Verification and Validation of UML and SysML Based Systems Engineering Design Models

YOSR JARRAYA

A THESIS

IN

THE DEPARTMENT

OF

ELECTRICAL AND COMPUTER ENGINEERING

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
CONCORDIA UNIVERSITY
MONTRÉAL, QUÉBEC, CANADA

APRIL 2010

Your file Votre référence
ISBN: 978-0-494-67341-6
Our file Notre référence
ISBN: 978-0-494-67341-6

Abstract

Verification and Validation of UML and SysML Based Systems Engineering Design Models

Yosr Jarraya, Ph.D.

Concordia University, 2010

In this thesis, we address the issue of model-based verification and validation of systems engineering design models expressed using UML/SysML. The main objectives are to assess the design from its structural and behavioral perspectives and to enable a qualitative as well as a quantitative appraisal of its conformance with respect to its requirements and a set of desired properties. To this end, we elaborate a heretofore unattempted unified approach composed of three well-established techniques that are model-checking, static analysis, and software engineering metrics. These techniques are synergistically combined so that they yield a comprehensive and enhanced assessment. Furthermore, we propose to extend this approach with performance analysis and probabilistic assessment of SysML activity diagrams. Thus, we devise an algorithm that systematically maps these diagrams into their corresponding probabilistic models encoded using the specification language of the probabilistic symbolic model-checker PRISM. Moreover, we define a first of its kind probabilistic calculus, namely activity calculus, dedicated to capture the essence of SysML activity diagrams and its underlying operational semantics in terms of Markov decision

processes. Furthermore, we propose a formal syntax and operational semantics for the input language of PRISM. Finally, we mathematically prove the soundness of our translation algorithm with respect to the devised operational semantics using a simulation preorder defined upon Markov decision processes.

Acknowledgments

Many people have contributed to my life and to my thesis to whom I would like to address these words.

First of all, I would like to thank my supervisor Dr. Mourad Debbabi for giving me the opportunity to lead this doctoral research work under his supervision. He guided me with many of his wise advices emanating from his profound knowledge and broad experience. I would like also to address my thanks to my co-supervisor Dr. Jamal Bentahar for supporting me. I am very grateful to both of them for their valuable suggestions and guidance throughout the preparation of this thesis. Next, I would like to thank Dr Fawzi Hassaïne, the responsible of the V&V project at Defence Research and Development Canada and my colleagues Andrei Soeanu, Luay Alawneh, and Payam Shahi with whom I worked in this project.

This experience would not have been the same without the assistance of my friends, the Computer Security Laboratory mates, as well as the nice and helpful CIISE staff.

Finally, I am deeply grateful to my parents for their endless support, to my husband and his parents for their understanding and encouragement, and to my little daughter Sarah who strengthened in me the art of patience.

Contents

List of Figures

List of Tables

Abbreviations

AC	Activity Calculus
CSL	Continuous Stochastic Logic
CTMC	Continuous Time Markov Chain
CTL	Computation Tree Logic
DTMC	Discrete Time Markov Chain
EFFBD	Enhanced Function Flow Block Diagram
HDL	Hardware Description Language
IDEF	Integrated DEFinition Language
IEEE	Institute of Electrical and Electronics Engineers
INCOSE	INternational Council On Systems Engineering
ISO	International Organization for Standardization
LCC	Life Cycle Cost
LTL	Linear Temporal Logic
LTS	Labeled Transition System
MBSE	Model-Based Systems Engineering
MDA	Model-Driven Architecture
MDP	Markov Decision Process
M&S	Modeling & Simulation
OCL	Object Constraint Language
OMG	Object Management Group
OMT	Object Modeling Technique
PCTL	Probabilistic Computation Tree Logic
PTS	Probabilistic Transition System
SE	Systems Engineering
SOS	Structural Operational Semantics
SysML	System Modeling Language
TeD	Telecommunications Description Language
UML	Unified Modeling Language
V&V	Verification and Validation
VV&A	Verification, Validation, and Accreditation
WebML	Web Modeling Language

Chapter 1

Introduction

Modern society relies heavily on systems. Nowadays, one can readily notice the omnipresence of systems in various domains including communications, healthcare, transportation, and industry. Every day, new systems are designed with the intention to improve the quality of life, increase the productivity, and make daily tasks easier. However, life can turn out to be a nightmare if these systems fail. Their failure may have profound implications ranging from serious endangerment of human lives and severe damage to equipments to money loss. Thus, the importance of building fail-safe systems that meet their design objectives is now greater than ever before. In addition to reliability, today's systems need to be sustainable, highly performing, and produced at reasonable costs. All these constraints have increased the challenge of developing profitable systems.

A system is defined as a collection of components, including people, hardware, and/or software, that are working together in order to accomplish a set of common specific objectives [1]. The design and realization of successful systems as well as the effective

management of engineering projects represent the prime concerns of Systems Engineering (SE) [2]. Notably, the critical aspect in the development of systems is not represented by conceptual difficulties or technical shortcomings, but it is rather related to the difficulty of ensuring specification-compliant products. This is due to many factors including the increased complexity of the engineered systems and the controversial effectiveness of the applied methods. In fact, the complexity of modern systems is continuously growing as more sophisticated products integrating new functionalities, electronics, and software components are in demand. With this increase in complexity and size of systems, the complexity of applying quality assurance methods skyrockets and testing becomes complicated and lengthy. Additionally, real life systems may exhibit stochastic behavior, where the notion of uncertainty is ubiquitous. Uncertainty can be viewed as a probabilistic behavior that models, for instance, risks of failure or randomness. As example of such systems we can cite lossy channel systems [3], randomized dining philosophers problem [4], and dynamic power management systems [5].

Verification, validation, and accreditation are expected to be an integral part of the SE process that span the product life cycle. Basically, verification is the assurance of the correctness with respect to the technical assumptions while validation is the assurance of the conformance to the requirements. The subsequent results of the V&V process are subjected to accreditation. The latter consists in inspecting the results in order to take an official decision on whether to accept the system or not. However, in practice, the VV&A effort is mostly concentrated on the final product and little to no effort is dedicated to the earlier products such as the design outcome. This is due to many causes: the mistaken

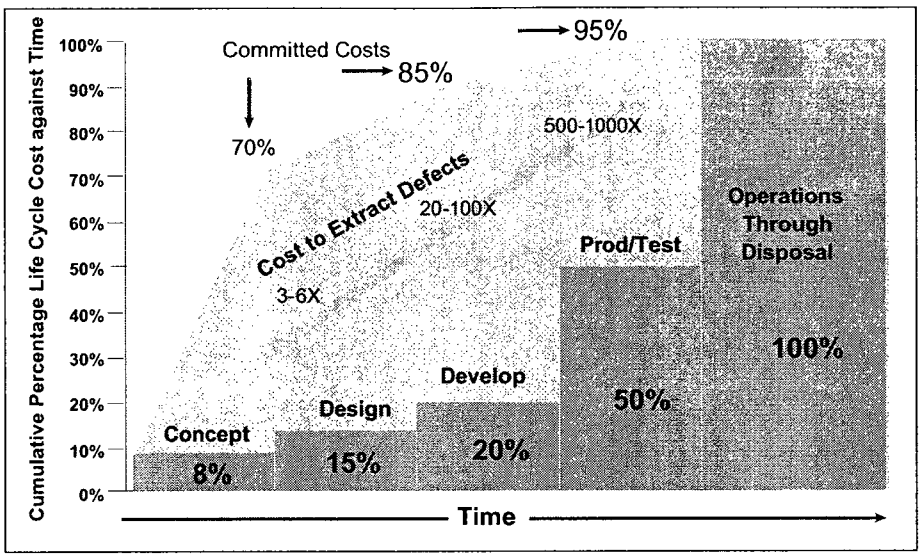


Figure 1: Committed Life Cycle Cost Against Time

belief that testing the final product is enough, the willing to minimize efforts, decrease the time-to-market, and save costs. Moreover, some limitations imposed by conventional quality assurance methods in the face of the increased complexity of systems may represent an additional contributing factor.

Generally, if quality assurance activities are performed according to the standard operating policies and procedures, they may be costly and time-consuming. Quality assurance costs include direct costs such as time and effort of V&V professionals and resources consumption (i.e. computer systems and support facilities). Furthermore, there are indirect costs such as training, acquisition, and support for related tools as well as meetings time [6]. Moreover, people usually believe that V&V costs may outweigh the earned benefits. However, this contradicts many studies that find out a return of investment in the case of the early detection of errors since this allows decreasing the maintenance time, effort, and cost. For instance, Bohem [7] provided interesting findings on software quality costs: "fixing a

defect after delivery can be one hundred times more expensive than fixing it during the requirement and the design phases". In the same vein, the INternational Council On Systems Engineering (INCOSE) confirms that the cost of finding the errors late during the system life cycle drastically increases [8]. As such, Figure 1 illustrates the Life Cycle Cost (LCC) accrued over time, the committed costs by the decisions taken, and the cost over time to extract defects. The light arrow under the curve indicates the multiplication factor of the expenses spent in order to remove errors depending on the considered life cycle phase.

Additionally, traditional SE design outcome is essentially composed of a set of documents informally describing the proposed solution that cannot be analyzed using any automated V&V means but human-based inspection of trained people. This may be tedious and complex and consequently contribute to the willing of delaying the V&V tasks to the latest development phase. In contrast to the traditional document-centric SE approaches, modern SE practices have undergone a fundamental transition to a model-based approach [9]. In a Model-Based Systems Engineering (MBSE), a system model, storing design decisions, is at the center of the development process (from requirements elicitation and design to implementation and testing). The advantage of such a model is its suitability to be subjected to systematic analyses using specific V&V techniques. In order to cope with this new model-based approach, SE community and standardization bodies developed interest in first using existing standard modeling languages, namely the Unified Modeling Language (UML) and then developing a dedicated systems modeling language, namely SysML [10].

In this thesis, we propose an innovative unified approach for the V&V of SE design models expressed using UML/SysML. It is part of a major research initiative supported

by Defence Research and Development Canada (DRDC)[1] and conducted in the Computer Security Laboratory at Concordia University. This chapter is organized as follows. Section 1.1 presents the motivations that determine the raison d'être of this thesis. Then, Section 1.2 describes the problem statement. Section 1.3 provides a general overview of the proposed approach. Next, Section 1.4 lists the objectives of this thesis. Section 1.5 highlights our main contributions. Finally, Section 1.6 summarizes the structure of the remainder chapters.

1.1 Motivations

As stated before, even though V&V is expected to be carried on along the life cycle of the system, most of the efforts are concentrated on testing the final product. Testing consists in exercising each test scenario developed by engineers on various fidelity levels testbeds ranging from simulators to actual hardware [11] and comparing the obtained results with the anticipated ones. Even though testing is essential in order to make sure that systems operate as expected, it can be complex and overwhelming. Also, it can reveal only the presence of faults and never their absence [12]. Moreover, it only allows the late discovery of errors whereas leaving some types of errors unexplored. Furthermore, testing some systems in their actual operational conditions can be costly and difficult to realize.

Concerning current trends in terms of V&V of design models, systems engineers rely

[1]The collaboration has started within the *Collaborative Capability Definition, Engineering and Management* (CapDEM) project, which is an R&D initiative within the Canadian department of defence. The latter aims to the development of a Systems-of-Systems engineering process and relies heavily on Modeling & Simulation.

on inspection and simulation or a combination of both. Inspection is a coordinated activity that includes a meeting or a series of meetings directed by a moderator [13] where the design is reviewed and compared with standards. It is based on the subjectiveness of human judgment, which cannot be regarded as the absolute truth because of its inherent nature of being error-prone. Furthermore, the success of such activity depends on the depth of planning, organization, and data preparation preceding the actual inspection activity [14] as well as on the expertise of the involved parties. However, this technique is based on documented procedures and policies that are difficult to manage and needs training the involved people. Furthermore, this task becomes more tedious and sometimes even impossible with the increase in size and complexity of design models. Alternatively, simulation is an experimental method performed with a simulation model in order to get information about the real system without actually having it. It involves an organized process of stimulating the model and measuring its responses [14] based on a pre-established plan, a predefined setup, and predicted responses. Though extremely useful, this technique is not comprehensive enough since it covers only predefined computation paths.

Integrating V&V during the design phase helps continuously identifying and correcting errors as well as gaining confidence in the system, which leads to significant reduction of costs incurred while fixing errors at the maintenance phase. Additionally, correcting errors before the actual realization of the system enables the reduction of project failure risks occurring while engineering complex systems. Furthermore, it improves the quality of systems and accelerates the time-to-market.

Finally, our motivations behind the integration of probabilistic aspects analysis early in

the systems life cycle is due to many factors. Firstly, a range of systems inherently exhibit uncertainty, which is usually expressed by means of probabilities. Consequently, taking into account this aspect leads to more realistic models. For example, many communication protocols are probabilistic in nature in the sense that sending correctly messages over a faulty media can only be guaranteed with a given probability. Secondly, most of quality attributes such as performance, reliability, and availability have a probabilistic nature. For instance, performance is generally expressed by means of expected probability. Reliability is by definition the probability that a system successfully operates and availability is the probability that a system is operating satisfactorily when needed for a particular mission or application [15]. Finally, performing quantitative assessment of systems after integration testing is generally the norm in the industry. However, quantitative assessment of the system early in the development life cycle may reveal important information that qualitative assessment misses.

1.2 Problem Statement

Various design V&V methodologies are proposed in the literature. While reviewing related works, the following remarks can be emphasized. Most of the proposals rely on a single verification technique, which application concentrates on a unique aspect of systems' characteristics. Furthermore, structural and behavioral perspectives are quite often addressed separately. Moreover, the verification of either functional or non-functional requirements is usually addressed, but rarely both. In addition, with respect to systems' behavior, state

machine diagrams are extensively studied whereas activities are considered as secondary derivative diagrams. Moreover, the proposals are rarely supported by formal foundations and proofs of soundness. From another side, SysML [10] is a very young language that augments a subset of UML with new features specific to systems modeling. Thus, we can hardly find significant related work on the subject.

Ideally, an efficient V&V approach needs to comply with the following guidelines:

- Enable automation as much as possible. This optimizes the V&V process and prevents potential errors that may be introduced by manual manipulation.
- Encompass formal and rigorous reasoning in order to minimize errors caused by subjective human-judgment.
- Support the graphical representation provided by the modeling language for the sake of conserving the usability of the visual notation and hide the intermediate transformations underlying the mechanisms implemented by the proposed approach.
- Combine quantitative as well as qualitative assessment techniques.

In the field of verification of systems and software, we pinpoint three well-established techniques that we propose in order to build our V&V framework. On the one hand, automatic formal verification techniques, namely model-checking, is reported to be a successful approach to the verification of the behavior of software and hardware applications. Model-checkers are generally capable of generating counter examples for failed properties. Also, their counterpart in the stochastic world, namely probabilistic model-checkers, are widely applied to quantitatively analyze specifications that encompass probabilistic information

about systems behavior [16]. On the other hand, static analysis that is usually applied on software programs [17], is used prior to testing [18] and model-checking [19]. Particularly, static slicing [19] yields smaller programs, which are less expensive to verify. Furthermore, empirical methods, specifically software engineering metrics, have proved to be successful in quantitatively measuring quality attributes of object-oriented design models. As we cannot compare what we cannot measure [20], metrics provide a means to evaluate the quality of proposed design solutions and help reviewing some design decisions.

In this light, this thesis aims at answering the following questions:

- How can we apply probabilistic and non-probabilistic model-checking on UML/SysML behavioral models?
- How can we synergistically integrate static analysis and metrics with model-checking in order to efficiently analyze behavioral diagrams?
- How can we benefit from the software engineering metrics by applying them on artifacts other than the structural diagrams?
- How can we assist systems engineers in their mission while ensuring a smooth learning curve of the applied approach and without sacrificing the benefits of the graphical notation?

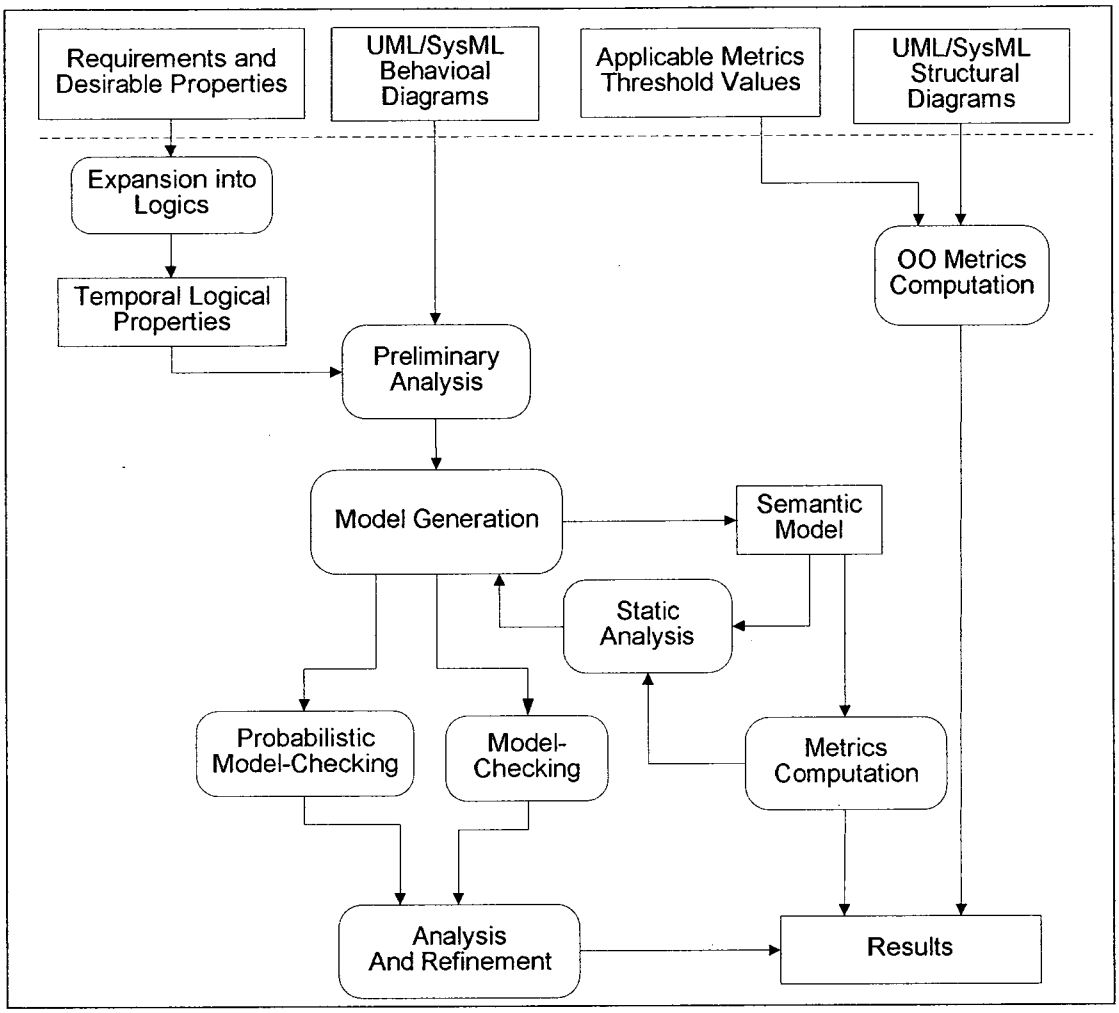


Figure 2: Proposed Approach

1.3 Approach

In this doctoral thesis, we aim at dealing with the issue of V&V of systems engineering design outcome. Thus, we propose an original unified approach for the V&V of SE design models expressed using UML/SysML modeling languages. The proposed approach synergistically integrates three well-established techniques that are model-checking, static analysis, and software engineering techniques. Figure 2 illustrates a summary of our approach. The main objectives are to enable a structural as well as a behavioral coverage of

the system design model in addition to providing the means to qualitatively and quantitatively verify its conformance with its requirements. With respect to behavioral diagrams, we propose to apply model-checking technique. Formally, the model-checker operates on the formal semantics describing the meaning of the model. It verifies the model by exploring the state space searching whether a given property holds or fails. In order to optimize the model-checking procedure from time and resources point of views, we advocate the use of static analysis techniques. More precisely, we inspired from static slicing of software programs in order to slice the semantic model prior to model-checking. This focuses the inquiry on specific parts of the design depending of the property of interest. Moreover, we propose to apply metrics on the semantic model generated from the behavioral diagrams in order to have an appraisal of its size and complexity. This allows the estimation of whether there is a need to apply static analysis. With respect to structural diagrams, we propose empirical metrics in order to quantitatively measure relevant quality attributes of the design. Besides, we extend this unified approach in order to cope with performance analysis and probabilistic behavior assessment. Therein, we focus essentially on SysML activity diagrams for three reasons. First, activity diagrams were most of the time treated as secondary, with a semantics tightly related to the statecharts semantics. However, this is no more the case since the release of the new revision of UML, namely UML 2.0 [21] and consequently SysML 1.0 [10]. Second, activity represents an important diagram for systems modeling due to its suitability for functional flow modeling commonly used by systems engineers [22]. Finally, SysML has added support for probabilistic information

modeling in activity diagrams. Therefore, we propose a translation algorithm that automatically generates the input of a probabilistic model-checker from a given SysML activity diagram. In order to add formal foundations to our approach, we define a hitherto unattempted calculus dedicated for SysML activity diagrams, namely Activity Calculus (AC). The latter captures the expressive power of activity diagrams in an algebraic fashion and is used in order to define an operational semantics for these diagrams. Finally, we demonstrate using a mathematical proof the soundness of the translation algorithm with respect to the derived operational semantics. This proof gives confidence to systems engineers that the results returned by the implementation of our algorithm are correct.

1.4 Objectives

The present thesis has as a principal objective to present an innovative unified approach for the V&V of systems engineering design models that are expressed using UML/SysML modeling languages. This can be decomposed into sub-objectives as follows:

- Study the state of the art in the following research areas: V&V of UML and SysML design models, performance analysis of such design models, formal semantics for UML and SysML activity diagrams, and software engineering metrics.
- Elaborate a unified approach for the V&V of UML/SysML design models based on three complementary well-established techniques that are:
  - Automatic formal verification,
  - Static analysis,

  - Empirical software engineering quantitative methods.

- Take into account the stochastic nature of real-life systems by supporting probabilistic behavior assessment using probabilistic model-checking technique.

- Define an algebraic calculus that captures the essence of SysML activity diagrams and investigate the formal foundations of its underlying operational semantics.

- Investigate the correctness of the proposed probabilistic verification approach.

- Implement and apply our approach on case studies to practically demonstrate its viability and benefits.

1.5 Contributions

The main contributions of this thesis can be summarized as follows:

- *An innovative V&V approach based on a synergy between model-checking, static analysis, and empirical software engineering quantitative methods*: We elaborate a practical framework for the V&V of UML/SysML design models that integrates three well-established techniques that are model-checking, static analysis, and software engineering metrics. On behavioral diagrams, we apply model-checking synergistically combined with static analysis and metrics. Moreover, we use metrics in order to measure relevant design quality attributes.

- *Performance analysis and probabilistic behavior assessment of SysML activity diagrams*: We extend the aforementioned approach by proposing a systematic translation algorithm that maps SysML activity diagrams into the specification language of the selected probabilistic symbolic model-checker PRISM. This enables quantitative as well as qualitative assessment of the design against functional and non-functional requirements.

- *A probabilistic calculus establishing the semantic foundations of SysML activity diagrams*: We define a dedicated heretofore unattempted calculus, namely Activity Calculus (AC) that captures the essence of SysML activity diagrams. Furthermore, we build AC underlying operational semantics in terms of Markov decision processes.

- *Timed actions assessment*: We annotate action nodes in SysML activity diagrams with time constraints and propose Markovian reward mechanism in order to analyze timing-related properties.

- *A formal syntax and semantics for PRISM language*: We propose a formal syntax and operational semantics for the specification language of the selected probabilistic model-checker PRISM.

- *Soundness of the translation algorithm*: We formulate and prove, using a simulation relation upon Markov decision processes, the soundness theorem for the translation algorithm that maps SysML activity diagrams into the input language of PRISM. The proof is performed with respect to the operational semantics of SysML activity diagrams and uses the defined operational semantics of PRISM language.

1.6 Thesis Structure

The rest of the thesis is organized as follows:

- Chapter 2 introduces some important aspects related to the topic background. First, we present systems engineering and its related concepts. We particularly focus on the verification and validation process as well as on the activities related to design and modeling. Then, we provide an overview of the modeling languages of our interest, namely UML and SysML. Afterwards, we discuss activity, sequence, and state machine diagrams related notations and their underlying meaning. After that, we lay down the existing verification and validation techniques. Next, we describe model-checking technique principles in both probabilistic and non-probabilistic cases. Then, the following section is dedicated for the probabilistic symbolic model-checker PRISM and its input language. Finally, the definitions of the formal models related to our solution are presented.

- Chapter 3 provides a comprehensive review of relevant research initiatives concerning the V&V of UML and SysML design models. Moreover, related work on performance analysis applied to such design models is presented. Following that, software engineering metrics are investigated. Finally, the research proposals on ascribing a formal semantics to activity diagrams are discussed.

- Chapter 4 is dedicated to the description of the proposed unified approach for the V&V of UML/SysML design models. First, the overall approach is presented. Then

the methodology is thoroughly explained . Therein, the different techniques composing our proposal are detailed. First, we explain the generation of the non-probabilistic model-checker input from UML behavioral diagrams. Thereafter, we present how we intend to synergistically integrate metrics and static analysis techniques prior to model-checking. After that, we present the application of software engineering metrics on structural diagrams. Next, we summarize the proposed extensions for supporting probabilistic verification. Finally, the framework V&V tool implementing our approach is described.

- Chapter 5 presents the probabilistic verification of SysML activity diagrams with and without time-annotation. Therein, we first explain the proposed time-annotation. Then, we explain the translation algorithm that maps this type of diagrams into the input language of the selected probabilistic model-checker and the use of rewards mechanism in order to assess timing aspects. Finally, we present a case study that demonstrates the proof of principle for our approach so that V&V as well as performance analysis are performed on SysML activity diagrams.

- Chapter 6 describes our probabilistic calculus, namely Activity Calculus (AC). First, we present AC language syntactic definition and summarize the mapping of activity diagram's graphical notation into AC terms. Afterwards, we elaborate the definition of its corresponding operational semantics. Finally, we illustrate the usefulness of such a formal semantics on a case study consisting of a SysML activity diagram modeling an hypothetical banking operation on an Automated Teller Machine (ATM).

- Chapter 7 examines essentially the soundness of the translation procedure described in Chapter 5. To this end, we first explain the methodology that we applied. Then, we present a formal description of the translation algorithm using a functional core language. After that, we propose an operational semantics for the specification language of the selected probabilistic model-checker, namely PRISM. Finally, we present a simulation preorder upon Markov decision processes that we use for formulating and proving the correctness of the translation algorithm.

- Chapter 8 concludes the thesis with a summary of our contributions and discusses new directions for potential future research in this topic.

Chapter 2

Background

This chapter introduces the fundamental background and the main concepts within the scope of this thesis. Section 2.1 provides an overview of the systems engineering discipline and the related principles and practices. This includes the main definitions, a description of the verification and validation process, and the main activities related to design and modeling. Section 2.2 focuses on the modeling languages of our interest, namely UML 2.1.2 [21] and SysML 1.0 [10]. Therein, the main concepts and notations of UML/SysML design models are introduced. We particularly focus on behavioral diagrams syntax and semantics as described in the corresponding standard, namely activity, state machine, and sequence diagrams. Section 2.3 briefly presents the existing verification and validation techniques. Section 2.4 reviews non-probabilistic and probabilistic model-checking techniques. The description of the probabilistic symbolic model-checker that we are using in this thesis, namely PRISM, can be found in Section 2.5. Finally, Section 2.6 discusses various formal models related to our work.

2.1 Systems Engineering

Systems Engineering (SE), as its name stands, is the engineering discipline that is concerned with systems. The most common definition used for a system is "a set of interrelated components working together toward some common objective" [1]. As many other engineering disciplines, SE is supported by a number of organizations and standardization bodies. One of the most active organization is the International Council on Systems Engineering (INCOSE). Its primary mission is "to advance the state of the art and practice of systems engineering in industry, academia, and government by promoting interdisciplinary, scalable approaches to produce technologically appropriate solutions that meet societal needs" [23].

SE is defined by INCOSE as an interdisciplinary approach that enables the realization of successful systems focusing on the system as a whole [2]. Although SE has been applied for a long time [2], it is sometimes referred to as an emerging discipline [24]. This can be understood in the sense that its importance has been recognized in the context of the increased complexity of today's problems. Ubiquitous systems such as hi-tech portable electronics, mobile devices, ATMs as well as many other advanced technologies like aerospace, defense or telecommunication platforms represent important application fields of systems engineering.

The ISO/IEC 15288 standard [25] distinguishes four system life cycle processes groups supporting SE: technical processes, project processes, enterprise processes, and agreement

processes. Project processes include but not limited to planning, control, and decision-making. Enterprise processes involve investment management, system life cycle processes management, and resource management. Agreement processes address acquisition and supply. Finally, technical processes group encompasses processes such as requirements definition, requirements analysis, architectural design, implementation, integration, verification, validation, and maintenance [8].

The mission of SE can be summarized as stated by IEEE [26]: "to derive, evolve, and verify a life cycle balanced system solution that satisfies customer expectations and meets public acceptability". In other words, besides deriving an effective system solution, SE duty is to ensure that the engineered system meets its requirements and its development objectives and that it performs successfully its intended operations [1]. To this end, systems engineers attempt to anticipate potential problems and to resolve them as early as possible in the development cycle. According to [8], six system product life cycle stages are identified in ISO/IEC 15288 [25]: concept, development, production, utilization, support, and retirement. Verification and Validation (V&V) activities are supposed to be performed continuously throughout the system product life cycle (from requirements elicitation through system delivery).

2.1.1 Verification, Validation, and Accreditation

There are many definitions for the terms verification and validation depending on the concerned group or the domain of application. In the SE world, the most widely used definitions are provided by the Defense Modeling and Simulation Organization (DMSO) [27,28].

On the one hand, verification is defined as "the process of determining that a model implementation and its associated data accurately represent the developer's conceptual description and specifications" [27]. On the other hand, validation is defined as "the process of determining the degree to which a model and its associated data provide an accurate representation of the real world from the perspective of the intended uses of the model" [27]. As the absence of a consensual definition for V&V raises ambiguities leading to incorrect use and misunderstanding [14, 29], the following illustrative example (inspired from [29]) is meant to clarify how we intend to use these two terms in this thesis. For instance, if a developer designs a system that complies with the specifications, but presents logical bugs, the system would fail the verification but successfully passes the validation. Conversely, if the system design is bug-free whereas it does not behave as expected, the model would fail the validation even though it passes the verification. In more common terms, the main purpose of V&V is to answer two key questions: 1) "Are we building the system right?" (Verification) and 2) "Are we building the right system?" (Validation).

At the start of system life cycle, the end users and the developers have to identify the systems' needs then to translate them into a set of specifications. Within this process a collection of functional and non-functional requirements are identified. Functional requirements specify what functions the system must perform, whereas the non-functional ones define how the system must behave, which might impose constraints upon the systems behavior (such as performance, security, or reliability). The collection of requirements represents a highly iterative process that ends up when the requirements reach a level of maturity sufficiently enough for initiating the development phase. Then, throughout the

system development phase, a series of technical reviews and technical demonstrations are held in order to answer the questions related to V&V [30]. At the end of the V&V process, the subsequent results are inspected in order to take an official decision on whether to accept the system or not. This is known as accreditation and it is commonly performed by an accreditation authority. Accreditation is defined as "the official certification that a model, a simulation, or a federation of models with simulations and the associated data is acceptable for use for a specific purpose" [27].

Basic V&V activities include inspection/examination, analysis, and testing [8]:

- *Inspection*: It consists in examining an item against the applicable documentation in order to confirm its compliance with requirements. Only observable properties can be verified by examination.

- *Analysis*: It uses analytical data or simulations under defined conditions to show theoretical compliance when testing the system in a realistic environment is difficult to realize or not cost-effective.

- *Test*: It consists in conducting specific actions in order to verify the operability, supportability, or performance capability of an item when subjected to real or simulated controlled conditions. It often involves the use of special test equipment or instrumentation in order to obtain accurate quantitative data for analysis.

2.1.2 Modeling and Simulation

Modeling and Simulation (M&S) is a design approach that is widely used by systems engineers. It helps gaining insights into the system structure and behavior before effectively producing it in order to manage the risk of failure to meet the system mission and performance requirements [8]. Modeling is defined in [31] as "the application of a standard, rigorous, structured methodology to create and validate a physical, mathematical, or otherwise logical representation of a system, entity, phenomenon, or process". With respect to simulation, it is defined by Shannon [32] as "the process of designing a model of a real system and conducting experiments with this model for the purpose of understanding the behavior of the system and/or evaluating various strategies for the operation of the system". Generally, it is the duty of subject matter experts to develop and validate the models, conduct the simulations, and analyze the results [8].

A model may be used to represent the system, its environment, and its interactions with other enabling and interfacing systems. M&S is important for decision-making since it enables to predict systems characteristics including performance, reliability, and operations. The predictions are used to guide decisions about the system design, construction, and operation, and to verify its acceptability [8]. However, simulation is hardly able to keep up with the rapidly increasing complexity of modern system design. In fact, the number of simulation cycles required in the verification process is continuously growing and simulation based methodologies require the time consuming step of creating the test inputs.

2.1.3 Model-Based Systems Engineering

MBSE is defined by INCOSE as the formalized application of modeling to support system requirements, design, analysis, verification and validation. It starts from the conceptual design phase and continues throughout the development and later life cycle phases [33]. MBSE for systems engineering is what Model-Driven Architecture (MDA) is for software engineering. They both target the use of a model-based approach to the development where the functionality and the behavior of the developed system are separated from the implementation details. In MBSE, the principal artifact is a coherent model of the system being developed. It is intended to support activities of systems engineering that have traditionally been performed according to a document-based approach. This aims at enhancing communications, specification and design precision, system design integration, and reuse of system artifacts [9]. Prominent visual modeling languages such as UML and SysML are supporting MBSE methodologies and are discussed in the next section.

2.2 Modeling Languages

Modeling is defined as a means to capture ideas, relationships, decisions, and requirements in a well-defined notation [34], namely a modeling language. Modeling languages are commonly used to specify, visualize, store, document, and exchange design models. In addition, they are domain-specific, thus containing all the syntactic, semantic, and presentation information regarding a given application domain. Various modeling languages have been defined by organizations and companies targeting different domains such as web

(WebML) [35], telecommunications (TeD) [36], hardware (HDL) [37], and software and lately systems (UML) [38]. Other languages such as IDEF [39] were designed for a broad range of uses including functional modeling, data modeling, and network design.

Although SE exists since more than five decades, until recently, there has been no dedicated modeling language for this discipline [40]. Traditionally, systems engineers rely heavily on documentation to express systems requirements and use various modeling languages in order to express the complete design solution, lacking of a specific standard language [41]. This diversity of techniques and approaches limits the cooperative work and the exchange of information. Among existing modeling languages that have been used by systems engineers we can cite HDL, IDEF, and EFFBD [42–44]. In order to remedy to this, OMG and INCOSE with a number of experts from the SE field have been collaborating in order to build a standard modeling language for SE. Being the modeling language par excellence for software engineering, UML has been selected to be customized for systems engineers needs. However, as the old version UML 1.x was found to be inadequate for systems engineering use [45, 46], the evolving revision of UML (i.e. UML 2.0) has been issued with features of interest for systems engineers. On April 2006, a proposal for a standard modeling language for systems modeling, namely SysML, has been submitted to the OMG in order to achieve the final standardization process. In the sequel, we provide a description of the main ideas concerning UML 2.x [21] and SysML 1.0 [10] modeling languages.

2.2.1 UML: Unified Modeling Language

UML stands for Unified Modeling Language and it had originally been built in order to serve modeling software systems. It is a result of the merging of three major notations: Grady Booch's methodology for describing a set of objects and their relationships [47], James Rumbaugh's Object-Modeling Technique (OMT) [48], and Ivar Jacobson's approach that includes "use case" methodology [49]. Its maintenance and revisions are assumed by OMG since 1997. It is a general-purpose visual modeling language that can be used for modeling standard software products, but also provides system architects, software engineers, and software developers with tools for analysis, design, and implementation of software-based systems as well as for modeling business processes and alike [21]. Furthermore, the strength of UML resides in its wide acceptance by many industrials and in the fact that it is non-proprietary, extensible, and supported by many tools and textbooks.

UML is defined using a meta-model, which is an abstraction of the UML model itself highlighting the properties of the language as well as the rules, constraints, and processes used to form the model. It offers a number of high-level modeling concepts allowing compact and abstract description of some systems properties. This abstractness capability offered by UML allows disregarding implementation details, which helps focusing on the essential business aspects of a solution. Furthermore, UML supports extension mechanisms (known as profiling mechanisms) such as constraints, stereotypes, and tagged values, which permit adapting it to a specific domain. A UML profile is a collection of extensions to the UML notations added for the purpose of tailoring the language to specific areas. Among UML profiles, we can cite UML Profile for CORBA [50], UML Profile for Modeling and

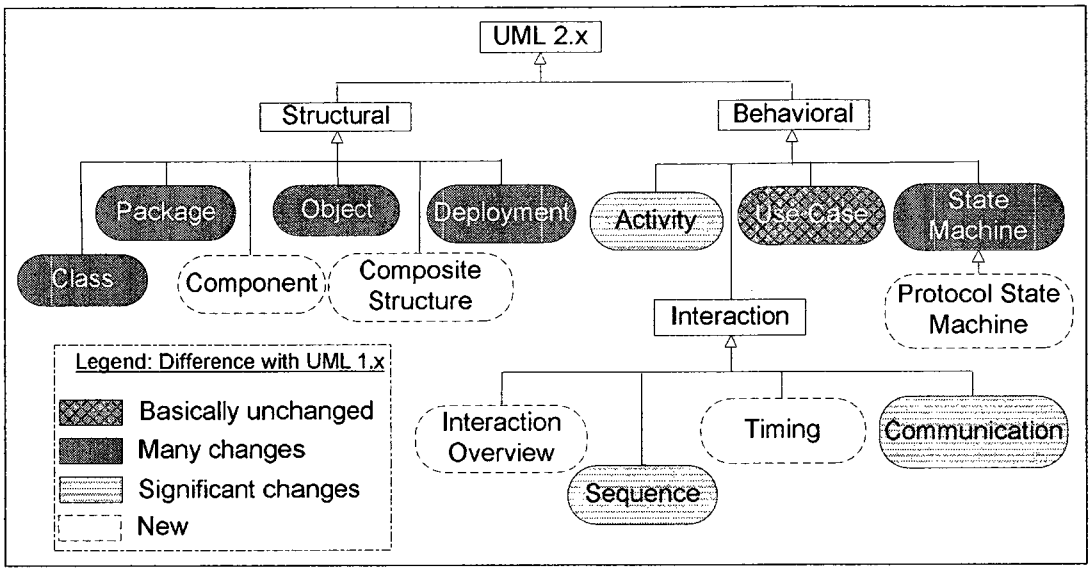


Figure 3: UML 2.1.1 Diagrams Taxonomy

Analysis of Real-time and Embedded Systems (MARTE) [51], and SysML [10].

UML standard has been revised many times, which results in the edition of many versions. In August 2003, a major revision has been issued in the form of UML 2.0 [52] in order to correct shortcomings discovered in the UML 1.x [38,53]. We base our work on the specification UML 2.1.2 [21]. The OMG distributes the UML standard as four specification documents [34]: The diagram interchange [54], the UML infrastructure [55], the UML superstructure [21], and the Object Constraint Language (OCL) [56]. The diagram interchange document provides a way to share the UML models between different modeling tools (elaborated by XML schema in the previous versions of UML). The UML infrastructure document defines its meta-model concept. The superstructure document consists of the definitions the user-level UML constructs. Finally, the OCL specification defines a simple language for writing constraints and expressions in UML models. Particularly, the superstructure document contains the description of the UML syntax, including the

diagrams specifications, and their underlying semantics explained in natural language. It defines 13 diagrams that can be classified into two main categories: structural diagrams and behavioral diagrams.

The structural diagrams category includes class, component, composite structure, deployment, object, and package diagrams. These diagrams show the static features of a model such as classes, associations, objects, links, and collaborations. These features provide the skeleton in which the dynamic elements of the model are executed. With respect to the behavioral diagrams category, it contains activity, use case, and state machine diagrams as well as a sub-class named interaction diagrams including communication, interaction overview, sequence, and timing diagrams. They show the behavioral features and the functionality of a system as well as the interactions between objects and resources modeled in the structural diagrams. There exists a strong relationship between the diagrams themselves and between the behavioral and the structural models. This relationship constitutes the basis for consistency of UML models. The classification of UML 2.x diagrams is shown in Figure 3, reported from [21]. It highlights the diagram taxonomy differences with respect to the UML version. For example, new diagrams are proposed in UML 2.x such as composite structure, interaction overview, and timing diagrams. Others are updated compared to their UML 1.x version like activity and sequence diagrams.

2.2.2 SysML: System Modeling Language

SysML [10] is a modeling language dedicated for systems engineering. It has its roots in UML 2. Indeed, it is a UML profile that reuses some UML packages and extends others

with SE specific features, in order to better fit the needed practices and methodologies. The mechanisms used in order to define these extensions are UML stereotypes, UML diagram extensions, and model libraries. The relationship between the two modeling languages UML and SysML is illustrated in Figure 4. The wide expressiveness of UML, its robustness and potential to be extended, as well as its large popularity have made it the best candidate to be customized for SE use [40]. In the process of customization, various UML elements that are not required in systems engineering have been excluded such as components that are more dedicated to model software. Currently, SysML is gaining increased popularity and many companies from various fields such as defense, automotive, aerospace, medical devices, and telecoms industries, are already using SysML, or are planning to switch to it very soon [40].

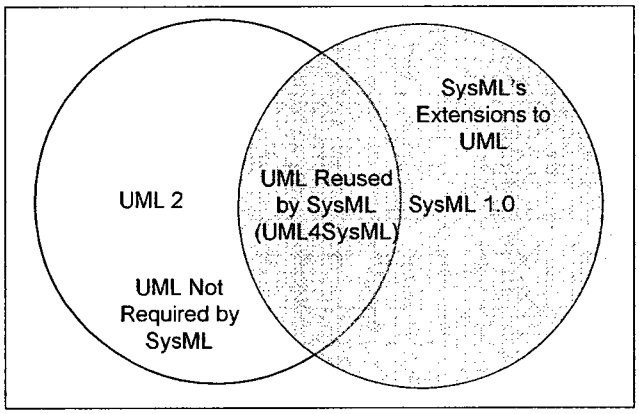


Figure 4: UML-SysML Relationship

SysML extends UML by adding new diagrams such as requirement and parametric diagrams and integrating new specification capabilities such as embedding allocation relationships into design in order to represent various types of allocation, including allocation of functions to components, logical to physical components, and software to hardware.

Furthermore, it has fundamentally modified some other UML diagrams such as class diagrams since they were no more suitable to SE. Instead, block definition and internal block diagrams have been introduced in order to replace class and composite structure diagrams respectively.

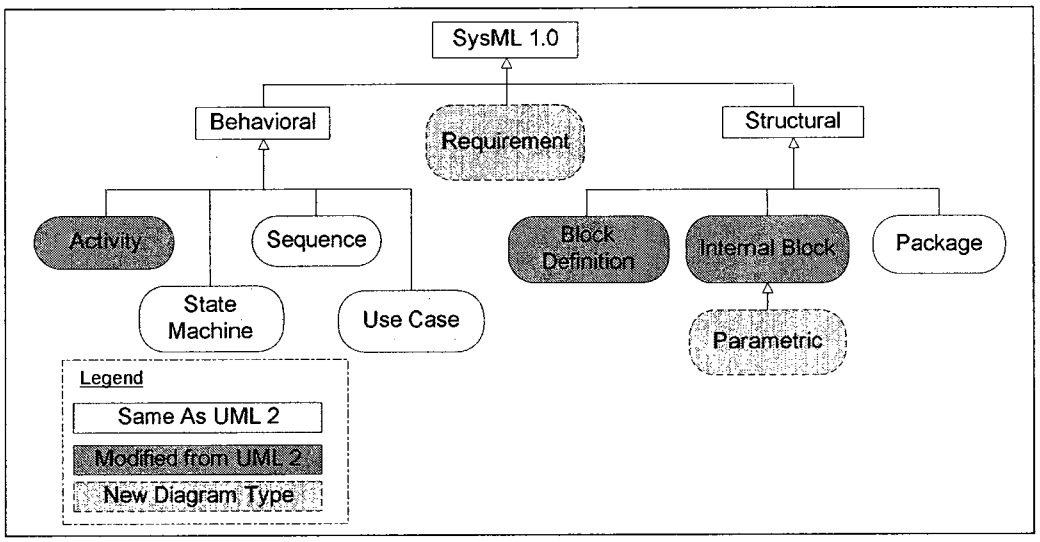


Figure 5: SysML 1.0 Diagrams Taxonomy

The most recent specification of OMG SysML v1.0 [10] has been published in September 2007. It describes, in the same way as the UML specification, the graphical constructs of the diagrams that can be used for the specification of systems architecture, structure, and behavior. It also explains informally, using text, the meaning of these constructs and the relationship among them. SysML diagrams cover four main perspectives of systems modeling: structure, behavior, requirements, and parametrics.

SysML diagrams taxonomy can be found in [10] and is reported in Figure 5. The correspondence between SysML and UML diagrams is summarized in Table 2.2.2 [57].

SysML Diagram	Purpose	UML Analog
Activity	Shows system behavior as control and data flows. Useful for functional analysis.	Activity
Block Definition	Shows system structure as components along with their properties, operations, and relationships.	Class
Internal Block	Shows the internal structures of components, including their parts and connectors.	Composite Structure
Package	Shows how a model is organized into packages, views, and viewpoints.	Package
Parametric	Shows parametric constraints between structural elements.	N/A
Requirement	Shows system requirements and their relationships with other elements.	N/A
Sequence	Shows system behavior as interactions between system components.	Sequence
State Machine	Shows system behavior as sequences of states that a component or interaction experience in response to events.	State Machine
Use Case	Shows systems' functions and the actors performing them.	Use Case

Table 1: Correspondence Between SysML and UML

Next, we detail activity, state machine, and sequence diagrams by describing their constructs and their corresponding meanings.

2.2.3 Activity Diagrams

Initially, UML 1.x defines activity modeling using activity graphs that are endowed with a statechart-based semantics. Later, this concept has been modified with the release of UML 2.0, where activity graphs have been replaced with activity diagrams. More precisely, UML 2.0 activity diagrams are endowed with a new semantics, which is independent of

statecharts semantics and supposedly based on Petri net semantics [21]. Generally, activity diagrams are used in modeling control flow and dataflow dependencies among the functions/processes that are defined within a system. They are widely used for instance in computational and business processes modeling and use cases detailing. Basically, an activity diagram is composed of a set of actions related in a specific order of invocation (or execution) by control flow paths, optionally emphasizing the input and the output dependencies using dataflow paths. An action represents the fundamental unit of a behavior specification and cannot be further decomposed within the activity. An activity may be composed of a set of actions coordinated sequentially, concurrently, or a combination of these. Furthermore, it may involve synchronization and/or branching. In order to enable these features, control nodes including fork, join, decision, and merge can be used. They support various forms of control routing. Additionally, it is possible to specify hierarchy among activities using call behavior action nodes, which may reference another activity definition. The diagram graphical artifacts corresponding to activity nodes and control flows are illustrated in Figure 6.

Concurrency and synchronization are modeled using forks and joins, whereas, branching is modeled using decision and merge nodes. While a decision node specifies a choice between two possible paths based on the evaluation of a guard condition (and/or a probability distribution), a fork node indicates the beginning of multiple parallel control threads. Moreover, a merge node specifies a point from where different incoming control paths have to follow the same path, whereas a join node allows multiple parallel control threads to synchronize and rejoin.

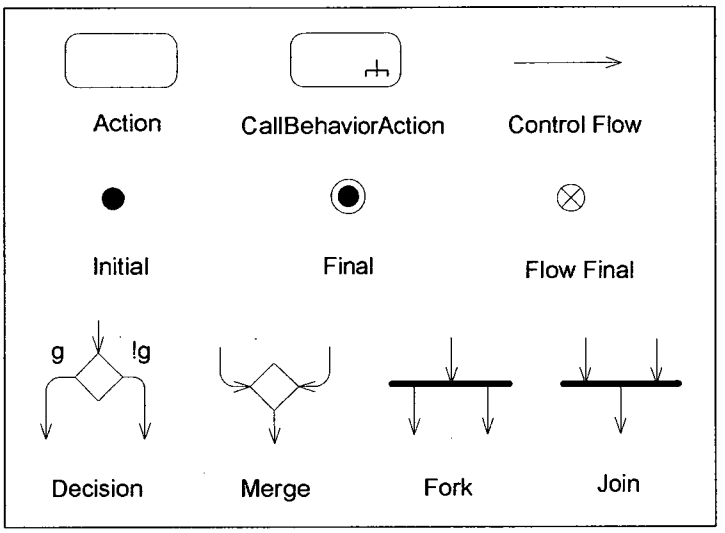


Figure 6: Activity Diagram Concrete Syntax

Activity diagrams behavior could be described in terms of tokens' flow. The UML superstructure [21] specifies basic rules for the execution of the various nodes by explaining textually how a token can be passed from one node to another. At the beginning, a first token starts flowing from the initial node and moves from one node to the next one(s) with respect to the foregoing set of control routing rules defined by the control nodes until reaching either an activity final or a flow final node. As soon as a given action receives a token, it starts executing and when it terminates, the token is removed from the corresponding node and then offered to the node's output edges. In the case of a fork node, the incoming token is duplicated as many times as there are outgoing paths. With respect to the join node, the traversal of the token downstream on the outgoing edge requires that all needed tokens on the incoming edges are available and merged into one token. More specifically, the join node requires a particular "join specification" requirement to be satisfied in order to issue a token on its single outgoing edge. By default, this token traversal condition requires to have

at least one token on each of the incoming edges of the join node. Finally, the first token that reaches an activity final node stops all the other active flows in the activity diagram. However, a token that reaches a final flow node ends only its corresponding control flow.

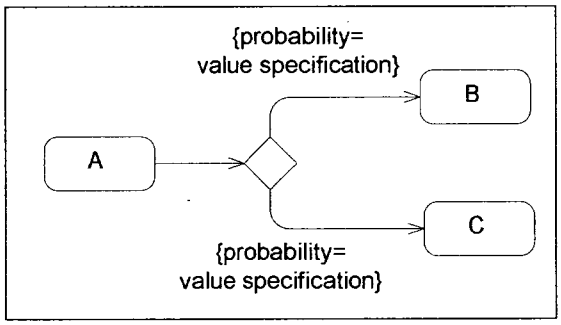


Figure 7: Probabilistic Decision Specification

As for SysML, apart from regular decision nodes, which describe choices between outgoing paths, it is possible to specify probabilistic behaviors in activity diagrams. There are two ways to use probabilities: On edges outgoing from a decision node and on output parameter sets (the set of outgoing edges holding data output from an action node) [10]. According to SysML specification, probability on a given edge expresses the likelihood that a token traverses this edge. An example of probabilistic decision node is shown in Figure 7.

2.2.4 State Machine Diagrams

State machine diagrams are used for modeling discrete behavior of entities building systems or software in terms of its transitions and states. Unlike classic Finite State Machine (FSM), UML state machine diagram may present hierarchy (i.e. clustering of states), where

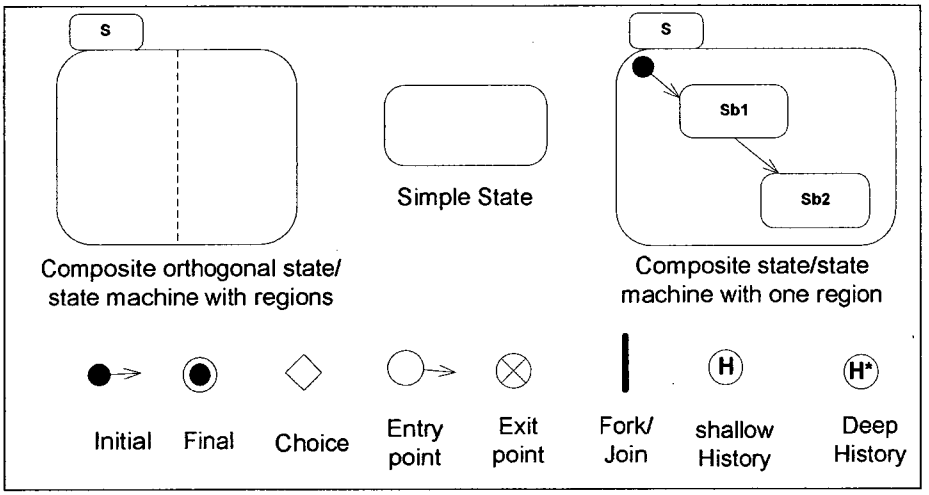


Figure 8: Syntax of State Machine Diagrams

some states can be decomposed into smaller units, namely substates, allowing the refinement of behavior. State machine diagrams are basically a structured aggregation of states, transitions, and a number of pseudo state components. A state can be either simple or composite (clustering a number of substates). A composite state can be either a simple composite state (with just one region) or an orthogonal state (with more than one region). A composite state has nested states that can be sequential or concurrent. A transition may be decorated using an event (trigger), a guard condition, and an effect (actions). A pseudostate is an abstraction that encompasses different types of transient vertices in the state machine graph [21]. Pseudostates include initial, fork, join, choice, shallow history, and deep history. The initial, join, fork, and choice pseudostates defined in state machines have a close meaning to those defined in activity diagrams. However, being pseudostates in state machine diagrams, the control tokens cannot reside in them (which is not the case in activity diagrams). Thus, pseudostates are not included in the execution configurations of state

machines and their main function serves to only model various forms of compound transitions. A shallow history is indicated by a small circle containing an 'H' and an asterisk "*" is added inside the shallow history symbol in order to denote deep history. Figure 8 illustrates the different syntactic elements of the state machine diagrams.

The state machine execution evolves in response to a set of incoming events (dispatched one at a time from an event queue). Each event may trigger one or more state machine transitions, which are in general associated with actions that are performed in response to the corresponding event. Once all triggered transitions are taken, a set of target states become active. Moreover, state machine diagrams are subject to the so-called run-to-completion semantics [21]. This means that events are processed one at a time and the next event will not be consumed until all actions related to the previous one are completed [40]. An event that is not triggering any transition cannot be consumed and is discarded. In this case, the state machine is said to stutter. As there can be more than one state active at a time, the state machine dynamics is configuration-based rather than state-based. A configuration denotes a set of active states and represents a stable point in the state machine dynamics while proceeding from one step to the next.

Concurrency in state machines can be described using orthogonal (AND) composite concurrent states, which may contain two or more concurrently active regions. Each region can be further clustering substates. A shallow History state represents the most recent active substate of its containing state (but not the substates of that substate). A deep history concept is an extension of shallow history in that it represents the most recent active configuration within the composite state that directly contains this pseudostate (e.g. the state

configuration that was active when the composite state was last exited). Consequently, it descends recursively into the most recently active substate until it reaches a basic active state. The syntax and semantics of UML state machine diagrams remained unchanged in SysML [10].

2.2.5 Sequence Diagrams

Among interaction diagrams, SysML includes only sequence diagrams. Both interaction overview and communication diagrams were excluded because of their overlapping functionalities and the fact that they are not offering other additional capabilities compared to sequence diagrams for system modeling applications. Furthermore, the timing diagram was also excluded due to its limited suitability for systems engineering needs [10].

Sequence diagrams describe the interactions within a system using communicating entities represented by lifelines. An interaction is a communication based on the exchange of messages in the form of operation calls or signals arranged in a temporal order [40]. Such entities are roles assumed by objects (blocks in SysML terminology). The body of a lifeline represents the life cycle of its corresponding object. A message is used for passing information and it can be exchanged between two lifelines in two possible modes: synchronous or asynchronous. There are four types of messages: operation call, signal, reply, and object creation/destruction. It is denoted by an arrow pointing from the sender to the receiver.

Apart from lifelines and messages, sequence diagrams define other constructs in order to organize the modeled interactions. The abstraction of the most general interaction

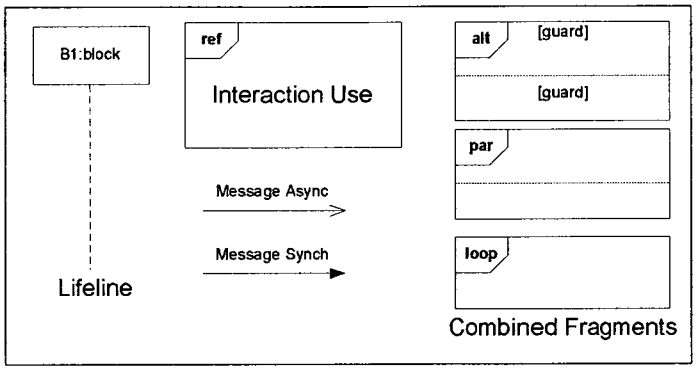


Figure 9: Syntax of Sequence Diagrams

unit is called *InteractionFragment* [21], which represents a generalization of a *Combined-Fragment*. The latter defines an expression of the former and it consists of an interaction operator and interaction operands [21]. Combined fragments enable the compact illustration of traces of exchanged messages. They are denoted using a rectangular frame with solid lines and a small pentagon in the upper left corner showing the operator and dashed lines separating the operands [40]. On the one hand, an interaction operand contains an ordered set of interaction fragments. On the other hand, the interaction operators include but not limited to conditional execution operator (denoted by *alt*), looping operator (denoted by *loop*), and parallel execution operator (denoted by *par*). Finally, sequence diagrams define interaction use constructs in order to reference other interactions. This allows the hierarchical organization of interactions and its decomposition into manageable units. Figure 9 illustrates a subset of sequence diagrams syntax.

2.3 Verification and Validation

According to the Recommended Practices Guide (RPG) [58] published by the Defense Modeling and Simulation Office (DMSO) in the United States Department of Defense, verification and validation techniques can be classified into four categories:

- Informal: They rely only on human interpretation and subjectivity without any underlying mathematical formalism. Even though they are applied with structure and guidelines, following standard policies and procedures is tremendous and it is not always effective. Among this category, we can enumerate audit, desk checking (called also self-inspection), inspection, and review.

- Static: They are applied to assess the static model design and the source code (implementation), without executing the model by a machine. They aim at checking the structure of the model, the dataflow and control flow, the syntactical accuracy, and the consistency. Therefore, in order to provide a full V&V coverage, they have to be applied in conjunction with dynamic techniques (defined in the next point). We can cite as examples cause-effect graphing, control flow analysis, dataflow analysis, fault/failure analysis, interface analysis, syntax analysis, and traceability assessment.

- Dynamic: In contrast to the static techniques, dynamic ones are based on the machine execution of the model in order to evaluate its behavior. They do not only examine the output of an execution but also watch the model as it is being executed. Thus, the insertion of additional code into the model is needed to collect or monitor the behavior during its execution. Debugging, execution testing, functional testing, and

visualization/animation are examples of dynamic techniques. Simulation turns out to be included in this category.

- Formal: These techniques are based on formal mathematical reasoning and proofs. Among them, we can find model-checking and theorem proving.

With respect to formal verification techniques, we focus mainly on automatic verification and in particular on probabilistic and non-probabilistic model-checking techniques, which are presented in the following section.

2.4 Formal Verification Techniques

Recently, formal verification methods have become more popular and usable by industry. There are two sorts of verification techniques: proof-based vs. model-based [59]. In a proof-based approach, the system is described using a set of formulae expressed in a given logic and the property to be verified is expressed as another formula. The verification consists in finding the proof that the specification formula holds in the set of description formulae. Conversely, in a model-based approach the system is described by a model that is verified against a specification property expressed generally in an appropriate logic or automaton [59]. In our case, we are interested in model-based approaches, namely temporal logic based model-checking. In the next section, we examine both probabilistic and non-probabilistic model-checking.

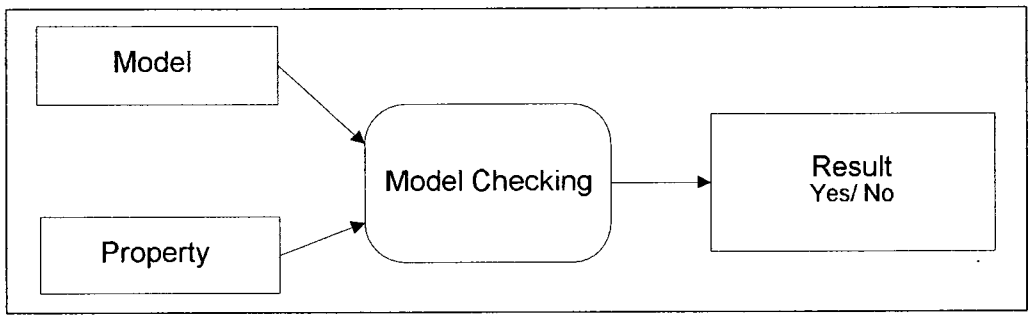


Figure 10: Model-Checking

2.4.1 Model-Checking

Model-checking is an automatic model-based verification approach [59], that has been successfully applied to automatically find errors in complex systems [60]. More specifically, it has been essentially used in software and hardware applications verification such as spacecrafts [61] and microprocessors [62]. Contrary to simulators, model-checkers perform a model analysis rather than a model execution. This technique consists of three main steps:

1. *Model the system*: Systems are represented using formalisms (describing the semantic model) such as automata or labeled transition systems with a generally finite number of states that have to be mapped into the model-checker's input language.

2. *Express the properties to be verified*: Temporal logics formulae that combine modal operators with boolean connectives and atomic propositions are generally used.

3. *Run the model-checker*: This checks whether or not the properties hold in the model.

The resulting output is either a positive answer (the system satisfies the specification) or a negative one (the system violates the specification). Most of the model-checkers generate

a trace of the system behavior as a counter example for the failed property. This approach is sufficient to reason about qualitative aspects of systems. Properties of the system that have to be verified have to be written in suitable temporal logic formulae. There are various temporal logics whose capabilities are tightly related to their expressiveness. Linear Temporal Logic (LTL) [63] and Computational Tree Logic (CTL) [64] are examples of prominent temporal logics.

In practice, due to its typical scalability issues, model-checking is generally limited to the verification of small to medium-scale design models. Nevertheless, numerous efforts have been deployed in order to address this problem in various ways, such as on-the-fly model-checking [65], symbolic model-checking [66], and distributed on-the-fly symbolic model-checking [67]. The main advantage of this technique is being fully-automatic as well as being based on the precision and rigor of formal methods. As examples of model-checkers we can cite SPIN [68], SMV [69], and NuSMV [70].

2.4.2 Probabilistic Model-Checking

As described earlier, non-probabilistic model-checking provides only a qualitative assessment of the property. However, some specification properties may fail qualitatively, if the scenarios satisfying these properties are very unlikely. Thus, probabilistic model-checking is needed in order to quantify the likelihood of satisfying a given property.

Probabilistic model-checking is a formal verification technique for the analysis of systems exhibiting probabilistic behavior. Typically, the input of non-probabilistic model-checker are transition systems, where transitions specify the evolution of the system's behavior while progressing from one state to another. In the case of a probabilistic model-checker, the transitions are labeled with probability information. The need of probabilistic model-checkers emerged from the existence of systems with stochastic behavior such as some algorithms or protocols, which need to make random choices when they execute. Moreover, probability enables modeling failure of unreliable systems and may be used for measuring system's performance. For instance, printing system performance can be measured by verifying the following property "the probability of the printing queue becoming full within a certain interval of time t is less than p". Among available probabilistic model-checkers we cite VESTA [71], PRISM [72], and MRMC [73].

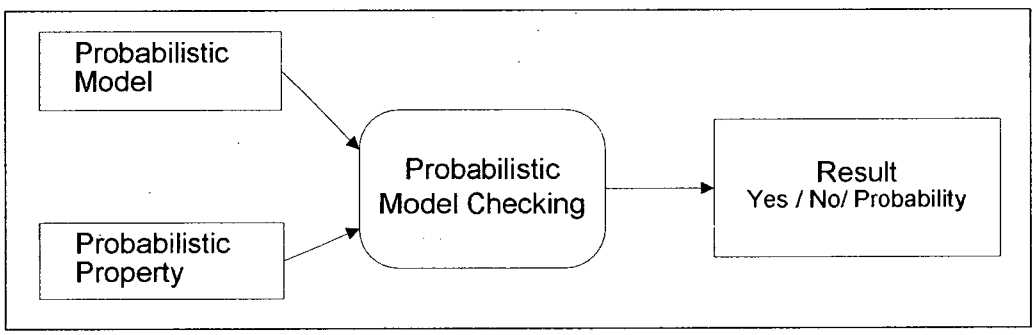


Figure 11: Probabilistic Model-Checking

Properties to be checked have to be expressed using suitable temporal logics. However, temporal logics such as CTL [64] and LTL [63] are not sufficient to express properties

involving the measure or the comparison of probability values. Thus, temporal logics extended with probability quantification are needed. For instance, model-checking systems specified as Markov Decision Processes (MDPs) requests expressing the properties using a suitable logic such as the Probabilistic Computation Tree Logic (PCTL) [74]. The latter is an extension of CTL [64] with probabilistic features. The next section is dedicated to the description of the probabilistic symbolic model-checker PRISM, which we use in order to support probabilistic behavior assessment.

2.5 PRISM: Probabilistic Symbolic Model-Checker

PRISM is a probabilistic symbolic model-checker developed first at the University of Birmingham [75] and then moved to the University of Oxford [76]. It has been widely applied to analyze systems from many application domains, including communication, multimedia, and security protocols [77]. The choice of PRISM among available probabilistic model-checkers is motivated by its wide application and other interesting features. Essentially, it is the only free and open source model-checker that supports MDP, which represents the probabilistic model of our interest in this thesis. Indeed, in real life systems, the notions of non-determinism, uncertainty, and probabilistic behavior are ubiquitous. The fact that MDP naturally encompass these features motivates our choice. Also, PRISM makes use of efficient data structures in order to have a compact model that are Multi-Terminal Binary Decision Diagram (MTBDD), sparse matrices, and a hybrid of these two. Moreover,

the applied numerical methods efficiency is reported to be acceptable in terms of execution time and memory consumption. A comprehensive comparative study of probabilistic model-checkers can be found in [78]. PRISM employs a state-based language that relies on the concept of reactive modules defined earlier by Alur and Henzinger in [79]. A comprehensive description of the PRISM language can be found within the manual published on PRISM website [80]. An informal overview of the semantics of PRISM input language can be found in [81]. In the following, we provide a brief informal description of the syntax of PRISM input language and its semantics.

2.5.1 Syntax

The input language of PRISM is based on the concepts of modules and variables. A given model is built using a set (possibly singleton) of modules and global variables declaration (if any). A module is composed of two main parts: local variables declaration and a set (possibly singleton) of commands. A command has three parts: a labeling action, a predicate guard over all variables of the model, and a set of updates over the local variables of the containing module. A given command has the following form:

$$\texttt{[action] guard} \rightarrow p_1 : update_1 + \cdots + p_n : update_n ;$$

where p_i is the probability of occurrence of the corresponding update $update_i$. In addition to the specification of models, PRISM supports rewards (known also as costs) mechanism. A reward structure is composed of reward items specified in association with the system model. A reward item can be assigned to one or many states and/or transitions in the model. A reward structure has the following form:

```
rewards ''rewardname''
guard : reward;
  [action] guard : reward;
endrewards
```

where `rewardname` is the identifier of the reward and `reward` is a real-valued expression. The reward value depends on the evaluation of the expression. Any state or transition that does not satisfy the guard and the action label (if any) of a reward item will have no reward assigned to it. If states and/or transitions satisfy multiple reward items, the assigned reward is the sum of the rewards for all the corresponding reward items. This mechanism allows the verification of properties based on reward such as the computation of the expected cumulated cost or reward before a certain state is reached [82]. Rewards allow wider range of quantitative measures relating to model behavior such as "expected time", "expected number of lost message", or "expected power consumption" [83].

2.5.2 Semantics

Each module within a PRISM model can be thought of as a process running concurrently with other modules and may interact with them. The values of the local variables define the local state of their enclosing module, whereas the commands encode the module's dynamics. The set of local states of all the modules defines the global state of the system. The state space is therefore the set of all possible valuations of the local and global variables. Given a model with a set of local and global variables, the initial state can be specified by assigning specific initial values to the variables. The set of commands defines the possible transitions between the states of a given model.

The actions labeling the commands are used for synchronization between a given set of commands located in different modules. The guard, which generally represents a predicate over the variables of the model, plays the role of a trigger for the enclosing command. When it is evaluated to $true$, the command is potentially enabled. Generally, a guard can be satisfied by a set of states. A given state s satisfies a predicate over the variables in V if by substituting all the variables in the predicate with their respective values defining the state s, the predicate evaluates to $true$. If S_c denotes the set of states satisfying g defined in the command c, each update unit of the command u_j is applied over the values of the variables defining the state $s \in S_c$. It consists of the assignment of new values to the variables resulting from the evaluation of some expression. An update is generally a probabilistic choice between a set of possible updates, where the probability values of all the updates for a single command have to sum up to one.

Modules building a given PRISM model can be composed in parallel according to three possible parallel composition modes: Full parallel composition with synchronization on common actions, full asynchronous parallel composition, and a restricted parallel composition with synchronization limited to only a specific set of actions. In the case of synchronous mode, all the commands that synchronize on the same action have to be simultaneously enabled to be executed. In an asynchronous mode, each command is executed independently of the others. The execution of the enabled command results in the occurrence of the specified updates and consequently in the modification of the global state of the model.

PRISM supports three types of probabilistic models [84]: Discrete-Time Markov Chain

(DTMC), Continuous-Time Markov Chain (CTMC), and Markov Decision Process (MDP). More precisely, the choice of the commands to be executed depends on both the guard evaluation and the specified model type. DTMCs and CTMCs are deterministic models. However, MDPs, which are generalizations of DTMCs, contain non-determinism and are limited to discrete-time like DTMCs. In the case of MDP, if there are multiple candidate commands, the choice of the one to be executed at a certain point in time is done non-deterministically. This choice is resolved by an entity called scheduler or adversary. In the following section, we review the formal definition of the probabilistic and non-probabilistic models that we encounter in this thesis.

2.6 Formal Models

In this section, we provide a description of the formal models that we may use later. We start by providing a formal definition of labeled transition systems in Section 2.6.1 and then we study briefly the probabilistic labeled transition systems in Section 2.6.2. Afterward, we discuss in Section 2.6.3 the Markovian models and focus on MDP.

2.6.1 Labeled Transition Systems

The operational behavior of systems is usually given in terms of transition systems. From an operational point of view, a system can be seen as a set of states (processes or configurations) and a set of transitions between these states. Usually, the transitions are labeled with external and/or internal actions (or events).

Definition 2.6.1. A Labeled Transition System (LTS) is a tuple $(S, s_0\ L, T)$ such that:

- S is a set of states,
- s_0 is the initial state,
- L is a set of actions,
- T: $S \times L \times S$ is a transition relation such that (s, a, s') is the transition obtained from a state s where an action a is selected and the target state s' is reached. □

A path through the LTS represents an execution sequence in the behavior of the corresponding system.

2.6.2 Probabilistic Labeled Transition Systems

Probabilistic Labeled Transition Systems (PLTS) are an extension of LTS where transitions are further annotated with probability values in the range $[0, 1]$. The probabilistic information denotes the likelihood of the transition to occur and it is generally introduced in transition systems in order to quantify the non-determinism.

Definition 2.6.2. A PLTS is a tuple (S, L, P) such that:

- S is a set of states,
- L is a set of actions,
- P: $S \times L \times S \longrightarrow [0, 1]$ is a probabilistic transition function such that $\forall a \in L$ and $s, s' \in S$, $P(s, a, s')$ is the probability that action a is selected from the state s and

the target state s' is reached with the condition that $\sum_{s' \in S} P(s, a, s') \in \{0, 1\}$. If the sum is equal 1 s is said to be stochastic, otherwise s is said to be absorbing or terminal. □

The choice of the action a can be either internal or external. Once an action a is chosen, the probabilistic transition function P will be used to select which transition to take.

2.6.3 Markovian Models

Markov chains are a probabilistic extensions of finite automata. For a given state, a finite automaton only specifies the possible next states within the next step, whereas a Markov chain precisely specifies the probability of the next transitions to each of the other states. The well-known property that characterizes Markov chains is the memoryless property: The future state (behavior) of the system is independent of the past (previously visited nodes) but it depends only on the current state. The latter is also called history-independence.

Markov decision processes describe both probabilistic and non-deterministic behaviors. They are used in various areas, such as robotics [85], automated control [86], and economics [87]. A formal definition of MDP is given in the following [84]:

Definition 2.6.3. A Markov decision process is a tuple M=(S, s_0, Act, $Steps$), where:

- S is a finite set of states,
- $s_0 \in S$ is the initial state,

- Act is a set of actions,

- $Steps{:}S \to 2^{Act \times Dist(S)}$ is the probabilistic transition function that assigns to each state s a set of pairs $(a, \mu) \in Act \times Dist(S)$ where $Dist(S)$ is the set of all probability distributions over S, i.e. the set of functions $\mu{:}\, S \to [0,1]$ such that $\sum_{s \in S} \mu(s) = 1$. □

We write $s \xrightarrow{a} \mu$ if and only if $s \in S$, $a \in Act$, and $(a, \mu) \in Steps(s)$ and refer to it as a step or a transition of s. The distribution μ is called an a-successor of s. For a specific action $\alpha \in Act$ and a state s, there is a single α-successor distribution μ for s. In each state s, there is a non-deterministic choice between elements of $Steps(s)$ (between the actions). Once an action-distribution pair (a, μ) is selected, the action is performed and the next state, say s', is determined probabilistically according to the distribution μ, i.e. with a probability $\mu(s')$. In the case of μ of the form $\mu^1_{s'}$ (meaning the unique distribution on s', i.e. $\mu(s') = 1$), we denote the transition by $s \xrightarrow{a} s'$ rather than $s \xrightarrow{a} \mu^1_{s'}$.

2.7 Conclusion

In this chapter, we introduced some important aspects that represent the background for our thesis topic. Specifically, some of the presented concepts are crucial for the understanding of the remaining chapters. In the next chapter, we present the reviewed related works on the assessment of design models that are expressed using UML and SysML.

Chapter 3

Related Work

This chapter discusses relevant research proposals from the literature on the V&V of design models expressed using either UML 2.x [21] or SysML 1.0 [10]. The state of the art with respect to UML design includes a significant number of initiatives. However, there is scarcely works on the analysis of OMG SysML models. This is mainly due to the youth of SysML. UML 2.0 has provided several improvements and modifications to the syntax and semantics of the diagrams compared to the previous UML versions. Thus, we focus our interest on proposals targeting mainly UML 2.x models but we briefly discuss the most prominent ones on UML 1.x.

This chapter is organized as follows. Section 3.1 presents the related works on the V&V of UML models. Therein, various proposed techniques such as model-checking, static analysis, simulation, and theorem proving are highlighted. Section 3.2 is dedicated to the review of the most relevant research initiatives on the V&V of SysML models. Section 3.3 presents a set of software engineering metrics that can be found in the literature.

Section 3.4 provides a detailed overview of propositions on the performance analysis of UML/SysML design. Finally, Section 3.5 presents the existing works on the formalization of UML/SysML activity diagrams semantics.

3.1 Verification and Validation of UML Models

There is a large body of research proposals that target the analysis of UML-based design models analysis. There are works that focus on the analysis of UML diagrams from consistency and data integrity point of views. The consistency issue is related to the fact that various artifacts representing different aspects of the system should be properly related to each other in order to form a consistent description of the developed system. Though these aspects are important, the present thesis focus on a no less important issue, which is the verification of the conformance of behavioral UML design to its stated requirements. Thus, we focus in this chapter on research proposals that fall in this area.

Some of the state of the art initiatives propose V&V approaches that jointly consider a set of diagrams, whereas the majority focus on a single diagram and particularly on a subset of its semantics. One can easily note that state machine diagrams have gained most of the attention. In addition, a single V&V technique is generally proposed (e.g. automatic formal verification, theorem proving, or simulation). Furthermore, there are works proposing a formalization of the semantics of the considered diagram, which is subjected to formal verification. However, other proposals prefer a direct mapping into the specification language of a particular verification tool.

There are various related work on simulation of UML design models. For instance, simulation is proposed in [88,89] for performance analysis, and in [90–92] for model execution and debugging. For instance, Hu and Shatz [91] propose to convert UML statecharts into Colored Petri Nets (CPN). Then, collaboration diagrams are used for connecting different model objects so that a single CPN of the whole system is obtained. Then, the Design/CPN tool is used for performing simulation. Sano et al. [92] propose a mechanism by which model simulation is performed on four behavioral diagrams, namely, statechart, activity, collaboration, and sequence diagrams.

A surge of interest has been expressed in the application of model-checking to the verification and validation of UML models. We can find a number of related papers including [93–100] for UML 1.x and [101–108] for UML 2.x. Therein, a number of V&V framework tools are proposed such as TABU [104], HIDE [98], PRIDE [96], HUGO [94], Hugo-RT [93], and VIATRA [109].

Gnesi and Mazzanti [101] provide an interpretation of a set of communicating UML 2.0 state machine diagrams in terms of Doubly Labeled Transition System (L^2TS). The state/event-based temporal logic μUCTL [110] is used for the description of the dynamic properties to be verified. A prototype environment is developed around the UMC on-the-fly model-checker [111]. Guelfi and Mammar [102] propose to verify UML activity diagrams extended with timed characteristics. This is based on the translation of activity diagrams into PROMELA code, the input language of the SPIN model-checker [68]. Eshuis [103] proposes two translations of UML activity diagrams into finite state machines, which are input to the NuSMV model checker [112]. The first is a requirement-level translation and

the second is implementation-level. Both translation are inspired by existent statechart semantics. The latter is inspired by the OMG statecharts semantics. The resulting models are used in model-checking data integrity constraints in activity diagrams and a set of class diagrams specifying the manipulated data. Activity diagrams are first transformed into activity hypergraphs by the means of transformation rules. Then, translation rules are defined for activity hypergraphs in order to obtain the NuSMV code. However, the considered activity semantics excludes multiple instances of activity nodes. Beato et al. [104] propose the verification of UML design models consisting of state machine and activity diagrams by means of formal verification techniques using the Symbolic Model Verifier (SMV) [69]. The diagrams are encoded into the SMV specification language via the XML Metadata Interchange (XMI) format. Mokhati et al. [105] propose the translation of UML 2.0 design models consisting of class, state machine, and communication diagrams into Maude language [113]. Properties expressed using Linear Temporal Logic (LTL) [63] are verified on the resulting models using Maude's integrated model-checker. Only basic state machine and communication diagrams with the most common features are considered. Xu et al. [106] propose an operational semantics for UML 2.0 activity diagrams by transforming it into Communicating Sequential Processes (CSP) [114]. The resulting CSP model is then used for the analysis using the model-checker FDR. Kaliappan et al. [107] propose to convert UML state machine into PROMELA code and map sequence diagrams into temporal properties for the verification of communication protocols using SPIN model-checker [68]. Engels et al. [108] propose Dynamic Meta Modeling (DMM) technique, which is based on graph transformation techniques, in order to define and analyze the semantics of UML

activity diagrams based on its meta-model. The considered activity diagrams are limited to workflow modeling thus imposing restrictions on their expressiveness as well as on their semantics since workflows have to adhere to specific syntactic and semantic requirements. DMM is used to generate the transition systems underlaying the semantic model of activity diagrams. The analysis is limited to the verification of the soundness property of workflows. The latter is expressed using CTL [64] temporal logic and then input into the GRaphs for Object-Oriented VErification (GROOVE) tool [115] in order to apply model-checking over the generated transition systems.

It is worthy to mention other related work concerning UML 1.x statecharts [93,99,100]. Latella et al. [99] as well as Mikk et al. [100] propose a translation of subsets of UML statecharts into SPIN/PROMELA [68] using an operational semantics as described in [116]. The approach consists in translating the statechart into an Extended Hierarchical Automaton (EHA). Then, the latter is modeled into PROMELA and subjected to model-checking. Knapp et al. [93] present a prototype tool, HUGO/RT, for the automatic verification of a subset of timed state machines and time-annotated UML 1.x collaboration diagrams. The model-checker UPPAAL [117] is used to verify state machine diagrams (compiled to timed automata) against the requirements described in the collaboration diagrams (compiled to observer timed automaton).

Concerning sequence diagrams, Grosu et al. [118] propose non-deterministic finite automata as their semantic model. A given diagram is translated into a hierarchical automaton and both safety and liveness Büchi automata are derived from it. These automata are subsequently used to define a compositional notion of refinement of UML 2.0 sequence

diagrams. Li et al. [119] define a static semantics for UML interaction diagrams to support verifying the well-formedness of interaction diagrams. The dynamic semantics is interpreted as a trace-based terminated CSP process that is used to capture the finite sequence of message calls. Cengarle et al. [120] propose a trace-based semantics for UML 2.0 interactions. Störrle [121] presents a partial order semantics for time constrained interaction diagrams. Korenblat et al. [122] present a formalization of sequence diagrams based on the π-calculus [123]. The state machine diagrams of the interacting objects are considered in order to identify feasible occurrence of sequences of messages. Accordingly, objects in a sequence diagram are modeled as π-calculus [123] processes and the exchanged messages as communications among these processes. The semantics of sequence diagram is defined based on the structured operational semantics of π-calculus [123]. The corresponding semantic model is a Labeled Transition System (LTS) that is used to generate the input of model-checker.

Some other initiatives use a model-checker and a theorem prover [124]. The latter proposes a V&V framework for UML 1.0 integrating multiple formalisms such as the Symbolic Analysis Laboratory (SAL) [125], CSP, and Higher Order Logic (HOL) [126] via the object-oriented specification language Object-Z [127]. However, it needs the developer intervention to choose the formalism to be applied, which constitutes a major inconvenient.

Other initiatives prefer the use of model-checking coupled with simulation [128, 129]. They emerged in the context of the IST *Omega* project[1]. Ober et al. [129] describe the application of model-checking and simulation techniques in order to validate the design

[1] `http://www-omega.imag.fr/`

models expressed in the *Omega* UML profile. This is achieved by mapping the design to a model of communicating extended timed automata in IF [130] format (an intermediate representation for asynchronous timed systems developed at *Verimag*). Properties to be verified are expressed in a formalism called UML observers. In [128], Ober et al. present a case study of validation of the control software of the Ariane-5 launcher. The experiment is done on a representative subset of the system, in which both functional and architectural aspects are modeled using *Omega* UML 1.x profile. The IFx, a toolset built on top of the IF environment, is used for the V&V of both functional and scheduling-related requirements using simulation and model-checking functionalities.

Some proposals concentrate only on ascribing a formal semantics to the selected UML diagram. An extensive survey on the formal semantics of UML state machine can be found in Crane and Dingel [131]. For instance, Fecher et al. [132] present an attempt to define a structured operational semantics for UML 2.0 state machine. Similarly, Zhan and Miao [133] propose a formalization of its semantics using the Z language. This allows the transformation of the diagram into the corresponding Flattened REgular Expression (FREE) state model. The latter is used to identify inconsistency and incompleteness and generate test cases.

3.2 Verification and Validation of SysML Models

Since SysML is relatively young, we can find only few initiatives that are concerned with the V&V of SysML design models [134–139]. Most of the proposals are concerned with

the use of simulation either directly of via Petri net formalism.

Viehl et al. [134] present an approach based on the analysis and simulation applied to a System-On-Chip (SoC) design specified using UML 2.0/SysML. Time-annotated sequence diagrams together with UML structured classes/SysML assemblies are considered for describing the system architecture. Moreover, the SysML version considered therein is not the one standardized by the OMG. Huang et al. [137] propose to apply simulation. A mapping of SysML models into their corresponding simulation meta-models is proposed. The latter is used to generate the simulation model. Similarly, Paredis and Johnson [138] propose to apply graph transformation approach to map the structural descriptions of a system into the corresponding simulation models.

Wang and Dagli [135] propose the translation of mainly SysML sequence diagrams and partly activity and block definition diagrams into Colored Petri Nets (CPNs). The resulting CPNs represent the executable architecture to be subjected to static and dynamic analyses (simulation). The behavior obtained by simulation is generated in the form of Message Sequence Charts (MSC), which are compared to sequence diagrams. This verification is based on the visual comparison of the simulated behavior against the intended behavior. Moreover, the assessment of non-functional requirements are not considered.

Carneiro et al. [136] consider SysML state machine diagrams annotated with MARTE profile [51]. This diagram is mapped manually into Timed Petri Nets with Energy constraints (ETPN) for the estimation of the energy consumption and execution time for embedded real-time systems. The analysis is performed using a simulation tool for timed Petri nets.

Jarraya et al. [139] consider the mapping of a synchronous version of time-annotated SysML activity diagrams into Discrete-Time Markov Chains (DTMC). The latter model is input to the probabilistic model-checker PRISM for the assessment of functional and non-functional properties.

3.3 Software Engineering Metrics

We found in the literature very few initiatives on the use of metrics in systems engineering. For instance, Tugwell et al. [140] outline the importance of metrics especially those related to complexity measurement. In the literature, many object-oriented metrics have been proposed in order to assess quality of UML design models from a structural perspective. Software metrics have been proposed for assessing the quality of UML structural diagrams. In the following, we review the main proposed metrics suites in the field of software engineering focusing on UML class and package diagrams.

The NASA technical report [141] discusses the use of metrics in order to measure the quality attributes of class and package diagrams. These metrics are classified in two categories. The first category deals with traditional metrics such as cyclomatic complexity while the second one is specifically related to object-oriented systems such as coupling, depth of inheritance, and the number of children.

Chidamber and Kemerer [142] propose a set of six metrics that aims to measure the complexity of diagrams according to different quality attributes such as maintainability, reusability, etc. Only three of these metrics can be applied to UML class diagrams. First,

the Coupling Between Object Classes (CBO) metric measures the level of coupling among classes in the diagram. High coupling harms design modularity and prohibits reuse and maintainability [141]. Second, the Depth of Inheritance Tree (DIT) metric represents the length of inheritance tree from a class to its root class. A deep class in the tree inherits a relatively high number of methods, which in turn increases its complexity. Finally, the Weighted Methods per Class (WMC) metric is the summation of the complexity of all methods in the class. If the complexity of each method is considered as unity, WMC returns the number of methods in the class. A high WMC value denotes high complexity and less reusability.

Brito et al. [143] propose a set of metrics such as enacapsulation, inheritance, polymorphism. The Metrics for Object-Oriented Design (MOOD) metrics suite can be applied to UML class diagrams. In the following, the relevant metrics for our case are the following:

- Method Hiding Factor (MHF) metric measures the encapsulation degree in a class. It is the ratio of the sum of hidden methods (private and protected) to the total number of methods defined in each class (public, private and protected). A high value of MHF (for instance unity) indicates that all the methods in the class are hidden and thus the considered class is not accessible and consequently not reusable. A null value for MHF assumes that all the methods of the class are public, which hinders encapsulation.

- Attribute Hiding Factor (AHF) metric represents the average of the invisibility of

attributes in the class diagram. It is computed as the ratio of the sum of hidden attributes (private and protected) for all the classes to the sum of all defined attributes (public, private, and protected). A high AHF value indicates the degree of data hiding.

- Method Inheritance Factor (MIF) and Attribute Inheritance Factor (AIF) metrics measure the class inheritance degree. MIF is calculated as the ratio of all inherited methods in the class diagram to total number of methods (defined and inherited) in the diagram. AIF is calculated as the ratio of all inherited attributes in the class diagram to the total number of attributes (defined and inherited) in the diagram. A zero value indicates no inheritance at all, which may be a flaw unless the class is a base class in the hierarchy.

- Polymorphism Factor (POF) metric measures methods overriding in a class diagram. It is the ratio between the number of overridden methods in a class and the maximum number of methods that can be overridden in the class. An appropriate use of polymorphism (low POF) should decrease the defect density as well as rework.

- Coupling Factor (COF) metric measures the coupling level in a class diagram. It is the ratio between the actual couplings among all classes and the maximum number of possible couplings among all the classes in the diagram. A class is coupled to another class if methods of the former access members of the latter. High values of COF indicate tight coupling, which increases the complexity and hinders maintainability and reusability.

Li and Henry [144] propose a metrics suite to measure several class diagram internal quality attributes such as coupling, complexity, and size. Two proposed metrics can be applied on UML class diagrams: Data Abstraction Coupling (DAC) and SIZE2. The DAC metric calculates the number of attributes in a class that have another class as their type (composition). It is related to the coupling complexity due to the existence of abstract data types (ADT). The more ADTs are defined within a class, the higher is the complexity due to coupling. The SIZE2 metric measures class diagram size. It is computed as the sum of the number of local attributes and local methods defined in a class.

Lorenz and Kidd [145] propose a set of metrics that measures the static characteristics of software design. A set of metrics measuring the size are proposed. Public Instance Methods (PIM) counts the number of public methods in a class. Number of Instance Methods (NIM) counts the number of all methods (public, protected and private) in a class. Finally, Number of Instance Variables (NIV) counts the total number of variables in a class. Furthermore, another set of metrics is proposed that measures the class inheritance usage degree. Number of Methods Overridden (NMO) gives a measure of the number of methods overridden by a subclass. Number of Methods Inherited (NMI) gives the total number of methods inherited by a subclass. Number of Methods Added (NMA) counts the number of methods added in a subclass. Specialization index (SIX) uses the NMO and DIT [142] metrics in order to calculate the class inheritance utilization.

Robert Martin [146] proposes a set of three metrics applicable to UML package diagrams. This set of metrics measures the interdependencies among packages. Highly interdependent packages tend to be not flexible since they are hardly reusable and maintainable.

The three defined metrics are Instability, Abstractness, and Distance from Main Sequence (DMS). The Instability metric measures the level of instability of a package. A package is unstable if it depends more on other packages than they depend on it. The Abstractness metric is a measure of the package's abstraction level, which depends on its stability level. Finally, the DMS metric measures the balance between the abstraction and instability of a package.

Bansiya et al. [147] define a set of five metrics to measure several object-oriented design properties such as data hiding, coupling, cohesion, composition, and inheritance. In the following, we present only those metrics that can be applied to UML class diagrams. The Data Access Metric (DAM) measures the level of data hiding in the class. DAM is the ratio of the private and protected (hidden) attributes to the total number of defined attributes in the class. The Direct Class Coupling (DCC) metric counts the total number of classes that a class is directly related to. The Measure Of Aggregation (MOA) metric computes the number of attributes, which types are classes (composition) defined in the same model.

Among a panoply of works on the subject, Briand et al. [148] propose a metrics suite to measure coupling among classes in a given class diagram. These metrics determine each type of coupling and the impact of each relationship type on the class diagram quality. Numerous types of coupling occurrences in a class diagram are covered. These types of relationships include coupling to ancestor and descendent classes, composition, class-method interactions, and import/export coupling. Genero et al. [149] illustrate the use of several object-oriented metrics to assess the complexity of a class diagram at the initial phases of

the development life cycle. Moreover, a set of metrics are proposed targeting UML relationships, mainly aggregations, associations, and dependencies in order to identify class and package diagrams complexity. Among the proposed metrics, we cite for instance, the Number of Associations of a Class metric (NAC), which represents the total number of associations that a class has in a class diagram. The Number of Dependencies Out (NDepOut) (respectively In (NDepIn)) metric is defined as the number of classes on which a given class depends (respectively that depend on a given class). Finally, quantitative design analysis on UML models is addressed in Gronback [150] where techniques such as audits and metrics are proposed.

3.4 Performance Analysis

Performance modeling and assessment of software and systems during the development process is still an active area of research. Particularly, we are interested in the analysis of SysML-based design, where the prediction and assessment of the performance coupled with V&V are important for a successful system solution.

In the literature, there are three major performance analysis techniques: analytical, simulative, and numerical [151]. Among various performance models, we cite four classes of performance models that can be distinguished: Queueing Networks (QN) [152], Stochastic Petri Nets (SPN) [153], Markov Chains (MC) [152], and Stochastic Process Algebras (SPA) [151]. The subsequent review is structured according to the used performance model.

Queuing Networks (QN) are applied to model and analyze resource sharing systems.

This model is generally analyzed using simulation and analytical methods. Within the family of queuing networks, we can find two classes: deterministic and probabilistic. Among the initiatives targeting the analysis of design models (including UML/SysML) using deterministic models, we cite for instance Wandeler et al. [154]. The latter applies modular performance analysis based on the Real-Time Calculus and use annotated sequence diagrams. In the context of probabilistic QN, [155–157] address performance modeling and analysis of UML 1.x design models. Cortellessa et al. [156] propose Extended Queuing Network (EQN) for UML 1.x sequence, deployment, and use case diagrams. Layered Queueing Network (LQN) is proposed by Petriu et al. [157] as the performance model for UML 1.3 activity and deployment diagrams. The derivation is based on graph-grammar transformations that are known to be complex and require a large number of transformation rules. In contrast to our work, time annotations are based on the UML SPT profile [158]. Balsamo et al. [155] target UML 1.x use case, activity, and deployment diagrams annotated according to the UML SPT profile. These diagrams are transformed into a multi-chain and multi-class QN models, which impose restrictions on the design. Specifically, activity diagrams should not contain forks and joins otherwise the obtained QN can only have an approximate solution [152].

Various research proposals such as [159–163] consider Stochastic Petri Net (SPN) models for performance modeling and analysis. King et al. [159] propose Generalized Stochastic Petri Nets (GSPN) as performance model for combined UML 1.x collaboration and statechart diagrams. Numerical evaluations of the derived Petri net are performed in order to approximately evaluate the performance. López-Grao et al. [160] present a prototype

tool for performance analysis of UML 1.4 sequence and activity diagrams based on the Labeled Generalized Stochastic Petri Nets (LGSPN). In the same trend, Trowitzsch et al. [162] present the derivation of Stochastic Petri Nets (SPNs) from restricted version of UML 2.0 state machines annotated with the SPT profile.

Alternatively, Stochastic Process Algebras (SPA) are also extensively used for performance modeling of UML design models [164–171]. Pooley [170] considers a systematic transformation of collaboration and statechart diagrams into the Performance Evaluation Process Algebra (PEPA). Canevet et al. [168] describe a PEPA-based methodology and a toolset for extracting performance measurements from UML 1.x statechart and collaboration diagrams. The state space generated by the PEPA workbench is used to derive the corresponding Continuous-Time Markov Chain (CTMC). In a subsequent work, Canevet et al. [167] present an approach for the analysis of UML 2.0 activity diagrams using PEPA. A mapping from activity to PEPA net model is provided, however, discarding join nodes. Tribastone and Gilmore propose a mapping of UML activity diagrams [164] and UML sequence diagrams [165] annotated with MARTE [51], the UML profile for model-driven development of Real Time and Embedded Systems, into the stochastic process algebra PEPA. Another type of process algebra is proposed by Lindemann et al. [169], which is the Generalized Semi-Markov Process (GSMP). The UML 1.x state machine and activity diagrams are addressed. Trigger events with deterministic or exponentially distributed delays are proposed for the analysis of timing in UML state diagrams and activity diagrams. The work presented by Bennett et al. [166] propose the application of performance engineering of UML diagrams annotated using the SPT UML profile. System behavior scenarios are

translated into the stochastic Finite State Processes (FSP). Stochastic FSP are analyzed using a discrete-event simulation tool. No algorithms for the inner workings of the approach is provided. Tabuchi et al. [171] propose a mapping of UML 2.0 activity diagrams annotated with SPT profile into Interactive Markov Chains (IMC) intended for performance analysis. Some features in activity diagrams are not considered such as guards on decision nodes and probabilistic decisions, while the duration of actions is expressed using a negative-exponential distribution of the delay.

More recently, Gallotti et al. [172] focus on model-based analysis of service compositions by proposing the assessment of their corresponding non-functional quality attributes, namely performance and reliability. The high-level description of the service composition given in terms of activity diagrams is employed to derive stochastic models (DTMC, MDP, and CTMC) according to the verification purpose and the characteristics of the activity diagram. The probabilistic model-checker PRISM is used for the actual verification. However, there is neither clear explanation of the translation steps nor a PRISM model example resulting from the proposed approach.

3.5 Formal Semantics for Activity Diagrams

Presently, to the best of our knowledge, there are no proposals on the formal semantics of SysML activity diagrams. Regarding the formalization of UML activity diagrams, some initiatives such as [173–176] are within UML 1.x. Other proposals [171, 177–181] study

the formal semantics for UML 2.0 activity diagrams and propose a mapping into an existing formalism with well-defined semantics. In the sequel, we present the related work divided into four distinct approaches given the targeted semantic domain of the mapping: (1) Mapping activity diagrams into a process algebra, (2) mapping activity diagrams into Petri-nets, (3) graph transformation techniques, (4) mapping activity into Abstract State Machines (ASM).

The ASM formalism is proposed in [173, 177]. Böger et al. [173] consider UML 1.3 activity diagrams and define their semantics by mapping their elements into transition rules of a multi-agent ASM (an extensions of ASM with concurrency). Similarly, Sarstedt and Guttmann [177] propose a token flow semantics for a subset of UML 2.0 activity diagrams based on the asynchronous multi-agent ASM model. However, this formalism impose certain restrictions on the supported control flows, for instance, it is mandatory that every fork be followed by a subsequent join node. The approaches in [182,183] deal with graph transformation techniques of UML 2.0 activity diagrams. Bisztray and Heckel [182] propose an approach that combines CSP and rule-based graph transformation technique. The mapping is based on the Triple Graph Grammars (TGGs) technique for graph transformations at the meta-model level. However, this approach is closely dependent on the semantic domain of CSP by considering only synchronous parallel composition. Hausmann [183] propose the specification of visual modeling languages semantics based on the Dynamic Meta Modeling (DMM), which is a combination of denotational meta modeling and operational graph transformation rules. However, this technique is quite complex and needs human intervention and understandability of a large set of rules.

Concerning process algebra, Rodrigues [175] considers the formalization of the UML 1.3 activity diagrams using Finite State Processes (FSP). A Labeled Transition System (LTS) that captures activity behavioral aspects is generated and the LTSA model-checker is used to assess the diagram. Yang et al. [176] propose a formalization of a subset of UML 1.4 activity diagrams using the π-calculus [123]. Activity diagram components are translated into π-calculus expressions, whereas activity edges are defined as relations linking these processes. For the UML 2.0 activity diagrams, Scuglík [178] proposes CSP as a formal framework. Many activity diagram constructs are covered. However, some constructs such as fork/join and merge have no direct mapping into the CSP syntax. They are handled by a combination of some elements from the CSP domain. Tabuchi et al. [171] propose a stochastic performance analysis of UML 2.0 state machines and activity diagrams annotated with the UML Profile for Schedulability, Performance, and Time. This is done using stochastic process algebraic semantics based on IMC. Finally, none of these proposals provides an intuitive mapping, since in most of the cases there is no one-to-one correspondence between the activity diagrams and process algebra neither syntactically nor semantically. This may result, for instance, in the difficulty to refer back to the original activity diagram from the corresponding process algebra term.

Among the approaches based on Petri net (PN) semantics, Lopez-Grao et al. [174] consider UML activity diagrams as a variant of the UML state machines and propose a mapping into the Labeled Generalized Stochastic Petri Nets (LGSPN). Störrle proposes PN-based semantics for UML 2.0 activity diagrams [179–181]. In [179], Störrle handles control flow using a mapping into Procedural Petri Nets (PPN), which is an extension of

PN for supporting calling subordinate activities or hierarchy and all kinds of control flow (well-formed or not) but neither data flow nor exception handling are supported. In [180], Störrle examines exception handling and provides a mapping into an extension of PPN, which is the Exception Petri Nets (EPN). The semantics is denotational and built on top of the semantics of [179]. Recently, Störrle has addressed data flow formalization [181] using Colored Petri Net (CPN). Although the work of Störrle seems to cover the majority of UML 2.0 activity diagram features, some of them still need more investigation such as streaming and expansion regions. Störrle and Hausmann [184] examine questions related to the appropriateness of the PN paradigm for expressing the UML 2.0 activity diagram semantics. Even though the UML standard claims that activity diagrams are redesigned to have a Petri-like semantics, the mapping of some features such as exceptions, streaming, and traverse-to-completion is not so natural and different variants of PN are needed to cover all the features. Moreover, some other problems are hindering the progress of investigations in this direction. This includes the absence of analysis tools and lack of a unified formalism combining all PN variants needed to cover all activity diagrams aspects [184].

3.6 Conclusion

In summary, this chapter presented research initiatives in four areas: (1) V&V of behavioral diagrams, (2) assessment of structural diagrams, (3) Performance analysis and probabilistic behavior assessment, and (4) Formalization of SysML activity diagrams. One can note that most of the works use a single technique focusing a unique aspect of the design. Moreover,

state machine diagrams caught most of the attention in terms of V&V and formalization. Finally, proposals targeting SysML just started to appear. In the next chapter, we present a unified approach that aims at addressing both structural and behavioral aspects of the design and enabling quantitative and qualitative assessment.

Chapter 4

Unified Verification and Validation Approach

In this chapter, we present the proposed verification and validation framework that was achieved within a major research initiative [185–187] that is part of a collaboration between the Computer Security Laboratory at the Concordia Institute for Information Systems Engineering (CIISE) and Defence Research and Development Canada (DRDC)[1]. The approach supports mainly the V&V of UML/SysML design models against a given set of functional requirements. It is based on three well-established techniques, namely formal analysis, static analysis, and software engineering metrics. In the present thesis, we will present our work within this project, then, we will enrich the underlying framework with a

[1] The collaboration has started within the *Collaborative Capability Definition, Engineering and Management* (CapDEM) project, which is an R&D initiative within the Canadian Department of National Defence. The latter aims at the development of a Systems-of-Systems engineering process and relies heavily on Modeling & Simulation.

performance analysis. The latter supports the quantitative assessment of probabilistic behavior. Additionally, we focus on the formal verification of SysML activity diagrams as being one of the most important and widely used diagrams in SE design models. Thus, we elaborate an operational semantics for SysML activity diagrams and establish the correctness of the V&V approach with respect to this diagram. These contributions will be detailed in the remaining chapters.

In this chapter, we intend to provide an overview of our V&V framework and describe its building components. Accordingly, this chapter is organized as follows. In Section 4.1, we give an overview of the overall approach. In Section 4.2, we detail the three techniques forming our unified approach. Then, we focus in Section 4.3 on behavioral verification and summarize the proposed extensions with probabilistic behavior assessment. Finally, we describe the design, architecture and implementation of our V&V tool in Section 4.4.

4.1 Verification and Validation Framework

Our main objective is to derive a unified approach for the V&V of design models in software and systems engineering. We cover both structural and behavioral aspects of the design. Our approach is based on a synergistic combination of three well-established techniques. These selected techniques are automatic formal verification (model-checking), software engineering techniques (metrics), and program analysis (static analysis). The choice of these three specific techniques is not done randomly, but, each one of them provides a means to tackle efficiently a specific issue and together they allow an enhanced design

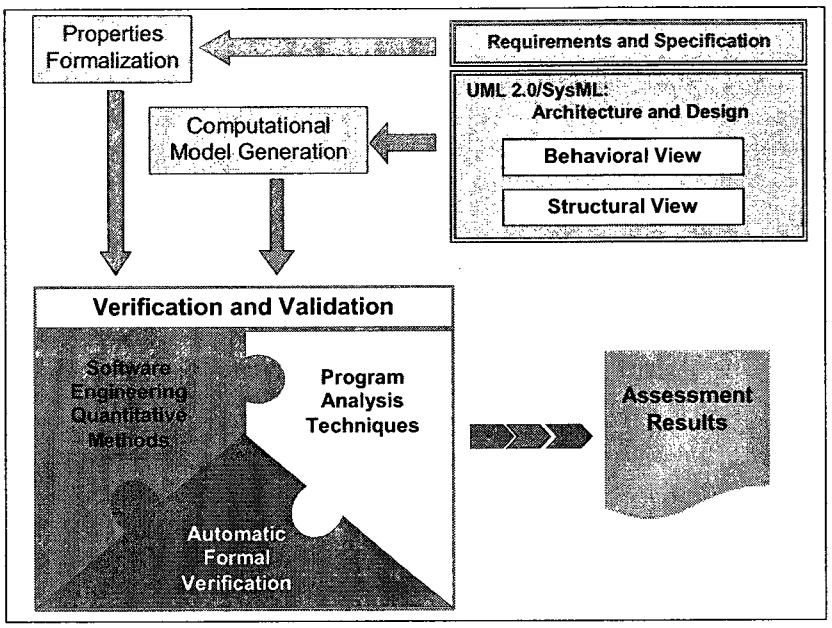


Figure 12: Verification and Validation Framework

assessment that is comprehensive to some extent. Specifically, for assessing the quality of the design from the structural point of view, we advocate the use of software engineering empirical methods such as metrics, which are extensively used to quantitatively measure quality attributes of object-oriented software design [142–150]. Conversely, with respect to the behavioral aspect, model-checking turns out to be an appropriate choice. Indeed, it has been successfully applied in the verification of the behavior of real applications (software as well as hardware systems) including digital circuits, communication protocols, and digital controllers. Moreover, model-checking is generally a fully automated formal verification technique that can explore thoroughly the state space of the system searching for potential errors. One of the benefits of many model-checkers is the ability to generate counterexamples for the violated property specifications. Finally, we propose to synergistically integrate static analysis techniques and software metrics with model-checking in

order to tackle scalability issues. Static analysis operates prior to the model-checker so that it helps narrowing the verification scope to the relevant parts of the model depending on the considered property. Additionally, specific metrics are used in order to assess the size and complexity of the model-checker's input so that it properly enables or disables static analysis. Figure 12 illustrates the synoptic of the overall approach. The V&V framework takes as input UML 2.0/SysML 1.0 design models of the system under analysis together with the related requirements. With respect to UML 2.0/SysML 1.0 design diagrams, the applied analysis depends on the type of the diagram under scope, whether it is structural or behavioral. The related results may help systems engineers have an appraisal of the quality of their design and take appropriate actions in order to remedy the detected deficiencies. In the following, we explain in details the proposed methodology.

4.2 Methodology

A software or system design model is fully characterized by its structural and its behavioral perspectives. The analysis of both views of the design is important in order to build a high-quality product. From a structural perspective, UML class and package diagrams, for instance, describe the organizational architecture of the software or the system. The quality of such diagrams is measurable in terms of object-oriented metrics. Such metrics provide valuable and objective insights into the quality characteristics of the design. In addition, behavioral diagrams not only focus on the behavior of elements in a system but also show the functional architecture of the underlying system (e.g. activity diagram). Thus,

we propose to apply metrics that can be used to measure quality attributes of the behavioral diagrams structure, namely the size and complexity metrics. From a behavioral perspective, simulation execution of state machine and activity diagrams, for instance, is not enough for a comprehensive assessment of the behavior. This is due to the increasing complexity of the behavior of modern software and systems that may exhibit concurrency and stochastic executions. Model-checking techniques have the ability to track such behaviors and provide faithful assessment based on the desired specifications. At this stage, program analysis techniques such as control flow and data flow analysis are integrated before the actual model-checking. They are applied on the semantic model in order to abstract it by focusing on the model fragments that are relevant to the properties that are being evaluated. This helps narrowing the verification scope and consequently leveraging the effectiveness of the model-checking procedure. In this context, quantitative metrics are used in order to appraise the size and complexity of the semantic model prior to static analysis. This enables the decision whether abstraction is actually needed before model-checking analysis take place. In the following, we present in details different components of our approach.

4.2.1 Semantic Models Generation

The semantics reflects the meaning of a given entity. Transition systems are widely accepted as semantic models for various systems. Indeed, it is generally accepted that any system that exhibits a given dynamic behavior can be abstracted to one that evolves within a discrete state space. Such a system is able to evolve through its state space assuming different configurations where a configuration is understood as the global state wherein the

system abides at a particular moment. Hence, all possible configurations summed up by the dynamics of the system and the transitions thereof can be coalesced into the semantic model of the system. We denote by a Configuration Transition System (CTS) the transition system specifying the semantic model of a given behavioral diagram. In essence, CTS is a form of automaton and it is characterized by a set of configurations that includes a set (usually a singleton) of initial ones and a transition relation that encodes the evolution of the CTS from one configuration to another. Configurations depend on the systems dynamic elements. Thus, a general parameterized CTS definition may be provided and tailored according to the concrete dynamic elements of the considered behavioral diagram. Each of these dynamic elements can be abstracted to a boolean variable. In a given configuration, boolean variables associated with the active dynamic elements are evaluated to true whereas, those associated with inactive dynamic elements are evaluated to false. An order relation among these variables has to be established.

The CTS concept can be conveniently adapted for each of the behavioral diagrams, including state machine, activity, and sequence diagrams. With respect to a given state machine diagram, its dynamic elements are represented by its states, guards, join specifications status, and dispatched events. At a certain point in time, we can define the current configuration of this diagram using the currently active states of the state machine (including sub-states or super-states in the hierarchy), the current guards evaluation, and the current status of the join nodes specifications. A join specification is a boolean expression attached to a join node specifying the condition for the tokens arriving at its incoming edges to synchronize. The evolution of the state machine diagram is triggered by the means of

dispatched events. Thus, events are used in order to label the transitions between pairs of configurations.

Concerning activity diagrams, a configuration is defined using the currently executing actions, the guards evaluations, and the join specifications status. The evolution of the activity diagram behavior to a new configuration is determined by the termination of some executing actions. As for sequence diagrams, the dynamic elements are the exchanged messages between the lifelines. The messages have to be encoded in the following format `S_ Msg_R`, where `Msg` represents the exchanged message, S denotes the sender, and R denotes the receiver. A Configuration in the semantic model of a given sequence diagram is composed of the set of tuples composed of the sender, the message, and the receiver that occur in parallel. Messages enclosed in a combined fragment of type `Alt` form multiple branching successor configurations. Messages enclosed in `Loop` combined fragment form a cycle in the CTS. Each message that is not enclosed in any combined fragment but in `Seq`, represents a singleton configuration. The transitions are derived from the ordered sequencing of events.

In order to assess a given system's behavior, we propose to systematically generate for each behavioral diagram its corresponding CTS that is used to encode the model-checker input. The procedure is based on a breadth-first iterative search approach that explores a given diagram on-the-fly and generates all reachable configurations and transitions thereof. Algorithm 13 presents the unified algorithm for the generation of the CTS from the behavioral diagram D, such that D can be of type *SM* for state machine diagram, *AD* for activity diagram, or *SQ* for sequence diagram. The algorithm for sequence diagrams is

```
1: procedure genCTS(D)
2:         /* Define two lists for respectively configurations and transitions. */
3:     CTSConfList = {}
4:     CTSTransList = {}
5:     FoundConfList = {initialConf}
6:         /* Check that there are still unexplored configuration. */
7:     while FoundConfList is not empty do
8:         crtConf := pop(FoundConfList)
9:         if crtConf not in CTSConfList then
10:            CTSConfList := CTSConfList ∪ crtConf
11:        else
12:            continue
13:        end if
14:        if Typeof(D) = SM then
15:            for all e in EventList do
16:        /* Compute the next configuration */
17:                nextConf := getConf(D,crtConf,e)
18:                if nextConf not in CTSConfList then
19:                    FoundConfList := FoundConfList ∪ nextConf
20:                    crtTrans = (crtConf,e,nextConf)
21:                    if crtTrans not in CTSTransList then
22:                        CTSTransList := CTSTransList ∪ crtTrans
23:                    end if
24:                end if
25:            end for
26:        end if
27:        if Typeof(D) = AD then
28:            for all a in crtConfActionList do
29:                nextConf := getConf(D,crtConf,execute(a))
30:                if nextConf not in CTSConfList then
31:                    FoundConfList := FoundConfList ∪ nextConf
32:                    crtTrans = (crtConf,nextConf)
33:                    if crtTrans not in CTSTransList then
34:                        CTSTransList := CTSTransList ∪ crtTrans
35:                    end if
36:                end if
37:            end for
38:        end if
39:    end while
40: end procedure
```

Figure 13: Generation of Configuration Transition Systems

very similar to the one for activity diagrams. The only difference resides in the exploration of the next configurations, which proceeds sequentially according to the type of the encountered enclosing combined fragment in the given sequence diagram. The CTS is defined using *CTSConfList* and *CTSTransList* initially empty, denoting respectively the list of configurations and the list of transitions thereof. *FoundConfList* is the list recording the so far identified but unexplored configurations. A configuration is of the form (*crtStateList*, *crtGList*,*crtJoinList*) where *crtStateList* is the list of the currently active states (or *crtConfActionList* in the case of actions), *crtGList* is the list of the current evaluations of all the guards, and *crtJoinList* is the list of the current status of the join specification for each join node. In each iteration, the current configuration to be explored is denoted by *crtConf*. The algorithm presents similarities in the processing of the different types of diagrams. In fact, one can note that the difference lies in the mechanism triggering the evolution of the diagram behavior. For the state machine diagram case, we rely on *EventList* from which events are picked up and dispatched one by one. For activity diagrams, we use *crtConfActionList* from which we select the action to be processed next. We use the auxiliary function *getConf*, which is overloaded according to the diagram type D, with parameter the variable *crtConf* denoting the current configuration and the variable e representing the event to be dispatched (or the action a to be executed). This function returns the next configuration *nextConf*.

In order to show practically the generation of CTS, we propose the state machine diagram example illustrated in Figure 14 modeling an hypothetical Automated Teller Machine (ATM) system. The top container state named `ATM` encloses four substates: `IDLE`,

VERIFY, EJECT and OPERATION. The IDLE state, wherein the system waits for a potential user, is the default initial substate of the top state. The VERIFY state represents the operations of verifying the validity of the card and the PIN. The EJECT state depicts the phase of termination of the user transaction. The OPERATION state is a composite state that includes states SELACCOUNT, PAYMENT and TRANSAC capturing several banking-related operations.

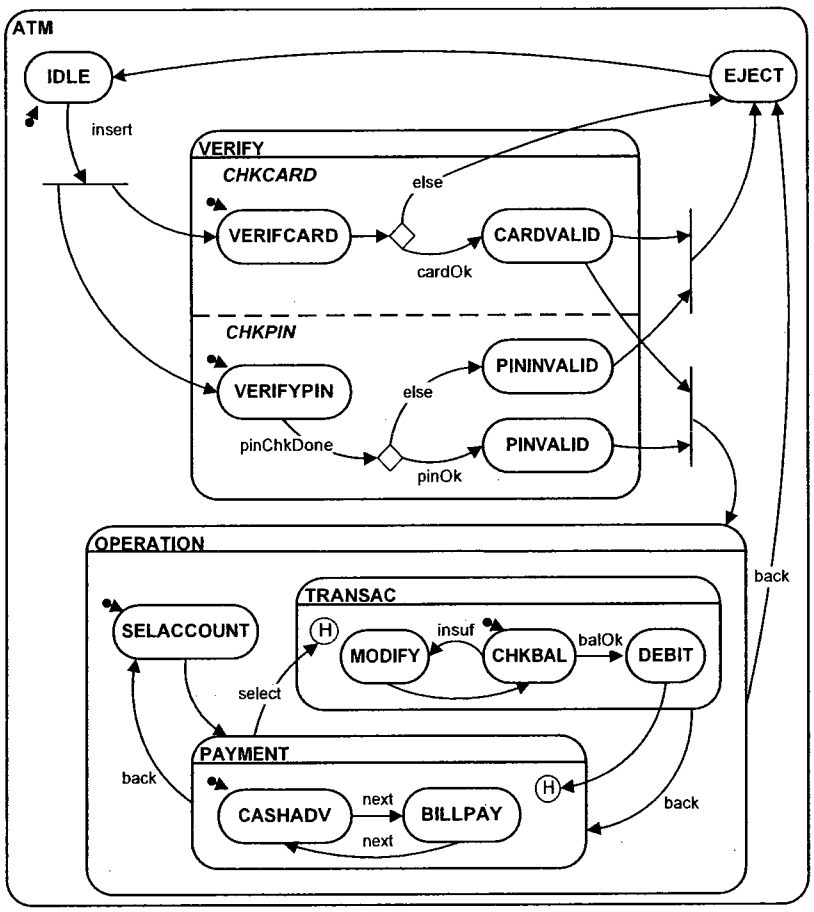


Figure 14: Case Study: ATM State Machine Diagram

The SELACCOUNT state is where an account belonging to the proprietary of the card has to be selected. When the state SELACCOUNT is active, and the user selects an account,

the next transition is enabled and the state PAYMENT is entered. The latter has two substates for cash advancing and bill payment, respectively. It represents a two-item menu, controlled by the event next. Finally, the TRANSAC state captures the transaction phase and includes three substates that corresponds each to checking the balance (CHKBAL), modifying the amount if necessary (MODIFY), and debiting the account (DEBIT), respectively. Each of the states PAYMENT and TRANSAC contains a shallow history pseudostate. If a transition targeting a shallow history is fired, the activated state is the most recent active substate in the composite state containing the history connector.

By applying our approach, we obtain the corresponding configuration transition system depicted in Figure 15. Each configuration is represented by a set (possibly singleton) of active states and guards evaluations of the state machine diagram. The join specifications status list is implicit. One can note that only active elements are shown in the configurations. The events are labeling the transitions.

4.2.2 CTL-Based Property Specification

In order to unfold the potential benefits of model-checking, properties are required to be precisely specified. In our V&V approach, we use the CTL temporal logic [64]. The latter is a branching-time logic (in contrast to linear-time logic). Its operators allow the description of properties on the branching structure of the computation tree unfolded from a given state transition graph of a system. Therein, a path is intended to represent a single possible computation in the model. It allows expressing an important set of systems properties including safety ("Nothing bad ever happens"), liveness ("Something good will eventually

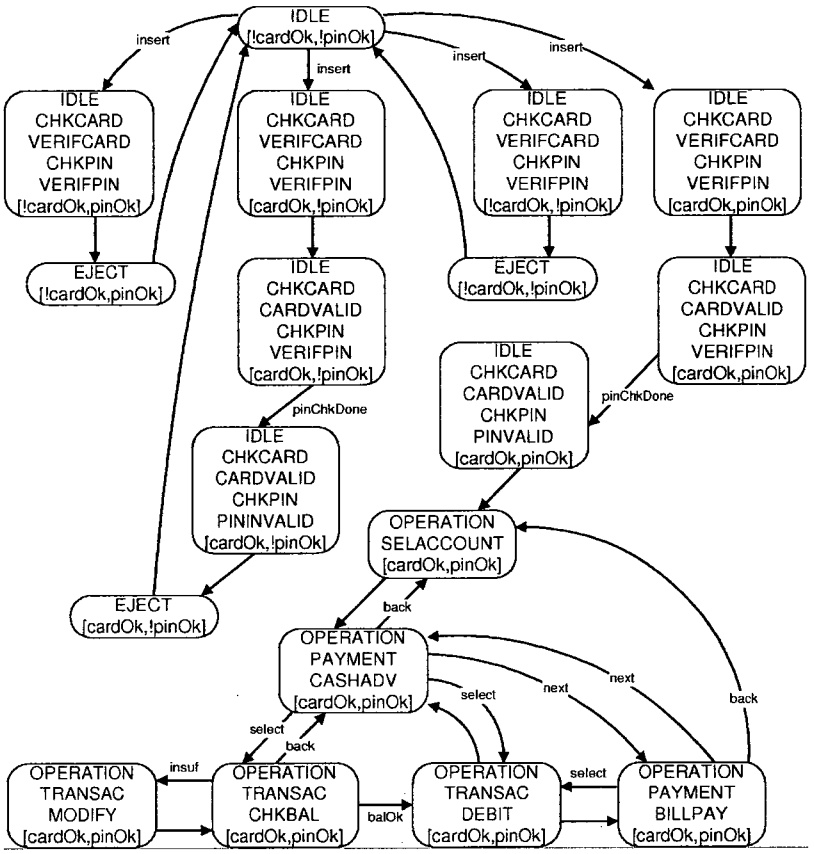


Figure 15: Configuration Transition System of the ATM State Machine Diagram

happen"), and reachability [188]. The CTL properties are built using atomic propositions, propositional logic, boolean connectives, and temporal operators. The atomic propositions correspond to the variables in the model while each temporal operator consists of two components: A path quantifier and an adjacent temporal modality. Since in general it is possible to have many execution paths starting at the current state, the path quantifier indicates whether the modality defines a property that should hold for all the possible paths (universal path quantifier `A`) or only on some of them (existential path quantifier `E`). The temporal operators are interpreted in the context of an implicit current state.

ϕ	::=	p	(Atomic propositions)
	|	$!\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \rightarrow \phi$	(Boolean Connectives)
	|	$AG\ \phi \mid EG\ \phi \mid AF\ \phi \mid EF\ \phi$	(Temporal Operators)
	|	$AX\ \phi \mid EX\ \phi \mid A[\ \phi\ U\ \phi] \mid E[\ \phi\ U\ \phi]$	(Temporal Operators)

Figure 16: CTL Syntax

Figure 16 presents the syntax of CTL, while Table 4.2.2 shows the underlying meaning of the temporal modalities.

$G\ p$	Globally, p is satisfied for the entire subsequent path
$F\ p$	Future (Eventually), p is satisfied somewhere on the subsequent path
$X\ p$	neXt, p is satisfied at the next state
$p\ U\ q$	Until, p has to hold until the point where q holds and q must eventually hold

Table 2: CTL Modalities

4.2.3 Model-Checking of Configuration Transition Systems

In order to apply automatic formal verification, we selected the NuSMV model-checker [70]. Our choice of NuSMV is motivated by the fact that it is open source and supports the analysis of specifications expressed in both Computation Tree Logic (CTL) and Linear Temporal Logic (LTL) [63]. NuSMV outperforms the SMV model-checker [69], its ancestor, especially for larger examples [112]. Furthermore, NuSMV supports both BDD and SAT techniques, which can be seen as complementary techniques since they solve different classes of problems [70]. Apart from its capability of generating counterexamples, NuSMV is able to verify a set of properties either in batch mode or interactively. The former mode allows better usability when dealing with large set of properties.

The back-end processing of model-checking requires the encoding of the CTS using the

NuSMV input language. The latter allows for the description of systems behavior based on Finite State Machines (FSM). The input model is built using three blocks. The first block is a syntactic declarative block wherein state variables are given a specific type and a specific range. The second block represents the initialization block, wherein the state variables are assigned their corresponding initial values or a range of possible initial values. Finally, the last block describes the dynamics of the transition system using `next` clauses. Therein, the logic governing the evolution of the state variables is specified. The latter consists in updating the state variables in every next step according to a logical valuation at the current step. In order to keep the NuSMV compact and simple, we encode the evolution of the dynamic elements contained within a configuration and not the configuration itself. Thus, we declare a NuSMV variable for each dynamic element of the diagram with its possible values. Using logical expressions, we define the conditions of activation and deactivation of each dynamic element. This is also useful when dealing with actual model-checking since properties to be verified ought to be expressed on the dynamic elements and not on configurations. The elaboration of the NuSMV code fragments describing the evolution of the dynamics is the most laborious part. It requires the analysis of the CTS configurations and transitions in order to determine the dynamic elements evolution. For every dynamic element, we need to specify a `next` block. The latter is built using a `case` expression specifying the activation or deactivation conditions of the current dynamic element. These conditions are logical expressions elaborated based on three parts: the configurations that contain the dynamic element as active, all transitions pointing to these configurations, and all the source configurations of these transitions. The first part is needed in order to identify the two following

```
Module main

DEFINE
insuf_2:=1;
balok_3:=2;
empty_1:=3;

Cand_modify0:= (evt=insuf_2) & chkbal;
Cand_all_modify:=Cand_modify0;
Cand_chkbal0:= (evt=empty_1) & modify;
Cand_all_chkbal:=Cand_chkbal0;
Can_debit0:= (evt=balok_3) & chkbal
Cand_all_debit:=Can_debit0;
Cand_any:= Cand_all_modify | Cand_all_chkbal | Cand_all_debit;

VAR
modify:boolean;
chkbal:boolean;
debit:boolean;
evt:1..3;

ASSIGN
init(modify):=1;
init(chkbal):=0;
init(debit):=0;

next(evt):= case
chkbal & !(modify | debit): {insuf_2,balok_3};
modify & !(chkbal | debit):{empty_1};
1:{1, 2, 3};
esac;

next(modify):= case
Cand_all_modify:1;
Cand_any:0;
1:modify;
esac;

next(chkbal):= case
Cand_all_chkbal:1;
Cand_any:0;
1:chkbal;
esac;

next(debit):= case
Cand_all_debit:1;
Cand_any:0;
1:debit;
esac;
FAIRNESS Cand_any
SPEC: EF modify
SPEC: AG(modify -> EF ! modify)
SPEC: EF chkbal
SPEC: AG(chkbal-> EF ! chkbal)
SPEC: EF debit
SPEC: AG(debit-> EF ! debit)
```

Figure 17: NuSMV Code Fragment of the ATM State Machine

parts, namely the transitions and the source configurations. The second part, which is concerned with transitions, is used in order to identify the events that trigger the activation of the current dynamic element. Finally, the source configurations are used to identify the dynamic elements responsible of the activation of the current dynamic element. Figure 17 shows a fragment of NuSMV code generated from the CTS of Figure 15. For example, we need to identify the configurations in the CTS of Figure 15 where the dynamic element `MODIFY` appear in order to build its corresponding `next` block (only one configuration in the left down corner). Then, one can note only one transition pointing to this configuration labeled with event `insuf`. Finally, the source configuration of the latter transition has the following active elements: `OPERATION`, `TRANSAC`, `CHKBAL`, `[cardOk,pinOk]`. We discard `OPERATION`, `TRANSAC`, and `[cardOk,pinOk]`, since they also appear in the configuration containing `MODIFY` (there is no change in their status). Thus, the condition of activating `MODIFY` (i.e. $modify = 1$) is the following logical expression $evt = insuf\_2 \wedge chkbal$.

After generating the NuSMV code of the behavioral diagram, properties that express deadlock absence and reachability are automatically generated for every state in the diagram and appended to the NuSMV code. In addition, user-defined properties specified using macros notation are automatically expanded into CTL formulas and appended to the input of the model-checker. Once the model-checking procedure is executed, the assessment results pinpointed some interesting problems in the ATM state machine design. Indeed, the model-checker determined that the `OPERATION` state exhibits deadlock, meaning that once entered, this state is never left. This is due to the fact that the transitions with

the same trigger are given higher priority when the source state is deeper in the containment hierarchy. Moreover, the transitions without a triggering event are fired as soon as the state machine reaches a stable configuration containing the corresponding source state. This is precisely the case of the transition from SELACCOUNT to PAYMENT. Thus, there is no transition that allows the OPERATION dynamic element to be deactivated. On the corresponding CTS, illustrated in Figure 15, one can notice that once a configuration containing OPERATION is reached, there is no transition to a configuration that deactivates OPERATION.

We present hereafter, some relevant user-defined properties described in both macro and CTL notations and their corresponding model-checking results. The first property (4.2.3.1) asserts that it is always the case that if the VERIFY state is reached then from that point on, the OPERATION state should be also reachable:

$$\textbf{Macro} \ : \ \texttt{ALWAYS}\ VERIFY \ \rightarrow \ \texttt{MAYREACH}\ OPERATION$$

$$\textbf{CTL} \ : \ \texttt{AG}(VERIFY \ \rightarrow \ (\texttt{E}[!(IDLE)\ \texttt{U}\ OPERATION])) \qquad (4.2.3.1)$$

The next property (4.2.3.2) asserts that whenever the state OPERATION is reached, it should be unavoidable to reach the state EJECT at a later point:

$$\textbf{Macro} \ : \ \texttt{ALWAYS}\ OPERATION \ \rightarrow \ \texttt{INEVIT}\ EJECT$$

$$\textbf{CTL} \ : \ \texttt{AG}(OPERATION \ \rightarrow \ (\texttt{A}[!(IDLE)\ \texttt{U}\ EJECT])) \qquad (4.2.3.2)$$

The last one (4.2.3.3) states that the CHKBAL state must precede the state DEBIT:

Macro : $CHKBAL$ PRECEDE $DEBIT$

CTL : !E[!($CHKBAL$) U $DEBIT$] (4.2.3.3)

The property (4.2.3.1) turned out to be satisfied when running the model-checker. However, the last two properties (4.2.3.2) and (4.2.3.3) failed. The failure of the property (4.2.3.2) was expected since, from the automatic specifications, we noticed that state `OPERATION` is never left once entered (deadlock state) and it does not contain the `eject` substate.

The failure of the last property (4.2.3.3) was demonstrated by a counterexample provided by the model-checker. Though the model-checker can provide a counterexample for any of the failed properties, we present this last one as it captures a critical unintended behavior. The following is a trace parsed using our V&V tool showing the counterexample:

IDLE [!cardOk,!pinOk];

(VERIFY,CHKCARD,VERIFCARD,CHKPIN,VERIFPIN [cardOk,pinOk]);

(VERIFY,CHKCARD,CARDVALID,CHKPIN,VERIFPIN [cardOk,pinOk]);

(VERIFY,CHKCARD,CARDVALID,CHKPIN,PINVALID [cardOk,pinOk]);

(OPERATION,SELACCOUNT [cardOk,pinOk]);

(OPERATION,PAYMENT,CASHADV [cardOk,pinOk]);

(OPERATION,TRANSAC,DEBIT [cardOk,pinOk]);

The foregoing counterexample is represented by a series of configurations separated by semicolons. Additionally, a comma is used to separate two or more states that are present simultaneously in a given configuration and the variable values are enclosed in square brackets. The failure of the last property is due to the presence of a transition from the

state PAYMENT to the shallow history connector of the state TRANSAC. This allows for the immediate activation of the state DEBIT when reentering the TRANSAC state by its history connector.

The counterexample is useful in the sense that it pinpoints the source of the problem in the design in order to help the designer infer the needed corrections. In the case of the presented ATM design, the first correction consists in adding a trigger event, `select` for instance, to the transition from the state SELACCOUNT to the state PAYMENT. This should eliminate the deadlock and correct the second user-defined property (4.2.3.2). In order to remedy to the failure of the property (4.2.3.3), the history connector of the state TRANSAC should be removed and the target of the transition should be changed to point directly to the state TRANSAC instead of pointing to the history connector. A reiteration of the V&V on the corrected design confirms that all properties are satisfied.

4.2.4 Static Analysis Integration

Program analysis has been used to automatically analyze software programs. It allows the collection of specific information from a program such as data and control flow dependencies, invariants, anomalous behavior, reliability, or conformance to specifications [189]. Information obtained from program analysis is used for program understanding, testing, compiler optimization, and security analysis [190]. There are two main approaches in program analysis: Static program analysis and dynamic program analysis. Particularly, we are interested in static program analysis techniques that can be used in order to slice (decompose) a program into independent meaningful parts that can be then analyzed separately.

Slicing, and subsequent manipulation of slices, has application in software engineering such as program debugging and testing [18].

In the context of our approach, we use data and control flow analysis techniques on the semantic models of the behavioral diagrams in order to slice them in a useful way. Basically, data flow analysis consists in searching for the presence of data invariants (e.g. specific variable values or relations) whereas control flow analysis is used in order to detect the control flow dependencies. Statically slicing the CTS of a given behavioral diagram allows getting subgraphs that are obviously less complex than the whole CTS. The obtained slices can be individually subjected to model-checking. This in turn has the potential to leverage the effectiveness of the model-checking procedure in terms of memory space and computation time. Nevertheless, the slicing procedure has to be performed safely under the assumptions that the properties to be verified fall into liveness or safety categories; This is important in order to keep the verification and validation feasible and flawless.

It should be noted that slicing does not preserve global properties of the sliced CTS. A given slice of the CTS may no longer contain a deadlock, for example. Due to the fact that the resulting subgraphs dynamics may be severely restricted in some cases, one has to take this fact into account when interpreting the model-checking results. Thus, even though it might be the case that a liveness property fails for a transition system corresponding to a particular subgraph, the property should not be declared as failed for the original model as long as there is at least one subgraph whose transition system satisfies the property in question. Conversely, whenever a safety property fails for a particular subgraph, then it is declared as failed for the original model as well. Notwithstanding, this task can be

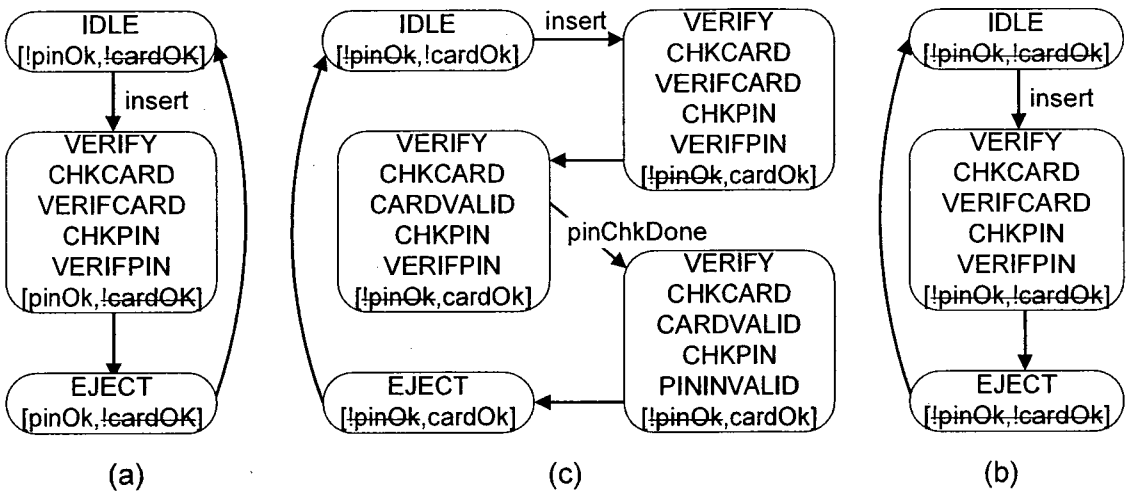


Figure 18: Data Flow Subgraphs

automated in a transparent way to the front-end of the verification framework.

In order to illustrate the slicing approach on the CTS, we use the example given in Figure 15. Therein, the guards `cardOk` and `pinOk` are present in every configuration with different evaluations. The exclamation mark preceding a variable in a particular configuration means that it is evaluated to false. Figure 18 illustrates three subgraphs corresponding each to a slice of the original CTS of Figure 15. The subgraph of Figure 18 (a) has the invariant `!cardOk` that holds in all its states. Similarly, the subgraph of Figure 18.(c) has the invariant `!pinOk`. Finally, Figure 18.(b) illustrates a slice with the invariants `!cardOk` and `!pinOk`. Furthermore, control flow analysis can be performed on the original CTS. Accordingly, the subgraph illustrated in Figure 19 presents a slice of the CTS that confines the control locus once `cardOk` and `pinOk` hold and `OPERATION` state is entered.

The complexity of the model-checking algorithms essentially depends on the size of the

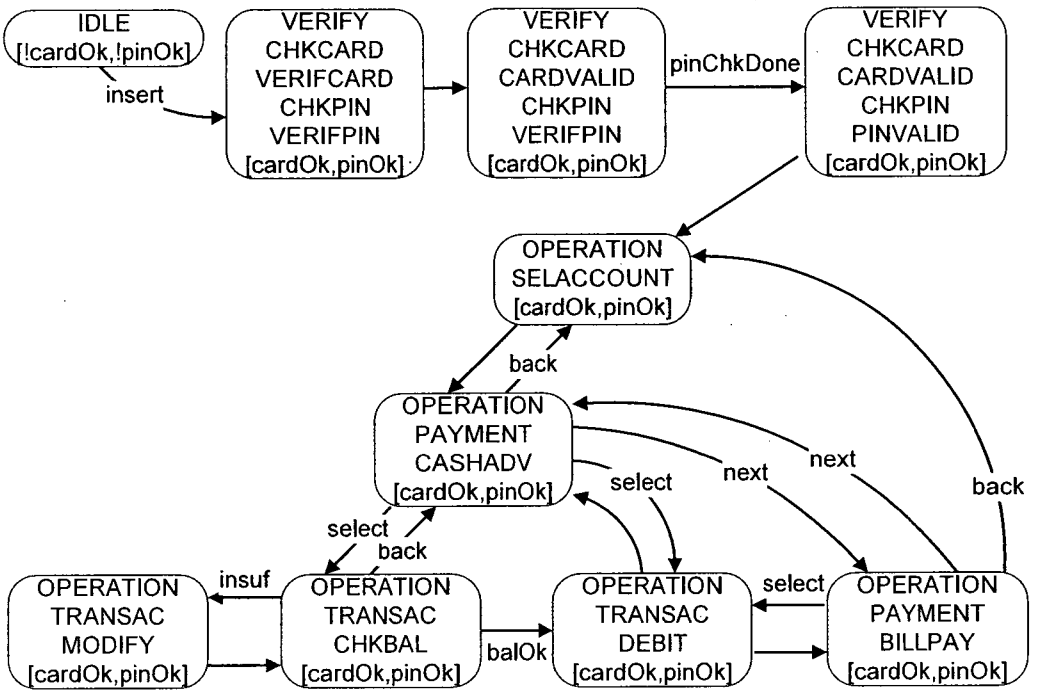


Figure 19: Control Flow Subgraph

input model, which corresponds to the number of state variables. Therefore, we use quantitative size metric on the semantic models in order to estimate their complexity. This represents an important feedback, which can be used in order to take the decision of whether to activate the static analysis module. In order to emphasize the benefits of slicing, we give some edifying statistics. While for the initial CTS graph, the model-checker allocated between 70 to 80 thousand BDD nodes (depending on the variable ordering), for the sliced subgraphs the allocated BDD nodes were significantly reduced as shown in Table 3.

Graph	Memory Footprint (BDD nodes)
Figure 15	70 000 to 80 000
Figure 18.a	≈ 4 000
Figure 18.b	≈ 4 000
Figure 18.c	≈ 8 000
Figure 19	28 000 to 33 000

Table 3: Memory Consumption Statistics

4.2.5 Design Quality Attributes Metrics

The quality of an object oriented system depends on different attributes such as complexity, understandability, maintainability, and stability. Empirical methods such as those involving software engineering quantitative methods, namely metrics, can be used to quantify such software quality attributes. Design metrics were essentially used on UML class and package diagrams in order to assess structural aspects of software engineering design models. In the context of our approach, we leveraged a set of fifteen metrics in order to assess the quality of UML class and package diagrams.

In addition to applying these quantitative methods to the design itself, we consider the use of specific metric on the semantic models and on the behavioral diagrams themselves. To do so, we consider metrics such as cyclomatic complexity [192], length of critical path [193], number of states, and number of transitions. On the one hand, the length of critical path metric can provide us with measurement to determine the length of the path that has to be traversed in a graph from an initial node to a destination node in order to achieve a given behavior with respect to a given criterion (e.g. timeliness). The critical path is different from a regular path in that it might traverse critical nodes. For example, a program

Metric	Diagram
Abstractness [146]	package
Instability [146]	package
Distance from the Main Sequence [146]	package
Number Of Attributes [145]	class
Number of Methods Added [145]	class
Number of Methods Overridden [145]	class
Number of Methods Inherited [145]	class
Specialization Index [145]	class
Class Responsibility [191]	class
Class Category Relational Cohesion [191]	class
Public Methods Ratio [191]	class
Number of Children [142]	class
Depth of Inheritance Tree [142]	class
Coupling Between Object Classes [142]	class
Number Of Methods [144]	class

Table 4: Implemented Metrics

represented by a call graph might have as critical path the one that is needed to achieve the execution of a given subroutine in the program within minimal time delay. In the same sense, we can measure the length of critical path on the semantic models derived from behavioral diagrams given some important criteria related to V&V.

On the other hand, we use the cyclomatic complexity metric in order to measure the complexity of the behavioral diagram and its semantic model. As the latter is a graph that unfolds the entire dynamics of the behavioral diagram, its complexity has to be greater or equal to the complexity of the diagram. If this is not the case, this implies that the corresponding diagram has some structural parts that are meaningless or redundant from the dynamics point of view.

4.3 Probabilistic Behavior Assessment

A range of systems inherently includes probabilistic information. Probabilities can be used in order to model unpredictable and unreliable behavior exhibited by a given system. It becomes insufficient to merely satisfy functional requirements of todays systems; Other quality standard attributes such as reliability, availability, safety, and performance have to be considered as well. In order to enable the specification of a larger spectrum of systems, SysML extends UML 2.1 activity diagrams with probabilistic features.

In this context, we propose to extend our aforementioned discussed framework with probabilistic verification of SysML activity diagrams. Accordingly, we propose to integrate probabilistic model-checking techniques within the automatic formal verification module. This consists in the systematic translation of SysML activity diagrams into the input language of an appropriate probabilistic model-checker. Furthermore, we propose to investigate mathematically the correctness of our approach. Thus, we define formally the semantics of SysML activity diagrams. Accordingly, we propose a dedicated probabilistic calculus, namely Activity Calculus (AC) that captures the essence of SysML activity diagrams syntactically and semantically. Our calculus is inspired by the domain of process algebra. The latter offers a mathematically well-elaborated framework for reasoning about concurrent and distributed systems [114, 123]. Moreover, we define formally the probabilistic model-checker input language, which represents the target domain of our translation. Finally, the soundness of the translation algorithm is proved using both defined formal semantics.

The next chapters detail these proposed extensions and contributions in order to build an efficient, automatic, and rigorous V&V approach.

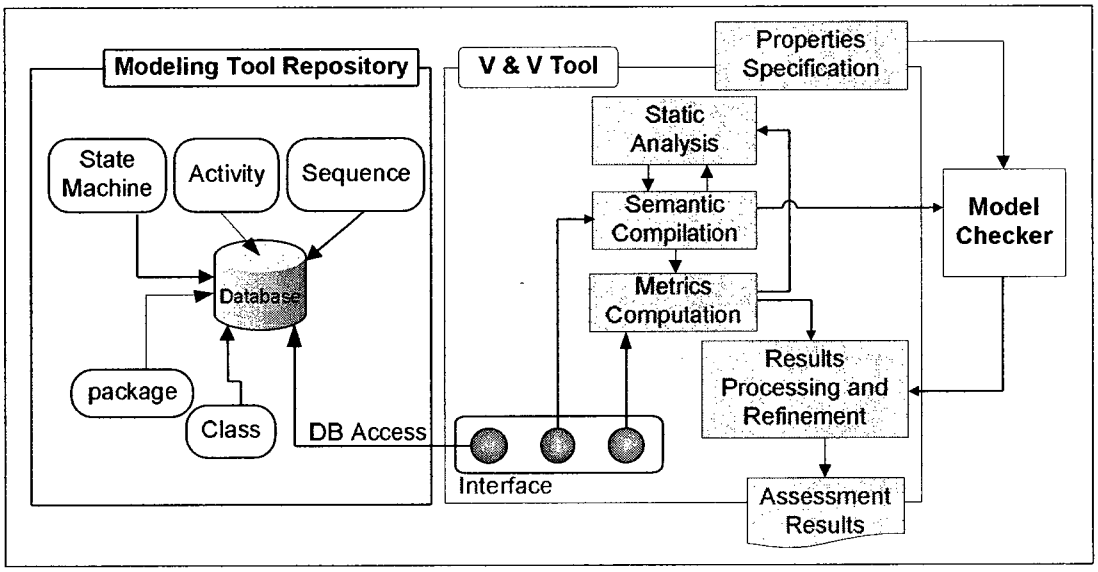


Figure 20: Architecture of the Verification and Validation Environment

4.4 Verification and Validation Tool

In order to put into practice our V&V approach, we design and implement a software tool whose architecture is illustrated in Figure 20. The tool is intended to be used in conjunction with a modeling environment wherefrom the design model under scope can be fetched and subjected to the V&V module. It is a Multiple Document Interface (MDI) application in which one can easily navigate among several views at once. The main interface is composed of a standard menu on the top and a vertical menu bar on the left.

The latter allows selecting a specific view of a given module and loading it into the MDI. The tool interfaces with the modeling environment Artisan Real-Time Studio [194]

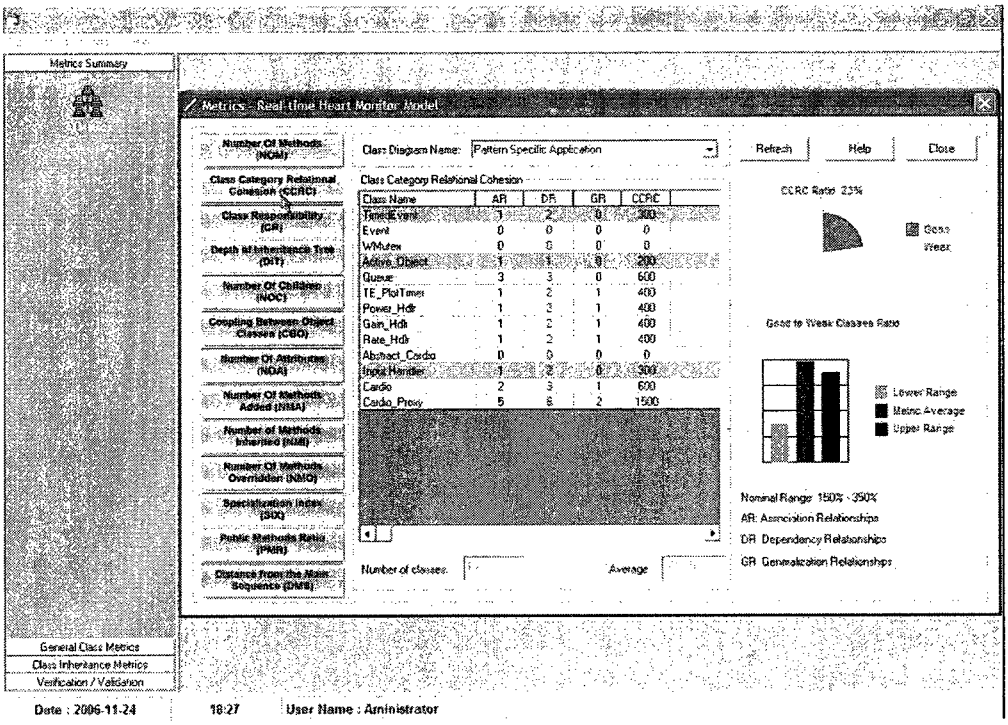


Figure 21: Environment Screenshot: Metrics Assessment

from where the designer can load the design model and select the diagram for assessment. Once started, the tool automatically loads the assessment module associated with the type of the selected diagram. For instance, if the opened diagram is a class diagram, the metric module is activated and the relevant measurements are performed. A set of quantitative measurement are provided with their relevant feedback to the designer. Figure 21 shows a screenshot example of metrics application. For behavioral diagrams, the corresponding model-checker (NuSMV) code is automatically generated and generic properties such as reachability and deadlock absence for each state of the model are automatically verified. An assessment example using model-checking is shown in Figure 22. Furthermore, the tool comprises an editor with a set of pre-programmed buttons through which the user

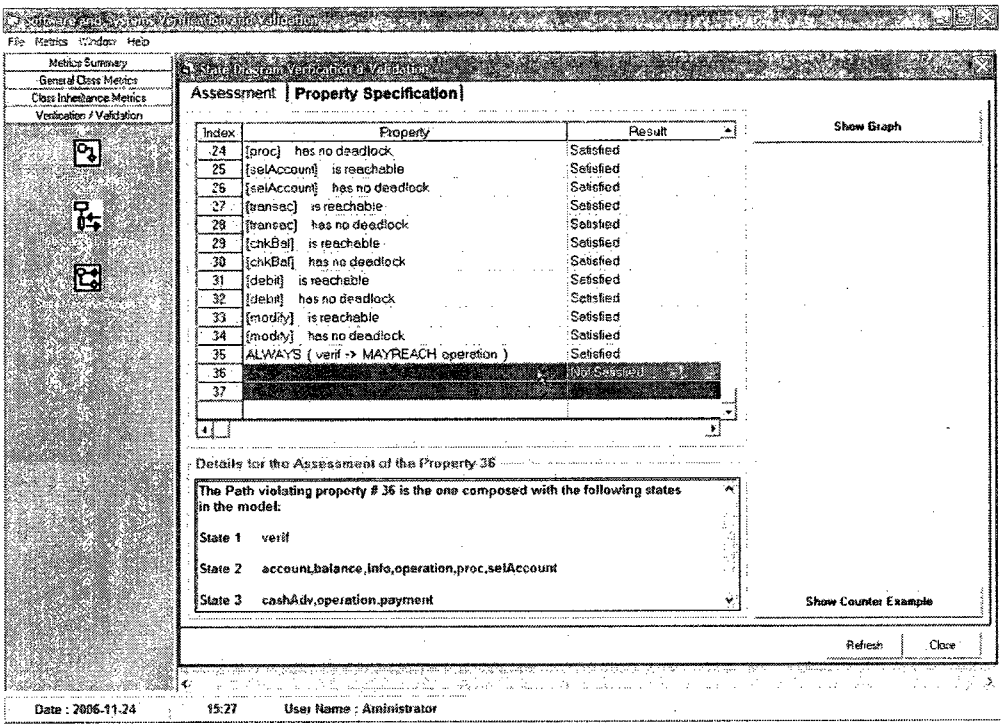


Figure 22: Environment Screenshot: Model-Checking Results

can specify custom properties. This is based on an intuitive easy-to-learn macro-based specification language that we defined. More precisely, we developed a set of macros using operators like always, mayreach, etc.that are systematically expanded into their corresponding Computation Tree Logic (CTL) operators. The editor interface screenshot with an example of custom properties specification is illustrated in Figure 23. Finally, a specific window frame is dedicated for the presentation of the assessment results. Since the feedback generated by the model-checker is not user-friendly and is not understandable by non-expert people, we built a back-end module that analyzes the provided output traces, in the case of failed properties, and renders relevant information about the counterexamples in a meaningful way, using a graphical visualization.

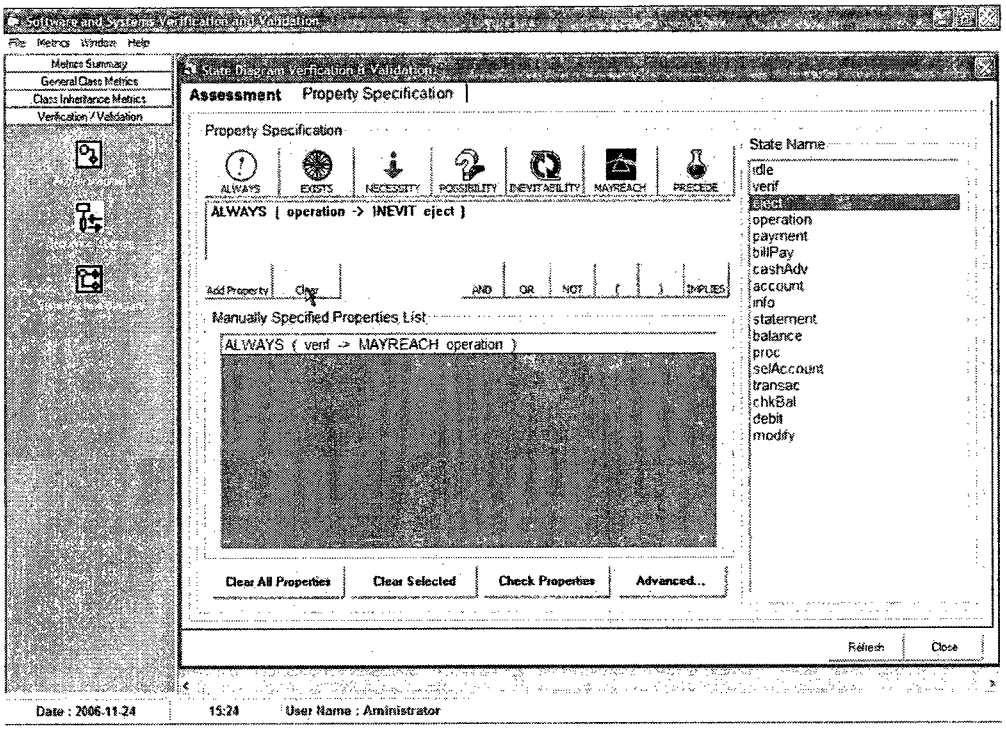


Figure 23: Environment Screenshot: Property Specification

4.5 Conclusion

In summary, we elaborated an innovative approach that contributes to the V&V of design models expressed using the modeling languages UML and SysML. It is based on a synergy between three well-established techniques: model-checking, static analysis, and empirical software engineering quantitative methods. The synergy relies on the fact that if each one of these techniques is applied alone, the resulting outcome is a partial assessment of the design (for instance either structural or behavioral). In addition to qualitative analysis, our approach enables quantitative assessment of design models. With respect to behavioral diagrams, the main challenge was to build a unified model that we called Configuration

Transition System (CTS). The latter represents a common parametrized model that describes the semantic models of state machine, sequence, and activity diagrams. In the next chapter, we present a practical framework for probabilistic verification of SysML activity Diagrams.

Chapter 5

Probabilistic Model-Checking of SysML Activity Diagrams

Incorporated modeling and analysis of both functional and non-functional aspects of todays systems behavior represents a challenging issue in the field of formal methods. In this chapter, we are interested in integrating such an analysis on SE design models. We focus on two aspects: probabilistic and timed behavior. SysML 1.0 [10] extends UML activity diagrams with stochastic features. Thus, we propose to translate SysML activity diagrams annotated with timing information into the input language of the probabilistic model-checker PRISM [72]. For the timed aspect, we need beforehand to investigate time-annotation on SysML activity diagrams. Thus, Section 5.1 presents how timing information is handled in SysML activity diagrams and describes the used time-annotation. In Section 5.2, we explain our approach for the verification of both untimed and time-annotated SysML activity diagrams. In Section 5.3, we present the algorithm implementing the translation of SysML

activity diagrams into PRISM input language. Section 5.4 is dedicated to the description of the property specification language, namely PCTL*. Finally, Section 5.5 illustrates the application of our approach on a SysML activity diagram case study.

5.1 Time-Annotated SysML Activity Diagrams

In order to carry out quantitative analysis of time-related properties, time constraints need to be specified on activity diagrams. However, time-annotations on top of SysML activity diagrams are not clearly defined. Two proposals have been advanced in [10]. The first proposal concerns the use of a model called "simple time model" defined in [21]. It is a UML 2.x sub-package related to the CommonBehavior package and allows for the specification of time constraints (e.g. time interval and duration) on sequence diagrams. Howbeit, the way to apply it on activity diagrams is not clearly specified. The second alternative is to use timing diagrams, even though these diagrams are not part of the SysML diagrams taxonomy [10]. The majority of reviewed works select the UML profile for Schedulability, Performance, and Time (SPT) [158] in order to annotate their diagrams with time and performance aspects. However, this profile is compatible with UML 1.4 and it has to be aligned with UML 2.x in order to be used on SysML diagrams. A new UML profile, called MARTE [51], has been recently developed by OMG in order to replace the existing UML SPT profile. It is recommended to be used for model-driven development of real-time and embedded systems. It aims at providing facilities to annotate models with the information required to perform specific analysis, especially, performance and schedulability analysis.

In the case of activity diagrams, we use the `RtFeature` stereotype, which extends the actions language unit [51]. Specifically, we use the attribute `relDl`, which denotes relative deadline specification. For the sake of clarity, the time annotation is performed directly inside the action nodes.

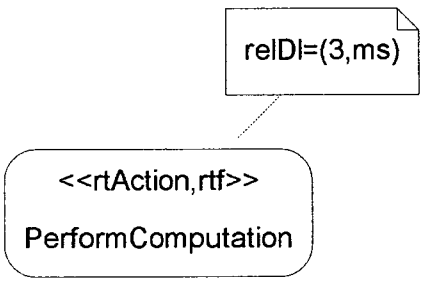


Figure 24: Time Annotation on Action Nodes

However, in order to keep our examples of SysML activity diagrams clear and uncrowded, we will annotate timing information directly inside the action node.

5.2 Probabilistic Verification Approach

Our objective is to provide a technique by which we can analyze SysML activity diagrams from functional and non-functional point of views in order to find out subtle errors in the design. This allows the reasoning about the correction of the design from these standpoints before the actual implementation. In these settings, probabilistic model-checking allows performing both qualitative and quantitative analysis of the model. It can be used to compute expectation on systems performance by quantifying the likelihood of a given property being violated or satisfied in the system model. In order to carry out this analysis, we

design and implement a translation algorithm that maps SysML activity diagrams into the input language of the selected probabilistic model-checker. Thus, an adequate performance model that correctly captures the meaning of these diagrams has to be derived. More precisely, the selection of a suitable performance model depends on the understanding of the behavior captured by the diagram and its underpinning characteristics. It has also to be supported by an available probabilistic model-checker. For the sake of generality, we study first the untimed SysML activity diagrams and then address time-annotated ones.

The global state of an activity diagram can be characterized using the location of the control tokens. A specific state can be described by the position of the token at a certain point in time. The modification in the global state occurs when some tokens are enabled to move from one node to another. This can be encoded using a transition relation that describes the evolution of the system within its state space. Therefore, the semantics of a given activity diagram can be described using a transition system (automata) defined by the set of all the states reachable during the system's evolution and the transition relation thereof. SysML activity diagrams present the possibility of modeling probabilistic behavior, using probabilistic decision nodes. The outgoing edges of these nodes quantified with probability values specify probabilistic branching transitions within the transition system. The probability label denotes the likelihood of a given transition's occurrence. In the case of deterministic transitions, all assigned probability labels are equal to 1. Furthermore, the behavior of activity diagrams presents non-determinism inherently due to parallel behavior and multiple instances execution. More precisely, fork nodes specify unrestricted parallelism, which can be described using non-determinism in order to model interleaving of

flows executions. This corresponds in the transition system to a set of branching transitions emanating from the same state, allowing the description of asynchronous behavior. In terms of probability labels, all transitions occurring due to non-determinism are labeled with probability equal 1.

In order to select the suitable model-checker, we need to define the right probabilistic model that captures the behavior depicted by SysML activity diagrams. To this end, we need a model that expresses non-determinism as well as probabilistic behavior. Thus, MDP might be a suitable model for SysML activity diagrams. Among the existing probabilistic model-checkers, we select PRISM model-checker. The latter is the only free and open source model-checker that supports MDPs analysis. Moreover, it is widely used in many application domains on various real-life case studies and is recognized for its efficiency in term of data structure and numerical methods. In summary, in order to apply probabilistic model-checking on SysML activity diagrams, we need to map these diagrams into the corresponding MDPs using PRISM input language. With respect to properties, they have to be expressed using Probabilistic Computation Tree Logic (PCTL*), which is commonly used in conjunction with discrete-time Markov chains and Markov decision processes [195]. Figure 25 illustrates the synopsis of the proposed approach.

In order to test our approach, we implemented our translation algorithm into a prototype tool written in Java that systematically maps SysML activity diagrams into their corresponding Markov decision processes expressed in the input language of PRISM model checker. The diagrams can be fetched from any modeling environment that supports SysML. Various model-driven development tools support UML, the defacto standard for software

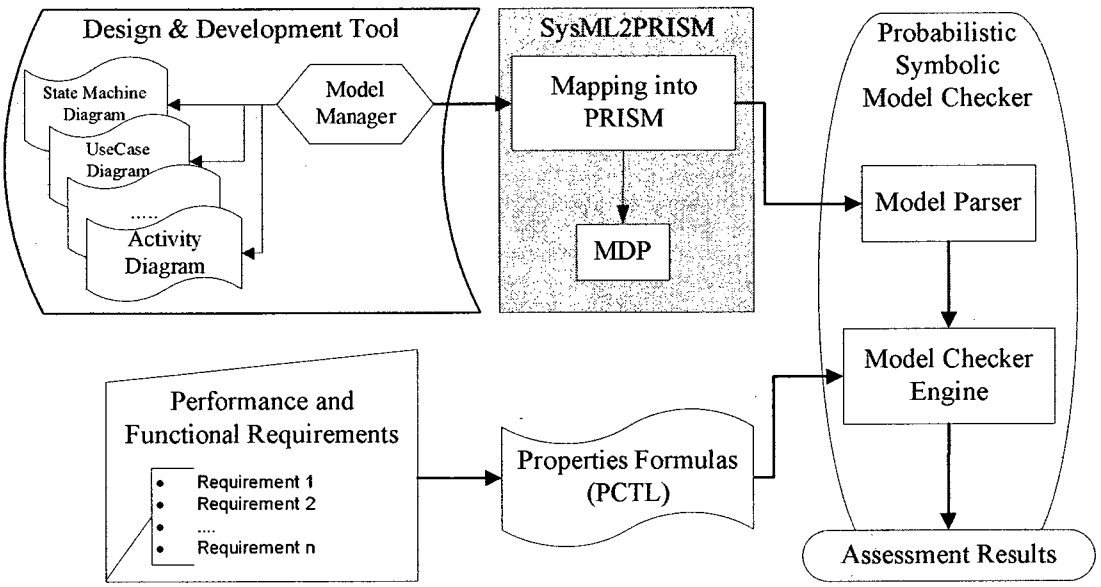


Figure 25: Probabilistic Model-Checking of SysML Activity Diagrams

development. Nowadays, many of these tools are upgraded in order to support the SysML modeling language (Artisan Real-time Studio [194], IBM Rational Software Delivery Platform [196], etc.). Generally, those tools provide also some advanced functionalities in order to access the design models in read and write modes.

In the sequel, we present the algorithm that we devise for the systematic mapping of SysML activity diagrams into the corresponding PRISM code.

5.3 Translation into PRISM

We assume a single initial node and a single activity final node. However, this is not a restriction since we can replace a set of initial nodes by one initial node connected to a fork node and a set of activity final nodes by a merge node connected to a single activity final

node. In the following, we present a data structure definition for SysML activity diagrams annotated with time.

Definition 5.3.1. A SysML activity diagram annotated with time on action nodes is a tuple $A = (N, N_0, type, \Delta, next, label)$ where:

- N is the set of activity nodes of types action, initial, final, flow final, fork, join, decision, and merge.
- N_0 is the initial node,
- *type*: $N \rightarrow \{action\ ,\ initial, final, flowfinal,\ fork, join,\ decision,\ merge\}$, that associates to each node its corresponding type,
- Δ: $N \rightarrow \mathbb{R}^+$, a function that associates for each node of type action a duration, where $\mathbb{R}$ is the set of real numbers. Control nodes are supposed to have no duration. As duration is a time measurement, we consider only positive real numbers,
- *next*: $N \rightarrow \wp(N)$ a function that returns for a given node the set (possibly singleton) of nodes that are directly connected to it via its outgoing edges,
- *label*: $N \times N \rightarrow Act \times]0, 1]$ a function that returns the pair of labels (g,p), namely the guard and the probability on the edge connecting two given nodes. □

5.3.1 Translation into MDP

We rely on a fine-grained iterative translation of SysML activity diagrams into MDP. Indeed, the control locus is tracked on both action and control nodes. Thus, each of these

```
1: nodes as Stack;
2: cNode as Node;
3: nNode as list_of_ Node;
4: vNode as list_of_ Node;
5: cmd as PrismCmd;
6: varfinal, var as PRISMVarId;
7: cmdtp as PrismCmd;
8: procedure T(A,N)
9:         /* Stores all newly discovered nodes in the stack */
10:    for all n in N do
11:        nodes.push(n);
12:    end for
13:    while not nodes.empty() do
14:        cNode := nodes.pop();
15:        if cNode not in vNode then
16:            vNode := vNode.add(cNode);
17:            if type(cNode)=final then
18:                cmdtp := C(cNode, eq(varfinal,1), raz(vars), null, 1.0);
19:            else
20:                nNode := next(cNode);
21:        /* Return the PRISM variable associated with the cNode */
22:                var := prismElement(cNode);
23:                if type(cNode)=initial then
24:                    cmdtp := C(cNode, eq(var,1), dec(var), nNode, 1.0)
25:                end if
26:                if type(cNode) in {action, merge} then
27:        /* Generate the final PRISM command for the edge cNode-nNode */
28:                    cmdtp := C(cNode, grt(var,0), dec(var), nNode, 1.0);
29:                end if
30:                if type(cNode)= join then
31:                    cmdtp := C(cNode, var, raz(pinsOf(var)), nNode, 1.0);
32:                end if
33:                if type(cNode)= fork then
34:                    cmdtp1 := C(cNode, grt(var,0), dec(var), nNode[0], 1.0);
35:                    cmdtp := C(cNode, cmdtp1.grd, cmdtp1.upd, nNode[1], 1.0));
36:                end if
```

Figure 26: Translation Algorithm of SysML Activity Diagrams into MDP - Part 1

```
37:                 if type(cNode)= decision then
38:                     g := Π(label(cNode, nNode[0]), 1);
39:                     upd := and(dec(var), set(g,true)) ;
40:                     cmdtp1 := C(cNode, grt(var,0), upd, nNode[0], 1.0);
41:                     g := Π(label(cNode, nNode[1]), 1);
42:                     upd := and(dec(var), set(g,true)) ;
43:                     cmdtp2 := C(cNode, grt(var,0), upd, nNode[1], 1.0);
44:         /* Append both generated commands together before final appending */
45:                     append(cmdtp, cmdtp1);
46:                     append(cmdtp, cmdtp2);
47:                 end if
48:                 if type(cNode)= pdecision then
49:                     g := Π(label(cNode,nNode[0]), 1);
50:                     p := Π(label(cNode,nNode[0]), 2);
51:                     upd := and(dec(var), set(g,true)) ;
52:                     cmdtp1 :=C(cNode, grt(var,0), upd, nNode[0], p);
53:                     g := Π(label(cNode,nNode[1]), 1);
54:                     q := Π(label(cNode,nNode[1]), 2);
55:                     upd := and(dec(var), set(g,true)) ;
56:                     cmdtp2 := C(cNode, grt(var,0), upd, nNode[1], q);
57:         /* Merge commands into one final command with a probabilistic choice */
58:                     cmdtp :=merge(cmdtp1,cmdtp2);
59:                 end if
60:             end if
61:         /* Append the newly generated command into the set of final commands */
62:             append(cmd, cmdtp);
63:             T(A,nNode);
64:         end if
65:     end while
66: end procedure
```

Figure 27: Translation Algorithm of SysML Activity Diagrams into MDP - Part 2

nodes is represented by a variable in the corresponding PRISM model. The join node represents a special case since the corresponding control passing rule is not straightforward [21] compared to the other control nodes rules. More precisely, a join node has to wait for a control locus on each incoming edge in order to be traversed. Thus, we need to keep a variable for each pin of a given join node. We also define a boolean formula corresponding to the condition of synchronization at each join node. Moreover, we allow multiple instances of execution and thus the number of tokens in a given node is represented by an integer number denoting active instances at a certain point in time. At this point, we consider that in realistic systems a certain number of instances are active at the same time. Therefore, we model each variable as being an integer within a range $[0..max\_inst]$ where the constant *max_inst* represents the maximum supported number of instances. This value can be tailored according to the application's needs.

Apart from the variables, the commands encode the behavior dynamics captured by the diagram. Thus, each possible progress of the control locus corresponds to a command in PRISM code. The predicate guard of a given command corresponds to the precondition for triggering the control passing and the updates represent its effect on the global state. A given predicate guard expresses the ability of the source nodes to pass the control and the destination nodes to accept it. A given update expresses the effect that has the passing of control on the number of active instances of the source and destination nodes. For instance, the fork node `F1` in Figure 29 passes the control to each of its outgoing edges if first it possesses at least one control locus and second the destination nodes are able to receive the token (did not reach their maximum number of instances). The modification in the control

```
1: function C(n, g, u, n', p)
2:     var := prismElement(n');
3:     if type(n')=flowfinal then
4:         /* Generate the final PRISM command */
5:         cmdtp := command(n,g,u,p);
6:     end if
7:     if type(n')=final then
8:         u' := inc(var);
9:         cmdtp := command(n, g, and(u,u'), p);
10:    end if
11:    if type(n')=join then
12:        /* Return the PRISM variable related to a specific pin of the join */
13:        varpin :=pinPrismElement(n,n');
14:        varn :=prismElement(n);
15:        g1 := not(varn);
16:        g2 :=less(varpin,max);
17:        g' := and(g1,g2);
18:        u' :=inc(varpin,1);
19:        cmdtp = command(n,and(g,g'),and(u,u'),p);
20:    end if
21:    if type(n') in {action, merge, fork, decision, pdecision} then
22:        g' := less(var,max);
23:        u' := inc(var,1);
24:        cmdtp = command(n,and(g,g'),and(u,u'),p);
25:    end if
   return cmdtp;
26: end function
```

Figure 28: Function Generating PRISM Commands

configuration has to be reflected in the updates of the command, where the fork node looses one control locus and the number of active instances of the destination nodes increases. The corresponding PRISM command can be written as follows:

```
[F1] F1>0 & Autofocus<max_inst & DetLight<max_inst &
                                  D3<max_inst & !End  →
            F1'=F1-1 & Autofocus'=Autofocus-1 &
                    DetLight'=DetLight-1 & D3'=D3-1;
```

This dependency of the predicates and updates on the nodes at source and at destination

of the control passing inspired us with the systematic mapping procedure. In fact, the principle underlying our algorithm is that the predicates and updates for the source and destination nodes are generated separately so that when composed together provide the whole final command. The commands are generated according to the type of the source node and the number of outgoing edges. For instance, in the case where the source node is a non-probabilistic decision node, the algorithm generates as many commands as outgoing edges. Concerning the probabilistic decision node, a single command is needed, where the updates are the sum of all probabilistic occurrences that are associated with different probabilistic choices. For a fork node, a single command enables all the outgoing target nodes. Finally, a single command is enough for nodes with a unique outgoing edge such as action, join, merge, and initial.

The algorithm translating SysML activity diagrams into the input language of PRISM is presented in Figure 26, Figure 27, and Figure 28. The algorithm visits the activity nodes using a depth-first search procedure and generates on the fly the PRISM commands. The main procedure $T(A,N)$ is illustrated in Figure 26 and continued in Figure 27. Initially, the main procedure $T(A,\{N_0\})$ is called where A is the data structure representing the activity diagram and N_0 is the initial node. Then, it is called recursively, where N represents the set (possibly singleton) of next nodes to be explored. The algorithm uses a function $C(n, g, u, n', p)$ illustrated in Figure 28 where n is the current node representing the action name of the command, g and u are expressions, n' is the destination node of n. The function C serves the generation of different expressions related to the destination node n' and it returns the final resulting command to be appended into the output of the main

algorithm.

We make use of the usual *Stack* data structure with the fundamental operations such as *pop*, *push*, and *empty*. We define user defined types such as:

- *PrismCmd* : a record type containing the fields act, grd, and upd corresponding respectively to the action, the guard, and the update of the command of type *PrismCmd*.
- *Node* : a type defined to handle activity nodes.
- *PRISMVarId* : a type defined to handle PRISM variables identifiers.

The variable *nodes* is of type $Stack$ and serves to store temporarily the nodes to be explored by the algorithm. At each while iteration, a current node *cNode* is popped from the stack *nodes* and its destination nodes in the activity diagram are stored in the list of nodes *nNode*. These destination nodes will be pushed in the stack in the next recursive call of the main algorithm. If the current node is already visited by the algorithm it is stored in the set of nodes *vNode*. According to the type of current node, the parameters to be passed to the function C are computed. We denote by *varfinal* the PRISM variable identifier of the final node and *vars* represents the set of all PRISM variables of the current activity diagram. Finally, *max* is a constant value specifying the maximum value of all PRISM variables (of type integer). The algorithm terminates when the stack is empty and no instance of the main algorithm is running. All the PRISM commands generated by the algorithm T are appended into a list of commands cmd (using the utility function *append*), which allows us to build the performance model.

We make use of the following utility functions:

- The functions *type*, *next*, and *label* are related to the access to the activity diagram structure and components.

- The function PRISMELEMENT takes a node as parameter and returns the PRISM element (either a variable of type integer or a formula) associated with the node.

- The function PINPRISMELEMENT takes two nodes as parameters where the second is a $join$ node and returns the PRISM variable related to the specific pin.

- Different functions are used in order to build expressions needed in the guard or the updates of the commands. The function *raz* returns the expression that is the conjunction of the reseting of the variables taken as parameter to their default values. The function *grt*(x,y) returns the expression $x > y$, while function *less*(x,y) returns the expression $x < y$. The function *dec*(x) returns the expression $x' = x - 1$. The function *inc*(x) returns the expression $x' = x + 1$. The function *not*(x) returns the expression $!x$. The function *and*(x,y) returns the expression $x\&y$. The function *eq*(x,y) returns the expression $x = y$. The function *set*(x,y) returns the expression $x' = y$.

- The Π is the conventional projection that takes two parameters, a pair (x, y) and an $index$ (1 or 2) and returns x, if $index = 1$, and y if $index = 2$.

- The function *pinsOf* takes as input the PRISM formula corresponding to a join node and extracts the corresponding pins variables into a list of prism variables.

- The function *command* takes as input the action name a, the guard g, the update u, the probability of the update p in this order and returns the expression $[a]\ g \rightarrow p : u$.

- The function *merge* merges two sub-commands taken as parameters into one command consisting of a set of probabilistic updates. More precisely, it takes two parameters $cmdtp1 = [a]\ g1 \rightarrow p : u1$ and $cmdtp1 = [a]\ g2 \rightarrow q : u2$ then generates the command $[a]\ g1\ \&\ g2 \rightarrow p : u1 + q : u2$.

5.3.2 Rewards Mechanism for Timed Actions

MDP is a discrete-probabilistic model that treats time as discrete steps. In order to handle quantitative assessment of timed actions, we propose to augment the generated MDP model with a Markov reward mechanism. The latter is a mechanism used in order to model quantitative measures (such as energy, cost, etc.) [197] and thus to analyze performance and reliability of systems. There are two possible types of rewards: state reward and transition reward. The former is a function $\rho : S \rightarrow \mathbb{R}$ that associates for each state a real value. The value $\rho(s)$ is the reward acquired in the state s per time step [198]. The latter is a function $\rho' : S \times S \rightarrow \mathbb{R}$ that associates for each transition a real value and $\rho'(s, s')$ denotes the reward acquired each time the transition is fired. We choose to use transition reward in order to model time-passing while executing related actions. According to the algorithm, we label each PRISM command with the action name of the source node, which passes the control locus to the destination node when the current command executes. More precisely, the label corresponds to the termination of the action at the source node. Consequently, each time that an action a terminates, which means the corresponding command is triggered, we add $\Delta(a)$ time units to the global reward "time".

5.4 PCTL* Property Specification

In order to apply probabilistic model-checking on the MDP model resulting from the translation algorithm, we need to express the properties in an appropriate temporal logic. For MDP models, we can use either LTL [63], PCTL [74] or PCTL* [63]. The Probabilistic Computation Tree Logic (PCTL) [74] is an extension of CTL [64] mainly with the probability operator $\mathcal{P}$. PCTL* subsumes PCTL and LTL [63]. It is based on PCTL where arbitrary combinations of paths formulas but only propositional state formulas are allowed [199].

PCTL* syntax according to [199] is as follows:

$$\phi ::= true \mid a \mid \neg \phi \mid \phi \wedge \phi \mid \mathcal{P}_{\bowtie p}[\psi] \mid \mathcal{R}_{\sim r}[F\phi]$$

$$\psi ::= \phi \mid \psi_1 \mathcal{U}^t \psi_2 \mid \psi_1 \mathcal{U} \psi_2 \mid \mathcal{X}\psi \mid \psi_1 \wedge \psi_2 \mid \neg \psi$$

where a is an atomic proposition, $t \in \mathbb{N}$, $p \in [0,1] \subset \mathbb{R}$, $\bowtie \in \{>, \geq, <, \leq\}$, and $\mathcal{R}$ represents the reward operator. It extends PCTL and PCTL* in order to handle reward properties.

PRISM extends the latter syntax in order to quantify probability values with the operator $\mathcal{P}=?$ and reward values with the operator $\mathcal{R}=?$. For the case of MDP, all non-determinism has to be resolved. Thus, properties quantifying the probability actually reason about the minimum or maximum probability, over all possible resolutions of non-determinism, that a certain type of behaviour is observed. Measuring the minimum/maximum probabilities provides the worst/best-case scenarios. Moreover, the reward property is expressed in terms of minimum/maximum reachability reward.

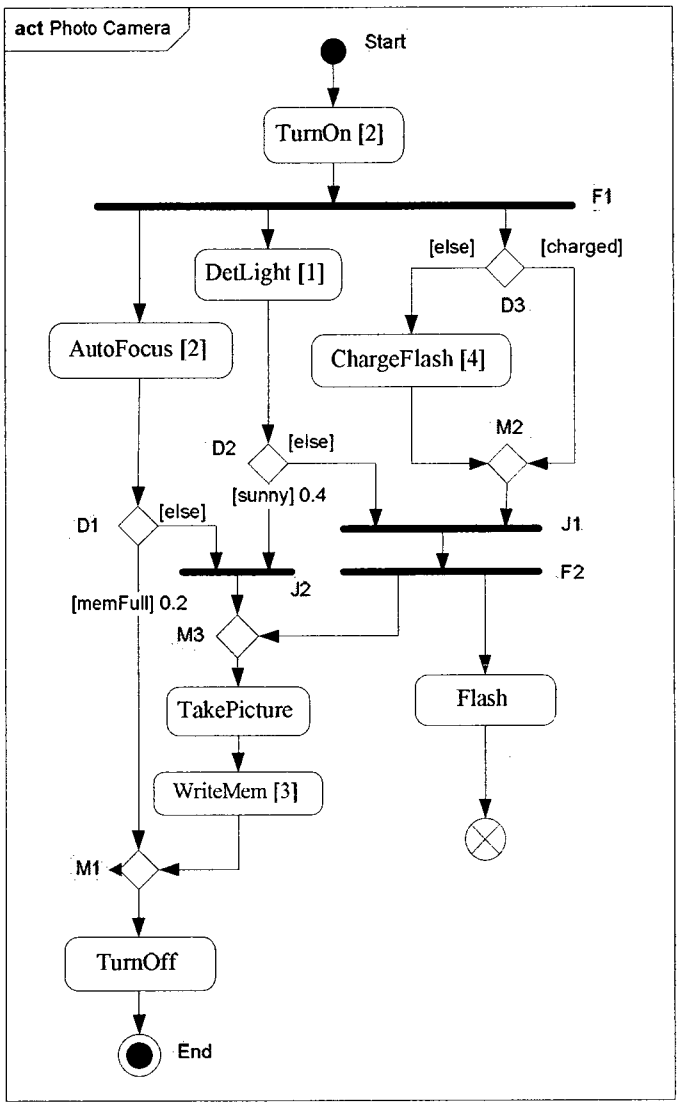


Figure 29: Case Study: Digital Camera Activity Diagram - Flawed Design

```
mdp
const int max_inst = 1;
formula J1 = J1_pin1>0 & J1_pin2>0 ;
formula J2 = J2_pin1>0 & J2_pin2>0 ;

module mainmod
memful : bool init false;
sunny : bool init false;
charged : bool init false;
Start : bool init true; TurnOn : [0 .. max_inst] init 0; F1 : [0 .. max_inst] init 0;
Autofocus : [0 .. max_inst] init 0; DetLight : [0 .. max_inst] init 0;
D3 : [0 .. max_inst] init 0; ChargeFlash : [0 .. max_inst] init 0;
D1 : [0 .. max_inst] init 0; D2 : [0 .. max_inst] init 0; F2 : [0 .. max_inst] init 0;
J1_pin1 : [0 .. max_inst] init 0; J1_pin2 : [0 .. max_inst] init 0;
J2_pin1 : [0 .. max_inst] init 0; J2_pin2 : [0 .. max_inst] init 0;
M1 : [0 .. max_inst] init 0; M2 : [0 .. max_inst] init 0; M3 : [0 .. max_inst] init 0;
TakePicture : [0 .. max_inst] init 0; WriteMem : [0 .. max_inst] init 0;
Flash : [0 .. max_inst] init 0; TurnOff : [0 .. max_inst] init 0; End : bool init false;

[Start] Start & TurnOn<max_inst & !End → Start'=false & TurnOn'=TurnOn + 1;

[TurnOn] TurnOn>0 & F1<max_inst & !End → TurnOn'=TurnOn - 1 & F1'=F1 + 1;

[F1] F1>0 & Autofocus<max_inst & DetLight<max_inst & D3<max_inst & !End →
 F1'= F1 - 1 & Autofocus'=Autofocus + 1 & DetLight'=DetLight + 1 & D3'=D3 + 1;

[Autofocus] Autofocus>0 & D1<max_inst& !End →
 Autofocus'=Autofocus - 1 & D1'=D1 + 1;

[DetLight] DetLight>0 & D2<max_inst& !End →
DetLight'= DetLight - 1 & D2' = D2 + 1 ;

[D3] D3>0 & ChargeFlash <max_inst& !End →
 ChargeFlash'= ChargeFlash + 1 & D3'=D3 - 1& (charged'=false);

[D3] D3>0 & M2<max_inst& !End →
  M2' = M2 + 1 & D3'= D3 - 1& (charged'=true);

[D1] D1>0   & M1<max_inst &  J2_pin1<max_inst & !J2 & !End →
0.2 : (M1'=M1 + 1) & (D1'=D1 - 1)& (memful' = true) +
0.8 : (J2_pin1'=J2_pin1 + 1) & (D1'=D1 - 1)& (memful' = false);

[D2] D2>0 & J2_pin2<max_inst & J1_pin1<max_inst & !J1 & !J2 & !End →
 0.6 : (J1_pin1'=J1_pin1 + 1) & (D2'=D2 - 1) & (sunny'=false) +
 0.4 : (J2_pin2'=J2_pin2 + 1) & (D2'=D2 - 1) & (sunny'=true);

[ChargeFlash] ChargeFlash>0 & M2<max_inst& !End →
 M2'= M2 + 1  & ChargeFlash'=ChargeFlash - 1;
```

Figure 30: PRISM Code for the Digital Camera Case Study - Part1

5.5 Case Study

In order to explain our approach, we present a case study of time-annotated SysML activity diagram of a hypothetical model of a digital photo-camera device. The diagram captures the functionality of taking a picture and is illustrated in Figure 29. The corresponding dynamics are rich enough to allow for the verification of several interesting properties that capture important functional aspects and performance characteristics. We deliberately modeled some flaws in the design in order to demonstrate the applicability and the benefits of our approach. The process captured by the digital photo-camera activity diagram starts by turning on the camera (TurnOn). Subsequently, three parallel execution flows are spawned. The first one begins by (AutoFocus) followed by a decision checking the status of the memory (memFull guard). In the case where the memory is full, the camera cannot be used and it is turned off. The second parallel flow is dedicated to the detection of the ambient lighting conditions (DetLight) and it determines whether the flash is needed in order to take a picture. The third flow allows charging the flash (ChargeFlash) if not already charged. The action (TakePicture) executes for two possible reasons: either it is sunny (sunny = true) and the memory is not full (memFull = false) or the flash (Flash) is needed because of the lack of luminosity (sunny=false). Thereafter, the picture is stored in the memory of the camera (WriteMem) and the activity diagram ends with turning off the camera (TurnOff).

By applying our algorithm on the SysML activity diagram case study, we end up with the MDP described using PRISM language as shown in Figure 30 and continued in Figure 31. The reward structure enabling the specification of timed action and the verification of

```
[M2] M2>0    & J1_pin2<max_inst & !J1 & !End  →
M2'= M2 - 1 & J1_pin2'=J1_pin2 + 1;

[M1]  M1>0 & TurnOff<max_inst    & !End →
 TurnOff'=TurnOff + 1 & M1'= M1  - 1;

[J2]  J2  & TakePicture<max_inst & !End →
  TakePicture'=TakePicture + 1 &  J2_pin1'=0 & J2_pin2'=0 ;

[J1]  J1 & F2<max_inst & !End →
F2'=F2 + 1 & J1_pin1'=0 & J1_pin2'=0 ;

[F2] F2>0    & Flash<max_inst & TakePicture<max_inst & !End →
  F2'=F2 - 1 & Flash'= Flash  + 1 & TakePicture'=TakePicture + 1;

[TakePicture] TakePicture>0 & WriteMem<max_inst & !End →
TakePicture' = TakePicture - 1 & WriteMem'= WriteMem  + 1;

[WriteMem] WriteMem>0 & M1<max_inst & !End →
  WriteMem'= WriteMem - 1 & M1'=M1  + 1;

[TurnOff] TurnOff>0 & !End →
  TurnOff'=TurnOff - 1 & End'=true;

[End] End    →
  TurnOn'=0 & F1'=0 & Autofocus'=0 & DetLight'=0 & D3'=0 & ChargeFlash'=0 & D1'=0
& D2'=0 & J1_pin1'=0 & J1_pin2'=0 & F2'=0  & J2_pin1'=0 & J2_pin2'=0 & M1'=0 &
 M2'=0 & M3'=0 & TakePicture'=0 &  WriteMem'=0 & Flash'=0 & TurnOff'=0&
 (memful' = false) & (sunny'=false) & (charged'=false);

endmodule
```

Figure 31: PRISM Code for the Digital Camera Case Study - Part2

time-related properties specifications is shown Figure 32 appended into the PRISM model. After supplying the model to PRISM, the latter constructs the reachable state space in the form of a state list and a transition probability matrix.

At the beginning, one can look for the presence of deadlock states in the model. This is expressed using the property 5.5.1. It is also possible to quantify the worst/best-case probability of such a scenario happening using properties 5.5.2 and 5.5.3.

```
rewards "time"
[TurnOn] true : 2;
[Autofocus] true : 2;
[DetLight] true : 1;
[ChargeFlash] true : 4;
[WriteMem] true : 3;
endrewards
```

Figure 32: Reward Structure for the Digital Camera Case Study

$$\texttt{"init"} \Rightarrow \texttt{P} > 0\,[\,\texttt{F "deadlock"}\,] \tag{5.5.1}$$

$$\texttt{Pmax} =?\,[\,\texttt{F "deadlock"}\,] \tag{5.5.2}$$

$$\texttt{Pmin} =?\,[\,\texttt{F "deadlock"}\,] \tag{5.5.3}$$

The labels `"init"` and `"deadlock"` in property 5.5.1 are built-in labels that are true for respectively initial and deadlocked states. Property 5.5.1 states that from an initial state the probability of reaching eventually a deadlocked state is greater than 0. This returns *true*, which means that the property is satisfied in some states of the model. However, after more investigations, we found that there is only one deadlocked state due to the activity final node in the activity diagram. This deadlock can be accepted since according to the desired execution, at the activity final node, the activity terminates and there are no outgoing transitions.

It is also important in the case of activity diagram to verify that we can eventually reach the activity final node once the activity diagram has started. Such a property is stated in Property 5.5.4. The properties 5.5.5 and 5.5.6 are used in order to quantify the probability of such a scenario to happen.

$$\texttt{TurnOn} >= 1 \;\Rightarrow\; \texttt{P} > 0\ [\texttt{F End}] \tag{5.5.4}$$

$$\texttt{Pmax} =?\ [\ \texttt{F End}\ ] \tag{5.5.5}$$

$$\texttt{Pmin} =?\ [\ \texttt{F End}\ ] \tag{5.5.6}$$

Property 5.5.4 returns *true* and properties 5.5.5 and 5.5.6 both return the probability value 1. This represents satisfactory results since the final activity is always reachable.

The first functional requirement states that the `TakePicture` action should not be activated if the memory is full `memfull=true` or if the `Autofocus` action is still on-going. Thus, we would like to evaluate the actual probability for this scenario to happen. Since, we are relying on MDP model, we need to compute the minimum (5.5.7) and the maximum (5.5.8) probabilities measures of reaching a state where, either the memory is full or the focus action is ongoing while taking a picture.

$$\texttt{Pmin} =?\ [\ \texttt{true U (memfull} \mid \texttt{Autofocus} \geq 1)\& \texttt{TakePicture} \geq 1\ ] \tag{5.5.7}$$

$$\texttt{Pmax} =?\ [\ \texttt{true U (memfull} \mid \texttt{Autofocus} \geq 1)\& \texttt{TakePicture} \geq 1\ ] \tag{5.5.8}$$

The expected likelihood for this scenario should be null (impossibility). However, the model-checker determines a non-zero probability value for the maximum measurement (`Pmax` = 0.6) and a null probability for the minimum. This shows that there is a path leading to such undesirable state, thus pointing out to a flaw in the design. On the activity diagram, this is caused by the existence of a control flow path leading to the `TakePicture` action independently of the evaluation of the `memfull` guard and of the termination of the action `AutoFocus`. In order to correct this misbehavior, the designer must alter the diagram

such that the control flow reaching the action AutoFocus and subsequently evaluating the guard memfull to false have to synchronize with all the possible paths leading to TakePicture. This might be done using a fork node that splits two thread each having to synchronize with a possible flow before activating TakePicture. Thus, we block the activation of TakePicture action unless AutoFocus eventually ends and memory space is available in the digital camera. Figure 33 illustrates the corrected SysML activity diagram taking into account the discovered problem.

As the main function of the digital photo camera device is to take pictures, we would like to measure the probability of taking a picture in the normal conditions. The corresponding properties are specified as follows:

$$\texttt{Pmin} =? [\ \texttt{true U TakePicture} \geq 1] \tag{5.5.9}$$

$$\texttt{Pmax} =? [\ \texttt{true U TakePicture} \geq 1] \tag{5.5.10}$$

The measures provided by the model-checker are respectively Pmin = 0.8 and Pmax = 0.92. These values have to be compared with the desired level of reliability of the system.

Another interesting time-related property concerns the time needed for the activity to be executed once. This requirement is related to the performance of the digital camera device in term of shots per time unit. This can be stated using a reward-related property as follows:

$$\texttt{Rmin} =? [\ \texttt{F End}] \tag{5.5.11}$$

$$\texttt{Rmax} =? [\ \texttt{F End}] \tag{5.5.12}$$

Property (5.5.11) and property (5.5.12) are the minimal and the maximal expected reachability rewards. The results after the verification provided by the model-checker are `Rmin` = 6.56 and `Rmax` = 11.4.

We applied probabilistic model-checking on the corrected design in order to compare both the flawed and corrected SysML activity diagrams. The comparison is summarized in Table 5. The correction of the design removed the flaw revealed by property (5.5.8) since the probability value became 0. However, we lost in terms of reliability in the best case scenario, since the maximum probability calculated for property (5.5.10) has dropped to 0.8 instead of 0.92. Moreover, the minimum time reward has increased in property (5.5.11) from 6.56 to 7.2, which decreases the performance of the digital camera in the worst case scenario.

Properties	Flawed Design	Corrected Design
(5.5.1)	*true*	*true*
(5.5.2)	1	1
(5.5.3)	1	1
(5.5.4)	*true*	*true*
(5.5.5)	1	1
(5.5.6)	1	1
(5.5.7)	0.0	0.0
(5.5.8)	0.6	0.0
(5.5.9)	0.8	0.8
(5.5.10)	0.92	0.8
(5.5.11)	6.56	7.2
(5.5.12)	11.4	11.4

Table 5: Comparative Assessment of Flawed and Corrected Design Models

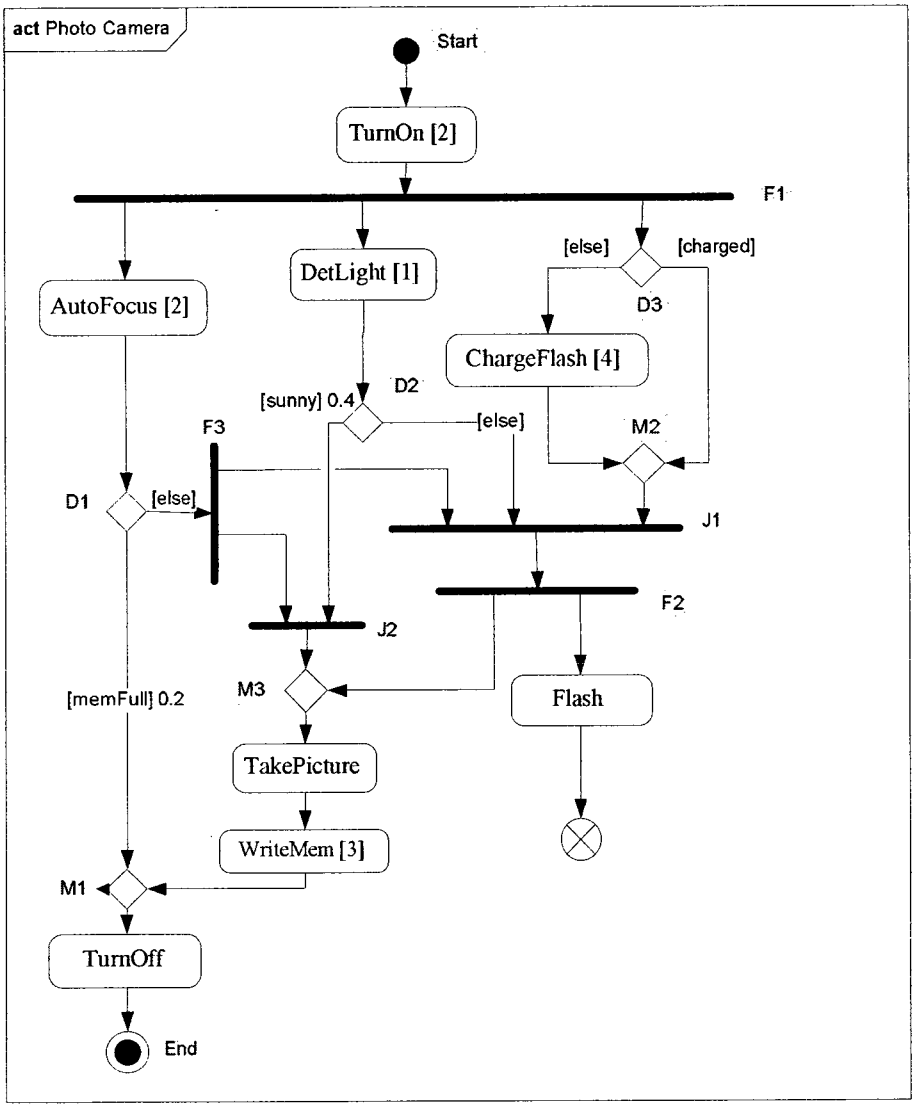


Figure 33: Case Study: Digital Camera Activity Diagram - Corrected Design

5.6 Conclusion

This chapter presented a translation algorithm that we designed and implemented in order to enable probabilistic model-checking of SysML activity diagrams. The algorithm maps SysML activity diagrams into code written using the input language of the selected probabilistic model-checker PRISM. Moreover, a case study is presented in order to show the practical use of our approach. Once the translation is done, establishing confidence in its correctness is necessary. Thus, we aim at focusing in the next chapters on the soundness property of the algorithm. One of the prerequisites for proving soundness is to define formally the semantic foundations of SysML activity diagrams. This represents the topic of the next chapter.

Chapter 6

Semantic Foundations of SysML Activity Diagrams

In this chapter, we propose to study the semantic foundations of SysML activity diagrams. A formalization of the semantics will allow us to build a sound and rigorous framework for the V&V of design models expressed using these diagrams. To this end, we design a dedicated formal language, called Activity Calculus (AC), used in order to mathematically express and analyze the behaviors captured by SysML activity diagrams. In the sequel, the syntactic and semantic definitions of the AC language are presented in Section 6.1. Therein, a summary of the informal mapping of the diagram constructs into AC terms with an illustrative example are also provided. In order to illustrate the usefulness of such a formal semantics, a case study is presented in Section 6.2 consisting of a SysML activity diagram for an hypothetical design of a banking operation on an Automated Teller Machine (ATM). We apply the semantic rules on the case study in order to show how this may

uncover subtle errors in the design. Finally, Section 6.3 defines the underlying Markov decision process that describes SysML activity diagram semantics.

6.1 Activity Calculus

The activity calculus is built with the goal in mind to provide a dedicated calculus that captures the rich expressiveness of activity diagrams and formally models the behavioral aspects using operational semantics framework. It is mainly inspired by the concept of process algebras, which represent a family of approaches used for modeling concurrent and distributed systems.

Apart from ascribing a rigorous meaning to the informally specified diagrams, formal semantics provides us with an effective technique to uncover design errors that could be missed by intuitive inspection. Furthermore, it allows the application of model transformations and model-checking. Practically, the manipulation of the graphical notations as it is defined in the standard does not provide the flexibility offered by a formal language. There is a real need to describe this behavior in a mathematical and a rigorous way. Thus, our formal framework allows the automation of the validation using existing techniques such as probabilistic model-checking. Moreover, it allows reasoning about potential relations between activity diagrams from the behavioral perspective and deriving related mathematical proofs.

To the best of our knowledge, this is the first calculus of its kind that is dedicated to capture the essence of SysML activity diagrams. While reviewing the state of the art, we

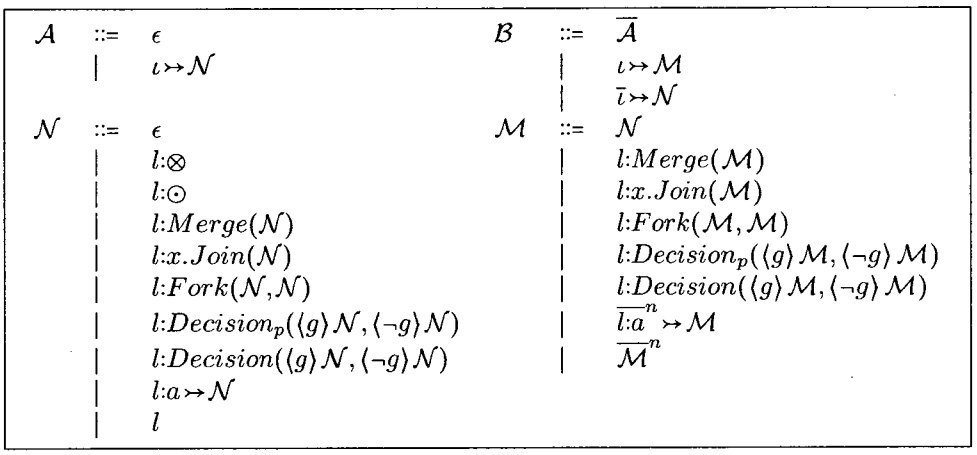
cannot find proposals along the same line as our activity calculus. With respect to UML 2.x activity diagrams, most of the research initiatives use existing formalisms such as CSP in [178], the Interactive Markov Chain (IMC) in [171], and variants of Petri nets formalism in [179–181]. Although these formalisms have well-established semantic domains, they impose some serious limitations to the expressiveness of activity diagrams (e.g. disallow multiple instance of actions). The majority of reviewed initiatives express the activity diagrams as data structure tuple. Very few proposals provide a dedicated algebraic-like notation [171, 200], where only [171] aims at defining a semantic framework. The main supported features that make our calculus distinguishable is its ability to express various control flows with mixed and nested forks and joins that are allowed by UML specification. Furthermore, AC allows multiple instances and includes both guarded and probabilistic decision. Finally, AC allows us to define an operational semantics for SysML activity diagrams that is intuitive and original based on tokens-propagation. In the following, we explain in details the syntax and semantics of our activity calculus.

6.1.1 Syntax

From the structural perspective, an activity diagram can be viewed as a directed graph with two types of nodes (action and control nodes) connected using directed edges. Alternatively, from the dynamic perspective, the activity diagram behavior amounts to a specifically ordered execution of its actions. This order depends on the propagation of the control locus (token) that starts from the initial node. When an action receives a token, it becomes

active and starts executing. When its execution terminates, it delivers the token to its outgoing edges. Moreover, multiple instances of the same action may execute concurrently if more than one control token is received. During the execution, the activity diagram structure remains unchanged, however, the location of the control tokens changes. Thus, the behavior (the meaning) depicted by the activity diagram can be described using a set of progress rules that dictates the tokens movement through the diagram. In order to specify the presence of control tokens, we use the word marking (borrowed from the Petri net formalism).

We assume that each activity node in the diagram (except initial) is assigned a unique label. Let $\mathcal{L}$ be a collection of labels ranged over by l, l_0, l_1, $\cdots$ and N any node (except initial) in the activity diagram. We write $l{:}N$ to denote an l-labeled activity node N. Labels serve different purposes. Mainly, a label l is used for uniquely referring to an l-labeled activity node in order to model a flow connection to the already defined node. Particularly, labels are useful for connecting multiple incoming flows towards merge and join nodes. The syntax of the AC language is defined using the Backus-Naur-Form (BNF) notation in Figure 34. The AC terms are generated using this syntax. We can distinguish two main syntactic categories: unmarked terms and marked terms. An unmarked AC term, typically given by $\mathcal{A}$, corresponds to the diagram without tokens. A marked AC term, typically given by $\mathcal{B}$, corresponds to an activity diagram with tokens. The difference between these two categories is the added "overbar" symbol for the marked terms (or sub-terms) denoting the presence and the location of a token. A marked term is typically used to denote an activity diagram while its execution is in progress. The idea of decorating the syntax was



$\mathcal{A}$	::=	ϵ	$\mathcal{B}$	::=	$\overline{\mathcal{A}}$
	|	$\iota \rightarrowtail \mathcal{N}$		|	$\iota \rightarrowtail \mathcal{M}$
				|	$\overline{\iota} \rightarrowtail \mathcal{N}$
$\mathcal{N}$	::=	ϵ	$\mathcal{M}$	::=	$\mathcal{N}$
	|	$l{:}\otimes$		|	$l{:}Merge(\mathcal{M})$
	|	$l{:}\odot$		|	$l{:}x.Join(\mathcal{M})$
	|	$l{:}Merge(\mathcal{N})$		|	$l{:}Fork(\mathcal{M},\mathcal{M})$
	|	$l{:}x.Join(\mathcal{N})$		|	$l{:}Decision_p(\langle g\rangle\mathcal{M},\langle\neg g\rangle\mathcal{M})$
	|	$l{:}Fork(\mathcal{N},\mathcal{N})$		|	$l{:}Decision(\langle g\rangle\mathcal{M},\langle\neg g\rangle\mathcal{M})$
	|	$l{:}Decision_p(\langle g\rangle\mathcal{N},\langle\neg g\rangle\mathcal{N})$		|	$\overline{l{:}a}^{n} \rightarrowtail \mathcal{M}$
	|	$l{:}Decision(\langle g\rangle\mathcal{N},\langle\neg g\rangle\mathcal{N})$		|	$\overline{\mathcal{M}}^{n}$
	|	$l{:}a \rightarrowtail \mathcal{N}$			
	|	l			

Figure 34: Unmarked Syntax (left) and Marked Syntax (right) of Activity Calculus

inspired by the work on Petri net algebra in [201]. However, we extended this concept in order to handle multiple tokens. We discard the intuitive but useless solution to write the expression $\overline{\overline{\mathcal{N}}}$ to denote a term $\mathcal{N}$ that is marked twice since it can result in overwhelming unmanageable marked AC terms if the number of tokens grows. Thus, we augment the "overbar" operator with an integer n such that $\overline{\mathcal{N}}^{n}$ denotes a term marked with n tokens. This allows us to consider loops in activity diagrams and so multiple instances.

Referring to Figure 34, the definition of the term $\mathcal{B}$ is based on $\mathcal{A}$, since $\mathcal{B}$ represents all valid sub-terms with all possible locations of the overbar symbol on top of $\mathcal{A}$ sub-terms. $\mathcal{N}$ defines an unmarked sub-term and $\mathcal{M}$ represents a marked sub-term of $\mathcal{A}$. An AC term $\mathcal{A}$ is either ϵ, to denote an empty activity or $\iota \rightarrowtail \mathcal{N}$, where ι specifies the initial node and $\mathcal{N}$ can be any labeled activity node (or control flows of nodes). The symbol$\rightarrowtail$is used to specify the activity control flow edge. The derivation of an AC term is based on a depth-first traversal of the corresponding activity diagram. Thus, the mapping of activity diagrams into AC terms is achieved systematically. It is important to note that, as a syntactic convention,

each time a new merge (or join) node is met, the definition of the node and its newly assigned label are considered. If the node is encountered later in the traversal process, only its corresponding label is used. This convention is important to ensure well-formedness of the AC terms.

Among the basic constructs of $\mathcal{N}$, we have:

- The term $l{:}\otimes$ (resp. $l{:}\odot$) specifies the flow final node (resp. the activity final node).
- The term $l{:}Merge(\mathcal{N})$ (resp. $l{:}x.Join(\mathcal{N})$) represents the definition of the merge (resp. join) node. This notation is used only when the corresponding node is firstly encountered during the depth-first traversal of the activity diagram. The parameter $\mathcal{N}$ inside the merge (resp. join) refers to the subsequent destination nodes (or flow) connected to the outgoing edge of the merge (resp. join) node. With respect to the join node, the entity x represents an integer specifying the number of incoming edges into this specific join node.
- The term $l{:}Fork(\mathcal{N}_1, \mathcal{N}_2)$ is the construct referring to the fork node. The parameters $\mathcal{N}_1$ and $\mathcal{N}_2$ represent the sub-terms corresponding to the destination of the outgoing edges of the fork node (i.e. the flows split in parallel).
- The term $l{:}Decision_p(\langle g\rangle \mathcal{N}_1, \langle \neg g\rangle \mathcal{N}_2)$ (resp. $l{:}Decision(\langle g\rangle \mathcal{N}_1, \langle \neg g\rangle \mathcal{N}_2)$) specifies the probabilistic (resp. non-probabilistic) decision node. It denotes a probabilistic (resp. non-probabilistic guarded) choice between alternative flows $\mathcal{N}_1$ and $\mathcal{N}_2$. For the probabilistic case, the sub-term $\mathcal{N}_1$ is selected with a probability p whereas, $\mathcal{N}_2$ is selected with probability $1 - p$.

- The term $l{:}a \rightarrowtail \mathcal{N}$ is the construct representing the prefix operator: The labeled action $l{:}a$ is connected to $\mathcal{N}$ using a control flow edge.

- The term l is a reference to a node labeled with l.

A marked term $\mathcal{B}$ is either $\overline{\mathcal{A}}$ or $\overline{\iota} \rightarrowtail \mathcal{N}$, which denotes the initial node ι marked with one token and connected to the unmarked sub-term $\mathcal{N}$, or $\iota \rightarrowtail \mathcal{M}$ that denotes an unmarked initial node that is connected to the marked sub-term $\mathcal{M}$. Among the basic constructs of $\mathcal{M}$, we have:

- The term $\mathcal{N}$ is a special case of an AC marked term where $n = 0$.

- The term $\overline{\mathcal{M}}^n$ denotes a term $\mathcal{M}$ that is marked with n other tokens such that $n \geq 0$.

- The term $l{:}Merge(\mathcal{M})$ (resp. $l{:}x.Join(\mathcal{M})$) represents the definition of an unmarked merge (resp. join) term with a marked sub-term $\mathcal{M}$.

- The term $l{:}Fork(\mathcal{M}_1, \mathcal{M}_2)$ represents an unmarked fork term with two marked sub-terms $\mathcal{M}_1$ and $\mathcal{M}_2$.

- The term $l{:}Decision(\langle g \rangle \mathcal{M}_1, \langle \neg g \rangle \mathcal{M}_2)$ (resp. $l{:}Decision_p(\langle g \rangle \mathcal{M}_1, \langle \neg g \rangle \mathcal{M}_2)$) denotes an unmarked decision (resp. probabilistic decision) term having two marked sub-terms $\mathcal{M}_1$ and $\mathcal{M}_2$.

- The term $\overline{l{:}a}^n \rightarrowtail \mathcal{M}$ denotes a prefix operator with a n-times marked action connected to a marked sub-term $\mathcal{M}$.

An important observation has to be made. Since the "overbar" symbol represents the presence (and eventually the location) of tokens, one may picture these tokens graphically

(M1)	$\overline{\iota \rightarrowtail \mathcal{N}} \preccurlyeq_M \bar{\iota} \rightarrowtail \mathcal{N}$
(M2)	$\overline{\overline{l{:}a}^{k} \rightarrowtail \mathcal{M}}^{n} \preccurlyeq_M \overline{l{:}a}^{n+k} \rightarrowtail \mathcal{M}$
(M3)	$\overline{l{:}\otimes}^{n} \preccurlyeq_M l{:}\otimes$
(M4)	$\overline{\overline{\mathcal{M}}^{k}}^{n} \preccurlyeq_M \overline{\mathcal{M}}^{n+k}$
(M5)	$\mathcal{B}[\overline{l{:}Merge(\mathcal{M})}^{n}, \bar{l}^{k}] \preccurlyeq_M \mathcal{B}[\overline{l{:}Merge(\mathcal{M})}^{n+k}, l]$

Figure 35: Marking Pre-order Definition

on the activity diagram using small solid squares (to not mix with initial node notation) similarly to the Petri-net tokens. This is not part of the UML notation, but it is only meant for illustration purposes. This exercise may reveal that two marked expressions may refer to the same activity diagram structure annotated with tokens that can be considered to be in the same locations. For instance, this is the case for the marked expressions $\overline{\iota \rightarrowtail \mathcal{N}}$ and $\bar{\iota} \rightarrowtail \mathcal{N}$. More precisely, the term $\overline{\iota \rightarrowtail \mathcal{N}}$ denotes an activity diagram with a token on the top of the whole diagram. This configuration is exactly the same as having the token placed in the initial element of the diagram, which is represented by the term $\bar{\iota} \rightarrowtail \mathcal{N}$. This is also the case of $\overline{l{:}a \rightarrowtail \mathcal{N}}$ and $\overline{l{:}a} \rightarrowtail \mathcal{N}$. This complies with [21] stating that "when an activity starts, a control token is placed at each action or structured node that has no incoming edges".

Thus, in order to identify these pairs of marked expressions, we define a pre-order relation denoted by $\preccurlyeq_M$ over the set of marked expressions.

Definition 6.1.1. Let $\preccurlyeq_M \subseteq \mathcal{M} \times \mathcal{M}$ be the smallest pre-order relation defined as specified in Figure 35.

This relation allows us to rewrite M_1 into M_1' in the case where $M_1 \preccurlyeq_M M_1'$ and then apply the semantic rule corresponding to M_1'. This simplifies considerably our operational semantics by keeping it concise. In these settings, we only need the pre-order concept, however $\preccurlyeq_M$ can be extended easily to an equivalence relation using the kernel of this pre-order.

Before discussing the operational semantics, we present first the translation of activity diagram constructs into their corresponding AC syntactic elements then, we express an activity diagram example using AC. The correspondence between the concrete syntax of activity diagrams and the syntax of the calculus is summarized in Figure 36.

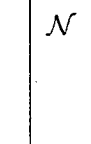
	AD Constructs	AC Syntax
$\mathcal{A}$		$\iota \rightarrowtail \mathcal{N}$
$\mathcal{N}$		$l{:}\odot$
		$l{:}\otimes$
		$l{:}a \rightarrowtail \mathcal{N}$
		$l{:}Decision(\langle g\rangle \mathcal{N}_1, \langle \neg g\rangle \mathcal{N}_2)$
		$l{:}Decision_p(\langle g\rangle \mathcal{N}_1, \langle \neg g\rangle \mathcal{N}_2)$
		$l{:}Merge(\mathcal{N})$ or l
		$l{:}Fork(\mathcal{N}_1, \mathcal{N}_2)$
		$l{:}x.Join(\mathcal{N})$ or l (x is the number of incoming edges)

Figure 36: Mapping Activity Diagram Constructs into AC Syntax

Example 6.1.1. The SysML activity diagram illustrated in Figure 37 denotes the design

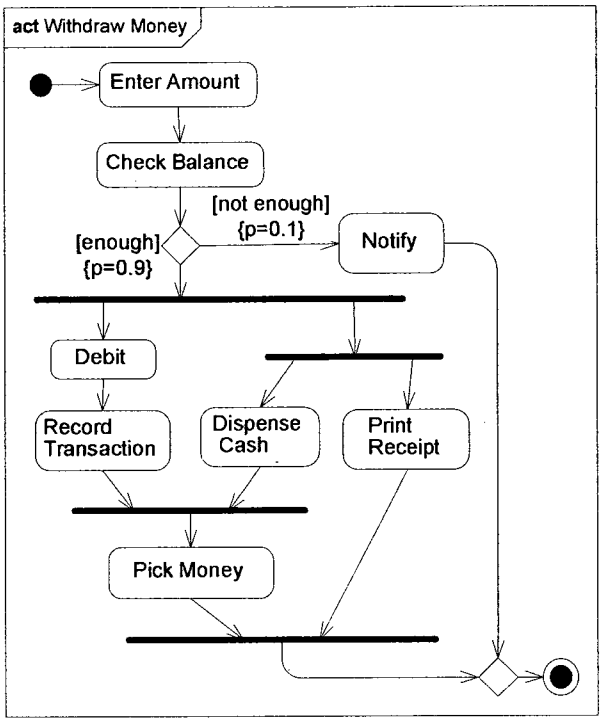


Figure 37: Case Study: Activity Diagram of Money Withdrawal

of a withdraw money banking operation. It can be expressed using the unmarked term $\mathcal{A}_{\texttt{withdraw}}$ as follows:

$$\mathcal{A}_{\texttt{withdraw}} = \iota \rightarrowtail l_1{:}\texttt{Enter} \rightarrowtail l_2{:}\texttt{Check} \rightarrowtail \mathcal{N}_1$$

$$\mathcal{N}_1 = l_3{:}\texttt{Decision}_{0.1}(\langle \text{notenough} \rangle \mathcal{N}_2,\ \langle \text{enough} \rangle \mathcal{N}_3)$$

$$\mathcal{N}_2 = l_4{:}\texttt{Notify} \rightarrowtail l_5{:}\texttt{Merge}(l_6{:}\odot)$$

$$\mathcal{N}_3 = l_7{:}\texttt{Fork}(\mathcal{N}_4,\ l_{13}{:}\texttt{Fork}(l_{14}{:}\texttt{Disp} \rightarrowtail l_{10},\ l_{15}{:}\texttt{Print} \rightarrowtail l_{12}))$$

$$\mathcal{N}_4 = l_8{:}\texttt{Debit} \rightarrowtail l_9{:}\texttt{Record} \rightarrowtail l_{10}{:}2.\texttt{Join}(l_{11}{:}\texttt{Pick} \rightarrowtail l_{12}{:}2.\texttt{Join}(l_5))$$

The $\mathcal{A}_{\texttt{withdraw}}$ term expresses the structure of the activity diagram. One can draw exactly the same activity diagram from its corresponding AC term.

6.1.2 Operational Semantics

In this section, we present the operational semantics of the activity calculus in the Structural Operational Semantics (SOS) style [202]. The latter is a well-established approach that provides a framework to give an operational semantics to many programming and specification languages [202]. It is also considerably applied in the study of the semantics of concurrent processes. Defining such a semantics (small-step semantics) consists in defining a set of axioms and inference rules that are used to describe the operational evolution. Since the propagation of the tokens within the diagram denotes its execution steps, the axioms and rules specify the tokens progress within the corresponding marked AC term. Each axiom and rule specifies the possible transitions between two marked terms of the activity diagram. In some cases, we might have more than one token present in the activity at a given instant. The selection of the progressing token is then performed non-deterministically.

The operational semantics is given by a Probabilistic Transition System (PTS) as presented in Definition 6.1.2. The initial state of the PTS corresponds to place a unique token on the initial node. The initially marked AC term corresponding to $\mathcal{A}$ is the term $\overline{\mathcal{A}}$ where the one overbar is placed on the sub-term ι (i.e. $\bar{\iota} \rightarrowtail \mathcal{N}$) according to (M1) in Figure 35. We denote this marked term by $\mathcal{B}_o$. The general form of a transition is $\mathcal{B} \xrightarrow{\alpha}_p \mathcal{B}'$ or $\mathcal{B} \xrightarrow{\alpha}_p \mathcal{A}$, such that $\mathcal{B}$ and $\mathcal{B}'$ are marked activity calculus terms, $\mathcal{A}$ is the unmarked activity calculus term, $\alpha \in \Sigma \cup \{o\}$, the set of actions ranged over by a, a_1, $\cdots$, b, o denotes the empty action, and $p, q \in [0, 1]$ are probabilities of transitions occurrences. This transition relation shows the marking evolution and means that a marked term $\mathcal{B}$ can be transformed into another marked term $\mathcal{B}'$ or to an unmarked term $\mathcal{A}$ by executing α with a probability p. If a marked

term is transformed into an unmarked term, the transition denotes the loss of the marking. This is the case when either a flow final or an activity final node is reached. For simplicity, we omit the label o on the transition relation, if no action is executed, i.e. $\mathcal{B} \longrightarrow_p \mathcal{B}'$ or $\mathcal{B} \longrightarrow_p \mathcal{A}$. The transition relation is defined using the semantic rules from Figure 38 to Figure 45.

Definition 6.1.2. The probabilistic transition system of the activity calculus term $\mathcal{A}$ is the tuple $\mathcal{T} = (S,\ s_0, \xrightarrow{\alpha}_p)$ where:

- S is the set of states, ranged over by s, each of which represents an AC term $\mathcal{B}$ corresponding to the unmarked term $\mathcal{A}$,
- $s_0 \in S$, the initial state representing the term $\mathcal{B}_o = \overline{\mathcal{A}}$,
- $\xrightarrow{\alpha}_p \subseteq S \times \Sigma \cup \{o\} \times [0,1] \times S$ is the probabilistic transition relation and it is the least relation satisfying the AC operational semantics rules. We write $s_1 \xrightarrow{\alpha}_p s_2$ in order to specify a probabilistic transition of the form $(s_1, (\alpha, p), s_2)$ for $s_1,\ s_2 \in S$ and (α, p) in $\Sigma \cup \{o\} \times [0,1]$. □

Let e be a marked term and f, f_1, $\cdots$, f_n specify marked (or unmarked) sub-terms. The term f is a sub-term (or a sub-expression) of e, denoted by $e[f]$, if f is a valid activity calculus term occurring once in the definition of e. We also use the notation $e[f\{x\}]$ to denote that f occurs exactly x times in the expression e. For simplification $e[f\{1\}] = e[f]$. We may generalize this notation to more than one sub-term, i.e. $e[f_1,\ f_2,\cdots,\ f_n]$. For instance, given a marked term $\mathcal{B}= \iota \rightarrowtail \overline{l_1{:}a_1} \rightarrowtail l_2{:}a_2 \rightarrowtail l_3{:}\odot$. We write $\mathcal{B}[\overline{l_1{:}a_1}]$ to specify that $\overline{l_1{:}a_1}$ is a sub-term of $\mathcal{B}$. Furthermore, we use the notation $|\mathcal{B}|$ to denotes the unmarked

activity calculus term obtained by removing the marking (all overbars) from the marked term $\mathcal{B}$.

In the sequel, we present the AC operational semantics.

Rules for Initial

The first set of rules in Figure 38 refers to the transitions related to the term $\iota \rightarrowtail \mathcal{N}$. "Tokens in an initial node are offered to outgoing edges" [21]. This is interpreted by our semantics using the axiom INIT-1, which means that if ι is marked, the marking propagates to the rest of the term $\mathcal{N}$ throughout its outgoing edge, with no observable action and a probability $q = 1$. Rule INIT-2 allows the marking to evolve in the rest of the activity term from $\iota \rightarrowtail \mathcal{M}$ to $\iota \rightarrowtail \mathcal{M}'$ with a probability q, by executing the action α if the marking on the sub-term $\mathcal{M}$ can evolve to another marking $\mathcal{M}'$ using the same transition.

INIT-1	$\overline{\iota} \rightarrowtail \mathcal{N} \longrightarrow_1 \iota \rightarrowtail \overline{\mathcal{N}}$
INIT-2	$\dfrac{\mathcal{M} \xrightarrow{\alpha}_q \mathcal{M}'}{\iota \rightarrowtail \mathcal{M} \xrightarrow{\alpha}_q \iota \rightarrowtail \mathcal{M}'}$

Figure 38: Semantic Rules for Initial

Rules for Action Prefixing

The second set of rules in Figure 39 concerns action prefixing. These rules illustrate the possible progress of the tokens in the expression $\overline{l{:}a}^{n} \rightarrowtail \mathcal{M}$. "The completion of the execution of an action may enable the execution of successor node" [21]. Accordingly, the

axiom ACT-1 specifies the progress of a token from $\overline{l{:}a}^{k}$, where action a terminates its execution, to the sub-term $\mathcal{M}$. Note that ACT-1 supports the case of multiple tokens, which is compliant with the specification stating that "start a new execution of the behavior with newly arrived tokens, even if the behavior is already executing from tokens arriving at the invocation earlier" [21]. Rule ACT-2 allows the marking to evolve in the rest of the activity term from $\overline{l{:}a}^{n} \rightarrowtail \mathcal{M}$ to $\overline{l{:}a}^{n} \rightarrowtail \mathcal{M}'$ by executing the action α and with a probability q, if the marked sub-term $\mathcal{M}$ can evolve to $\mathcal{M}'$.

ACT-1	$\overline{l{:}a}^{n} \rightarrowtail \mathcal{M} \xrightarrow{a}_{1} \overline{l{:}a}^{n-1} \rightarrowtail \overline{\mathcal{M}} \quad \forall n > 0$
ACT-2	$\dfrac{\mathcal{M} \xrightarrow{\alpha}_{q} \mathcal{M}'}{\overline{l{:}a}^{n} \rightarrowtail \mathcal{M} \xrightarrow{\alpha}_{q} \overline{l{:}a}^{n} \rightarrowtail \mathcal{M}'}$

Figure 39: Semantic Rules for Action Prefixing

Rules for Final

The rules for activity final are given in Figure 40. "A token reaching an activity final node terminates the activity. In particular, it stops all executing actions in the activity, and destroys all tokens" [21]. Once marked (one token is enough), the activity final node imposes the abrupt termination of all the other normal flows in the activity. Accordingly, the axiom FINAL states that if $\overline{l{:}\odot}$ is a subterm of a marked term $\mathcal{B}$, the latter can do a transition with a probability $q = 1$ and no action, which results in the deletion of all overbars (tokens) from the marked activity term $\mathcal{B}$.

FINAL	$\mathcal{B}[\overline{l{:}\odot}] \longrightarrow_{1} \lfloor \mathcal{B} \rfloor$

Figure 40: Semantic Rules for Final

FORK-1

$$\overline{l{:}Fork(\mathcal{M}_1,\mathcal{M}_2)}^{n} \longrightarrow_1 \overline{l{:}Fork(\overline{\mathcal{M}_1},\overline{\mathcal{M}_2})}^{n-1} \quad \forall n > 0$$

FORK-2

$$\frac{\mathcal{M}_1 \xrightarrow{\alpha}_q \mathcal{M}_1'}{\begin{array}{l}\overline{l{:}Fork(\mathcal{M}_1,\mathcal{M}_2)}^{n} \xrightarrow{\alpha}_q \overline{l{:}Fork(\mathcal{M}_1',\mathcal{M}_2)}^{n} \\ \overline{l{:}Fork(\mathcal{M}_2,\mathcal{M}_1)}^{n} \xrightarrow{\alpha}_q \overline{l{:}Fork(\mathcal{M}_2,\mathcal{M}_1')}^{n}\end{array}}$$

Figure 41: Semantic Rules for Fork

Rules for Fork

The rules for fork are listed in Figure 41. "Tokens arriving at a fork are duplicated across the outgoing edges" [21]. Accordingly, the axiom FORK-1 shows the propagation of the tokens to the sub-terms of the fork in the case where the fork expression is marked. A fork expression is marked means that one or many tokens are offred at the incoming edge of the fork node. Rules FORK-2 illustrates two symmetric rules showing the evolution of the marking within the sub-terms of the fork expression. According to the activity diagram specification, "UML 2.0 activity forks model unrestricted parallelism", which is contrasted with the earlier semantics of UML 1.x, where there is a required synchronization between parallel flows [21]. Thus, the marking evolves asynchronously according to an interleaving semantics on both left and right sub-terms.

Rules for Decision

The next set of rules concerns the non-probabilistic decision shown in Figure 42 and the probabilistic decision provided in Figure 43. With respect to non-probabilistic decision nodes, the specification document states the following: "Each token arriving at a decision node can traverse only one outgoing edge. Guards of the outgoing edges are evaluated

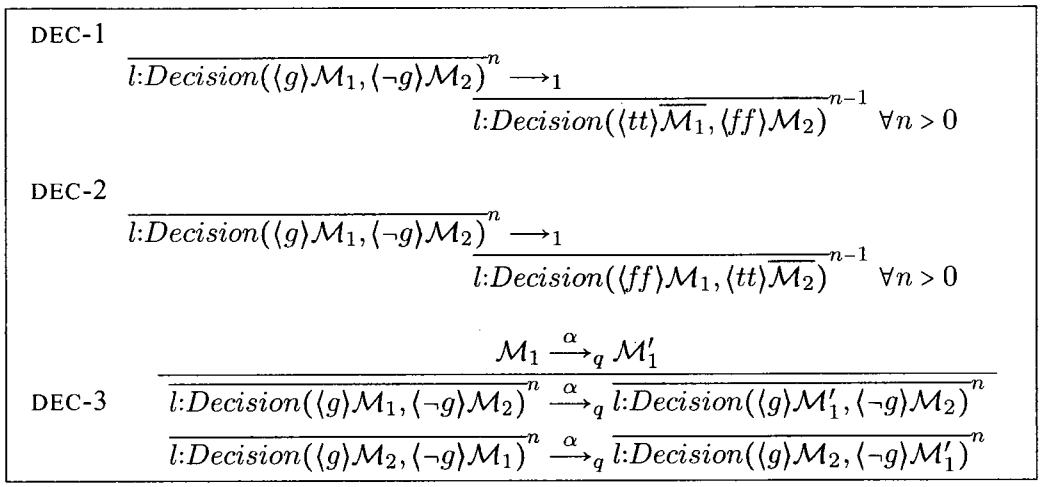


Figure 42: Semantic Rules for Non-Probabilistic Guarded Decision

to determine which edge should be traversed." [21]. Axioms DEC-1 and DEC-2 describe the evolution of tokens reaching a non-probabilistic decision node. For the probabilistic counterpart, the axioms PDEC-1 and PDEC-2 specify the likelihood of a token reaching a probabilistic decision node to traverse one of its branches. The choice is probabilistic; The marking will propagate either to the first branch with a probability p (PDEC-1) or to the second branch with a probability $1-p$ (PDEC-2). This complies with the specification [10].

Rule PDEC-3 (respectively DEC-3) groups two symmetric cases that are related to the marking evolution through the decision sub-terms. If a possible transition $\mathcal{M}_1 \xrightarrow{\alpha}_q \mathcal{M}_1'$ exists and $\mathcal{M}_1$ is a subexpression of $\overline{l{:}Decision_p(\langle g\rangle \mathcal{M}_1, \langle \neg g\rangle \mathcal{M}_2)}^{n}$, then we can deduce the transition $\overline{l{:}Decision_p(\langle g\rangle \mathcal{M}_1, \langle \neg g\rangle \mathcal{M}_2)}^{n} \xrightarrow{\alpha}_q \overline{l{:}Decision_p(\langle g\rangle \mathcal{M}_1', \langle \neg g\rangle \mathcal{M}_2)}^{n}$.

Rules for Merge

Rules for merge are presented in Figure 44. The semantics of merge node according to [21] is defined as follows: "All tokens offered on incoming edges are offered to the outgoing

PDEC-1

$$\overline{l{:}Decision_p(\langle g\rangle \mathcal{M}_1, \langle \neg g\rangle \mathcal{M}_2)}^{n} \longrightarrow_p \overline{l{:}Decision_p(\langle tt\rangle \overline{\mathcal{M}_1}, \langle ff\rangle \mathcal{M}_2)}^{n-1} \quad \forall n > 0$$

PDEC-2

$$\overline{l{:}Decision_p(\langle g\rangle \mathcal{M}_1, \langle \neg g\rangle \mathcal{M}_2)}^{n} \longrightarrow_{1-p} \overline{l{:}Decision_p(\langle ff\rangle \mathcal{M}_1, \langle tt\rangle \overline{\mathcal{M}_2})}^{n-1} \quad \forall n > 0$$

PDEC-3

$$\frac{\mathcal{M}_1 \xrightarrow{\alpha}_q \mathcal{M}_1'}{\begin{array}{l}\overline{l{:}Decision_p(\langle g\rangle \mathcal{M}_1, \langle \neg g\rangle \mathcal{M}_2)}^{n} \xrightarrow{\alpha}_q \overline{l{:}Decision_p(\langle g\rangle \mathcal{M}_1', \langle \neg g\rangle \mathcal{M}_2)}^{n} \\ \overline{l{:}Decision_p(\langle g\rangle \mathcal{M}_2, \langle \neg g\rangle \mathcal{M}_1)}^{n} \xrightarrow{\alpha}_q \overline{l{:}Decision_p(\langle g\rangle \mathcal{M}_2, \langle \neg g\rangle \mathcal{M}_1')}^{n}\end{array}}$$

Figure 43: Semantic Rules for Probabilistic Decision

MERG-1

$$\overline{l{:}Merge(\mathcal{M})}^{n} \longrightarrow_1 \overline{l{:}Merge(\overline{\mathcal{M}})}^{n-1} \quad \forall n \geq 1$$

MERG-2

$$\frac{\mathcal{M} \xrightarrow{\alpha}_q \mathcal{M}'}{\overline{l{:}Merge(\mathcal{M})}^{n} \xrightarrow{\alpha}_q \overline{l{:}Merge(\mathcal{M}')}^{n}}$$

Figure 44: Semantic Rules for Merge

edge. There is no synchronization of flows or joining of tokens." Thus, the axiom MERG-1 states that the marking on top of the merge evolves with a probability 1 and no action to its sub-term $\mathcal{M}$. Rule MERG-2 allows the marking to evolve in $\overline{l{:}Merge(\mathcal{M})}^{n}$ if there is a possible transition such that $\mathcal{M} \xrightarrow{\alpha}_q \mathcal{M}'$.

Rules for Join

Rules for join are presented in Figure 45. "If there is a token offered on all incoming edges, then one control token is offered on the outgoing edge" [21]. Axioms JOIN-1 and JOIN-2 describe the propagation of a token on the top of the join definition expression, namely $\overline{l{:}x.Join(\mathcal{M})}^{n}$ and the referencing labels. Unlike the merge node, the join traversal requires

JOIN-1

$$\mathcal{B}[\overline{l{:}x.Join(\mathcal{M})}^{n}, \bar{l}^{k_x}\{x-1\}] \longrightarrow_1 \mathcal{B}[l{:}x.Join(\overline{\mathcal{M}}), l\{x-1\}] \; x > 1; \;\; n, k_x \geq 1$$

JOIN-2

$$\overline{l{:}1.Join(\mathcal{M})}^{n} \longrightarrow_1 \overline{l{:}1.Join(\overline{\mathcal{M}})}^{n-1} \quad n \geq 1$$

JOIN-3

$$\frac{\mathcal{M} \xrightarrow{\alpha}_q \mathcal{M}'}{\overline{l{:}x.Join(\mathcal{M})}^{n} \xrightarrow{\alpha}_q \overline{l{:}x.Join(\mathcal{M}')}^{n}}$$

Figure 45: Semantic Rules for Join

all references to itself to be marked, which is described using the "join specification" requirement in [21]. More precisely, all the sub-terms l corresponding to a given join node in the AC term, including the definition of the join itself, have to be marked so that the token can progress to the rest of the expression. The number of occurrences of the sub-term $\bar{l}$ in the whole marked term is known and it corresponds to the value of x–1. If so, only one control token propagates to the subsequent subterm $\mathcal{M}$ with a probability $q = 1$. Moreover, [21] states that: "Multiple control tokens offered on the same incoming edge are combined into one before" the traversal, which is specified in axiom JOIN-1. Axiom JOIN-2 corresponds to the special case where $x = 1$. According to [21], there is no restriction on the use of a join node with a single incoming edge even though this is qualified therein as not useful. Rule JOIN-3 shows the possible evolution of the marking in $\overline{l{:}x.Join(\mathcal{M})}^{n}$ to $\overline{l{:}x.Join(\mathcal{M}')}^{n}$, if the marking in $\mathcal{M}$ evolves to $\mathcal{M}'$ with the same transition.

6.2 Case Study

In the sequel, we present a SysML activity diagram case study depicting a hypothetical design of the behavior corresponding to banking operations on an ATM system illustrated

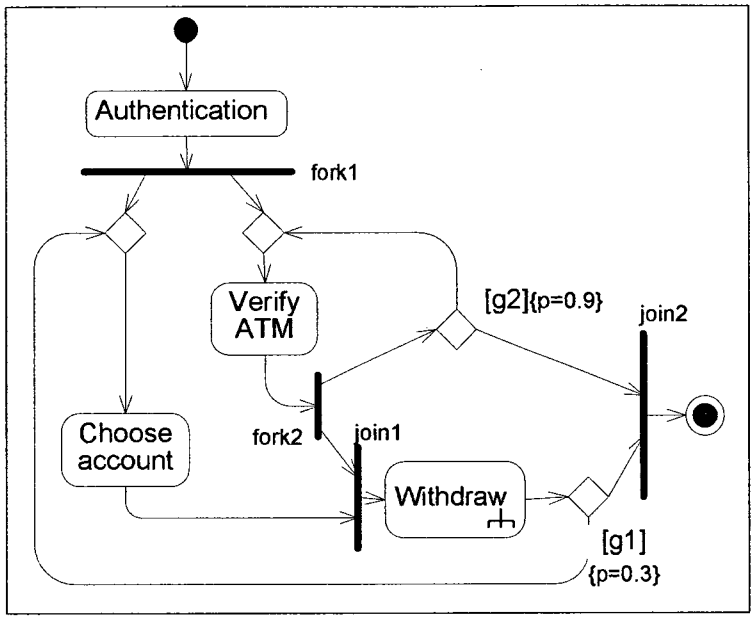


Figure 46: Case Study: Activity Diagram of a Banking Operation

in Figure 46. We first show how we can express the activity diagram using the AC language and then demonstrate the benefit and usefulness of the proposed formal semantics.

The actions in an activity diagram can be refined using structured activity nodes in order to expand their internal behavior. For instance, the node labeled `Withdraw` in Figure 46 is actually a structured node that calls the activity diagram pictured in Figure 37. Using the operational semantics defined earlier, a compositional assessment of the design can be performed. For instance, the detailed activities are abstracted away at a first step and the global behavior is validated. Then, the assessment of the refined behavior can be performed. The compositionality and abstraction features allow handling real-world systems without compromising the validation process. For instance, we consider the activity diagram of Figure 46 and assume that `Withdraw` action is an atomic action denoted by the abbreviation d. Moreover, considering the actions a, b, and c as the abbreviations of the

actions `Authentication`, `Verify ATM`, and `Choose account` respectively, the corresponding unmarked term $\mathcal{A}_1$, is as follows:

$$\mathcal{A}_1 = \iota \rightarrowtail l_1{:}a \rightarrowtail l_2{:}\texttt{Fork}(\mathcal{N}_1, l_{12})$$

$$\mathcal{N}_1 = l_3{:}\texttt{Merge}(l_4{:}b \rightarrowtail l_5{:}\texttt{Fork}(\mathcal{N}_2, \mathcal{N}_3))$$

$$\mathcal{N}_2 = l_6{:}\texttt{Decision}_{0.9}(\langle g2\rangle\, l_3, \langle \neg g2\rangle\, l_7{:}2.\texttt{Join}(l_8{:}\odot))$$

$$\mathcal{N}_3 = l_9{:}2.\texttt{Join}(l_{10}{:}d \rightarrowtail \mathcal{N}_4)$$

$$\mathcal{N}_4 = l_{11}{:}\texttt{Decision}_{0.3}(\langle g1\rangle\, \mathcal{N}_5, \langle \neg g1\rangle\, l_7)$$

$$\mathcal{N}_5 = l_{12}{:}\texttt{Merge}(l_{13}{:}c \rightarrowtail l_9)$$

The abbreviations are used to simplify the presentation of the AC term . The guard $g1$ denotes the possibility of triggering a new operation if evaluated to $true$ and guard $g2$ denotes the result of evaluating the status of the connection. Applying the operational rules on the marked $\overline{\mathcal{A}_1}$, we can derive a run that leads to a deadlock, which means that we reached a configuration where the expression is marked but no progress can be made (no operational rule can be applied). This derivation may reveal a design error in the activity diagram, which is not obvious using only inspection. Even though one may suspect the `join2` to cause the deadlock due to the presence of a prior decision node, the deadlock actually occurs due to the other join node (i.e. node `join1`).

More precisely, the run consists in executing the action c twice (because the guard $g1$ is true) and the action b once ($g2$ evaluated to false). The deadlocked configuration reached by the derivation run has the following marked sub-terms:

$$\mathcal{M}_2 = l_6{:}\texttt{Decision}_{0.9}(\langle g2\rangle\, l_3, \langle \neg g2\rangle\, \overline{l_7{:}\texttt{Join}(l_8{:}\odot)})$$

$$\mathcal{M}_5 = l_{12}{:}\texttt{Merge}(l_{13}{:}c \rightarrowtail \overline{l_9})$$

A possible derivation run leading to this deadlocked configuration is presented in Figure

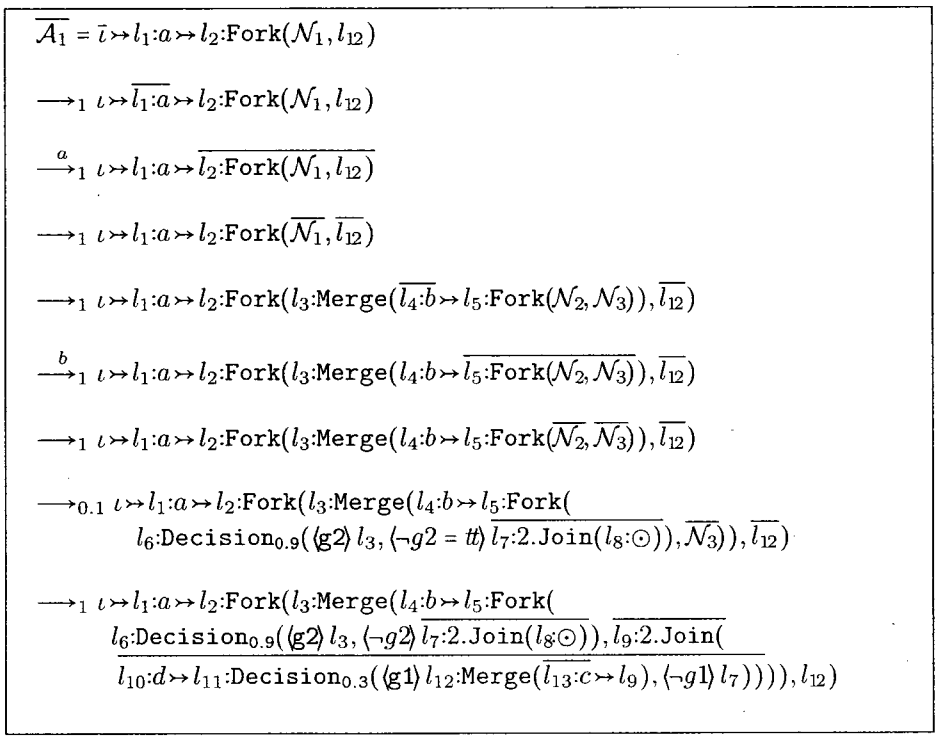


Figure 47: Derivation Run Leading to a Deadlock - Part 1

47 and Figure 48. This has been obtained by applying the AC operational semantics rules on the term $\overline{\mathcal{A}_1}$, which corresponds to the initial state of the probabilistic transition system. This run represents a single path in the probabilistic transition system corresponding to the semantic model of the activity diagram of Figure 46. Informally, the deadlock occurs because both join nodes `join1` and `join2` are waiting for a token that will never be delivered on one of their incoming edges. There is no possible token progress from the deadlocked configuration since no rule can be applied.

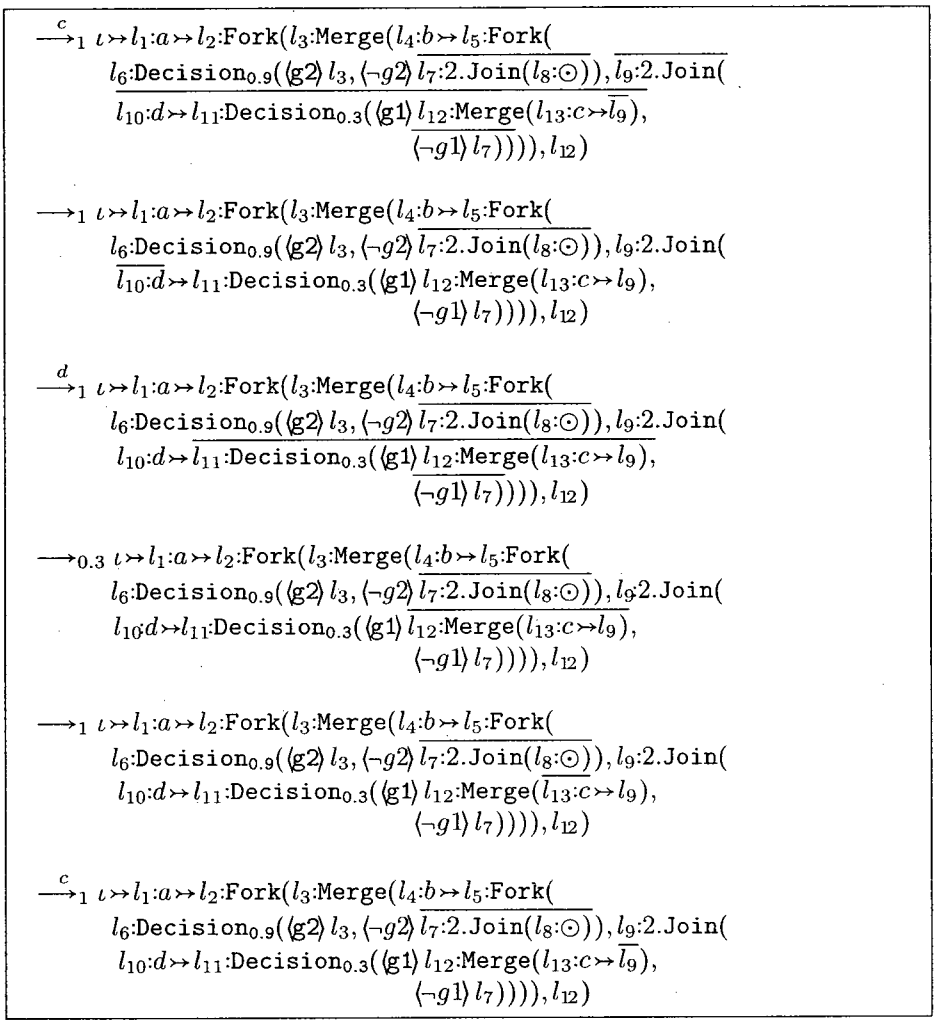


Figure 48: Derivation Run Leading to a Deadlock - Part 2

6.3 Markov Decision Process

The MDP underlying the PTS corresponding to the semantic model of a given SysML activity diagram can be described using to the following definition.

Definition 6.3.1. The Markov Decision Process $\mathcal{M}_{\mathcal{T}}$ underlying the Probabilistic Transition System $\mathcal{T}$=(S, s_0,$\xrightarrow{\alpha}_p$) is the tuple $\mathcal{M}_{\mathcal{T}}$=($S$, s_0, Act, $Steps$) such that:

- Act=$\Sigma \cup \{o\}$,

- $Steps: S \rightarrow 2^{Act \times Dist(S)}$ is the probabilistic transition function defined over S such that, for each $s \in S$, $Steps(s)$ is defined as follows:
  - For each set of transitions $\Gamma_\alpha = \{s \xrightarrow{\alpha}_{p_j} s_j,\ j \in J,\ p_j < 1, \text{ and } \sum_j p_j = 1\}$, $(\alpha, \mu_\Gamma) \in Steps(s)$ such that $\mu_\Gamma(s_j) = p_j$ and $\mu_\Gamma(s') = 0$ for $s' \in S \setminus \{s_j\}_{j \in J}$.
  - For each transition $\tau = s \xrightarrow{\alpha}_1 s'$, $(\alpha, \mu_\tau) \in Steps(s)$ such that $\mu_\tau(s') = 1$ and $\mu_\tau(s) = 0$ for $s \neq s'$. □

6.4 Conclusion

In this chapter, we defined a probabilistic calculus that we called Activity Calculus (AC). The latter allows expressing algebraically SysML activity diagrams and providing its formal semantic foundations using operational semantics framework. Our calculus serves proving the soundness of the translation algorithm that we presented in the previous chapter, but also, opens up new directions to explore other properties and applications using the formal semantics of SysML activity diagrams. The following chapter defines a formal syntax and semantics for PRISM specification language and examines the soundness of the proposed translation algorithm.

Chapter 7

Soundness of the Translation Algorithm

In this chapter, our main objective is to closely examine the correctness of the translation procedure proposed earlier that maps SysML activity diagrams into the input language of the probabilistic model-checker PRISM. In order to provide a systematic proof, we rely on formal methods, which enable us with solid mathematical basis. To do so, four main ingredients are needed. First, we need to express formally the translation algorithm. This enables its manipulation forward deriving the corresponding proofs. Second, the formal syntax and semantics for SysML activity diagrams need to be defined. This has been proposed in the previous chapter by the means of the activity calculus language. Third, the formal syntax and semantics of PRISM input language have to be defined. Finally, a suitable relation is needed in order to compare the semantics of the diagram with the semantics of the resulting PRISM model.

We start by exposing the notation that we use in Section 7.1. Then, in Section 7.2 we

explain the followed methodology for establishing the correctness proof. After that, we describe in Section 7.3 the formal syntax and semantics definitions of PRISM input language. Section 7.4 is dedicated for formalizing the translation algorithm using a functional core language. Section 7.6 defines a simulation relation over Markov decision processes, which can be used in order to compare the semantics of both SysML activity diagrams and their corresponding PRISM models. Finally, Section 7.7 presents the soundness theorem, which formally defines the soundness property of the translation algorithm. Therein, we provide the details of the related proof.

7.1 Notation

In the following, we present the notation that we are going to use in this chapter. A multiset is denoted by (A, m), where A is the underlying set of elements and $m: A \longrightarrow \mathbb{N}$ is the multiplicity function that associates a positive natural number in $\mathbb{N}$ with each element of A. For each element $a \in \mathcal{A}$, $m(a)$ is the number of occurrences of a. The notation $\{|\}$ is used to designate the empty multiset, and $\{|\ (a \hookrightarrow n)\ |\}$ denotes the multiset containing the element a occurring $m(a) = n$ times. The operator $\uplus$ denotes the union of two multisets, such that if (A_1, m_1) and (A_2, m_2) are two multisets, the union of these two multisets is a multiset $(A, m) = (A_1, m_1) \uplus (A_2, m_2)$ such that $A = A_1 \cup A_2$ and $\forall a \in \mathcal{A}$, we have $m(a) = m_1(a) + m_2(a)$.

A discrete probability distribution over a countable set S is a function $\mu: S \rightarrow [0, 1]$ such that $\sum_{s \in S} \mu(s) = 1$ where $\mu(s)$ denotes the probability for s under the distribution μ.

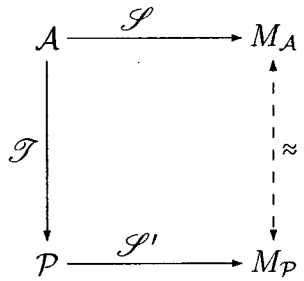


Figure 49: Approach to Prove the Correctness of the Translation

The support of the distribution μ is the set $Supp(\mu) = \{s \in S : \mu(s) > 0\}$. We write μ_s^1 for $s \in S$ to designate a distribution that assigns a probability 1 to s and 0 to any other element in S. Also, sub-distributions μ are considered where $\sum_{s \in S} \mu(s) < 1$ and μ_s^λ denotes a probability distribution that assigns λ probability to s and 0 to any other element in S. The set of probability distributions over S is denoted by $Dist(S)$.

7.2 Methodology

Let $\mathcal{A}$ be the unmarked AC term corresponding to a given SysML activity diagram. Let $\mathcal{P}$ be the corresponding PRISM model description written in the PRISM input language. We denote by $\mathscr{T}$ the translation algorithm that maps $\mathcal{A}$ into $\mathcal{P}$, i.e. $\mathscr{T}(\mathcal{A}) = \mathcal{P}$. If we denote by $\mathscr{S}$ the semantic function that associates for each SysML activity diagram its formal meaning, $\mathscr{S}(\mathcal{A})$ denotes the corresponding semantic model. According to our previous results, the semantics of the activity diagram can be expressed as an MDP as defined in Definition 6.3.1. Let denote it by $\mathscr{S}(\mathcal{A}) = M_{\mathcal{A}}$. Similarly, let $\mathscr{S}'$ be the semantic function that associates with a PRISM model description its formal semantics. Since we are dealing with MDP models, $\mathscr{S}'(\mathcal{P}) = M_{\mathcal{P}}$ represents the MDP semantics of $\mathcal{P}$.

Our main objective is to prove the correctness of the translation algorithm with respect to the SysML activity diagram semantics. This can be reduced to prove the commutativity of the diagram presented in Figure 49. To this end, we aim at defining a relation that we can use to compare $M_{\mathcal{P}}$ with $M_{\mathcal{A}}$. Let $\approx$ denote this relation, we aim at proving that there exists such a relation so that $M_{\mathcal{P}} \approx M_{\mathcal{A}}$.

7.3 Formalization of the PRISM Input Language

We describe in this section the formal syntax and the semantics of the PRISM input language. By doing so, we greatly simplify the manipulation of the output of our translation algorithm for the sake of proofs. Moreover, defining a formal semantics for the PRISM language itself leads to more precise soundness concepts and more rigorous proofs. While reviewing the literature, there were no initiatives in this direction. The informal description of the syntax and semantics of PRISM language is provided in Chapter 2 Section 2.5.

7.3.1 Syntax

The formal syntax of PRISM input language is presented in a BNF style in Figure 50 and Figure 51. A PRISM model, namely *prism_model*, starts with the specification of the model type *model_type* (i.e. MDP, CTMC, or DTMC). A model consists of two main parts:

- The declaration of the constants, the formulas, and the global variables corresponding to the model,
- The specification of the modules composing the model each consisting of a set of

prism_model	::=	*model_type*	
		global_declaration	(Global Declarations)
		modules	(Modules Specification)
modules	::=	**module** *module_name*	
		localvar_dec	(Local Variables Declarations)
		c	(Commands)
		endmodule	
	|	*modules* || *modules*	(Modules Composition)
global_declaration	::=	*const_dec*	(Constants Declarations)
		formula_dec	(Formulas Declarations)
		globalvar_dec	(Global Variables Declarations)
model_type	::=	**mdp**	
	|	**ctmc**	
	|	**dtmc**	

Figure 50: Syntax of the PRISM Input Language - Part 1

local variables declaration followed by a set of commands.

We focus on the commands since they describe the intrinsic behavior of the model. The formal descriptions of constants, formulas, and local and global variables declarations are not provided in details since they are supposed to be pre-determined and generated before the actual definition of the commands. In addition, we assume that each variable declaration contains an initial value. We denote by x_0 the initial value of the variable x.

A command c is of the form $[\alpha]\ w \rightharpoonup u$ where α represents the action label of the command, w is the corresponding (boolean) guard, and u is its update representing the effect of the command on the values of the variables. An update is build as the probabilistic choice over unit updates d_i, denoted by $\sum_{i \in I} \lambda_i : d_i$ such that $\sum_{i \in I} \lambda_i = 1$. A given unit update d is the conjunction of assignments of the form $x' = e$, where x' represents the new value of the variable x and e is an expression over the variables, constants, and/or formulas

$\mathcal{C} \ni c$	::=	$[\alpha]\, w \rightharpoonup u$	(Command)
	|	$c \cup c$	
$\mathcal{W} \ni w$	::=	e	(Guard)
$\mathcal{U} \ni u$	::=	$\lambda : d$	(Update)
	|	$u_1 + u_2$	
$\mathcal{D} \ni d$	::=	$skip$	(Update unit)
	|	$x' = e$	
	|	$d_1 \wedge d_2$	
$\mathcal{E} \ni e$	::=	x	(Expression)
	|	v	
	|	$e_1\ op\ e_2$	
	|	$\neg\, e_1$	
$\mathcal{V} \ni v$	::=	i	(Value)
	|	d	
	|	b	
op	$\in$	$\{*, /, +, -, <, \leq, >, \geq, =, \neq, \wedge, \vee\}$	(Operators)
λ	$\in$	$[0, 1]$	(Probability Value)
x	$\in$	*variables*	(Variable)
α	$\in$	*actions*	(Action)
i	$\in$	*integer*	(Integer)
d	$\in$	*double*	(Double)
b	$\in$	*boolean*	(Boolean)

Figure 51: Syntax of the PRISM Input Language - Part 2

of the model. Thus, we require type-consistency of the variable x and the expression e. A trivial update unit $skip$ stands for the update that does not affect the values of the variables. Finally, a guard w is build using a logical expression over the variables and formulas of the model.

7.3.2 Operational Semantics

In this section, we focus on the semantics of a program written in PRISM input language limiting ourselves to the fragment that has an MDP semantics. We define the operational semantics of the PRISM language following the style of SOS [202]. We consider PRISM models consisting of a single system module, since any PRISM model described as a composition of a set of modules can be reduced to a single module system according to a set of construction rules described in [81]. In the case of a single module, actions labeling the commands are not as much useful as in the case of multiple modules.

A configuration represents the state of the system at a certain moment during its evolution. It is build as a pair $\langle \mathscr{C}, s \rangle$ where $\mathscr{C}$ denotes the set of commands to be executed and s the associated store, which models the memory used in order to keep track of the current values associated with the variables of the system. Let $\mathscr{V}$ be the set of values and $\mathcal{S}$ be the set of stores ranged over by s, s_1, s_2, etc.We write $s[x \mapsto v_x]$ to denote the store s that assigns to the variable x the value v_x and the value $s(y)$ to the variable $y \neq x$. We denote by $[\![\_]\!](\_)$ the semantic function used to evaluate expressions or guards defined in Figure 51. Let E be the set of expressions and W the set of guards. We have $[\![\_]\!](\_): E \cup W \to \mathcal{S} \to \mathscr{V}$ a function that takes as argument an expression e (or a guard) and a store s and returns the value of the expression e where each variable x is interpreted by $s(x)$.

We define an auxiliary function $f(\_)(\_): \mathit{Dist}(\mathcal{S}) \to \mathit{Dist}(\mathcal{S}) \to \mathit{Dist}(\mathcal{S})$ such that

(SKIP)	$\langle skip, s\rangle \rightarrow s$
(UPD-EVAL)	$\langle x' = e, s\rangle \rightarrow s[x \mapsto [\![e]\!](s)]$
(UPD-PROCESSING)	$\dfrac{\langle d_1, s\rangle \rightarrow s_1}{\langle d_1 \wedge d_2, s\rangle \rightarrow \langle d_2, s_1\rangle}$
(PROB-UPD)	$\dfrac{\langle d, s\rangle \rightarrow s_1}{\langle \lambda : d, s\rangle \rightarrow \mu^{\lambda}_{s_1}}$
(PROBCHOICE-UPD)	$\dfrac{\langle u_1, s\rangle \rightarrow \mu_1 \qquad \langle u_2, s\rangle \rightarrow \mu_2}{\langle u_1 + u_2, s\rangle \rightarrow f(\mu_1)(\mu_2)}$
(ENABLED-CMD)	$\dfrac{[\![w]\!](s) = true}{\langle [\alpha]\ w \rightarrow u, s\rangle \xrightarrow{\alpha} \langle u, s\rangle}$
(CMD-PROCESSING)	$\dfrac{\langle c, s\rangle \xrightarrow{\alpha} \langle u, s\rangle \qquad \langle u, s\rangle \rightarrow \mu}{\langle \{c\} \cup C, s\rangle \xrightarrow{\alpha} \langle \{c\} \cup C, \mu\rangle}$

Figure 52: Semantic Inference Rules for PRISM's Input Language

$\forall \mu_1, \mu_2 \in Dist(\mathcal{S})$,

$$f(\mu_1)(\mu_2) = \mu = \begin{cases} \sum_{\substack{i \in I \\ s_i = s}} \mu_1(s_i) + \sum_{\substack{j \in J \\ s_j = s}} \mu_2(s_j) & \\ 0 & \text{otherwise.} \end{cases}$$

The inference rules corresponding to the operational semantics of PRISM model are listed in Figure 52. Basically, rule (SKIP) denotes that the trivial unit update $skip$ does not affect the store (i.e. the values of the variables). Rule (UPD-EVAL) expresses the effect on the store s of a new value assignment to the variable x using the evaluation of the expression e. Rule (UPD-PROCESSING) is used to process a conjunction of updates $\langle d_1 \wedge$

$d_2, s\rangle$ given the evaluation of $\langle d_1, s\rangle$. The unit update d_2 is applied on the store s_1 resulting from applying the unit update d_1. This rule allows a recursive application of unit updates until processing all the components of an update of the form $\lambda : d$, which results in a new state of the system, i.e. a new store reflecting the new variables values of the system. Rule (PROB-UPD) denotes the processing of a probabilistic update on the system. The result of processing $\langle \lambda : d, s\rangle$ is a probability sub-distribution that associates a probability λ to a store s_1 obtained by applying the update. If $\lambda < 1$, the definition of the probability distribution is partial since it is a part of a probabilistic choice over the related command's updates.

Rule (PROBCHOICE-UPD) processes the probabilistic choice between different updates as a probability distribution over the set of resulting stores. It uses the function f in order to build the resulting probability distribution from partial probability distribution definitions taking into account the possibility that different updates may lead to the same store. Rule (ENABLED-CMD) is used to evaluate the guard of a given command and thus enabling its execution if its corresponding guard is true. Finally, rule (CMD-PROCESSING) states that if a command is non-deterministically selected from the set of available enabled commands (since their guards are true) for a given store s (first premise) and if the set of corresponding updates leads to the probability distribution μ (second premise), then a transition can be fired from the configuration $\langle \{c\} \cup C, s\rangle$ resulting in a set of new possible configurations where all reachable states are defined using the probability distribution μ.

An operational semantics of a PRISM MDP program $\mathcal{P}$ is provided by means of the MPD $M_{\mathcal{P}}$ where the states are of the form $\langle \mathscr{C}, s\rangle$, the initial state is $\langle \mathscr{C}, s_0\rangle$ such that s_0 is the store where each variable is assigned its default value, the set of actions are the

action labeling the commands, and the probabilistic transition relation is obtained by the rule (CMD-PROCESSING) in Figure 52 such that $Steps(s) = (\alpha, \mu)$. We omit the set of commands $\mathscr{C}$ from the configuration and write simply $s \xrightarrow{\alpha} \mu$ to denote $Steps(s)$.

7.4 Formal Translation

We focus hereafter on the formal translation of SysML activity diagrams into their corresponding MDP using the input language of PRISM. We presented in Chapter 5 the translation algorithm written in an imperative language representing an abstraction of the implementation code. In order to simplify the analysis of the translation algorithm, we need to express it in some functional core language. The latter allows making assertions about programs and prove their correctness easier than having them written in an imperative language. Thus, we use the ML functional language [203]. The input to the translation algorithm corresponds to the AC term expressing formally the structure of the SysML activity diagram. The output of the translation algorithm represents the PRISM MDP model. The latter contains two parts: variables declarations and the set of PRISM commands enclosed in the main module. We suppose that the declarations of variables, constants, and formulas are performed separately of the actual translation using our algorithm.

Before detailing the translation algorithm, we first clarify our choices for the constants, the formulas, and the variables in the model and their correspondences with the elements of the diagram. First, we need to define a constant of type integer, namely *max_inst*, that specifies the maximum number of supported executions instances (i.e. the maximum number of

control tokens). Each node (action or control) that might receive tokens is associated with a variable of type integer, which values range over the interval $[0, max\_inst]$. Exceptionally, the flow final nodes are the only nodes that are not represented in the PRISM model since they only absorb the tokens reaching them. The value assigned to each variable at a point in the time denotes the number of active instances of the corresponding activity node. An activity node that is not active will have its corresponding variable assigned the value 0. An activity node that reached the maximum supported number of active instances will have its corresponding variable assigned the maximum value *max_inst*. However, there are two exceptions to this rule: the first corresponds to the initial and the final nodes and the second to the join nodes.

Firstly, each of the initial and final nodes is associated with an integer variable that takes two possible values 0 or 1. This is because these nodes are supposed to have a boolean state (active or inactive). Secondly, the join node represents also an exceptional case because of the specific processing of the join condition. The latter states that each of the incoming edges has to receive at least one control token in order to produce a single token that traverses the join node. Thus, we assign an integer variable for each incoming edge of a join node. Their values range over the interval $[0, max\_inst]$. Then, we also assign a boolean formula to the join node in order to express the join condition. This formula is a conjunction of boolean conditions stating that each variable associated with an incoming edge has a value greater or equal to 1. Finally, we also consider the guards of all decision nodes. These are helpful in describing properties to be verified on the model. Thus, we assign a boolean PRISM variable for each boolean guard of a decision node.

We use the labels associated with the activity nodes as defined in the activity calculus term as the identifiers of their corresponding PRISM variables. For the exceptional cases (meaning the initial, final, and join nodes), we use other adequate notation. As for the initial and the final nodes, we use respectively the variables identifiers l_ι and l_f. Concerning the join nodes denoted in the AC term either as the subterm $l{:}x.join(\mathcal{N})$ for the definition of the join or as l for referencing each of its incoming edges, we need to assign x distinct variables. Thus, we use the label l concatenated with an integer number k such that $k \in [1, x]$, which results in a variable $l[k]$ associated with each incoming edge. By convention, we use $l[1]$ for specifying the variable related to $l{:}x.join(\mathcal{N})$. We denote by $\mathcal{L}_\mathcal{A}$ the set of labels associated with the AC term $\mathcal{A}$ representing the identifiers of the variables in the corresponding MDP model.

A generated PRISM command c is expressed formally using the syntax definition presented in Figure 51. The main mapping function denoted by $\mathscr{T}$ is described in Listing 7.1. It makes use of the function $\mathscr{E}$ described in Listing 7.2. It also employs an utility function $\mathscr{L}$ in order to identify the label of an element of the AC term. The signatures of the two main functions are provided in their respective listings. We denote by $\mathcal{AC}$ the set of unmarked AC terms and $\overline{\mathcal{AC}}$ the set of marked AC terms. Let $\mathcal{L}$ be the universal set of labels ranged over by l. Moreover, let C be the set of commands ranged over by c, W be the set of guard expressions ranged over by w, Act be the set of actions ranged over by α, and D be the set of updates units ranged over by d.

Listing 7.1: Formal SysML Activity Diagram Translation Algorithm

$\mathcal{T}: \mathcal{AC} \rightarrow \mathcal{C}$
$\mathcal{T}(\mathcal{N})$ = Case $(\mathcal{N})$ **of**
$\iota \mapsto \mathcal{N}'$ ⇒ **let**
$c = \mathcal{E}(\mathcal{N}')(l_\iota = 1)(l'_\iota = 0)(1.0)(l_\iota)$
in
$\{c\} \cup \mathcal{T}(\mathcal{N}')$
end
$l{:}a \mapsto \mathcal{N}'$ ⇒ **let**
$c = \mathcal{E}(\mathcal{N}')(l > 0)(l' = l - 1)(1.0)(l)$
in
$\{c\} \cup \mathcal{T}(\mathcal{N}')$
end
$l{:}Merge(\mathcal{N}')$ ⇒ **let**
$c = \mathcal{E}(\mathcal{N}')(l > 0)(l' = l - 1)(1.0)(l)$
in
$\{c\} \cup \mathcal{T}(\mathcal{N}')$
end
$l{:}x.Join(\mathcal{N}')$ ⇒ **let**
$c = \mathcal{E}(\mathcal{N}')(\wedge_{1 \le k \le x}(l[k] > 0))(\wedge_{1 \le k \le x}(l[k]' = 0))(1.0)(l[1])$
in
$\{c\} \cup \mathcal{T}(\mathcal{N}')$
end
$l{:}Fork(\mathcal{N}_1, \mathcal{N}_2)$ ⇒ **let**
$[\alpha]\, w \rightarrow \lambda : d = \mathcal{E}(\mathcal{N}_1)(l > 0)(l' = l - 1)(1.0)(l)$
in
let
$c = \mathcal{E}(\mathcal{N}_2)(w)(d)(\lambda)(\alpha)$
in
$\{c\} \cup \mathcal{T}(\mathcal{N}_1) \cup \mathcal{T}(\mathcal{N}_2)$
end
end
$l{:}Decision_p(\langle g \rangle \mathcal{N}_1, \langle \neg g \rangle \mathcal{N}_2)$ ⇒ **let**
$[\alpha_1]\, w_1 \rightarrow \lambda_1 : d_1 = \mathcal{E}(\mathcal{N}_1)(l > 0)((l' = l - 1) \wedge (g' = true))(p)(l)$
$[\alpha_2]\, w_2 \rightarrow \lambda_2 : d_2 = \mathcal{E}(\mathcal{N}_2)(true)((l' = l - 1) \wedge (g' = false))(1 - p)(l)$
in
$([l]\, w_1 \wedge w_2 \rightarrow \lambda_1 : d_1 + \lambda_2 : d_2) \cup \mathcal{T}(\mathcal{N}_1) \cup \mathcal{T}(\mathcal{N}_2)$
end
$l{:}Decision(\langle g \rangle \mathcal{N}_1, \langle \neg g \rangle \mathcal{N}_2)$ ⇒ **let**
$[\alpha_1]\, w_1 \rightarrow \lambda_1 : d_1 = \mathcal{E}(\mathcal{N}_1)(l > 0)((l' = l - 1) \wedge (g' = true))(1.0)(l)$
$[\alpha_2]\, w_2 \rightarrow \lambda_2 : d_2 = \mathcal{E}(\mathcal{N}_2)(l > 0)((l' = l - 1) \wedge (g' = false))(1.0)(l)$
in
$([l]\, w_1 \rightarrow \lambda_1 : d_1) \cup \mathcal{T}(\mathcal{N}_1) \cup ([l]\, w_2 \rightarrow \lambda_2 : d_2) \cup \mathcal{T}(\mathcal{N}_2)$
end
$l_f{:}\odot$ ⇒ **let**
$w = (l_f = 1)$
$d = \wedge_{l \in \mathcal{L}_\mathcal{A}}(l' = 0)$
$\lambda = 1.0$
in
$([l_f]\, w \rightarrow \lambda : d)$
otherwise ⇒ skip

Listing 7.2: Definition of the $\mathscr{E}$ Function

$\mathscr{E}: \mathcal{AC} \to W \to D \to [0,1] \to Act \to \mathcal{C}$
$\mathscr{E}(\mathcal{N})(w)(d)(\lambda)(\alpha)$ = Case($\mathcal{N}$) **of**
$l{:}\otimes \Rightarrow ([\alpha]\, w \wedge (l_f = 0) \rightharpoonup \lambda : d)$
$l_f{:}\odot \Rightarrow ([\alpha]\, w \wedge (l_f = 0) \rightharpoonup \lambda : (l'_f = 1) \wedge d)$
$l{:}x.Join(\mathcal{N}') \Rightarrow$ **let**
$w_1 = w \wedge (l_f = 0) \wedge \neg(\bigwedge_{1 \le k \le x}(l[k] > 0)) \wedge (l[1] < max\_inst)$
$d_1 = (l[1]' = l[1] + 1) \wedge d$
in
$([\alpha]\, w_1 \rightharpoonup \lambda : d_1)$
end
$l \Rightarrow$ **if** $l = \mathscr{L}(l{:}x.Join(\mathcal{N}'))$ **then**
let
$w_1 = w \wedge (l_f = 0) \wedge \neg(\bigwedge_{1 \le k \le x}(l[k] > 0)) \wedge (l[j] < max\_inst)$
$d_1 = (l[j]' = l[j] + 1) \wedge d$
in
$([\alpha]\, w_1 \rightharpoonup \lambda : d_1)$
end
end
if $l = \mathscr{L}(l{:}Merge(\mathcal{N}'))$ **then**
let
$w_1 = w \wedge (l_f = 0) \wedge (l < max\_inst)$
$d_1 = (l' = l + 1) \wedge d$
in
$([\alpha]\, w_1 \rightharpoonup \lambda : d_1)$
end
end
otherwise $\Rightarrow$ **let**
$w_1 = w \wedge (l_f = 0) \wedge (l < max\_inst)$
$d_1 = (l' = l + 1) \wedge d$
in
$([\alpha]\, w_1 \rightharpoonup \lambda : d_1)$
end

7.5 Case Study

In the sequel, we present a SysML activity diagram case study depicting a hypothetical design of the behavior corresponding to banking operations on an ATM system illustrated in Figure 46. It is designed intentionally with flaws in order to demonstrate the viability of our approach. The activity starts by `Authentication` then a fork node indicates the initiation of concurrent behavior. Thus, `Verify ATM` and `Choose account` are triggered

together. This activity diagram presents mixed disposition of fork and join nodes. Thus, `Withdraw` money action cannot start until both `Verify ATM` and `Choose account` terminate. The guard $g1$, if evaluated to $true$, denotes the possibility of triggering a new operation. The probability on the latter decision node models the probabilistic user behavior. The guard $g2$ denotes the result of evaluating the status of the connection presenting functional uncertainty. We first show how we can express the activity diagram using the AC language and then explain its mapping into PRISM code.

In order to simplify the presentation of the corresponding AC term, namely $\mathcal{A}_1$, we use abbreviated notation for action designations. Thus, we assume the `Withdraw` action to be atomic denoted by the abbreviation d. Moreover, we consider the actions a, b, and c as the abbreviations of the actions `Authentication`, `Verify ATM`, and `Choose account` respectively. The corresponding unmarked term $\mathcal{A}_1$, is as follows:

$$\mathcal{A}_1 = \iota \rightarrowtail l_1{:}a \rightarrowtail l_2{:}\texttt{Fork}(\mathcal{N}_1, l_{12})$$

$$\mathcal{N}_1 = l_3{:}\texttt{Merge}(l_4{:}b \rightarrowtail l_5{:}\texttt{Fork}(\mathcal{N}_2, \mathcal{N}_3))$$

$$\mathcal{N}_2 = l_6{:}\texttt{Decision}_{0.9}(\langle \texttt{g2} \rangle\, l_3, \langle \neg g2 \rangle\, l_7{:}2.\texttt{Join}(l_8{:}\odot))$$

$$\mathcal{N}_3 = l_9{:}2.\texttt{Join}(l_{10}{:}d \rightarrowtail \mathcal{N}_4)$$

$$\mathcal{N}_4 = l_{11}{:}\texttt{Decision}_{0.3}(\langle \texttt{g1} \rangle\, \mathcal{N}_5, \langle \neg g1 \rangle\, l_7)$$

$$\mathcal{N}_5 = l_{12}{:}\texttt{Merge}(l_{13}{:}c \rightarrowtail l_9)$$

First, PRISM variables identifiers are deduced from the AC term. We use li and lf as variable identifiers for respectively the initial node ι and the final node $l_8{:}\odot$. For the join node $l_7{:}2.\texttt{Join}$ (resp. $l_9{:}2.\texttt{Join}$), we use $l7$ (resp. $l9$) as identifier for the formula specifying the join conditions and we use the PRISM variables $l7\_1$ and $l7\_2$ (resp. $l9\_1$ and

$l9\_2$) as variables identifiers for the incoming edges of the join node. Once the declaration of variables and formulas is done, the module commands are generated using the algorithm $\mathscr{T}$ described in Listing 7.1. For instance, the first command labeled $[li]$ is generated in the first iteration while calling $\mathscr{T}(\mathcal{A}_1)$. It corresponds to the first case such that $\mathcal{A}_1 = \iota \rightarrowtail \mathcal{N}'$ and $\mathcal{N}' = l_1{:}a \rightarrowtail l_2{:}\texttt{Fork}(\mathcal{N}_1, l_{12})$. Thus, the function $\mathscr{E}$ described in Listing 7.2 is called $\mathscr{E}(\mathcal{N}')((li = 1))((li' = 0))(1.0)(li)$. The last case of the latter function is triggered since $\mathcal{N}'$ is of the form $l{:}a \rightarrowtail \mathcal{N}$. This allows to generate the first command and then a call of $\mathscr{T}(\mathcal{N}')$ is triggered which launches the algorithm again. The translation halts when all the nodes are visited and all instances of the algorithm $\mathscr{T}$ stop their executions. The PRISM MDP code obtained for the activity diagram $\mathcal{A}_1$ is shown in Figure 53. Once, the PRISM code is generated, one can input the code to PRISM model-checker for assessment. However, the properties that have to be verified on the model need to be expressed in adequate temporal logic. The property specification language of PRISM subsumes several well-known probabilistic temporal logics, including PCTL [74], CSL [204], LTL [63] and PCTL* [72]. Moreover, PRISM also extend and customize this logics with additional features. For instance, PRISM adds the ability to determine the actual probability of satisfying a formula, rather than only placing a bound on it.

In order to verify MDP model, we use PCTL* temporal logic. A property that can be verify on the model is the presence/absence of a deadlock. The property (7.5.1) specifies the eventuality of reaching a deadlock state (with probability $P > 0$) from any configuration

```
mdp

const int max_inst=1;
formula l9 = l9_1>0 & l9_2>0 ;
formula l7 = l7_1> 0 & l7_2>0;

module mainmod

g1 : bool init false;
g2 : bool init false;
li : [0..1] init 1; lf : [0..1] init 0;
l1 : [0..max_inst] init 0; l2 : [0..max_inst] init 0;
l3 : [0..max_inst] init 0; l4 : [0..max_inst] init 0;
l5 : [0..max_inst] init 0; l6 : [0..max_inst] init 0;
l7_1 : [0..max_inst] init 0; l7_2 : [0..max_inst] init 0;
l9_1 : [0..max_inst] init 0; l9_2 : [0..max_inst] init 0;
l10 : [0..max_inst] init 0;
l11 : [0..max_inst] init 0; l12 : [0..max_inst] init 0;
l13 : [0..max_inst] init 0;

[li] li=1 & l1<max_inst & lf=0 → 1.0 : (l1'=l1 + 1) & (li'=0);
[l1] l1>0 & l2<max_inst & lf=0 → 1.0 : (l2'=l2 + 1) & (l1'=l1 − 1);
[l2] l2>0 & l3<max_inst & l12<max_inst & lf=0
                                → 1.0 : (l3'=l3 + 1) & (l12'=l12 + 1) & (l2'=l2 − 1);
[l3] l3>0 & l4<max_inst & lf=0 → 1.0 : (l4'=l4 + 1)&(l3'=l3 − 1);
[l4] l4>0 & l5<max_inst & lf=0 → 1.0 : (l5'=l5 + 1)&(l4'=l4 − 1);
[l5] l5>0 & l6<max_inst & !l9 & l9_1<max_inst & lf=0
                                → 1.0 : (l6'=l6 + 1) &(l9_1'=l9_1 + 1) & (l5'=l5 − 1);
[l6] l6>0 & l3<max_inst & !l7 & l7_1<max_inst &lf=0 →
                                    0.9 : (l3'=l3 + 1) & (l6'=l6 − 1) & (g2'=true)
                                    + 0.1 : (l7_1'=l7_1 + 1) & (l6'=l6 − 1) & (g2'=false);
[l7] l7 & lf=0 → 1.0 : (l7_1'=0) & (l7_2'=0) & (lf'=1);
[l9] l9 & l10<max_inst & lf=0 → 1.0 : (l10'=l10 + 1) & (l9_1'=0) & (l9_2'=0);
[l10]l10>0 & l11<max_inst & lf=0 → 1.0 : (l11'=l11 + 1) & (l10'=l10 − 1);
[l11]l11>0 & l12<max_inst & ! l7 &l7_2<max_inst & lf=0 →
                                    0.3 : (l12'=l12 + 1) & (l11'=l11 − 1) & (g1'=true)
                                    + 0.7 : (l7_2'=l7_2 + 1) & (l11'=l11 − 1) & (g1'=false);
[l12] l12>0 & l13<max_inst & lf=0 → 1.0 : (l13'=l13 + 1) & (l12'=l12 − 1);
[l13] l13>0 & !l9 & l9_2<max_inst & lf=0 → 1.0 : (l9_2'=l9_2 + 1) & (l13'=l13 − 1);
[lf] lf=1 → 1.0 : (li'=0)& (lf'=0) & (l1'=0) & (l2'=0) & (l3'=0) &
            (l4'=0) & (l5'=0) & (l6'=0) & (l7_1'=0)& (l7_2'=0)& (l9_1'=0)& (l9_2'=0)&
            (l10'=0)& (l11'=0)& (l2'=0)& (l3'=0) & (g1'=false) & (g2'=false);
endmodule
```

Figure 53: PRISM Code for the SysML Activity Diagram Case Study

starting at the initial state can be expressed as follows:

$$\text{“}init\text{”} \Rightarrow \texttt{P} > 0 \; [\; \texttt{F} \; \text{“}\texttt{deadlock}\text{”} \;] \tag{7.5.1}$$

Using PRISM model-checker, this property returns true. In fact, the execution of the action `Choose account` twice (because the guard $g1$ is true) and the action `Verify ATM` only once (because $g2$ evaluated to false) result in a deadlock configuration where the condition of the join node `join1` is never fulfilled.

7.6 Simulation Preorder for Markov Decision Processes

Simulation preorder represents one example of relations that have been defined in both non-probabilistic and probabilistic settings in order to establish a step-by-step correspondences between two systems. Segala and Lynch have defined in their seminal work [205] several extensions of the classical simulation and bisimulation relations to the probabilistic settings. These definitions have been reused and tailored in Baier and Kwiatkowska [206] and recently in Kattenbelt and Huth [207]. Simulations are unidirectional relations that have proved to be successful in formal verification of systems. Indeed, they allow to perform abstractions of the models while preserving safe CTL properties [208]. Simulation relations are preorders on the state space such that a state s simulates state s' (written $s \sqsubseteq s'$) if and only if s' can mimic all stepwise behavior of s. However, the inverse is not always true; s' may perform steps that cannot be matched by s.

In probabilistic settings, strong simulation have been introduced, where $s \sqsubseteq s'$ (meaning

s' strongly simulates s) requires that every α-successor distribution of s, has a corresponding α-successor at s'. This correspondence between distribution is defined based on the concept of weight functions [209]. States related with strong simulation have to be related via weight functions on their distributions [208]. Let $\mathscr{M}$ be the class of all MDPs. A formal definition of an MDP is provided in Chapter 2 Section 2.6 Definition 2.6.3. In the following, we recall the definitions related to strong simulation applied on MDPs. First, we define the concept of weight functions as follows.

Definition 7.6.1. Let $\mu \in Dist(S)$ and $\mu' \in Dist(S')$ and $R \subseteq S \times S'$. A weight function for (μ, μ') w.r.t. R is a function $\delta: S \times S' \to [0, 1]$ satisfying the following:

- $\delta(s, s') > 0$ implies $(s, s') \in R$
- For all $s \in S$ and $s' \in S'$, $\sum_{s' \in S'} \delta(s, s') = \mu(s)$ and $\sum_{s \in S} \delta(s, s') = \mu'(s')$.

We write $\mu \preceq_R \mu'$ if there exists such a weight function δ for (μ, μ') with respect to R. □

Definition 7.6.2. Let $M = (S, s_0, Act, Steps)$ and $M' = (S', s'_0, Act', Steps')$ be two MDPs. We say M' simulates M via a relation $R \subseteq S \times S'$, denoted by $M \sqsubseteq^R_{\mathscr{M}} M'$, if and only if for all s and s': $(s, s') \in R$, if $s \xrightarrow{\alpha} \mu$ then there is a transition $s' \xrightarrow{\alpha} \mu'$ with $\mu \preceq_R \mu'$. □

Basically, we say that M' strongly simulates M, denoted $M \sqsubseteq^R_{\mathscr{M}} M'$, iff there exists a strong simulation R between M and M' such that for every $s \in S$ and $s' \in M'$ each α-successor of s has a corresponding α-successor of s' and there exist a weight function δ that can be defined between the successor distributions of s and s'.

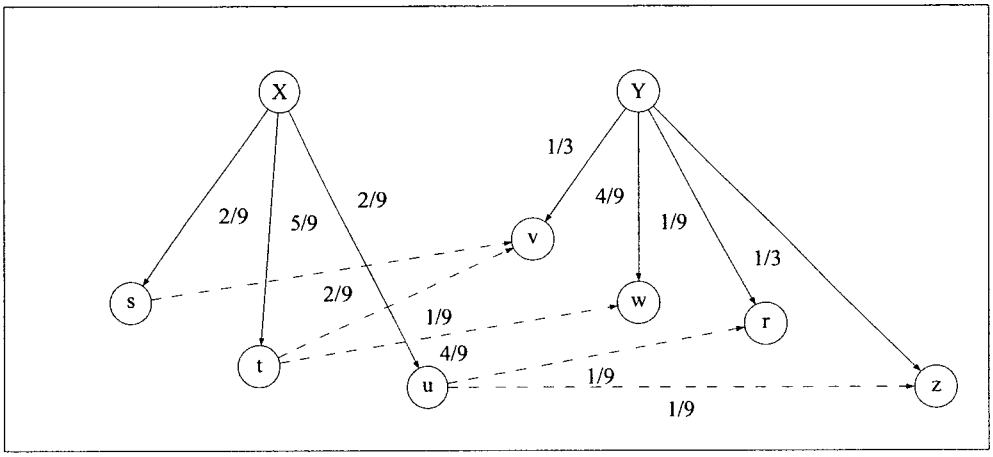


Figure 54: Example of Simulation Relation using Weight Function

Example 7.6.1. Let consider the example illustrated in Figure 54. We consider two set of states $S = \{s, t, u\}$ destination of X and $S' = \{v, w, r, z\}$ destination states of Y. The distribution μ over S is defined as follows: $\mu(s) = 2/9$, $\mu(t) = 5/9$, and $\mu(u) = 2/9$, whereas the distribution μ' over S' is defined such that $\mu'(v) = 1/3$, $\mu'(w) = 4/9$, $\mu'(r) = 1/9$, and $\mu'(z) = 1/3$. If we consider the relation R such that $R = \{(s, v), (t, v), (t, w), (u, r), (u, z)\}$, we can find out if R is a simulation relation provided that we can define a weight function that fulfills the constraint of being a weight function relating μ and μ'. Let δ be a weight function such that $\delta(s, v) = \frac{2}{9}$, $\delta(t, v) = \frac{1}{9}$, $\delta(t, w) = \frac{4}{9}$, $\delta(u, r) = \frac{1}{9}$, and $\delta(u, z) = \frac{1}{9}$ fulfills the constraints of being a weight function. According to Definition 7.6.1, the first condition is satisfied. For the second condition we have $\sum_{s' \in S'} \delta(t, s') = \delta(t, v) + \delta(t, w) = \frac{5}{9} = \mu(t)$, $\sum_{s_1 \in S} \delta(s_1, v) = \delta(s, v) + \delta(t, v) = \frac{3}{9} = \mu(t)$, and $\sum_{s' \in S'} \delta(u, s') = \delta(u, r) + \delta(u, z) = \frac{2}{9} = \mu(u)$. It follows that $\mu \preceq_R \mu'$. Thus, $X \sqsubseteq^R_{\mathscr{M}} Y'$.

7.7 Soundness of the Translation Algorithm

In this section, we aim at ensuring that the translation function $\mathscr{T}$ defined in Listing 7.1 generates a model that correctly captures the behavior of the activity diagram. More precisely, we look forward to prove the soundness of the translation. To this end, we use the operational semantics defined for both the SysML activity diagrams and the PRISM input language. Before formalizing the soundness theorem, we need first to make some important definitions.

We use the function $\lfloor\_\rfloor$ specified in Listing 7.3. The latter takes as input a term $\mathcal{B}$ in $\overline{\mathcal{AC}} \cup \mathcal{AC}$ and returns a multiset of labels $(\mathcal{L}_{\mathcal{B}}, m)$ corresponding to the marked nodes in the corresponding activity calculus term, i.e. $\lfloor\mathcal{B}\rfloor = \{\!|\, l_j \in \mathcal{L}_{\mathcal{B}} \mid m(l_j) > 0 \,|\!\}$.

In the next definition, we make use of the function $[\![\_]\!](\_)$ defined in Section 7.3.2 in order to define how an activity calculus term $\mathcal{B}$ satisfies a boolean expression. This is needed in order to define a relation between a state in the semantic model of PRISM model and another state in the semantic model of the corresponding SysML activity diagram.

Definition 7.7.1. An Activity Calculus term $\mathcal{B}$ such that $\lfloor\mathcal{B}\rfloor=(\mathcal{L}_{\mathcal{B}}, m)$, satisfies a boolean expression e and we write $[\![e]\!](\mathcal{B}) = true$ iff $[\![e]\!](s[x_i \mapsto m(l_i)]) = true$, $\forall\ l_i \in \mathcal{L}_{\mathcal{B}}$ and $x_i \in$ *variables*. □

The evaluation of the boolean expression e using the term $\mathcal{B}$ consists of two steps. First, a store s is defined where we assign to each variable x_i the marking of the node labeled l_i. The second step is to replace in the boolean expression e each variable x_i with $s(x_i)$.

Let $M_{\mathcal{P}_{\mathcal{A}}} = (S_{\mathcal{P}_{\mathcal{A}}}, s_0, Act, Steps_{\mathcal{P}_{\mathcal{A}}})$ and $M_{\mathcal{A}} = (S_{\mathcal{A}}, s_0, Act, Steps_{\mathcal{A}})$ be the MDPs

corresponding respectively to the PRISM model $\mathcal{P}_{\mathcal{A}}$ and the SysML activity diagram $\mathcal{A}$.

We need to define in the following a relation $\mathscr{R} \subseteq S_{\mathcal{P}_{\mathcal{A}}} \times S_{\mathcal{A}}$.

Listing 7.3: Function $\lfloor - \rfloor$ Definition

$\lfloor - \rfloor : \overline{\mathcal{AC}} \longrightarrow P^{\mathcal{L}}$

$\lfloor \mathcal{M} \rfloor$ = Case $(\mathcal{M})$ **of**

$\overline{\iota} \mapsto \mathcal{N}$ $\Rightarrow$ $\{\!|\, (l_\iota \hookrightarrow 1) \,|\!\}$

$\iota \mapsto \mathcal{M}'$ $\Rightarrow$ $\lfloor \mathcal{M}' \rfloor$

$\overline{l_f : \odot}^n$ $\Rightarrow$ **if** $n > 0$ **then** $\{\!|\, (l_f \hookrightarrow 1) \,|\!\}$ **else** $\{\!|\!\}$

$\overline{l : Merge(\mathcal{M}')}^n$ $\Rightarrow \{\!|\, (l \hookrightarrow n) \,|\!\} \uplus \lfloor \mathcal{M}' \rfloor$

$\overline{l : x.Join(\mathcal{M}')}^n$ $\Rightarrow \{\!|\, (l[1] \hookrightarrow n) \,|\!\} \uplus \lfloor \mathcal{M}' \rfloor$

$\overline{l : Fork(\mathcal{M}_1, \mathcal{M}_2)}^n$ $\Rightarrow \{\!|\, (l \hookrightarrow n) \,|\!\} \uplus \lfloor \mathcal{M}_1 \rfloor \uplus \lfloor \mathcal{M}_2 \rfloor$

$\overline{l : Decision_p(\langle g \rangle \mathcal{M}_1, \langle \neg g \rangle \mathcal{M}_2)}^n$ $\Rightarrow \{\!|\, (l \hookrightarrow n) \,|\!\} \uplus \lfloor \mathcal{M}_1 \rfloor \uplus \lfloor \mathcal{M}_2 \rfloor$

$\overline{l : Decision(\langle g \rangle \mathcal{M}_1, \langle \neg g \rangle \mathcal{M}_2)}^n$ $\Rightarrow \{\!|\, (l \hookrightarrow n) \,|\!\} \uplus \lfloor \mathcal{M}_1 \rfloor \uplus \lfloor \mathcal{M}_2 \rfloor$

$\overline{l : a}^n \rightarrowtail \mathcal{M}'$ $\Rightarrow$ $\{\!|\, (l \hookrightarrow n) \,|\!\} \uplus \lfloor \mathcal{M}' \rfloor$

$\overline{l}^n$ $\Rightarrow$ **if** $l = \mathscr{L}(l : x.join(\mathcal{N}))$ **then**

$\{\!|\, (l[k] \hookrightarrow n) \,|\!\}$

else

$\{\!|\, (l \hookrightarrow n) \,|\!\}$

end

Otherwise $\Rightarrow$ $\{\!|\!\}$

Definition 7.7.2. Let $\mathscr{R} \subseteq S_{\mathcal{P}_{\mathcal{A}}} \times S_{\mathcal{A}}$ be a relation defined as follows.

For all $s_p \in S_{\mathcal{P}_{\mathcal{A}}}$ and $\mathcal{B} \in S_{\mathcal{A}}$, $s_p \mathscr{R} \mathcal{B}$ iff for any expression $w \in \mathcal{W}$, $[\![w]\!](s_p) = true$ implies $[\![w]\!](\mathcal{B}) = true$, $\forall\ l_i \in \mathcal{L}_{\mathcal{B}}$ and $x_i \in$ *variables*. □

This definition states that an AC term $\mathcal{B}$ is in relation with a state s if they both satisfy

the same boolean expression. Based on this relation, we present hereafter the soundness theorem.

Theorem 7.7.1. (Soundness) Let $\mathcal{A}$ be an AC term of a given SysML activity diagram and $M_{\mathcal{A}}$ its corresponding semantic model. Let $\mathcal{T}(\mathcal{A})=\mathcal{P}_{\mathcal{A}}$ be the corresponding PRISM model and $M_{\mathcal{P}_{\mathcal{A}}}$ be its MDP. We say that the translation algorithm $\mathcal{T}$ is sound if $M_{\mathcal{P}_{\mathcal{A}}} \sqsubseteq^{\mathcal{R}}_{\mathcal{M}} M_{\mathcal{A}}$. □

Proof of Soundness. In order to prove the theorem, we have to prove the existence of a simulation relation between both the MDPs $M_{\mathcal{A}}$ and $M_{\mathcal{P}_{\mathcal{A}}}$. We reason by structural induction on the syntax of the AC term describing the activity diagram. The proof process consists in proving that the soundness holds for the base cases, which are $l{:}\otimes$ and $l{:}\odot$ and then proving it for the inductive cases.

- **Case of** $l{:}\otimes$

  The algorithm generates neither a PRISM variable nor a PRISM command associated with this element. So, there is no transition in $M_{\mathcal{P}_{\mathcal{A}}}$ corresponding to this element. According to the operational semantics, we have $\overline{l{:}\otimes}^{n} \equiv_M l{:}\otimes$ (according to (M3) in Figure 35) and there is no operational inference rule associated with this element. So there is no transition in $M_{\mathcal{A}}$ associated with this element. Thus, the theorem holds for this case.

- **Case of** $l{:}\odot$

  The algorithm generates a single PRISM command such that:

  $[l_f]\ l_f = 1 \rightharpoonup 1.0 : l'_f = 0.$

There exists a state s_0 satisfying the corresponding guard, meaning $[\![(l_f = 1)]\!](s_0) = true$, from which emanates a transition of the form $s_0 \longrightarrow \mu_2$ where $\mu_2 = \mu_1^{s_1}$.

The activity calculus term $\mathcal{B} = \overline{l_f{:}\odot}$ is related to s_0 via $\mathscr{R}$, since $[\![(l_f = 1)]\!](\overline{l_f{:}\odot}) = true$ (Definition 7.7.2). This can be justified because $\lfloor \overline{l_f{:}\odot} \rfloor = \{\!|\ (l_f \hookrightarrow 1)\ |\!\}$ (Listing 7.3).

According to the operational Rule FINAL, we have $\overline{l_f{:}\odot} \longrightarrow \mu$ such that $\mu = \mu_1^{l_f{:}\odot}$.

For $\mathscr{R} = \{(s_0, \overline{l_f{:}\odot}), (s_1, l_f{:}\odot)\}$, it follows that $\mu \preceq_{\mathscr{R}} \mu_2$ as δ defined such that $\delta(s_1, l_f{:}\odot) = 1$ fulfills the constraints of being a weight function. Thus, the theorem is proved for this case.

Let $M_{\mathcal{N}}$ denote the semantics of $\mathcal{N}$ and $M_{\mathcal{P}_{\mathcal{N}}}$ denotes the semantics of the corresponding PRISM model obtained from $\mathscr{T}(\mathcal{N})$. We assume that $M_{\mathcal{P}_{\mathcal{N}}} \sqsubseteq^{\mathscr{R}}_{\mathscr{M}} M_{\mathcal{N}}$. We also assume a bounded number of instances max_inst, i.e. $\forall\ l, l \leq max\_inst$ (ASSUMPTION 1).

- **Case of $\iota \rightarrowtail \mathcal{N}$**

  According to the translation algorithm, we have $\mathscr{T}(\iota \rightarrowtail \mathcal{N}) = \{c\} \cup \mathscr{T}(\mathcal{N})$. By assumption of the inductive step we have $M_{\mathcal{P}_{\mathcal{N}}} \sqsubseteq^{\mathscr{R}}_{\mathscr{M}} M_{\mathcal{N}}$. Thus, we need to prove the theorem for the command c such as:

  $c = \mathscr{E}(\mathcal{N})(l_\iota = 1)(l'_\iota = 0)(1.0)(l_\iota)$.

  Let w be the guard and d the update generated by $\mathscr{E}(\mathcal{N})$, the command c can be written as follows:

$c = [l_\iota]\ w \wedge (l_\iota = 1) \wedge (l_f = 0) \rightharpoonup 1.0 : d \wedge (l'_\iota = 0)$

There exists a state s_0 satisfying the guard, i.e. $[\![w \wedge (l_\iota = 1) \wedge (l_f = 0)]\!](s_0) = true$ and a transition $s_0 \longrightarrow \mu$ where $\mu = \mu_1^{s_1}$.

Given that $\lfloor \bar{\iota} \mapsto \mathcal{N} \rfloor = \{\!|\ (l_\iota \hookrightarrow 1)\ |\!\}$, we can easily verify that $[\![(l_\iota = 1) \wedge (l_f = 0)]\!](\bar{\iota} \mapsto \mathcal{N}) = true$. It remains to verify that $\forall \mathcal{N}, [\![w]\!](\bar{\iota} \mapsto \mathcal{N}) = true$.

There are two cases: $w = \neg(\wedge_{1 \leq k \leq x}(l[k] > 0)) \wedge (l[j] < max\_inst)$ or $w = (l < max\_inst)$. Since $\forall l \neq l_\iota$, $m(l) = 0$ and having ASSUMPTION 1, we can conclude that $[\![w]\!](\bar{\iota} \mapsto \mathcal{N}) = true$. Thus, by Definition 7.7.2 $s_0\ \mathscr{R}\ \bar{\iota} \mapsto \mathcal{N}$.

The operational semantics rule INIT-1 allows a transition $\bar{\iota} \mapsto \mathcal{N} \longrightarrow \mu'$ such that $\mu'(\iota \mapsto \overline{\mathcal{N}}) = 1$.

For $\mathscr{R} = \{(s_0, \bar{\iota} \mapsto \mathcal{N}), (s_1, \iota \mapsto \overline{\mathcal{N}})\}$, it follows that $\mu' \preceq_{\mathscr{R}} \mu$ as δ defined such that $\delta(s_1, \iota \mapsto \overline{\mathcal{N}}) = 1$ fulfills the constraints of being a weight function. Thus, the theorem is proved for this case.

- **Case of** $l{:}a \mapsto \mathcal{N}$

This case can be proved similarly to the previous one. We need to prove the theorem for the command c expressed as follows:

$c = \mathscr{E}^e(\mathcal{N})(l > 0)(l' = l - 1)(1.0)(l)$.

Let w be the guard and d the update such that:

$c = [l]\ w \wedge (l > 0) \wedge (l_f = 0) \rightharpoonup 1.0 : d \wedge (l' = l - 1)$

The corresponding transition is $s_0 \longrightarrow \mu$ where $\mu = \mu_1^{s_1}$ and $[\![w \wedge (l > 0) \wedge (l_f = 0)]\!](s_0) = true$.

We can verify easily that $[\![w \wedge (l > 0) \wedge (l_f = 0)]\!](\overline{l{:}a \rightarrowtail \mathcal{N}}) = true$.

Let $\mathscr{R} = \{(s_0, \overline{l{:}a \rightarrowtail \mathcal{N}}), (s_1, l{:}a \rightarrowtail \overline{\mathcal{N}})\}$.

The operational semantics rule ACT-1 allows a transition $\overline{l{:}a \rightarrowtail \mathcal{N}} \xrightarrow{a} \mu'$

such that $\mu'(l{:}a \rightarrowtail \overline{\mathcal{N}}) = 1$.

It follows that $\mu' \preceq_{\mathscr{R}} \mu$ as δ defined such that $\delta(s_1, l{:}a \rightarrowtail \overline{\mathcal{N}}) = 1$ fulfills the constraints of being a weight function. Thus, the theorem is proved for this case.

- **Case of** $l{:}Merge(\mathcal{N})$

  $\mathscr{T}(l{:}Merge(\mathcal{N})) = \{c\} \cup \mathscr{T}(\mathcal{N})$. In order to prove this case, we need to prove the theorem for the command c.

  $c = \mathscr{E}(\mathcal{N})(l > 0)(l' = l - 1)(1.0)(l)$.

  Let denote by w the guard and d the update such that:

  $c = [l]\, w \wedge (l > 0) \wedge (l_f = 0) \rightharpoonup 1.0 : d \wedge (l' = l - 1)$

  The state s_0 such that $[\![w \wedge (l > 0) \wedge (l_f = 0)]\!](s_0) = true$ is the source of a transition of the form $s_0 \longrightarrow \mu$ where $\mu = \mu_1^{s_1}$.

  We can easily verify that $[\![w \wedge (l > 0) \wedge (l_f = 0)]\!](\overline{l{:}Merge(\mathcal{N})}) = true$. Thus, by Definition 7.7.2, $(s_0, \overline{l{:}Merge(\mathcal{N})}) \in \mathscr{R}$.

  Let $\mathscr{R} = \{(s_0, \overline{l{:}Merge(\mathcal{N})}), (s_1, l{:}Merge(\overline{\mathcal{N}}))\}$.

  The operational semantics rule MERGE-1 allows a transition $\overline{l{:}Merge(\mathcal{N})} \longrightarrow \mu'$

  such that $\mu'(l{:}Merge(\overline{\mathcal{N}})) = 1$.

It follows that $\mu' \preceq_{\mathscr{R}} \mu$ as δ defined such that: $\delta(s_1, l{:}Merge(\overline{\mathcal{N}})) = 1$ fulfills the constraints of being a weight function. Thus, the theorem is proved for this case.

- **Case of** $l{:}x.Join(\mathcal{N})$

This case is jointly treated with the case where l is a reference of a join node. $\mathscr{T}(l{:}x.Join(\mathcal{N})) = \{c\} \cup \mathscr{T}(\mathcal{N})$ such that:

$c = \mathscr{E}(\mathcal{N})(\bigwedge_{1\leq k\leq x}(l[k] > 0))(\bigwedge_{1\leq k\leq x}(l[k]' = 0))(1.0)(l[1])$.

Let denote by w the guard and d the update such that:

$c = [l]\, w \wedge (\bigwedge_{1\leq k\leq x}(l[k] > 0)) \wedge (l_f = 0) \rightharpoonup 1.0 : d \wedge (\bigwedge_{1\leq k\leq x}(l[k]' = 0)$

The corresponding transition is $s_0 \longrightarrow \mu$ where $\mu = \mu_1^{s_1}$ and $[\![w \wedge (\bigwedge_{1\leq k\leq x}(l[k] > 0)) \wedge (l_f = 0)]\!](s_0) = true$.

The activity calculus term $\mathcal{B}[\overline{l{:}x.Join(\mathcal{N})}, \bar{l}\{x{-}1\}]$ satisfies the guard $w \wedge (\bigwedge_{1\leq k\leq x}(l[k] > 0)) \wedge (l_f = 0)$. Thus by Definition 7.7.2, $s_0 \mathscr{R} \mathcal{B}$.

The operational semantics rule JOIN-1 allows a transition such that:

$\mathcal{B}[\overline{l{:}x.Join(\mathcal{N})}, \bar{l}\{x-1\}] \longrightarrow_1 \mathcal{B}[l{:}x.Join(\overline{\mathcal{N}}), l\{x-1\}]$

For $\mathcal{B}' = \mathcal{B}[l{:}x.Join(\overline{\mathcal{N}}), l\{x-1\}]$, we can write $\mathcal{B} \longrightarrow \mu'$ such that $\mu'(\mathcal{B}') = 1$.

Let $\mathscr{R} = \{(s_0, \mathcal{B}), (s_1, \mathcal{B}')\}$. It follows that $\mu' \preceq_{\mathscr{R}} \mu$ as δ defined such that: $\delta(s_1, \mathcal{B}') = 1$ fulfills the constraints of being a weight function. Thus, the theorem is proved for this case.

- **Case of** l

l is a reference to $l{:}x.Join(\mathcal{N})$ or to $l{:}Merge(\mathcal{N})$. Thus, the proof can be inferred from the previous corresponding cases.

Let $M_{\mathcal{N}_1}$ and $M_{\mathcal{N}_2}$ respectively denote the semantics of $\mathcal{N}_1$ and $\mathcal{N}_2$. Also, $M_{\mathcal{P}_{\mathcal{N}_1}}$ denotes the semantics of the corresponding PRISM translation $\mathcal{T}(\mathcal{N}_1)$ and $M_{\mathcal{P}_{\mathcal{N}_2}}$ the one of $\mathcal{T}(\mathcal{N}_2)$. We assume that $M_{\mathcal{P}_{\mathcal{N}_1}} \sqsubseteq^{\mathscr{R}}_{\mathscr{M}} M_{\mathcal{N}_1}$ and $M_{\mathcal{P}_{\mathcal{N}_2}} \sqsubseteq^{\mathscr{R}}_{\mathscr{M}} M_{\mathcal{N}_2}$.

- **Case of** $l{:}Fork(\mathcal{N}_1, \mathcal{N}_2)$

  According to the translation algorithm, we have $\mathcal{T}(\iota \rightarrowtail \mathcal{N}) = \{c\} \cup \mathcal{T}(\mathcal{N}_1) \cup \mathcal{T}(\mathcal{N}_2)$. By assumption of the inductive step we have $M_{\mathcal{P}_{\mathcal{N}_1}} \sqsubseteq^{\mathscr{R}}_{\mathscr{M}} M_{\mathcal{N}_1}$ and $M_{\mathcal{P}_{\mathcal{N}_2}} \sqsubseteq^{\mathscr{R}}_{\mathscr{M}} M_{\mathcal{N}_2}$. Thus, we need to prove the theorem for the command c.

  Let denote by w_1 and w_2 the guards generated respectively for $\mathcal{N}_1$ and $\mathcal{N}_2$ and d_1 and d_2 the corresponding updates such that:

  $c = [l]\ w_1 \wedge w_2 \wedge (l > 0) \wedge (l_f = 0) \rightharpoonup 1.0 : d_1 \wedge d_2 \wedge (l' = l - 1)$

  Thus, there exists a state s_0 such that $[\![w_1 \wedge w_2 \wedge (l > 0) \wedge (l_f = 0)]\!](s_0) = true$ and a transition $s_0 \longrightarrow \mu$ where $\mu = \mu_1^{s_1}$.

  We can easily verify that $[\![w_1 \wedge w_2 \wedge (l > 0) \wedge (l_f = 0)]\!](\overline{l{:}Fork(\mathcal{N}_1, \mathcal{N}_2)}) = true$. So, $s_0\ \mathscr{R}\ \overline{l{:}Fork(\mathcal{N}_1, \mathcal{N}_2)}$

  According to Rule FORK-1, we have $\overline{l{:}Fork(\mathcal{N}_1, \mathcal{N}_2)} \longrightarrow_1 l{:}Fork(\overline{\mathcal{N}_1}, \overline{\mathcal{N}_2})$, or $\overline{l{:}Fork(\mathcal{N}_1, \mathcal{N}_2)} \longrightarrow \mu'$, where $\mu' = \mu_1^{l:Fork(\overline{\mathcal{N}_1}, \overline{\mathcal{N}_2})}$.

  Let $\mathscr{R} = \{(s_0, \overline{l{:}Fork(\mathcal{N}_1, \mathcal{N}_2)}), (s_1, l{:}Fork(\overline{\mathcal{N}_1}, \overline{\mathcal{N}_2}))\}$. It follows that $\mu' \preceq_{\mathscr{R}} \mu$ as δ defined such that: $\delta(s_1, l{:}Fork(\overline{\mathcal{N}_1}, \overline{\mathcal{N}_2})) = 1$ fulfills the constraints of being a weight function. Thus, the theorem is proved for this case.

- **Case of** $l{:}Decision_p(\langle g \rangle\ \mathcal{N}_1,\ \langle \neg g \rangle\ \mathcal{N}_2)$

  The translation algorithm results in the following:

$\mathscr{T}(l{:}Decision_p(\langle g\rangle \mathcal{N}_1,\ \langle \neg g\rangle \mathcal{N}_2))$= $([l]\ w_1 \wedge w_2 \wedge (l > 0) \wedge (l_f = 0) \rightharpoonup \lambda_1 : d_1 \wedge (l' = l - 1) \wedge (g' = true) + \lambda_2 : d_2 \wedge (l' = l - 1) \wedge (g' = false)) \cup \mathscr{T}(\mathcal{N}_1) \cup \mathscr{T}(\mathcal{N}_2)$.

Given the assumption of the inductive step, we have to prove the theorem for the following command:

$c = [l]\ w_1 \wedge w_2 \wedge (l > 0) \wedge (l_f = 0) \rightharpoonup p : d_1 \wedge (l' = l - 1) \wedge (g' = true) + (1 - p) : d_2 \wedge l' = l - 1 \wedge (g' = false)$.

There exists a state s_0 such that the command c is enabled, i.e. $[\![w_1 \wedge w_2 \wedge (l > 0) \wedge (l_f = 0)]\!](s_0) = true$. The enabled transition is of the form $s_0 \longrightarrow \mu$ such that $\mu(s_1) = p$ and $\mu(s_2) = 1 - p$.

$[\![w_1 \wedge w_2 \wedge (l > 0) \wedge (l_f = 0)]\!](\overline{l{:}Decision_p(\langle g\rangle \mathcal{N}_1,\ \langle \neg g\rangle \mathcal{N}_2)}) = true$.

Thus, $s_0\ \mathscr{R}\ \overline{l{:}Decision_p(\langle g\rangle \mathcal{N}_1,\ \langle \neg g\rangle \mathcal{N}_2)}$.

According to the operational semantics, there are two possible transitions emanating from the configuration $\overline{l{:}Decision_p(\langle g\rangle \mathcal{N}_1,\ \langle \neg g\rangle \mathcal{N}_2)}$. The transition enabled by Rule PDEC-1:

$\overline{l{:}Decision_p(\langle g\rangle \mathcal{N}_1, \langle \neg g\rangle \mathcal{N}_2)} \longrightarrow_p l{:}Decision_p(\langle tt\rangle \overline{\mathcal{N}_1}, \langle ff\rangle \mathcal{N}_2)$.

The transition enabled by Rule PDEC-2:

$\overline{l{:}Decision_p(\langle g\rangle \mathcal{N}_1, \langle \neg g\rangle \mathcal{N}_2)} \longrightarrow_{1-p} l{:}Decision_p(\langle ff\rangle \mathcal{N}_1, \langle tt\rangle \overline{\mathcal{N}_2})$.

Definition 6.3.1 defines a transition in the Markov decision process M_A such that $\overline{l{:}Decision_p(\langle g\rangle \mathcal{N}_1, \langle \neg g\rangle \mathcal{N}_2)} \longrightarrow \mu'$ where $\mu'(l{:}Decision_p(\langle tt\rangle \overline{\mathcal{N}_1}, \langle ff\rangle \mathcal{N}_2)) = p$ and $\mu'(l{:}Decision_p(\langle ff\rangle \mathcal{N}_1, \langle tt\rangle \overline{\mathcal{N}_2})) = 1 - p$.

Let $\mathscr{R} = \{(s_0, \overline{l{:}Decision_p(\langle g\rangle \mathcal{N}_1, \langle \neg g\rangle \mathcal{N}_2)}), (s_1, l{:}Decision_p(\langle tt\rangle \overline{\mathcal{N}_1}, \langle ff\rangle \mathcal{N}_2)),$

$(s_2, l{:}Decision_p(\langle ff\rangle \mathcal{N}_1, \langle tt\rangle \overline{\mathcal{N}_2}))\}$. It follows that $\mu' \preceq_{\mathscr{R}} \mu$ as δ defined such that: $\delta(s_1, l{:}Decision_p(\langle tt\rangle \overline{\mathcal{N}_1}, \langle ff\rangle \mathcal{N}_2)) = p$ and $\delta(s_2, l{:}Decision_p(\langle ff\rangle \mathcal{N}_1, \langle tt\rangle \overline{\mathcal{N}_2})) = 1 - p$ fulfills the constraints of being a weight function. Thus, the theorem is proved for this case.

- **Case of** $l{:}Decision(\langle g\rangle \mathcal{N}_1, \langle \neg g\rangle \mathcal{N}_2)$

  The translation algorithm results in the following:

  $\mathscr{T}(l{:}Decision(\langle g\rangle \mathcal{N}_1, \langle \neg g\rangle \mathcal{N}_2)) = ([l]\ w \rightharpoonup 1.0 : d) \cup \mathscr{T}(\mathcal{N}_1) \cup ([l]\ w' \rightharpoonup 1.0 : d') \cup \mathscr{T}(\mathcal{N}_2)$

  Given the assumption of the inductive step, we have to prove the theorem for two commands c_1 and c_2:

  $c_1 = [l]\ w_1 \wedge (l > 0) \wedge (l_f = 0) \rightharpoonup 1.0 : d_1 \wedge (l' = l - 1) \wedge (g' = true)$.

  $c_2 = [l]\ w_2 \wedge (l > 0) \wedge (l_f = 0) \rightharpoonup 1.0 : d_2 \wedge (l' = l - 1) \wedge (g' = false)$.

  There exists a state s_0 such that the commands c_1 and c_2 are enabled, i.e. $[\![w_1 \wedge (l > 0) \wedge (l_f = 0)]\!](s_0) = true$ and $[\![w_2 \wedge (l > 0) \wedge (l_f = 0)]\!](s_0) = true$. Two enabled transitions from s_0 such that $s_0 \longrightarrow \mu_1$ where $\mu_1(s_1) = 1$ and $s_0 \longrightarrow \mu_2$ where $\mu_2(s_2) = 1$.

  We have $[\![w_1 \wedge (l > 0) \wedge (l_f = 0)]\!](\overline{l{:}Decision(\langle g\rangle \mathcal{N}_1, \langle \neg g\rangle \mathcal{N}_2)}) = true$.

  Also, $[\![w_2 \wedge (l > 0) \wedge (l_f = 0)]\!](\overline{l{:}Decision(\langle g\rangle \mathcal{N}_1, \langle \neg g\rangle \mathcal{N}_2)}) = true$.

  Thus, $s_0\ \mathscr{R}\ \overline{l{:}Decision(\langle g\rangle \mathcal{N}_1, \langle \neg g\rangle \mathcal{N}_2)}$.

  According to the operational semantics, there are two possible transitions emanating from the configuration $\overline{l{:}Decision(\langle g\rangle \mathcal{N}_1, \langle \neg g\rangle \mathcal{N}_2)}$. The transition enabled by Rule

DEC-1:

$\overline{l{:}Decision(\langle g\rangle\mathcal{N}_1, \langle\neg g\rangle\mathcal{N}_2)} \longrightarrow_1 l{:}Decision(\langle tt\rangle\overline{\mathcal{N}_1}, \langle ff\rangle\mathcal{N}_2)$.

The transition enabled by Rule DEC-2:

$\overline{l{:}Decision(\langle g\rangle\mathcal{N}_1, \langle\neg g\rangle\mathcal{N}_2)} \longrightarrow_1 l{:}Decision(\langle ff\rangle\mathcal{N}_1, \langle tt\rangle\overline{\mathcal{N}_2})$.

Definition 6.3.1 defines two transitions in the Markov decision process M_A emanating from the same state such that $\overline{l{:}Decision(\langle g\rangle\mathcal{N}_1, \langle\neg g\rangle\mathcal{N}_2)} \longrightarrow \mu_1'$ where $\mu_1'(l{:}Decision(\langle tt\rangle\overline{\mathcal{N}_1}, \langle ff\rangle\mathcal{N}_2)) = 1$ and $\overline{l{:}Decision(\langle g\rangle\mathcal{N}_1, \langle\neg g\rangle\mathcal{N}_2)} \longrightarrow \mu_1'$ where $\mu_2'(l{:}Decision(\langle ff\rangle\mathcal{N}_1, \langle tt\rangle\overline{\mathcal{N}_2})) = 1$.

Let $\mathscr{R} = \{(s_0, \overline{l{:}Decision(\langle g\rangle\mathcal{N}_1, \langle\neg g\rangle\mathcal{N}_2)}), (s_1, l{:}Decision(\langle tt\rangle\overline{\mathcal{N}_1}, \langle ff\rangle\mathcal{N}_2)),$ $(s_2, l{:}Decision(\langle ff\rangle\mathcal{N}_1, \langle tt\rangle\overline{\mathcal{N}_2}))\}$. It follows that $\mu_1' \preceq_{\mathscr{R}} \mu_1$ and $\mu_2' \preceq_{\mathscr{R}} \mu_2$ as δ_1 defined such that: $\delta_1(s_1, l{:}Decision(\langle tt\rangle\overline{\mathcal{N}_1}, \langle ff\rangle\mathcal{N}_2)) = 1$

and δ_2 defined such that $\delta_2(s_2, l{:}Decision(\langle ff\rangle\mathcal{N}_1, \langle tt\rangle\overline{\mathcal{N}_2})) = 1$ fulfill both the constraints of being weight functions. Thus, the theorem is proved for this case.

□

7.8 Conclusion

The main result of this chapter was the proof that our translation algorithm is sound. This establishes confidence in that the PRISM code generated by our algorithm correctly captures the behavior intended by the SysML activity diagram given as input. This ensures the correctness of our probabilistic verification approach. The next chapter concludes our thesis with a summary of our contributions and open future research directions.

Chapter 8

Conclusion

The main intent of this thesis is to propose an innovative V&V approach in order to assist systems engineers in their mission of building fail-safe and highly performing systems that meet the stakeholders' vision. Thus, we proposed a heretofore unattempted model-based approach to the verification and validation of systems engineering models expressed using UML 2.1 [21] and SysML1.0 [10]. The proposed approach supports the new model-based systems engineering paradigm by enabling the systematic assessment of the design solution early in the system's life cycle. It provides a critical appraisal of the design quality and verifies its alignment with the design objectives and requirements. These results help the systems engineers take appropriate actions in order to remedy the detected deficiencies before the cost to repair skyrockets.

Our main contributions can be summarized as follows. First, we elaborated a unified approach composed of three well-established techniques that are: model-checking, software engineering metrics, and static analysis. These techniques are synergistically composed

together in order to quantitatively and qualitatively analyze the design and cover the assessment of its structural as well as its behavioral perspectives. Each selected technique target a specific issue and together they allow an efficient and enhanced evaluation of the design model. From the behavioral perspective, we applied qualitative model-checking synergistically combined with static analysis and software metrics on UML/SysML state machine, sequence, and activity diagrams. We integrated static analysis techniques and software metrics prior to model-checking in order to tackle model-checking scalability issues. Moreover, we used software metrics on behavioral diagrams in order to appraise their size and complexity. With respect to requirements and other properties that need to be verified, we defined an intuitive easy-to-learn macro-based language that supports custom properties specification. From the structural perspective, we proposed to apply a set of software engineering metrics on UML class and package diagrams that quantitatively measure important design quality attributes.

Additionally, we applied probabilistic model-checking on SysML activity diagrams, which enables a quantitative and qualitative verification of the underlying probabilistic behavior. To this end, we devised an algorithm that systematically maps activity into the input language of the probabilistic symbolic model-checker PRISM. This allows to generate a performance model for SysML activity diagrams in terms of Markov decision processes. We also augmented our approach in order to support the verification of timeliness-related properties. Therefore, we specified timing information on the diagram using a UML standard profile, namely MARTE [51] and employed Markov reward mechanism for the verification of time-related properties. Furthermore, we developed an innovative dedicated

language, namely activity calculus, which is to the best of our knowledge the first calculus of its kind that captures the essence of SysML activity diagrams endowed with probabilistic features. The devised calculus was then used in order to built the underlying semantic foundations of SysML activity diagrams in terms of Markov decision processes using operational semantics framework.

In order to verify the mathematical basis for our approach, we demonstrated formally the correctness of the proposed translation algorithm with respect to the devised SysML activity diagrams operational semantics. This guarantees that the properties verified on the generated PRISM model actually hold on the analyzed diagram. Accordingly, we formulated the soundness theorem based on a simulation pre-order upon Markov decision processes. The latter establishes a step-by-step unidirectional correspondence between the SysML activity diagrams semantics and the semantics of the resulting PRISM model generated by the translation algorithm. Thus, we also developed a formal syntax and semantics for the fragment of PRISM input language that has MDP semantics. The proof of soundness was derived using structural induction on the activity calculus syntax and the so-called weight function concept. Finally, in order to put into practice our V&V approach and enable its automation, we developed a software V&V tool implementing the aforementioned features that interfaces existing modeling environments.

This thesis comes up with an innovative approach to the analysis of systems engineering design models expressed using UML/SysML. Our approach enables the automation of the V&V process and it aims at conserving the usability of the visual notation provided by

the graphical modeling language. Additionally, it encompasses formal and rigorous reasoning. Having a flawless design enables other model-based activities such as automatic generation of test vectors and translation between design and implementation. The results of our research efforts have been rewarded with several publications [139,185–187,210] in international conferences and journals. We also submitted a journal paper [211] in a highly reputable journal and a book [212] for publication.

This thesis opens up some new research directions with solid prospectives. For instance, our activity calculus can be easily extended in order to support other features defined in SysML activity diagrams such as continuous behaviors. This can be done using a syntactic enrichment of the language and the modification of the semantic rules. In this case, Markov decision process model has to be extended with continuous-time features. This results in a new underlying model, namely the Continuous-Time Markov Decision Processes (CT-MDP). Furthermore, one can investigate the possibility to synergistically combine static analysis and metrics with probabilistic model-checking. Finally, it is also possible to apply the same approach that we propose in this thesis for SysML activity diagrams on other behavioral diagrams, such as sequence and state machine diagrams.

Bibliography

[1] A. Kossiakoff and W. N. Sweet. *Systems Engineering Principles and Practice.* John Wiley & Sons, 2003.

[2] Technical Board. Systems Engineering Handbook: A "What To" Guide For All SE Practitioners. Technical Report INCOSE-TP-2003-016-02, Version 2a, International Council on Systems Engineering, June 2004.

[3] Ph. Schnoebelen. The Verification of Probabilistic Lossy Channel Systems. In *Validation of Stochastic Systems - A Guide to Current Research*, volume 2925 of *Lecture Notes in Computer Science*, pages 445–465. Springer, Berlin / Heidelberg, 2004.

[4] D. Lehmann and M. Rabin. On the Advantage of Free Choice: A Symmetric and Fully Distributed Solution to the Dining Philosophers Problem (Extended Abstract). In *the Proceedings of the 8th Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 133–138. ACM, 1981.

[5] L. Benini, A. Bogliolo, G. Paleologo, and G. De Micheli. Policy Optimization for Dynamic Power Management. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 8(3):299–316, 2000.

[6] J. Tian. *Software Quality Engineering: Testing, Quality Assurance, and Quantifiable Improvement.* Wiley-IEEE Computer Society Press, 1st edition, 2005.

[7] B. W. Boehm and V. R. Basili. Software Defect Reduction Top 10 List. *IEEE Computer*, 34(1):135–137, 2001.

[8] Technical Board. Systems Engineering Handbook: A Guide for System Life Cycle Processes and Activities. Technical Report INCOSE-TP-2003-002-03, Version 3, International Council on Systems Engineering, June 2006.

[9] S. Friedenthal, A. Moore, and R. Steiner. *A Practical Guide to SysML: The Systems Modeling Language.* Elsevier Science & Technology Books, July 2008.

[10] Object Management Group. *OMG Systems Modeling Language (OMG SysML) Specification*, September 2007. OMG Available Specification.

[11] NASA ARC. V&V of Advanced Systems at NASA. Technical Report 10 TA-5.3.3 (WBS 1.4.4.5.3), National Aeronautics and Space Administration, January 2002.

[12] E. W. Dijkstra. Notes on Structured Programming. Circulated privately, April 1970.

[13] B. S. Blanchard and W. J. Fabrycky. *Systems Engineering and Analysis.* International Series in Industrial and Systems Engineering. Prentice Hall, 1981.

[14] J. O. Grady. *System Validation and Verification.* Systems Engineering Series. Boca Raton : CRC Press, 1998.

[15] H. Eisner. *Essentials of Project and Systems Engineering Management*. Wiley, New York, 2002.

[16] G. Norman and V. Shmatikov. Analysis of Probabilistic Contract Signing. *Journal of Computer Security*, 14(6):561–589, 2006.

[17] G. Brat and W. Visser. Combining Static Analysis and Model Checking for Software Analysis. In *the Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE)*, page 262, Washington, DC, USA, 2001. IEEE Computer Society.

[18] D. Binkley. The Application of Program Slicing to Regression Testing. In *Information and Software Technology Special Issue on Program Slicing*, pages 583–594. Elsevier, 1999.

[19] F. Tip. A Survey of Program Slicing Techniques. *Journal of Programming Languages*, 3:121–189, 1995.

[20] T. DeMarco. *Controlling Software Projects: Management, Measurement & Estimation*. Yourdon Inc.: Prentice Hall, 1982.

[21] Object Management Group. *OMG Unified Modeling Language: Superstructure 2.1.2*, November 2007.

[22] C. Bock. Systems Engineering in the Product Lifecycle. *International Journal of Product Development*, 2:123–137, 2005.

[23] International Council on Systems Engineering (INCOSE) Website. `http://www.incose.org/`. Last Visited: February 2010.

[24] R. Davis. Systems Engineering Experiences Growth as Emerging Discipline, November 2001. Engineering Times, National Society of Professional Engineers.

[25] International Organization for Standardization (ISO). *Systems and Software Engineering – System Life Cycle Processes*, 2008.

[26] Institute of Electrical and Electronics Engineers(IEEE). *IEEE Std. 1220-1998, IEEE Standard for Application and Management of the Systems Engineering Process*, 1998.

[27] Department of Defense. *DoD Modeling and Simulation (M&S) Verification, Validation and Accreditation (VV&A)*, May 2003.

[28] Navy Modeling and Simulation Management Office. Modeling and Simulation Verification, Validation, and Accreditation Implementation Handbook. Technical report, Department of the navy, U.S, March 2004.

[29] D.A Cook and J.M. Skinner. How to Perform Credible Verification, Validation, and Accreditation for Modeling and Simulation. *CrossTalk: The Journal of Defense Software Engineering*, 18:20–24, May 2005.

[30] C. S. Wasson. *System Analysis, Design, and Development: Concepts, Principles, and Practices*. Wiley Series in Systems Engineering and Management. Wiley-Interscience, Hoboken, NJ, 2006.

[31] Department of Defense, United States of America. *Data Administration Procedures*, March 1994.

[32] R. Shannon. Introduction to the Art and Science of Simulation. In *Proceedings of the 1998 Winter Simulation Conference*, volume 1, pages 7–14. IEEE, 1998.

[33] INCOSE. Systems Engineering Vision 2020. Technical Report TP-2004-004-02, International Council on Systems Engineering (INCOSE), September 2007.

[34] D. Pilone and N. Pitman. *UML 2.0 in a Nutshell*. O'Reilly, 2005.

[35] S. Ceri, P. Fraternali, and A. Bongio. Web Modeling Language (WebML): a Modeling Language for Designing Web Sites. *Computer Networks*, 33(1–6):137–157, 2000.

[36] D. M. Nicol. Special Issue on the Telecommunications Description Language. *SIGMETRICS Performance Evaluation Review*, 25(4):3, 1998.

[37] D. Agnew, L. J. M. Claesen, and R. Camposano, editors. *Computer Hardware Description Languages and their Applications*, volume A-32 of *IFIP Transactions*. North-Holland, 1993.

[38] Object Management Group. Unified Modeling Language. `http://www.uml.org/`.

[39] R. J. Mayer, M. K. Painter, and P. S. Dewitte. IDEF Family of Methods for Concurrent Engineering and Business Re-engineering Applications. Technical report, Knowledge-Based Systems, Inc., 1992.

[40] T. Weilkiens. *Systems Engineering with SysML/UML: Modeling, Analysis, Design.* Morgan Kaufmann Publishers Inc., 2008.

[41] Object Management Group. *UML for Systems Engineering, Request For Proposal key=ad/03-03-41*, March 2003.

[42] H.P. Hoffmann. UML 2.0-Based Systems Engineering Using a Model-Driven Development Approach. *CrossTalk: The Journal of Defense Software Engineering*, 18:17–22, November 2005.

[43] A. Hohl. HDL for System Design. In H. Schwärtzel and I. Mizin, editors, *Advanced Information Processing: Proceedings of a Joint Symposium Information Processing and Software Systems Design Automation*, pages 313–326. Springer, Berlin, Heidelberg, 1990.

[44] J. Long. Relationships Between Common Graphical Representations in Systems Engineering. Technical report, Vitech Corporation, 2002.

[45] L. Doyle and M. Pennotti. Systems Engineering Experience with UML on a Complex System. In *the Proceedings of the 3rd Annual Conference on Systems Engineering Research*, Department of Systems Engineering and Engineering Management, Stevens Institute of Technology, 2005.

[46] S. White, M. Cantor, S. Friedenthal, C. Kobryn, and B. Purves. Panel: Extending UML from Software to Systems Engineering. In *the Proceedings of the*

10th IEEE International Conference on Engineering of Computer-Based Systems (ECBS), pages 271–. IEEE Computer Society, April 2003.

[47] G. Booch. *Object-Oriented Analysis and Design with Applications (2nd Edition).* Addison Wesley Longman Publishing Co., Inc., Amsterdam, 2007.

[48] J. Rumbaugh, M. Blaha, W. Lorensen, F. Eddy, and W. Premerlani. *Object-Oriented Modeling and Design.* Prentice Hall, 1991.

[49] I. Jacobson, M. Christerson, and P. Jonsson. *Object-Oriented Software Engineering.* Addison-Wesley Professional, 1992.

[50] Object Management Group. *UML Profile for CORBA specification Version 1.0*, April 2002.

[51] Object Management Group. *A UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems, Beta 2*, June 2008. OMG Adopted Specification.

[52] Object Management Group. *OMG Unified Modeling Language: Superstructure Version 2.0*, March 2005.

[53] M. Broy. Semantik der UML 2.0, October 2004.

[54] Object Management Group. *Diagram Interchange*, April 2006. Version 1.0.

[55] Object Management Group. *OMG Unified Modeling Language: Infrastructure 2.1.2*, November 2007.

[56] Object Management Group. *Object Constraint Language (OCL)*, May 2006. Version 2.0.

[57] SysML Forum. SysML Forum-Frequently Asked Questions. `http://www.sysmlforum.com/FAQ.htm`. Last visited: December 2009.

[58] Defense Modeling and Simulation Office. Verification and Validation Techniques. `http://vva.dmso.mil/Ref_Docs/VVTechniques/VVtechniques-pr.pdf`, August 2001. Published as a Recommended Practices Guide (RPG).

[59] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 2nd edition, 2004.

[60] O. Grumberg and H. Veith. *25 Years of Model Checking: History, Achievements, Perspectives*. Springer Publishing Company, Incorporated, 2008.

[61] P.J. Pingree, E. Mikk, G.J. Holzmann, M.H. Smith, and D. Dams. Validation of Mission Critical Software Design and Implementation using Model Checking [Spacecraft]. In *the Proceedings of the 21st Digital Avionics Systems Conference*, volume 1, pages 1–12, October 2002.

[62] L. Fix. Fifteen Years of Formal Property Verification in Intel. In *25 Years of Model Checking: History, Achievements, Perspectives*, pages 139–144. Springer-Verlag, Berlin, Heidelberg, 2008.

[63] M. Y. Vardi. Branching vs. Linear Time: Final Showdown. In *the Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 1–22, London, UK, 2001. Springer-Verlag.

[64] E. M. Clarke and E. A. Emerson. Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic. In *25 Years of Model Checking*, volume 5000 of *Lecture Notes in Computer Science*, pages 196–215. Springer, 2008.

[65] D. Peled. Combining Partial Order Reductions with On-the-fly Model-Checking. In *the Proceedings of the 6th International Conference on Computer Aided Verification (CAV'94)*, pages 377–390, London, UK, 1994. Springer-Verlag.

[66] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *the Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 1–33, Washington, D.C., 1990. IEEE Computer Society Press.

[67] S. Ben-David, T. Heyman, O. Grumberg, and A. Schuster. Scalable Distributed On-the-Fly Symbolic Model Checking. In *Formal Methods in Computer-Aided Design*, pages 390–404. Springer-Verlag, 2000.

[68] G.J. Holzmann. The Model Checker SPIN. *IEEE Transaction on Software Engineering*, 23(5):279–295, May 1997. Special issue on Formal Methods in Software Practice.

[69] K.L. McMillan. The SMV System. Technical Report CMU-CS-92-131, Carnegie Mellon University, 1992.

[70] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *the Proceedings of the International Conference on Computer-Aided Verification (CAV)*, volume 2404 of *LNCS*, Copenhagen, Denmark, July 2002. Springer.

[71] K. Sen. VESTA: A Statistical Model-checker and Analyzer for Probabilistic Systems. `http://osl.cs.uiuc.edu/~ksen/vesta2/`. Last visited: September 2009.

[72] PRISM Team. PRISM - Probabilistic Symbolic Model Checker. `http://www.prismmodelchecker.org/index.php`. Last visited: September 2009.

[73] I. S. Zapreev. MRMC Home-page. `http://www.mrmc-tool.org/trac/`. Last visited: September 2009.

[74] F. Ciesinski and M. Größer. On Probabilistic Computation Tree Logic. In C. Baier, B. R. Haverkort, H. Hermanns, J.-P. Katoen, and M. Siegle, editors, *Validation of Stochastic Systems*, volume 2925 of *Lecture Notes in Computer Science*, pages 147–188. Springer, 2004.

[75] University of Birmingham. `http://www.bham.ac.uk/`. Last visited: January 2010.

[76] University of Oxford. `http://www.ox.ac.uk/`. Last visited: January 2010.

[77] M. Kwiatkowska, G. Norman, and D. Parker. Probabilistic Model Checking in Practice: Case Studies with PRISM. *ACM SIGMETRICS Performance Evaluation Review*, 32(4):16–21, March 2005.

[78] H.A. Oldenkamp. Probabilistic Model Checking-A Comparison of Tools. Master's thesis, University of Twenty, Netherlands, May 2007.

[79] R. Alur and T. A. Henzinger. Reactive Modules. *Formal Methods in System Design*, 15(1):7–48, 1999.

[80] M. Kwiatkowska, G. Norman, D. Parker, and M. Kattenbelt. PRISM - Probabilistic Symbolic Model Checker, July 2008. Last Visited: February 2009.

[81] PRISM Team. The PRISM Language - Semantics. Last Visited: March 2009.

[82] M. Kwiatkowska, G. Norman, and D. Parker. PRISM: Probabilistic Model Checking for Performance and Reliability Analysis. *SIGMETRICS Performance Evaluation Review*, 36(4):40–45, 2009.

[83] A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker. PRISM: A Tool for Automatic Verification of Probabilistic Systems. In H. Hermanns and J. Palsberg, editors, *the Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 3920 of *LNCS*, pages 441–444. Springer, 2006.

[84] J. Rutten, M. Kwiatkowska, G. Norman, and D. Parker. *Mathematical Techniques for Analyzing Concurrent and Probabilistic Systems,* P. Panangaden and F. van Breugel (eds.), volume 23 of *CRM Monograph Series*. American Mathematical Society, Panangaden, P. and Van Breugel, F. edition, 2004.

[85] J. Bahuguna, B. Ravindran, and K. M. Krishna. MDP-Based Active Localization for Multiple Robots. In *the Proceedings of the Fifth Annual IEEE Conference on Automation Science and Engineering (CASE)*, pages 635–640. IEEE Press, 2009.

[86] R. Haijema and J. Van der wal. An MDP Decomposition Approach for Traffic Control at Isolated Signalized Intersections. *Probability in the Engineering and Informational Sciences*, 22(4):587–602, 2008.

[87] Q. Hu and W. Yue. *Markov Decision Processes with their Applications* . Springer US, 2008.

[88] O. Constant, W. Monin, and S. Graf. A Model Transformation Tool for Performance Simulation of Complex UML Models. In *Companion of the 30th International Conference on Software Engineering*, pages 923–924, New York, NY, USA, 2008. ACM.

[89] V. Cortellessa, P. Pierini, R. Spalazzese, and A. Vianale. MOSES: Modeling Software and Platform Architecture in UML 2 for Simulation-Based Performance Analysis. In *the Proceedings of the 4th International Conference on Quality of Software-Architectures*, pages 86–102, Berlin, Heidelberg, 2008. Springer-Verlag.

[90] A. Kirshin, D. Dotan, and A. Hartman. A UML Simulator Based on a Generic Model Execution Engine. *Models in Software Engineering*, pages 324–326, 2007.

[91] Z. Hu and S. M. Shatz. Mapping UML Diagrams to a Petri Net Notation for System Simulation. In *the Proceedings of the Sixteenth International Conference on Software Engineering and Knowledge Engineering (SEKE'04)*, pages 213–219, 2004.

[92] M. Sano and T. Hikita. Dynamic Semantic Checking for UML Models in the IIOSS System. In *the proceedings of the International Symposium on Future Software Technology (ISFST)*, Xian, China, October 2004.

[93] A. Knapp, S. Merz, and C. Rauh. Model Checking - Timed UML State Machines and Collaborations. In *the Proceedings of the 7th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems, Oldenburg, Germany (FTRTFT'02)*, pages 395–414. Springer-Verlag, 2002.

[94] T. Schäfer, A. Knapp, and S. Merz. Model Checking UML State Machines and Collaborations. *Electronic Notes in Theoretical Computer Science*, 55(3):13, 2001.

[95] V. Del Bianco, L. Lavazza, and M. Mauri. Model Checking UML Specifications of Real-Time Software. In *the Proceedings of the Eighth International Conference on Engineering of Complex Computer Systems (ICECCS'02)*, page 203, Los Alamitos, CA, USA, 2002. IEEE Computer Society.

[96] S. Mazzini, D. Latella, and D. Viva. PRIDE: An Integrated Software Development Environment for Dependable Systems. In *the Proceedings of the Conference on*

Data Systems in Aerospace, Nice, France (DASIA'04). ESA Publications Division, 2004.

[97] R. Eshuis and R. Wieringa. Tool Support for Verifying UML Activity Diagrams. *IEEE Transactions on Software Engineering*, 30(7):437–447, 2004.

[98] A. Bondavalli, D. Latella, M. Dal Cin, and A. Pataricza. High-Level Integrated Design Environment for Dependability (HIDE). In *the Proceedings of the Fifth International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS'99)*, page 87, Washington, DC, USA, 1999. IEEE Computer Society.

[99] D. Latella, I. Majzik, and M. Massink. Automatic Verification of a Behavioural Subset of UML Statechart Diagrams Using the SPIN Model Checker. *Formal Aspects of Computing*, 11(6):637–664, 1999.

[100] E. Mikk, Y. Lakhnech, M. Siegel, and G. J. Holzmann. Implementing Statecharts in PROMELA/SPIN. In *the Proceedings of the Second IEEE Workshop on Industrial Strength Formal Specification Techniques (WIFT'98)*, page 90. IEEE Computer Society, 1998.

[101] S. Gnesi and F. Mazzanti. A Model Checking Verification Environment for UML Statecharts. In *the XLIII AICA Annual Conference, University of Udine*, October 2005.

[102] N. Guelfi and A. Mammar. A Formal Semantics of Timed Activity Diagrams and its PROMELA Translation. In *the Proceedings of the 12th Asia-Pacific Software Engineering Conference (APSEC'05) Taiwan*, pages 283–290. IEEE Computer Society, 2005.

[103] R. Eshuis. Symbolic Model Checking of UML Activity Diagrams. *ACM Transactions on Software Engineering Methodologies*, 15(1):1–38, 2006.

[104] M. E.Beato, M. Barrio-Solórzano, and C. E. Cuesta. UML Automatic Verification Tool (TABU). In *the Proceedings of the 12th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Specification and Verification of Component-Based Systems, Newport Beach, California, USA (SAVCBS'04)*. Department of Computer Science, Iowa State University, 2004.

[105] F. Mokhati, P. Gagnon, and M. Badri. Verifying UML Diagrams with Model Checking: A Rewriting Logic Based Approach. In *the Proceedings of the Seventh International Conference on Quality Software, (QSIC'07)*, pages 356–362, October 2007.

[106] D. Xu, H. Miao, and N. Philbert. Model Checking UML Activity Diagrams in FDR. In *the Proceedings of the ACIS International Conference on Computer and Information Science*, pages 1035–1040, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[107] P. S. Kaliappan, H. Koenig, and V. K. Kaliappan. Designing and Verifying Communication Protocols Using Model Driven Architecture and SPIN Model Checker.

In *the Proceedings of the International Conference on Computer Science and Software Engineering (CSSE'08)*, pages 227–230, Washington, DC, USA, 2008. IEEE Computer Society.

[108] G. Engels, C. Soltenborn, and H. Wehrheim. Analysis of UML Activities Using Dynamic Meta Modeling. In M. M. Bonsangue and E. B. Johnsen, editors, *the Proceedings of the International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, volume 4468 of *Lecture Notes in Computer Science*, pages 76–90. Springer, 2007.

[109] G. Csertan, G. Huszerl, I. Majzik, Z. Pap, A. Pataricza, and D. Varro. VIATRA: Visual Automated Transformations for Formal Verification and Validation of UML Models. In *the Proceedings of the 17th IEEE International Conference on Automated Software Engineering, Edinburgh, UK, (ASE'02)*, Septembre 2002.

[110] S. Gnesi and F. Mazzanti. Mu-UCTL: A Temporal Logic for UML Statecharts. Technical report, ISTI, 2004.

[111] S. Gnesi and F. Mazzanti. On-the-Fly Model Checking of Communicating UML State Machines. In IEE INSPEC, editor, *the Proceedings of the Second International Conference on Software Engineering Research and Applications (SERA'04), Los Angeles, CA, USA*, pages 331–338. Springer-Verlag, 2004.

[112] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a New Symbolic Model Checker. *International Journal on Software Tools for Technology Transfer*, 2:2000, 2000.

[113] Maude system. http://maude.cs.uiuc.edu/. Last Visited: January 2010.

[114] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 26(1):100–106, 1983.

[115] GRaphs for Object-Oriented VErification (GROOVE). http://groove.sourceforge.net/groove-index.html. Last Visited: January 2010.

[116] D. Latella, I. Majzik, and M. Massink. Towards a Formal Operational Semantics of UML Statechart Diagrams. In *the Proceedings of the Third International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, page 465. Kluwer, B.V., 1999.

[117] UPPAAL. http://www.uppaal.com/. Last Visited: January 2010.

[118] R. Grosu and S. A. Smolka. Safety-Liveness Semantics for UML 2.0 Sequence Diagrams. In *the Proceedings of the Fifth International Conference on Applications of Concurrency to System Design (ACSD'05)*, pages 6–14, Washington, DC, USA, June 2005. IEEE Computer Society.

[119] X. Li, Z. Liu, and J. He. A Formal Semantics of UML Sequence Diagrams. In *Proc. of Australian Software Engineering Conference (ASWEC'04)*, pages 13–16, Melbourne, Australia, April 2004. IEEE Computer Society.

[120] M. V. Cengarle and A. Knapp. UML 2.0 Interactions: Semantics and Refinement. In *the Proceeding of the 3rd International Workshop on Critical Systems Development with UML (CSDUML'04)*, pages 85–99. Technische Universität München, 2004.

[121] H. Störrle. Semantics of Interactions in UML 2.0. In *the Proceedings of the IEEE Symposium on Human Centric Computing Languages and Environments (HCC'03)*, pages 129–136. IEEE Computer Society, 2003.

[122] K. Korenblat and C. Priami. Extraction of PI-calculus Specifications from UML Sequence and State Diagrams. Technical Report DIT-03-007, Informatica e Telecomunicazioni, University of Trento, Trento, Italy, February 2003.

[123] R. Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, June 1999.

[124] S. K. Kim and D. A Carrington. A Formal V&V Framework for UML Models Based on Model Transformation Techniques. In *the Proceedings of the 2nd MoDeVa Workshop - Model Design and Validation*, INRIA, France, 2005.

[125] S. Bensalem, V. Ganesh, Y. Lakhnech, C. Mu noz, S. Owre, H. Rueß, J. Rushby, V. Rusu, H. Saïdi, N. Shankar, E. Singerman, and A. Tiwari. An Overview of SAL. In C. Michael Holloway, editor, *the Proceedings of the Fifth NASA Langley Formal Methods Workshop (LFM)*, pages 187–196, Hampton, VA, June 2000. NASA Langley Research Center.

[126] D. Miller. Higher-Order Logic Programming. In *the Proceedings of the Eighth International Conference on Logic Programming (ICLP)*, page 784, 1990.

[127] G. Smith. *The Object-Z Specification Language*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[128] I. Ober, S. Graf, and D. Lesens. Modeling and Validation of a Software Architecture for the Ariane-5 Launcher . In *Formal Methods for Open Object-Based Distributed Systems (FMOODS'06)*, volume 4037 of *Lecture Notes in Computer Science*, pages 48–62. Springer Berlin / Heidelberg, 2006.

[129] I. Ober, S. Graf, and I. Ober. Validating Timed UML Models by Simulation and Verification. In *the Workshop on Specification and Validation of UML Models for Real-Time and Embedded Systems (SVERTS'03), San Francisco*, October 2003.

[130] M. Bozga, J.C. Fernandez, L. Ghirvu, S. Graf, J.P. Krimm, and L. Mounier. IF: An Intermediate Representation and Validation Environment for Timed Asynchronous Systems. In *the World Congress on Formal Methods (1)*, pages 307–327. Springer-Verlag, 1999.

[131] M.L. Crane and J. Dingel. On the Semantics of UML State Machines: Categorization and Comparison. Technical Report 2005-501, School of Computing, Queen's University, 2005.

[132] H. Fecher, M. Kyas, and J. Schönborn. Semantic Issues in UML 2.0 State Machines. Technical Report 0507, Christian-Albrechts-Universität zu Kiel, 2005.

[133] X. Zhan and H. Miao. An Approach to Formalizing the Semantics of UML Statecharts. In *the Proceedings of the 23rd International Conference on Conceptual Modeling, Shanghai, China*, pages 753–765. Springer Berlin / Heidelberg, November 2004.

[134] A. Viehl, T. Schänwald, O. Bringmann, and W. Rosenstiel. Formal Performance Analysis and Simulation of UML/SysML Models for ESL Design. In *the Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, pages 242–247. European Design and Automation Assoc., 2006.

[135] R. Wang and C. H. Dagli. An Executable System Architecture Approach to Discrete Events System Modeling Using SysML in Conjunction with Colored Petri Net. In *the Proceedings of the 2nd Annual IEEE Systems Conference*, pages 1–8. IEEE, April 2008.

[136] E. Carneiro, P. Maciel, G. Callou, E. Tavares, and B. Nogueira. Mapping SysML State Machine Diagram to Time Petri Net for Analysis and Verification of Embedded Real-Time Systems with Energy Constraints. In *the Proceedings of the International Conference on Advances in Electronics and Micro-electronics (ENICS'08)*, pages 1–6, Washington, DC, USA, 2008. IEEE Computer Society.

[137] E. Huang, R. Ramamurthy, and L. F. McGinnis. System and Simulation Modeling Using SysML. In *the Proceedings of the 39th conference on Winter simulation (WSC'07)*, pages 796–803, Piscataway, NJ, USA, 2007. IEEE Press.

[138] C.J. J. Paredis and T. Johnson. Using OMG's SysML to Support Simulation. In *the Proceedings of the 40th Conference on Winter Simulation (WSC'08)*, pages 2350–2352. Winter Simulation Conference, 2008.

[139] Y. Jarraya, A. Soeanu, M. Debbabi, and F. Hassaïne. Automatic Verification and Performance Analysis of Time-Constrained SysML Activity Diagrams. In *the Proceedings of the 14th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS)*, pages 515–522. IEEE Computer Society, March 2007.

[140] G.C. Tugwell, J.D. Holt, C.J. Neill, and C.P. Jobling. Metrics for Full Systems Engineering Lifecycle Activities (MeFuSELA). In *Proceedings of the 9th International Symposium of the International Council on Systems Engineering (INCOSE 99)*, Brighton, U.K., 1999.

[141] NASA. Software Quality Metrics for Object-Oriented System Environments. Technical Report SATC-TR-95-1001, National Aeronautics and Space Administration, Goddard Space Flight Center, Greenbelt Maryland, June 1995.

[142] S. R. Chidamber and C. F. Kemerer. A Metrics Suite for Object-Oriented Design. *IEEE Transaction on Software Engineering*, 20(6):476–493, 1994.

[143] F. Brito, e Abreu, and W. Melo. Evaluating the Impact of Object-Oriented Design on Software Quality. In *the Proceedings of the Third International Software Metrics Symposium*, pages 90–99, 1996.

[144] W. Li and S. Henry. Object-Oriented Metrics that Predict Maintainability. *Journal of Systems and Software*, 23(2):111–122, 1993.

[145] M. Lorenz and J. Kidd. *Object-Oriented Software Metrics: a Practical Guide.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.

[146] R. C. Martin. OO Design Quality Metrics. `http://www.comp.nus.edu.sg/~bimlesh/oometrics/6/oodmetrc.pdf`, 1994.

[147] J. Bansiya and C. G. Davis. A Hierarchical Model for Object-Oriented Design Quality Assessment. *IEEE Trans. Softw. Eng.*, 28(1):4–17, 2002.

[148] L. C. Briand, P. T. Devanbu, and W. L. Melo. An Investigation into Coupling Measures for C++. In *the Proceedings of the 19th International Conference on Software Engineering (ICSE'97)*, pages 412–421, New York, NY, USA, 1997. ACM.

[149] M. Genero, M. Piattini, and C. Calero. Early Measures for UML Class Diagrams. *L'OBJET*, 6(4), 2000.

[150] R. Gronback. Model Validation: Applying Audits and Metrics to UML Models. In *Borland Developer Conference.* Borland Software Corporation, 2004.

[151] J. Hillston. Process Algebras for Quantitative Analysis. In *the Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 239–248, Washington, DC, USA, 2005. IEEE Computer Society.

[152] G. Bolch, S. Greiner, H. de Meer, and K. S. Trivedi. *Queueing Networks and Markov Chains : Modeling and Performance Evaluation with Computer Science Applications.* John Wiley & Sons, 2006.

[153] P. J. Haas. *Stochastic Petri Nets: Modelling, Stability, Simulation*. Operations Research and Financial Engineering. Springer-Verlag, New York, 2002.

[154] E. Wandeler, L. Thiele, M. Verhoef, and P. Lieverse. System Architecture Evaluation Using Modular Performance Analysis: A Case Study. *International Journal on Software Tools for Technology Transfer*, 8(6):649–667, 2006.

[155] S. Balsamo and M. Marzolla. Performance Evaluation of UML Software Architectures with Multiclass Queueing Network Models. In *the Proceedings of the 5th International Workshop on Software and Performance (WOSP)*, pages 37–42, New York, USA, 2005. ACM Press.

[156] V. Cortellessa and R. Mirandola. Deriving a Queueing Network-Based Performance Model from UML Diagrams. In *the Proceedings of the 2nd International Workshop on Software and Performance (WOSP)*, pages 58–70, New York, NY, USA, 2000. ACM Press.

[157] D. C. Petriu and H. Shen. Applying the UML Performance Profile: Graph Grammar-Based Derivation of LQN Models from UML Specifications. In *the Proceedings of the 12th International Conference on Computer Performance Evaluation, Modelling Techniques and Tools (TOOLS)*, pages 159–177, London, UK, 2002. Springer-Verlag.

[158] Object Management Group. *UML Profile for Schedulability, Performance and Time*, January 2005.

[159] P. J. B. King and R. Pooley. Derivation of Petri Net Performance Models from UML Specifications of Communications Software. In *the Proceedings of the 11th International Conference on Computer Performance Evaluation: Modelling Techniques and Tools (TOOLS)*, pages 262–276, London, UK, 2000. Springer-Verlag.

[160] J.P. López-Grao, J. Merseguer, and J. Campos. Performance Engineering Based on UML and SPN: A Software Performance Tool. In *the Proceedings of the Seventeenth International Symposium on Computer and Information Sciences*, pages 405–409. CRC Press, October 2002.

[161] J. Merseguer and J. Campos. Software Performance Modelling Using UML and Petri Nets. *Lecture Notes in Computer Science*, 2965:265–289, 2004.

[162] J. Trowitzsch, A. Zimmermann, and G. Hommel. Towards Quantitative Analysis of Real-Time UML Using Stochastic Petri Nets. In *the Proceedings of the 19th IEEE International Workshop on Parallel and Distributed Processing Symposium (IPDPS)*, page 139.b, Washington, DC, USA, 2005. IEEE Computer Society.

[163] J. Campos and J. Merseguer. On the Integration of UML and Petri Nets in Software Development. In *the Proceedings of the 27th International Conference on Applications and Theory of Petri Nets and Other Models of Concurrency (ICATPN), June 26-30*, volume 4024 of *Lecture Notes in Computer Science*, pages 19–36. Springer, 2006.

[164] M. Tribastone and S. Gilmore. Automatic Extraction of PEPA Performance Models from UML Activity Diagrams Annotated with the MARTE Profile. In *the Proceedings of the 7th International Workshop on Software and Performance (WOSP)*, pages 67–78, New York, NY, USA, 2008. ACM.

[165] M. Tribastone and S. Gilmore. Automatic Translation of UML Sequence Diagrams into PEPA Models. In *the Proceedings of the Fifth International Conference on Quantitative Evaluation of Systems September 2008 (QEST)*, pages 205–214. IEEE Press, 2008.

[166] A. Bennett and A. J. Field. Performance Engineering with the UML Profile for Schedulability, Performance and Time: a Case Study. In *the Proceedings of the 12th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS)*, pages 67–75, October 2004.

[167] C. Canevet, S. Gilmore, J. Hillston, L. Kloul, and P. Stevens. Analysing UML 2.0 Activity Diagrams in the Software Performance Engineering Process. In *the Proceedings of the Fourth International Workshop on Software and Performance*, pages 74–78, Redwood Shores, California, USA, January 2004. ACM Press.

[168] C. Canevet, S. Gilmore, J. Hillston, M. Prowse, and P. Stevens. Performance Modelling with the Unified Modelling Language and Stochastic Process Algebras. *the IEE Proceedings: Computers and Digital Techniques*, 150(2):107–120, March 2003.

[169] C. Lindemann, A. Thümmler, A. Klemm, M. Lohmann, and O. P. Waldhorst. Performance Analysis of Time-Enhanced UML Diagrams Based on Stochastic Processes.

In *the Proceedings of the 3rd International Workshop on Software and Performance (WOSP)*, pages 25–34, New York, NY, USA, 2002. ACM Press.

[170] R. Pooley. Using UML to Derive Stochastic Process Algebra Models. In Davies and Bradley, editors, *the Proceedings of the Fifteenth Performance Engineering Workshop, Department of Computer Science, The University of Bristol, UK*, pages 23–33, July 1999.

[171] N. Tabuchi, N. Sato, and H. Nakamura. Model-Driven Performance Analysis of UML Design Models Based on Stochastic Process Algebra. In *the Proceedings of the First European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA)*, volume 3748 of *Lecture Notes in Computer Science*, pages 41–58. Springer, November 2005.

[172] S. Gallotti, C. Ghezzi, R. Mirandola, and G. Tamburrelli. Quality Prediction of Service Compositions through Probabilistic Model Checking. In *the Proceedings of the 4th International Conference on Quality of Software-Architectures (QoSA'08)*, pages 119–134, Berlin, Heidelberg, 2008. Springer-Verlag.

[173] E. Börger, A. Cavarra, and E. Riccobene. An ASM Semantics for UML Activity Diagrams. In *the Proceedings of the 8th International Conference on Algebraic Methodology and Software Technology (AMAST)*, pages 293–308, London, UK, 2000. Springer-Verlag.

[174] J. P. López-Grao, J. Merseguer, and J. Campos. From UML Activity Diagrams to Stochastic Petri Nets: Application to Software Performance Engineering. *ACM SIGSOFT Software Engineering Notes*, 29(1):25–36, 2004.

[175] R. W. S. Rodrigues. Formalising UML Activity Diagrams Using Finite State Processes. Online Proceedings of UML 2000 Workshop on Dynamic Behaviour in UML Models: Semantic Questions, 2000.

[176] D. Yang and S. s. Zhang. Using π-Calculus to Formalize UML Activity Diagram for Business Process Modeling. In *the 10th International Conference and Workshop on the Engineering of Computer Based Systems (ECBS)*, pages 47–54, 2003.

[177] S. Sarstedt and W. Guttmann. An ASM Semantics of Token Flow in UML 2 Activity Diagrams. In *the Proceedings of the 6th International Andrei Ershov Memorial Conference, Perspectives of Systems Informatics (PSI)*, volume 4378 of *Lecture Notes in Computer Science*, pages 349–362. Springer, 2007.

[178] F. Scuglík. Relation Between UML 2 Activity Diagrams and CSP Algebra. *WSEAS Transactions on Computers*, 4(10):1234–1240, 2005.

[179] H. Störrle. Semantics of Control-Flow in UML 2.0 Activities. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 235–242. IEEE Computer Society, 2004.

[180] H. Storrle. Semantics of Exceptions in UML 2.0 Activities. Technical Report 0403, Ludwig-Maximilians-Universität München, Institut für Informatik, 2004.

[181] H. Störrle. Semantics and Verification of Data Flow in UML 2.0 Activities. *Electronic Notes in Theoretical Computer Science*, 127(4):35–52, 2005.

[182] B. Dénes and R. Heckel. Rule-Level Verification of Business Process Transformations using CSP. In Karsten Ehrig and Holger Giese, editors, *the Proceedings of the 6th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT)*, volume 6, pages 13–27, United Kingdom, May 2007.

[183] J. H. Hausmann. *Dynamic Meta Modeling: A Semantics Description Technique for Visual Modeling Languages*. PhD thesis, University Paderborn, 2005.

[184] H. Störrle and J. H. Hausmann. Towards a Formal Semantics of UML 2.0 Activities. In *Software Engineering 2005, Fachtagung des GI-Fachbereichs Softwaretechnik, 8.-11.3.2005 in Essen*, volume 64 of *Lecture Notes in Informatics*, pages 117–128. GI, 2005.

[185] L. Alawneh, M. Debbabi, F. Hassaïne, Y. Jarraya, P. Shahi, and A. Soeanu. Towards a Unified Paradigm for Verification and Validation of Systems Engineering Design Models. In *the Proceedings of the International Conference on Software Engineering*, pages 282–287. ACTA Press, February 2006.

[186] L. Alawneh, M. Debbabi, Y. Jarraya, A. Soeanu, and F. Hassaïne. A Unified Approach for Verification and Validation of Systems and Software Engineering Models. In *the Proceedings of the 13th Annual IEEE International Conference and Workshop on Engineering of Computer Based Systems (ECBS)*, pages 409–418. IEEE Computer Society, March 2006.

[187] Y. Jarraya, A. Soeanu, L. Alawneh, M. Debbabi, and F. Hassaine. Synergistic Verification and Validation of Systems and Software Engineering Models. *International Journal of General Systems*, 38(7):719–746, October 2009.

[188] M. G. Hinchey J. F. Monin. *Understanding Formal Methods*. Springer, 2003.

[189] J. R. Ruthruff, S. Elbaum, and G. Rothermel. Experimental Program Analysis: A New Program Analysis Paradigm. In *the Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'06)*, pages 49–60, New York, NY, USA, 2006. ACM Press.

[190] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[191] USAF Research Group. Object Oriented Model Metrics. Technical report, The United States Air Force Space and Warning Product-Line Systems, 1996.

[192] T. J. McCabe. A Complexity Measure. *IEEE Trans. Software Eng.*, SE-2:308–320, 1976.

[193] J. K. Hollingsworth. Critical Path Profiling of Message Passing and Shared-Memory Programs. *IEEE Transactions on Parallel and Distributed Systems*, 09(10):1029–1040, 1998.

[194] ARTiSAN Software. ARTiSAN Real-time Studio. `http://www.artisansoftwaretools.com/`. Last Visited: October 2009.

[195] A. Bianco and L. De Alfaro. Model Checking of Probabilistic and Nondeterministic Systems. In *Foundations of Software Technology and Theoretical Computer Science*, volume 1026 of *Lecture Notes in Computer Science*, pages 499–Ű513. Springer Berlin / Heidelberg, 1995.

[196] IBM. IBM Rational Software Delivery. `http://www-304.ibm.com/jct09002c/gsdod/solutiondetails.do?solution=27895&expand=true&lc=en`. Last Visited: October 2009.

[197] M. Kwiatkowska. Quantitative Verification: Models, Techniques and Tools. In *the Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 449–458. ACM Press, September 2007.

[198] M. Kwiatkowska, G. Norman, and D. Parker. Quantitative Analysis with the Probabilistic Model Checker PRISM. *Electronic Notes in Theoretical Computer Science*, 153(2):5–31, 2005.

[199] C. Baier. *On the Algorithmic Verification of Probabilistic Systems*. Habilitation, Universität Mannheim, 1998.

[200] D. Flater, P. A. Martin, and M. L. Crane. Rendering UML Activity Diagrams as Human-Readable Text. Technical Report NISTIR 7469, National Institute of Standards and Technology (NIST), November 2007.

[201] E. Best, R. Devillers, and M. Koutny. *Petri Net Algebra.* Springer-Verlag New York, Inc., New York, NY, USA, 2001.

[202] G. D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

[203] R. Harper. *Programming in Standard ML.* Online working draft, February 13, 2009.

[204] C. Baier, B. Haverkort, H. Hermanns, and J.-P. Katoen. Model-Checking Algorithms for Continuous-Time Markov Chains. *IEEE Transactions on Software Engineering*, 29(7):2003, 2003.

[205] R. Segala and N. A. Lynch. Probabilistic Simulations for Probabilistic Processes. In *the Proceedings of the Concurrency Theory (CONCUR'94)*, pages 481–496, London, UK, 1994. Springer-Verlag.

[206] C. Baier and M. Kwiatkowska. Domain Equations for Probabilistic Processes. *Mathematical Structures in Computer Science*, 10(6):665–717, 2000.

[207] M. Kattenbelt and M. Huth. Abstraction Framework for Markov Decision Processes and PCTL via Games. Technical Report RR-09-01, Oxford University Computing Laboratory, 2009.

[208] R. Milner. An Algebraic Definition of Simulation Between Programs. In *the Proceedings of the 2nd International Joint Conference on Artificial Intelligence (IJCAI), London, UK*, pages 481–489, San Francesco, CA, 1971. William Kaufmann.

[209] B. Jonsson and K. G. Larsen. Specification and Refinement of Probabilistic Processes. In *the Proceedings of the 6th Annual IEEE Symposium on Logic in Computer Science (LICS), Amsterdam, Holland*, pages 266–277. IEEE Computer Society, 1991.

[210] Y. Jarraya, M. Debbabi, and J. Bentahar. On the Meaning of SysML Activity Diagrams. In *the Proceedings of the 16th Annual IEEE International Conf. on the Engineering of Computer Based Systems (ECBS)*, pages 95–105, Los Alamitos, CA, USA, April 2009. IEEE Computer Society.

[211] Y. Jarraya, M. Debbabi, and J. Bentahar. Sound Probabilistic Model Checking of SysML Activity Diagrams. *ACM Transaction on Software Engineering Methodologies*, 2009. Submitted on February 2010.

[212] L. Alawneh, M. Debbabi, F. Hassaïne, Y. Jarraya, and A. Soeanu. *Verification and Validation in Systems Engineering: A practical approach for automating the assessment of design models expressed in UML and SysML.* Springer, 2010. Submitted for publication.