applications - Model-Based Design*. 1st. Boca Raton, FL, USA: CRC Press, Inc., 2009. ISBN: 1420072331, 9781420072334 (cit. on p. 40).

[WF89] Dolores R. Wallace and Roger U. Fujii. “Software Verification and Validation: An Overview”. In: *IEEE Softw.* 6.3 (1989), pp. 10–17 (cit. on pp. 47, 48).

[Wan06] Bow-Yaw Wang. “Automatic Verification of a Model Checker by Reflection”. In: *Practical Aspects of Declarative Languages: 8th International Symposium, PADL 2006, Charleston, SC, USA, January 9-10, 2006. Proceedings*. Ed. by Pascal Van Hentenryck. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 45–59 (cit. on p. 134).

[WL07] Zhongshi Wang and Axel Lehmann. “A Framework for Verification and Validation of Simulation Models and Applications”. In: *AsiaSim 2007: Asia Simulation Conference 2007, Seoul, Korea, October 10-12, 2007. Proceedings*. Ed. by Jin-Woo Park, Tag- Gon Kim, and Yun-Bae Kim. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 237–246 (cit. on p. 72).

[WWZ10] Zizhen Wang, Hanpin Wang, and Naijun Zhan. “Refinement of Models of Software Components”. In: *Proceedings of the 2010 ACM Symposium on Applied Computing*. SAC ’10. Sierre, Switzerland: ACM, 2010, pp. 2311–2318. ISBN: 978-1-60558-639-7 (cit. on p. 29).

[YHF14a] A. Yacoub, M. Hamri, and C. Frydman. “A method for improving the verification and validation of systems by the combined use of simulation and formal methods”. In: *Proceedings - IEEE International Symposium on Distributed Simulation and Real-Time Applications, DS-RT*. 2014, pp. 155–162 (cit. on p. 70).

[Yac+15] A. Yacoub, M. Hamri, C. Frydman, et al. “Towards an extension of Promela for the modeling, simulation and verification of discrete-event systems”. In: *27th European Modeling and Simulation Symposium, EMSS 2015* (2015), pp. 340–348 (cit. on pp. 82, 141).

[YHF16a] Aznam Yacoub, Maamar el-amine Hamri, and Claudia Frydman. “Formal Methods and Discrete-Event Simulation”. In: *JDF 2016 - Les Journées DEVS Francophones - Theorie et Applications*. Cepadues, 2016 (cit. on p. 77).

[YHF16b] Aznam Yacoub, Maamar el-amine Hamri, and Claudia Frydman. “Using DEv-PROMELA for Modelling and Verification of Software”. In: *Proceedings of the 2016 Annual ACM Conference on SIGSIM Principles of Advanced Discrete Simulation*. SIGSIM-PADS ’16. Banff, Alberta, Canada: ACM, 2016, pp. 245–253. ISBN: 978-1-4503-3742-7 (cit. on p. 143).

[YHF14b] Aznam Yacoub, Maamar El-Amine Hamri, and Claudia S. Frydman. “Complementarity between simulation and formal verification transformation of PROMELA models into FDDEVS models: Application to a case study”. In: *4th International Conference On Simulation And Modeling Methodologies, Technologies And Applications, SIMULTECH 2014, Vienna, Austria, August 28-30, 2014*. 2014, pp. 421–426 (cit. on p. 70).

[Zei76] Bernard P. Zeigler. *Theory of Modeling and Simulation*. John Wiley, 1976 (cit. on pp. 23, 30–35, 38, 39, 79).

[Zei84] Bernard P. Zeigler. *Multifacetted Modelling and Discrete Event Simulation*. San Diego, CA, USA: Academic Press Professional, Inc., 1984 (cit. on pp. 40, 78).

[ZKP00] Bernard P. Zeigler, Tag Gon Kim, and Herbert Praehofer. *Theory of Modeling and Simulation*. 2nd. Academic Press, Inc., 2000 (cit. on pp. 23, 26, 28–30, 33, 35, 40, 78, 79, 117, 123, 132).

[ZN15] Bernard P Zeigler and James J Nutaro. “Towards a framework for more robust validation and verification of simulation models for systems of systems”. In: *The Journal of Defense Modeling and Simulation: Applications, Methodology, Technology* (2015) (cit. on p. 72).

[ZNS16] Bernard P. Zeigler, James Nutaro, and Chungman Seo. “Combining DEVS and model-checking: Concepts and tools for integrating simulation and analysis”. In: *To appear in Int. J. Process Modeling and Simulation* (2016) (cit. on p. 72).

[Zer+13] Fokion Zervoudakis, David S. Rosenblum, Sebastian Elbaum, et al. “Cascading Verification: An Integrated Method for Domain-specific Model Checking”. In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 2013, pp. 400–410 (cit. on p. 55).