AUTOMATIC VERIFICATION OF BEHAVIORAL SPECIFICATIONS IN SOFTWARE INTENSIVE SYSTEMS

ANDREI SOEANU CAVAL

A THESIS
IN
THE DEPARTMENT
OF
ELECTRICAL ENGINEERING

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF MASTER OF APPLIED SCIENCE
CONCORDIA UNIVERSITY
MONTRÉAL, QUÉBEC, CANADA

MAY 2007

© ANDREI SOEANU CAVAL, 2007

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 978-0-494-34650-1
Our file Notre référence
ISBN: 978-0-494-34650-1

NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:
L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Abstract

Automatic Verification of Behavioral Specifications in Software Intensive Systems

Andrei Soeanu Caval

Modern systems tend to exhibit an ever increasing complexity especially due to their software design components and programmable aspects which are nowadays ubiquitous. Consequently, in order to assure reliable and dependable systems, sustained efforts are required in the process of system verification and validation. However, conventional verification and validation techniques that are primarily based on testing and simulation, while being helpful and useful, may lack in many cases the desired level of rigor and completeness and are generally costly, laborious and time consuming. In contrast, using verification techniques that are based on formal foundations, such as model-checking and program analysis in a complementary manner to the traditional verification techniques can provide an increased level of reliability and dependability. In this context, applying such techniques for verifying the correctness and validity of the engineered systems early in the design phase can greatly improve the quality and performance of the design. Moreover, using such a verification methodology can alleviate the high cost of maintaining the systems later in their development phases. Presently, modern system design can benefit from a wide range of development paradigms including those that are using techniques traditionally employed in software engineering such as the object oriented design paradigm. In order to standardize the process of system design and development, several modeling languages emerged in order to provide the means for capturing and modeling various system specifications and requirements. The Unified Modeling Language (UML) 2.0 and more recently the Systems Modeling Languages (SysML) represent the most prominent standardized modeling languages for software and systems engineering. In this setting, the research initiative that this work addresses, is introducing a unified paradigm for the verification and validation of software intensive systems engineering design models by using formal verification techniques that can be applied in order to assess different behavioral diagrams belonging to the aforementioned modeling languages.

Acknowledgments

This research effort was conducted under the distinct coordination of our research supervisor, Prof. Dr. M. Debbabi and it was supported by the Collaborative Capability Definition, Engineering and Management (Cap-DEM) project which is an R&D initiative within the Canadian Department of National Defence. It represents the result of a fruitful collaboration between the Computer Security Laboratory (CSL) at Concordia University and Defence Research and Development Canada (DRDC) at Ottawa. Also, a very special thanks is due to all the V&V team colleagues for all their help and support offered.

Contents

List of Figures viii

List of Tables ix

1 Foreword and Motivations 1

2 Introduction and Background 4

2.1 The Unified Modeling Language 5

2.2 Design Perspectives 7

2.3 Verification and Validation 8

2.4 Objectives 9

2.5 Structure of the Thesis 10

3 Methodologies for Design Verification 12

3.1 Verification Primer 12

3.2 Simulation Based Verification and Validation 13

3.2.1 Random Test Generation 14

3.2.2 Monitoring Internal Nodes 15

3.3 Reference Model Equivalence 15

3.4 Model Checking 16

3.5 Theorem Proving 17

4 V&V in Software and Systems Engineering 18

4.1 Verification and Validation of UML State Machines 18

4.2 Verification and Validation of UML Activity Diagrams 20

4.3 Structural Design Assessment 22

4.4 UML Verification Frameworks 22

5 Automatic Verification Approach **24**

5.1 Verification Techniques for Object Oriented Design 25

5.1.1 Software Engineering Techniques 26

5.1.2 Formal Verification Techniques 26

5.1.3 Program Analysis Techniques 27

5.2 Verification and Validation Framework 28

6 UML Models for System Behavior **32**

6.1 UML State Machine Diagram 34

6.1.1 State Machine Diagram Elements 35

6.1.2 Transitions 38

6.1.3 State Configurations 40

6.1.4 Run-to-Completion Step 40

6.2 UML Activity Diagram 41

6.2.1 Activity Actions 42

6.2.2 Activity Flows 42

6.2.3 Activity Building Blocks 43

6.2.4 Activity Diagram Execution 45

7 Verification and Validation of UML Behavioral Diagrams **47**

7.1 Configuration Transition System 48

7.2 Model Checking of Configuration Transition Systems 50

7.3 Property Specification using CTL 53

7.4 Program Analysis of Configuration Transition Systems 54

8 Verification and Validation of UML State Machine Diagram **57**

8.1 Derivation of the State Machine Diagram Semantic Model 58

8.2 Case Study 62

8.3 Program Analysis Example of the State Machine Diagram 68

9 Verification and Validation of UML Activity Diagram **71**

9.1 Derivation of the Activity Diagram Semantic Model 72

9.2 Activity Diagram Case Study 73

10 Performance Analysis of Time Constrained SysML Activity Diagrams **81**
10.1 Time Annotated SysML Activity Diagrams 82
10.2 Modeling Time Constrained Activity Diagrams 83
10.3 Mapping SysML Activity to DTMC 87
10.4 Performance Evaluation Case Study 88

11 Conclusion **94**

Bibliography **96**

List of Figures

2.1 UML 2.0 Diagram Taxonomy 6
5.1 Synoptic Overview of the Toolkit. 28
5.2 Architecture of the Framework. 30
6.1 State Machine Hierarchical Clustering 34
6.2 State Machine Components 35
6.3 Activity Control Flow Artifacts 44
6.4 Activity Control Flow Patterns 46
8.1 ATM State Machine Diagram Example 63
8.2 CTS of the ATM State Machine Example 65
8.3 Data Flow Sub-Graphs 68
8.4 Control Flow Sub-graph 69
9.1 ATM Activity Diagram Example 75
9.2 CTS of the ATM Cash Withdrawal Activity Diagram 77
9.3 Fixed ATM Activity Diagram Example 80
10.4 Probability and Execution Duration Annotation 83
10.5 Proposed Approach for Assesing SysML Activity Diagrams 85
10.6 Digital Camera SysML Activity Diagram Example 89
10.7 Reachability Graph of the Digital Camera Activity Diagram Example 91

List of Tables

7.1 CTL Syntax 54

7.2 CTL Modalities 54

8.3 State Machine Counter-Example 67

9.4 Activity Diagram Counter-Example for Property Number Two 79

9.5 Activity Diagram Counter-example for Property Number Four 79

Chapter 1

Foreword and Motivations

Factual evidence shows that the modern scientific process proved itself to be very useful in solving a very wide range of problems in various domains. However, though the quality of life seems to be constantly improving as a result of the scientific progress, one cannot fail to notice upon thorough examination that modern society is still in its early stages of understanding the human quest for scientific research. More precisely, we can readily notice that nowadays, while physical comfort is ubiquitous in many parts of the word, the minds of the people living their lives surrounded by technological innovations are under an ever increasing stress and pressure mainly generated by the same factors that contribute to or represent the byproduct of technological progress. In other words, the process of research and investigation has in general little concern about the internal mind environment associated with each individual that is interacting with various technologies.

In this setting, it becomes apparent that modern technological development necessitates a shift of focus towards those technical and engineering solutions that have the ability to enhance both the external and the internal environments of the individuals using them. Specifically, there is a clear

benefit in adopting more exhaustive design and development paradigms that have the potential to minimize and even eliminate the variance between the design intent and the actual design quality and performance, a problem that nowadays is encountered at different degrees in most of the engineering disciplines and especially in the engineering of software intensive systems.

We can readily see that in the context of maximizing the immediate economical gains that are associated with modern product development, many aspects such as ergonomics, different environmental concerns, thorough verification and validation or computational efficiency are frequently ignored, especially in those areas where the effects are not immediately apparent as is the case in software (e.g. operating systems, web browsers, office suites) or software intensive systems (e.g. computer and mobile networking, portable/wearable electronics and pervasive computing).

However, while we can notice an increased awareness about ergonomics and various environmental issues, the same can not be said about the necessity for thorough verification and validation of modern software and software intensive systems. Nowadays, this is usually done by means of simulation, which is very rarely thorough. Moreover, it is quite obvious that modern society is increasingly dependant on technologies based on software intensive systems that are presently accompanied by an ever increasing difficulty of assuring appropriate bug-free design models.

Thus, we make the case that an appropriate science for conscience philosophical setting should be present to some extent in any research and development initiative. Furthermore, it is not so difficult to acknowledge in this setting that the most precious resource that the human society is benefiting from is not represented by a physical material datum but is rather represented by the human resources themselves. As such, we argue that the more technology dependant the human

society becomes, a very important concern is to assure robust, bug free and high quality software and system design. Consequently, this can significantly contribute to the quality of the minds of the individuals using modern technologies and thus to the quality of the human society as a whole.

To that effect, the present research initiative that this work is concerned with, aims to introduce and explain a unified, automated and cost effective methodology for verification and validation of software intensive system design models. The emphasis will be on the assessment of behavioral design diagrams by means of model checking and program analysis as well as performance appraisal. Furthermore, this research represents a part of a larger project concerning the verification and validation of system engineering design models where it contributed to the module dedicated to the automated assessment of UML and SysML behavioral diagrams against specially devised intuitive macro-based specifications that can be systematically expanded to corresponding temporal logic properties. More precisely, the present research is concerned with the automated generation of the model-checker transition system and input code compilation along with the corresponding temporal logic properties to be verified. Moreover, it is addressing the performance assessment of SysML activity diagrams annotated with time constraints and probability artifacts.

Chapter 2

Introduction and Background

Nowadays, various forms of programming and computation are becoming ubiquitous in our immediate urban surroundings often times embedded in sensors, traffic and other driving related assistance, public advertisement, hotspots, smart elevators and many other micro-controller or CPU based systems. Moreover, wearable electronics like mobile phones, PDAs and the like are more popular than ever. In this context, we can notice that in modern engineering fields and especially in those related to software intensive systems, the solution space available to the designers and engineers is significantly enlarged due to the presence of the programmable aspect.

In this context, mere intuition and ingeniosity, though still playing a significant part, can hardly assure strong, flaw free and cohesive designs, especially after reaching an elevated level of complexity. Moreover, the programmable aspect allows for a broader specialization range of a given design and is usually inviting many engineers to benefit from design reuse. Hence, various aspects such as understandability, extensibility and modularity are becoming increasingly important

aspects of modern system design along with general requirements like cost and performance. Consequently, there is a clear trend toward the systematization and standardization of system design engineering practices and methodologies.

Systems engineering is defined by INCOSE [17] as an interdisciplinary approach that enables the realization of successful systems. Its application consists in the use of the "systems" approach in order to design complex systems such as Systems on-top-of Systems (SoS). Thus, systems engineers can use various approaches when specifying and designing their systems engineering models, including Model Driven Architecture Development [47] and Object Oriented (OO) techniques. In this context, UML [61] and SysML [28] emerged as the most prominent modeling languages.

2.1 The Unified Modeling Language

Modeling languages for software and systems engineering emerged in response to the continuous advancements in the field in order to allow for an abstract, high level description of a design and the components thereof. Thus it gives the designers the ability to successfully cope with an increased complexity.

Several object-oriented modeling languages were developed in the software engineering community in the early 1990s. However, as it is usually the case, these early initiatives were hardly satisfactory for the community, which needed a unified solution. Many modeling languages have been defined by different organizations, targeting domains such as SDL for telecommunications [48], SysML for hardware [50], and UML for software systems [40].

The envisioned objective of using modeling languages is to thoroughly and unambiguously

specify, visualize as well as to document system design models. Furthermore, unified modeling languages such as UML can be used as a basis for common communication channels that can help professionals to exchange their design models clearly and effectively. Another benefit consists in the potential to bridge different understanding or intuition gaps that generally exist in enlarged design teams. The impact can be more significant especially when used as a "de facto" standard for software intensive systems engineering.

Grady Booch of Rational Software Corporation [10] was one of the pioneers that in late 1994, contributed to one of the most expressive and rich modeling language, namely UML. In June 1996, UML 0.9 was released and the standard has progressed since then through several versions, culminating with a major revision of the standard, namely the newly adopted UML 2.0. Figure 2.1 depicts the diagram taxonomy of the UML 2.0 modeling language.

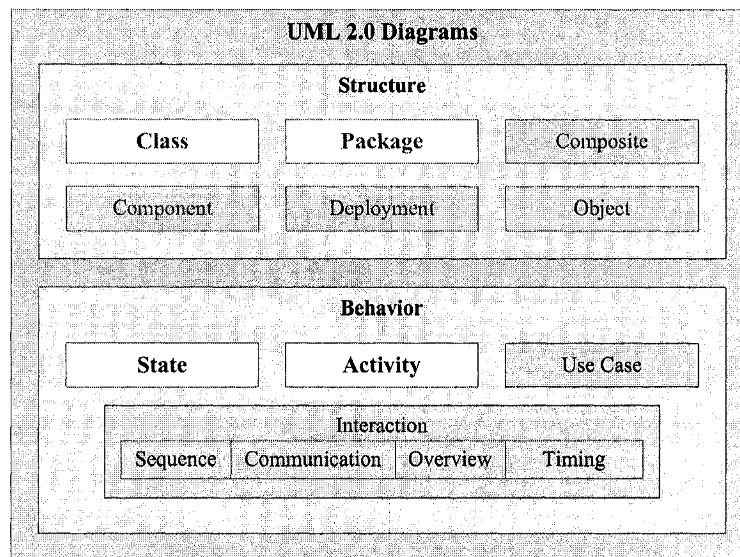

Figure 2.1: UML 2.0 Diagram Taxonomy

2.2 Design Perspectives

When modeling a system using languages like UML, the design can in general be viewed from two main perspectives, namely the structural description and the exhibited behavior. The former can be captured in a visual, diagrammatic notation by specifying the distinctive attributes of the system and its components along with their relation with respect to each other. The latter can be encoded by appropriate diagrams that can capture the dynamics of various state parameters in the system or its components as well as different internal or external interactions of the system. Both perspectives have an accompanying complexity degree that can be the subject of different processes of analysis and assessment. Moreover, structural analysis can be used in order to evaluate numerous quality attributes and may be used as a feedback in many tuning and optimization mechanisms. In contrast, the behavioral analysis is in general much more demanding and involves more rigor and preciseness. Furthermore, it is a known fact that a relatively short encoding of a behavioral model may have a very complex associated dynamic as it is the case with some forms of automata (e.g. cellular automata).

The programmable aspect of a system can be divided into configuration related programming (customization) and execution related programming. The latter is in general the one responsible for a highly dynamical system behavior. Thus, we may argue that the focus of various assessment procedures should be on the execution aspect of a design as it is in general more difficult to evaluate the behavioral aspects of a system than its architectural ones.

2.3 Verification and Validation

Verification can be understood as the process of evaluating a system in order to determine if the outcome of a given development phase is compliant to the initial conditions that were considered in the beginning of that phase [31]. The basic principle is related to the question of whether "we are building the product right" or not. In contrast, validation is concerned with the evaluation of a product or system in order to determine if it satisfies its specified requirements [31]. In this case, the emphasis is on the question of whether "we are building the right product" or not.

The V&V phase is nowadays a major bottleneck in the development life cycle of any complex software or systems engineering product and it was determined that it can represent from about a half and up to more than three quarters of the total design effort [32]. Moreover, many engineering solutions and especially those related to safety-critical areas, require a strong level of confidence with respect to their reliability, security, and performance. Thus, a challenging issue consists in assuring that both their performance and requirements are satisfied.

Presently, due to the increased complexity and intricacy inherent in software intensive systems design, traditional V&V methods involving testing and simulation are no longer the most adequate and have become increasingly time consuming and less useful. In contrast, formal automatic verification techniques like model checking can be used complementary to simulation in order to provide a strong degree of confidence in the quality and robustness of modern system design.

2.4 Objectives

The main objective of this thesis consists in proposing a unified methodology targeting the V&V process of software and systems engineering design models described in UML or SysML focusing on the assessment of the behavioral diagrams. Specifically, the target objectives are as follows:

- Investigate the state of the art approaches in the area of V&V in software and systems engineering.
- Elaborate a dedicated approach for the verification and validation of UML and SysML behavioral diagrams, namely state machine and activity diagrams by integrating the following two techniques in a synergetic manner:
  - Automatic formal verification,
  - Program analysis.
- Devise a methodology for assessing performance characteristics of systems engineering design models expressed as SysML activity diagrams annotated with timing and probability artifacts by using a probabilistic model checking technique.
- Prototype the approach into a framework for V&V of software and systems engineering design models.
- Conduct case studies in order to demonstrate the effectiveness and the viability of the approach.

2.5 Structure of the Thesis

The remainder of the thesis is organized as follows:

- Chapter 3 presents an outline of the existing verification methodologies in the field of software and systems engineering design. It starts with a verification primer followed by presenting an overview of various techniques including simulation, reference model equivalence, model checking and theorem proving.

- Chapter 4 provides a review of some relevant research initiatives in the field of verification and validation of software and systems engineering design models.

- Chapter 5 begins by introducing the appropriate techniques for the proposed approach. Thereafter, it presents the verification and validation framework embodying the proposed methodology.

- Chapter 6 presents the artifacts and specification of the targeted behavioral diagrams, namely the state machine diagram and the activity diagram.

- Chapter 7 introduces a unified semantic model that is appropriate for capturing behavioral models such as the state machine and the activity diagram and describes the procedure involved in applying the model checking technique. Furthermore, it discusses model property specification in the context of the CTL temporal logic. Subsequently it explains the use of program analysis techniques in order to tackle the state explosion problem that is characteristic to the model checking procedure.

- Chapter 8 describes the methodology involved in verifying the state machine diagram. It provides case study while demonstrating the benefit of using the synergy emerging from using program analysis techniques in the process of verification by means of model checking.
- Chapter 9 presents the methodology involved in verifying the activity diagram and provides as well a relevant case study.
- Chapter 10 describes the SysML activity diagram annotated with probability artifacts and extended with time constraints. Subsequently, it presents a transformation procedure to discrete time Markov chains that can be analyzed by a probabilistic model checker. Thereafter it discusses a performance evaluation case study of a SysML activity diagram design model capturing a functional aspect of a digital photo camera device.
- Chapter 11 presents the summarizing conclusions of the thesis.

Chapter 3

Methodologies for Design Verification

There is a continuous growth in the market for sophisticated systems encompassing complex software and electronics. These products are underpinned by the significant advances in software engineering, integrated circuit design and manufacturing technology. Design automation software has been an important enabling technology that helped designers to achieve rapid advances by automating much of the design process, bringing greater efficiency and productivity to system design. In this context, the process of design and development requires a sound and cost-effective verification and validation (V&V) phase. Generally, verification aims at assuring that we design the right system whereas validation asks whether we design the system the right way.

3.1 Verification Primer

Verification is typically performed after each step of the design process in order to ensure that each particular step has been performed correctly. Physical system design verification is done in

order to ensure that the design meets the manufacturing rules and it can be automated in most of the cases. However, verification of the other design steps requires further automation to keep the design development cycles under control.

Verification of the behavioral and logic design steps is currently relying heavily on the use of simulation tools, that are used to predict the functional response of the design or its components to specified inputs. The input values are typically specified manually by the designer or by the verification engineer and provided as input to the logic simulator together with the design description. The simulator's output, which is a prediction of the response of the system or component to the applied inputs, is then compared to the expected response in order to verify that the responses are consistent.

Design verification is performed after every step of the design process. The design process typically begins with an architectural design phase accompanied by a behavioral design specification using various modeling tools. Behavioral simulation tools can also be used during the behavioral design step. Currently, the design validation step is heavily based on simulation of user generated tests. However, formal verification is emerging as a way of supplementing simulation based design verification that currently spans a wide spectrum of techniques including brute-force, manual or random test generation [6].

3.2 Simulation Based Verification and Validation

Simulation is to a large extent the workhorse of today's design verification and validation methodologies. However, despite the increases in simulation speed and computer performance, simulation

is hardly able to keep up with the rapidly increasing complexity of system design in the last decade. For example, a large server ranch was used in order to simulate thousands of tests corresponding to a figure of 2 billion instruction cycles required to verify the 64 bit Sun UltraSparc™processor [45]. Moreover, the future prospects of simulation are not very encouraging as the number of simulation cycles required in the verification process is growing at an alarming rate, (e.g., from hundreds of million less than a decade ago to hundreds of billion for modern CPU based systems).

An additional issue with simulation based methodologies is that they require the time consuming step of creating the test inputs. The most commonly used test generation method relies primarily on the manual generation of test vectors. However, the complexity and laborious nature of manual test generation makes it highly impractical as the design size increases.

3.2.1 Random Test Generation

The huge search spaces associated with large designs have led to the adoption of random test generation. This approach has the typical undesired side effect of generating a very large number of inefficient test vectors and so results in very long simulation time even when directed or constraint driven random test generation techniques are used. For example, many months of simulation using large "computer farms" consisting of hundreds of workstations are typically required to validate today's microprocessor designs.

Pure random test generation has severe limitations. Despite the large number of patterns generated, the effectiveness in detecting design flaws is rather limited as typically a large percentage of the tests are not useful, especially in the absence of appropriate constrains that may even lead to

invalid test vectors. To address the problems associated with pure random test generation, weighting or biasing is generally used to attempt to constrain the test generator to interesting areas of the design space with the intention of attempting to cover the "corner cases". The problem with this technique is that the design verification engineer may not always know or be able to determine the direction in which to guide the test generator.

3.2.2 Monitoring Internal Nodes

Because of the very low ratio of system ports to internal nodes and interfaces, the ability to observe the internal system logic is typically very low and it can usually be increased by using monitors and assertions. Monitors inspect and log the state of internal nodes or interface signals, while assertions are "self-checking" monitors that assert the truth of certain properties and trigger warning or error messages if these properties are violated. The use of monitors and assertions can help in boosting the coverage achieved when generating tests. However, the construction of assertion checkers and monitors is still generally done manually and is labor intensive. The emergence of libraries containing application specific assertion monitors is providing some hope to alleviate this problem.

3.3 Reference Model Equivalence

A widely used formal verification technique is reference model equivalence checking, which allows two behavioral models to be compared. In general, one of the two is taken as the reference model and represents the so called "golden model". Model equivalence checking can work well

and is often used in the design process in order to verify the results of employing various design techniques and/or applying different optimization and tuning procedures. Model equivalence checking verifies that the behavior of two models is the same; it does not actually verify that the design is bug-free. In addition, when a variance is encountered, the error diagnosis capability of the model equivalence checking tools is in most of the cases rather limited and so it is difficult to determine the exact cause of the difference.

3.4 Model Checking

Model checking is an automated and thorough verification technique that can be used to check that the properties specified for a given design or its components are satisfied for all legal design inputs. Temporal logics allow the users to express the properties of the system over various trajectories (state paths). Model checking is primarily useful in verifying the control parts of a system, which represent the critical area of concern in most of the cases. It is in general impractical for thorough data flow analysis, since it suffers from the well known state explosion problem. In the worst case scenario, the state space of the design that must be explored can grow exponentially with the number of state variables. Model checking can also be performed on the components of a design but it requires in this case the specification of precise interfaces for those components so that only legal inputs are considered. However, a potential issue in this case consists in the fact that in practice, the interface specifications of the design component modules are subject to change during the design process.

Furthermore, in the absence of automated tools that can be used in order to easily specify the

design properties, the model checking technique is heavily dependent on experienced users that are able to properly encode the properties of the design or its components into temporal logic formulas. Thus, the need to rely on experienced design engineers with strong background in temporal logics has restricted the adoption of this technology. An additional issue consists in the limited ability to find a well suited metric that can be used to evaluate the design property coverage and thus, it is relatively difficult to determine if all design properties have been specified and verified. Notwithstanding, this issue may be circumvented by preforming appropriate requirements analysis especially in the cases where it is possible to express the requirements in clear manner by using specific diagrams belonging to modelling languages like UML or SysML.

3.5 Theorem Proving

Theorem proving involves verifying the truth of mathematical theorems that are postulated or inferred about the design, using a formal specification language [55]. The procedure followed when proving such theorems usually involves two main components, namely a proof checker and an inference engine. However, while the former can be completely automated in most of the cases, the latter may require occasional human guidance thus impeding the automation of the whole process. Moreover, there may be rare cases where due to the formalism involved (e.g. hidden circular references leading to a logical paradox), a given theorem conjecture cannot be either proven or disproven (refuted). The aforementioned issues represent some of the main reasons why presently, this technique is not widely adopted for performing verification and validation in system design.

Chapter 4

V&V in Software and Systems Engineering

In the following sections of this chapter, we present the most relevant research endeavors with respect to the verification and validation of software and systems engineering design models. The state of the art in this respect includes a significant number of initiatives that target the verification of various design models. However, for the present research material, we have focused on a number of work proposals that are concerned with the verification of UML or SysML behavioral diagrams like Activity and State machine.

4.1 Verification and Validation of UML State Machines

Several initiatives explored the V&V of the UML state machine diagram also known as state chart. The adopted approaches can be classified with respect to the model checker used. Some researchers have chosen SPIN [30] while others preferred SMV [36]. Latella et al. [41] as well as Mikk et al. [46] proposed a translation of a subset of UML state-charts to SPIN/PROMELA

using an operational semantics described in [42]. The translation procedure has two main steps. First, the state-chart is used to construct an equivalent Extended Hierarchical Automaton (EHA). Then, the generated EHA is modeled in PROMELA and subjected to model checking. Von der Beeck introduces in [60] a semantics for a subset of UML state machines based on the previously mentioned work of Latella et. al. The semantics for history states is defined in this work but neither join and fork transitions nor guards are considered. Moreover, the semantics focuses on communication and control while the employed transition priority scheme is different from the UML scheme. Moreover, the configuration is encoded directly at the syntactic level whereas the behavioral semantics is described by structural operational semantics rules. In [39], the author proposed a term rewriting based semantic model where the active states are encoded as terms but without a clear separation between the syntax and semantics. Also, the transitions are expressed as conditional rewriting rules requiring knowledge of the syntax of rewrite rules and terms. The approach is presented using the SMV model checker language. Furthermore, the work considers only simple state machines with no pseudostates. The semantic model employs a global step but the semantics of the interlevel transitions is not UML compliant.

A few approaches have been advanced for the assessment of the UML 2.0 state machine diagram. An attempt to define a structured operational semantics for UML 2.0 state machine is presented by Fecher et al. in [23] in terms of sets and relations. The work deals with major issues in UML 2.0 features including shallow and deep history, join, and fork pseudostates together with entry/exit actions. However, junction and choice pseudostates, completion event/transition, and communicating state machines are not considered. In a similar manner, Zhan et al. [62] present a

formalization based on the general purpose Z language. The semantic model is used to transform the state machine diagram into a so called Flattened REgular Expression (FREE) state model. The latter can be used in order to identify inconsistencies and incompleteness issues and also in the process of automatic test generation. The suggested approach is taking into account well-formed state machine diagrams covering simple and composite states, concurrent and non-concurrent sub-states, simple and compound transitions as well as solving transition firing priority. Nevertheless, pseudostates such as fork/join and history are not considered. The work of Damm et. al. [18] proposes a semantics for UML state machines, which departs in some respects from the informally defined UML 2.0 semantics. Thus, the run-to-completion step is defined by choosing one enabled transition and not of a set of transitions. Moreover, the semantics is defined on flat state machines. However, a translation from hierarchical to flat state machines is given at the expense of state explosion in the flattened state machine.

4.2 Verification and Validation of UML Activity Diagrams

Activity diagrams are very useful in order to capture business processes modeling and workflows. They are also suitable for specifying system behavior. Several approaches have been proposed for the V&V of activity diagrams. These approaches can be classified with respect to the derived semantic model. Some researchers such as Van der Aalst [58] and Ellis et al. [20] found that Petri nets are suitable for representing activity diagrams. Thus, activity diagrams capturing workflow systems can be described using interval timed colored Petri nets [34], which are ordinary Petri nets [52] extended with colored tokens required to model data and timing intervals for transitions.

Börger et al. [11] provide Abstract State Machine (ASM) semantics for activity diagrams. Eshuis et al. [22] use a mapping of activity diagrams to equivalent activity hypergraphs by flattening the structure of the former. Then, the activity hypergraph of a given diagram is mapped to a Clocked Labeled Kripke Structure (CLKS), which is an extension of Kripke systems with real variables.

More recent initiatives are addressing the formalization of UML 2.0 activity diagram. This category includes the work of Vitolins et al. [59], where a more formalized definition of UML 2.0 activity diagram semantics is introduced. This work is considering a subset of activity diagram, suitable for business process modeling. The semantics is based on the original token flow methodology, but using a more constructive approach that employs the concept of activity diagram virtual machine that can be constructed from any given activity diagram. In the context of V&V, the authors consider the defined virtual machine as a basis for UML activity diagram simulation engines.

In [13], Canevet et al. propose the analysis of UML 2.0 activity diagrams in the context of software performance analysis using the Performance Evaluation Process Algebra (PEPA) net models. The authors are detailing the mapping from UML activity diagrams to their corresponding PEPA net models. In order to apply performance analysis, these models are used to generate the corresponding continuous-time Markov chain (CTMC) model wherein an exponential delay is associated with each corresponding activity node.

4.3 Structural Design Assessment

Even though the present material is mainly focusing on the verification and validation of the behavioral modeling of the system design, we briefly mention some of the initiatives that address the assessment of structural design descriptions such as the UML Class and Package diagrams. In [26], the authors illustrate the use of several object oriented metrics to assess the complexity of class diagrams at the initial phases of the development life cycle. Also, Tugwell et al. [57] outline the importance of metrics in systems engineering especially related to complexity measurement. The BorCon 2004 proceedings [27], held under Borlands umbrella, address topics like validation by applying audits and metrics to UML models. Audits refer to conformance to standards while metrics are viewed as numerical measurements that allow the analysis of a model with respect to an already established scale that denotes a good design.

4.4 UML Verification Frameworks

In [44], Lilius et al. present the vUML tool that they developed the for automatic verification of UML state-chart diagrams. The intended use of vUML is to verify concurrent and distributed design models that contain active objects but it can also be used to verify sequential designs since it supports synchronous communication. The SPIN model checker is used to perform the verification while the state chart diagrams are modelled using PROMELA (the input language of SPIN). When an unsatisfied property is detected the tool can create a UML sequence diagram that shows the scenario that reproduces the error in the model.

The framework proposed by Guerra et al. [29] for the verification of UML models is building meta-models for UML diagrams and then translates them into a formalism that allows to verify their properties. The translation (denotational semantics) is described along with the operational semantics of the formalism by means of graph grammars.

Chapter 5

Automatic Verification Approach

Systems modeling languages such as UML 2.0 and SysML are used to compile the requirements in order to create executable specifications. These specifications model the system at a high-level of abstraction, thus helping in the verification of the correctness of a system. Moreover, specific diagrams are used to capture important system aspects such as:

- **Requirements**, which are a description of what a system should do. They are captured by using requirement diagrams in SysML or using sequence diagrams and use case diagrams in UML 2.0.

- **Concurrency**, which is an aspect that identifies how activities, events, and processes are composed (sequence, branching, alternative, parallel composition, etc.). It could be specified using sequence or activity diagrams.

- **Structure**, which is shown in class and composite structure diagrams. The class diagram shows the relationships between different classes of the system. The composite structure

diagram shows the internal structure of the building blocks of the system and how these blocks are interfacing to other components of the system.

- **Interface**, which identifies the shared boundaries of the different components of the system whereby the information is passed. This aspect is shown using class diagrams in UML 2.0 and SysML, composite structure diagrams in UML 2.0, and assembly diagrams in SysML.
- **Control**, which determines the order in which actions, states, events, and/or processes are arranged. It is captured using state machine, activity, and sequence diagrams in UML 2.0 and SysML.

5.1 Verification Techniques for Object Oriented Design

Object oriented design is characterized by its corresponding structural and behavioral perspectives. Thus, when analyzing a given design, both perspectives have to be assessed with appropriate techniques. Moreover, object oriented design models exhibit specific features such as modularity, hierarchical structure, inheritance, encapsulation, etc. These features are reflected in various related attributes such as complexity, understandability, reusability, maintainability, and others. Consequently, based on the evaluation of the aforementioned attributes, one can determine the quality of an object oriented system design. In this setting, empirical methodologies, such as those involving software metrics, can help assess the quality of the structural architecture of the design. Conversely, complementary automatic verification techniques based of formal methods such as model checking, can achieve a thorough behavioral assessment of the model (or the components

thereof) against a set of specified properties that capture the system's intended behavior. However, such exhaustive techniques are generally accompanied by corresponding scalability shortcomings (e.g., state explosion). In this context, techniques like program analysis, mainly data and control flow analysis have the potential to address, among other things, some of the scalability issues, thus allowing for an increased efficiency of the model checking procedure.

5.1.1 Software Engineering Techniques

A set of fifteen metrics from the software engineering field [1, 4, 43] can be used in order to assess quality attributes of various structural models. We found in the literature some support about the use of metrics in systems engineering. For instance, Tugwell et al. [57] outline the importance of metrics in systems engineering especially related to complexity measurement. In this context, a potential synergy can be achieved by applying the metric concept to behavioral specifications. Accordingly, in addition to applying metrics on the structural diagrams such as class diagram, they can also be applied on the semantic model that is derived from different behavioral diagrams. For example, cyclomatic complexity and length of critical path can be applied on the semantic model. Thus, the quality assessment of a design can combine both the static and dynamic perspectives.

5.1.2 Formal Verification Techniques

Formal verification techniques, such as model checking, establish a solid confidence for a reliable V&V. Moreover, model checking can be fully automated for design verification. In fact, it has been successfully used in the verification of real applications including digital circuits, communication

protocols, and digital controllers. It yields results much more quickly than theorem proving [37]. In order to use it, first we have to map the design to a formal model that is accepted by the model checking tool (semantic model, which is usually a kind of a transition system). Second, we have to express the properties that the design must satisfy in temporal logic formulas (derived from the specifications). Then, by providing these two ingredients to the model checker, the latter exhaustively explores the state space of the transition system during the verification stage and checks automatically if the specifications are satisfied. One of the benefits of many model checking tools (e.g. NuSMV, which we use) is that, if a specification is violated, a counterexample is produced.

5.1.3 Program Analysis Techniques

Program analysis techniques [49] are used to analyze software systems with the intent to collect or infer specific information about them in order to evaluate and verify system properties such as data dependencies, control dependencies, invariants, anomalous behaviors, reliability, and compliance to certain specifications. The information is useful for various software engineering activities such as testing, fault localization, and program understanding. There are two approaches in program analysis techniques that are static and dynamic analysis. Static analysis is performed before program execution while dynamic analysis focuses on a specific execution. The first is mainly used to determine the static aspects of a program and it can be used to check whether the implementation complies with the static aspects of the specification. The second is less reliable but can achieve greater precision by showing the presence of errors by dynamically verifying the behavior of a

program on a finite set of possible executions selected from a possibly infinite domain. Static program analysis techniques can also be used in order to slice (decompose) a program or a transition system into independent parts that can be then analyzed separately.

5.2 Verification and Validation Framework

Our verification framework is harmoniously and synergically combining software engineering techniques (metrics), automatic verification (model checking) and program analysis (static analysis) in order to perform V&V of systems engineering design models. Figure 5.1 outlines the synoptic overview of the V&V process. In what follows, we provide a more detailed description of our V&V approach and framework and show how the aforementioned techniques are not simply used together, but rather synergically combined.

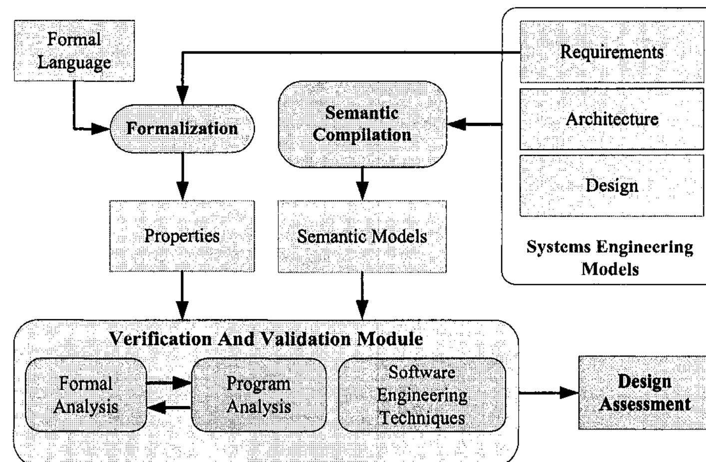

Figure 5.1: Synoptic Overview of the V&V Process

Model checking is a model-based verification technique that can be fully automated. It was

used for the verification of real applications, in both software and hardware fields. Examples of model checkers are SPIN [30], SMV [36], and NuSMV [15]. We selected the latter, a modified version of the SMV, since it supports fairness constrains along with branching-time logic for property specification, namely the Computation Tree Logic (CTL). Due to its interesting expressiveness, this temporal logic can capture many useful properties (e.g., deadlock, reachability, state sequencing etc.). Thus, in order to check whether the dynamic aspect of a model satisfies the specified properties, we have to extract the semantic model from the behavioral diagram that we wish to verify (e.g. state machine or activity). In practice, due to its typical scalability issues, model checking is generally limited to the verification of small and medium-sized complexity design models. Nonetheless, a number efforts are addressing the scalability issues in various ways, such as on-the-fly model checking [51], symbolic model checking [35], and distributed on-the-fly symbolic model checking [7]. In contrast to these techniques, our research makes use of model slicing based on flow analysis (data and control) as a complementary technique. The objective is to narrow the scope of the model checking on the part of the model that exhibits the dynamic subject of checking. Thus, we use static program analysis techniques in order to leverage the effectiveness of the model checking procedure by decomposing the transition system that is supplied to the model checker into independent parts that can be analyzed separately, thereby reducing the state space that has to be explored in the verification process.

Moreover, our V&V framework requires an underlying modeling tool wherefrom various models can be fetched and assessed. The current version of our framework is composed of three core components, as shown in Figure 5.2.

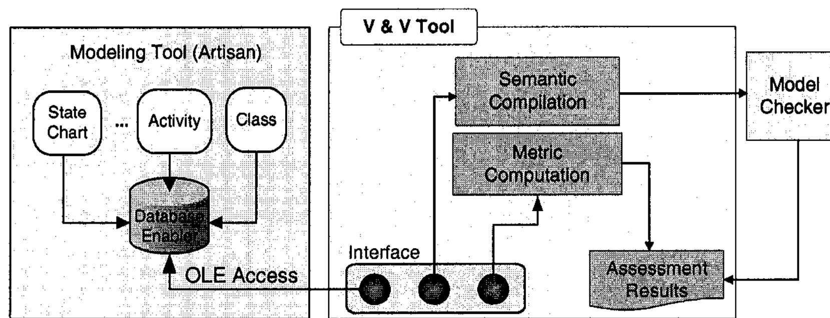

Figure 5.2: Architecture of the Framework.

The metric computation component is used for applying metric algorithms. We make use of an interface that accesses the object repository of the modeling tool and retrieves the needed information about the diagram to be assessed. According to the type of diagram, we can use a class of metrics for structural diagrams and another one for behavioral ones. In the literature, many metrics were developed to measure the quality of software systems, especially for structural diagrams namely class and package. We advocate the use of such metrics for the verification and validation of systems engineering design models.

The semantic compilation component is responsible for deriving the semantic model of a specific diagram. After converting a diagram into its corresponding semantic model, we can automatically specify CTL properties for deadlock and reachability. Manual specification of properties is also possible by using intuitive macros that can be automatically expanded to corresponding CTL properties. This allows the designers to easily express properties without being required to know formal logics or temporal formulas. Thus, the semantic compilation component is generating the semantic model along with the properties to be verified and encoding this information into the model checker input language. At this stage, the synergy is achieved by applying static analysis

(control and data flow) on the semantic model, before performing the model checking procedure.

The assessment results component is devoted to the presentation of interpreted results. When a property fails, the trace provided by the model-checker is analyzed and the relevant information is provided as feedback to the designer.

Chapter 6

UML Models for System Behavior

The UML modeling language is intended to have a rich expressive power in order to model complete systems across a broad spectrum of application domains. In many cases, such models require more than modeling software and computation as is the case with many designs of real-time and embedded software systems where it may be necessary to model the behavior of diverse real-world physical entities such as a hardware devices or human users. However, the physical components of a system tend in general to be highly heterogenous and much more complex than most mathematical formalisms can capture. Moreover, it is often the case that for a given entity, different viewpoints or perspectives related to different sets of concerns may be required in the process of modeling and design (e.g., dissimilar sets of concerns are considered in modeling the performance of the system when compared to modeling the same system from the view-point of human-machine interaction).

This may be one of the reasons behind the informal semantics of UML, as one of its primary objectives is to unify a set of broadly applicable modeling mechanisms in a common conceptual

framework, a task which any specific concrete mathematical formalism would likely restrict. That is, the commonality of UML is what underpins to use the same tools, techniques, knowledge, and experience in various domains and situations. Notwithstanding, the community is generally acknowledging that there is a compelling need for the formalization of many of the UML features [56]. In the context of modeling system behavior using UML, diagrams like state machine and activity are typically used in order to capture and specify the behavior of the system or its components. With respect to the features of the two diagrams, there are some similarities such as concurrent execution and synchronization, as well as important differences especially from the perspective of structure and hierarchy. Moreover, one can notice that while both of them have a pretty rich syntax and expressive power, in practice, the state machine diagram is especially useful when modeling reactive systems whereas the activity diagram is often used to describe workflow systems.

In general, a state machine description can be broadly conceived as the dynamical abstraction of a component instance in terms of its state attributes along with the evolution thereof in response to various stimuli. Conversely, the activity diagram purpose can be understood in providing the abstraction of a compound operation that usually involves many components that may interact in a complex but precise way.

In the next two sections, we will detail the features of each of the two aforementioned diagrams along with some appropriate comments regarding their specific usage and expressiveness.

6.1 UML State Machine Diagram

The primary motivation underlying the development of hierarchical state machines in general and their UML formalism in particular was the necessity to overcome the limitations of the conventional state machines in describing large and complex system behavior.

The UML state machines are hierarchical in nature while supporting orthogonal regions, (i.e., concurrency) and can be used to express the behavior of a system or its components in a visual, intuitive and compact manner. The state machine is evolving in response to set of possible incoming events (dispatched one at a time from an event queue) that can trigger the state machine transitions, which in turn, are in general associated with a series of actions (corresponding to the elements of the state machine) that are executed.

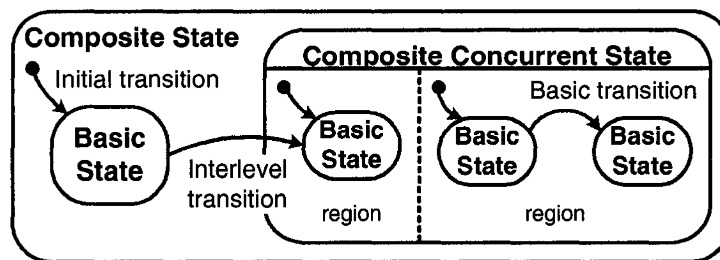

Figure 6.1: State Machine Hierarchical Clustering

Some key features of UML state-machines include the ability to cluster states into composite (OR) super-states and refine abstract states into sub-sates thus providing the hierarchical structure. Moreover, concurrency can be described by orthogonal (AND) composite concurrent states that contain two or more concurrently active regions, each of which is a further clustering of states as depicted in Figure 6.1. Hence, when a system is in an AND state, then each of its regions will

contain at least one active state. Furthermore, as there can be more than one state active at a time, the state machine dynamic 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.

6.1.1 State Machine Diagram Elements

State machines are basically a structured aggregation of states, transitions and a number of other pseudo state components. We can see the building blocks of a state machine in Figure 6.2. The states are either simple or composite (clustering a number of substates). Moreover, the states are nested in a containment hierarchy such that the states contained inside a region of a composite state are denoted as the substates of the composite state.

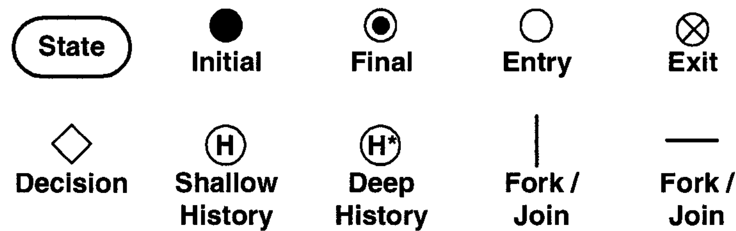

Figure 6.2: State Machine Components

Regions

A region is a placeholder in a composite state and contains the substates of that composite state. Moreover, every composite state may contain one or more regions and in such case, the different

regions of a composite state are separated by a dashed line. Each region of a composite state is said to be orthogonal to each other region of that composite state. Hence, concurrency is present whenever a composite state is containing more than one region.

States

Each state has a unique name or label and it may represent the source or target for a transition. States may also have *entry*, *exit* and *do* associated actions. Final states may only be a target of transitions and have no substates or associated actions and denote that the parent region is to be completed.

Simple or basic states represent the leaves of the state hierarchy tree and as such, they do not contain substates or regions. Conversely, the composite states are represented by all the other non-leaf nodes and as such, they have at least one region containing substates.

Moreover, whenever a composite state is active, all of its regions have exactly one nested substate which is also active. A single region composite state is called a non-orthogonal composite state or sequential (OR) state whereas a multi-region composite state is called an orthogonal (AND) composite state.

Pseudo States

The UML state machine formalism draws a distinction between states and pseudo states in that the latter ones are not included in configurations and their main function serves to model various forms of compound transitions. Moreover, pseudo states have no names or associated actions and are used as an abstraction for various types of intermediary vertices that are used to link different

states in the state machine tree hierarchy.

The following itemized list presents the different type of pseudo states:

- $initial$: It is used to indicate the default state of a region by way of only one outgoing transition to the target state. Moreover, at most one initial vertex can be present in each region of a composite state. The graphical representation of an $initial$ pseudo state is a small black disk.

- $fork$: It signifies that the incoming transition originates from a single state while having multiple outgoing transitions that occur concurrently, requiring the targets to be located in concurrent regions. Graphically, a thick black bar is used to represent the $fork$ pseudo state while the incoming transition and the outgoing transitions are touching the opposite sides of the thick black bar.

- $join$: It combines two or more transitions emanating form multiple states located in concurrent regions, into a compound synchronized transition with a single target state. The graphical representation of $join$ pseudo state is likewise a thick black bar, such that the incoming transitions and the outgoing transition are touching the opposite sides of the thick black bar.

- $shallowHistory$: It is used to represent the most recently active substate enclosed in the region of a composite state and corresponds to the state configuration that was active when the composite state was exited the last time. A single transition is allowed to indicate the default shallow history state for the case where the composite state has never been visited before.

The capital letter H enclosed in a circle is used to graphically represent the $shallowHistory$ pseudo state.

- $deepHistory$: Its use is an extension of $shallowHistory$ in that it is used represent the most recently active configuration enclosed in the region of a composite state and consequently it descends recursively into the most recently active substate until it reaches a basic active state. The graphical representation is similar with an additional "*" sign concatenated with the capital letter H.

6.1.2 Transitions

Transition are relating pairs of states and are used to indicate that a dynamic element (e.g. an object) is changing state in response to a trigger (event) provided that some specified condition (guard) is satisfied.

The guard is evaluated after the event is dispatched, but before the corresponding transition is fired. If the guard is evaluated to $true$, the transition is enabled; otherwise, it is disabled. Each transition allows for an optional action (e.g., issuing a new event) to be specified and if so, it is understood as the effect of the transition.

Transitions from composite states are called high-level or group transitions. When triggered, the state machine is also exiting all the substates of that composite state. Furthermore, a compound transition is an acyclic chain of transitions linked by various pseudo states and represents a path from a set (possibly a singleton) of source states to a set (possibly a singleton) of destination states. When the source and the destination sets are both singleton, the transition is said to be basic or

simple. Moreover, whenever the intersection of the source states belonging to two or more enabled transitions is not empty, the transitions are said to be in conflict. In this case, it is possible that one has higher priority than the other. Thus, in the case of simple transitions, the UML standard assigns higher priority to the transition having the most deeply nested source state.

The UML 2.0 specification is also indicating a number of transition constrains:

- The set of source states of a transition involving a join pseudo state is a set of at least two orthogonal states.
- A join vertex must have at least two incoming transitions and exactly one outgoing transition.
- All transitions incoming a join vertex must originate in different regions of an orthogonal state.
- A join segment must not have guards or triggers. The transitions entering a join vertex cannot have guards or triggers.
- A fork vertex must have at least two outgoing transitions and exactly one incoming transition.
- All target states of a fork vertex must belong to different regions of an orthogonal state.
- Transitions from fork pseudo-states may not target pseudo-states
- Transitions outgoing pseudo-states may not have a trigger.
- Initial and history transitions are restricted to point only to their default target state.
- Transitions from one region to another in the same immediate enclosing composite state are not allowed and require that the two regions must be part of two different composite states.

6.1.3 State Configurations

In the UML state machine, due to the containment hierarchy and concurrency, more than one state can be active at a time. If a simple substate of a composite state is active, then its parent and all the composite states that transitively contain the simple state are also active. Furthermore, some of the composite states in the hierarchy may be orthogonal (AND) and hence potentially concurrently active. Thus, the currently active states of the state machine are actually represented by a sub-tree of the state machine hierarchy tree. Consequently, the states contained in such a sub-tree are denoting a configuration of the state machine.

6.1.4 Run-to-Completion Step

The execution semantics of the UML state machine is described in the specification as a sequence of run-to-completion steps. Each step represents a move from an active configuration to another active configuration in response to the dispatched events, which are stored in an event pool. The UML 2.0 specification is silent about the kind of order imposed on the event pool, leaving it at the discretion of the modeler. However, the events are required to be dispatched and processed one at a time. Consequently, the run-to-completion means that an event can be popped from the event pool and dispatched only if the processing of the previously selected event finished. Thus, the processing of a single event in a run-to-completion step is achieved by firing the maximal set of enabled and non-conflicting transitions of the state machine. This results in a consistent change in the set of states that are currently active along with the execution of the corresponding actions if any, and assures that before and after each run-to-completion step the state machine is

in a stable active configuration. Furthermore, multiple transitions can be fired provided that they reside in mutually orthogonal regions while the order of firing can be arbitrary. In the case that an event is not triggering any transitions in a particular configuration (there is an empty set of enabled transitions), this amounts to an immediate completion of the run-to-completion step but with no configuration change. In this case the state machine is said to stutter and the event is discarded.

6.2 UML Activity Diagram

Activity diagrams are generally used in order to depict the flow of a given process by emphasizing the input/output dependencies, sequencing and other conditions (e.g. synchronizations) that are required for coordinating the process behavior.

Moreover, UML activity diagrams [53] can capture the behavior of a process or system using a control flow and data flow model that can be typically applied in a wide variety of domains such as computational, business and other workflow related systems in general.

UML activity diagrams are used to depict the sequencing of activities in a system from the start point to the termination point. Furthermore, an activity diagram may encompass various processing paths consisting in decision points and concurrent processing while the behavior of the system or component is focusing on activities or action states. Activity diagrams include several elements that show the behavior of a system using control flow modeling.

6.2.1 Activity Actions

Activity actions are used in order to specify the fine-grained behavior, similar to the kind of behavior corresponding to the executable instructions in ordinary programming languages. In essence, an action can be understood as the value transforming of a set of inputs into a set of outputs. Furthermore, for a given action, the inputs are specified by the set of incoming edges whereas the outputs are defined by the set of outgoing edges. However, when using elementary action nodes, only one incoming and one outgoing edge are likely to be specified.

The run-time effect of a given action can be described in terms of the difference in the state of the system from the pre-condition of the action to its post-condition. Thus, the pre-condition holds at the moment immediately before the action is executed whereas the post-condition holds at the instant just after the execution of the action completes. In this context, an important feature of the UML specification is that it makes no restrictions on the duration of the actions. Therefore, depending on the needs and constrains of the designer, both instantaneous (i.e. zero-time) semantic models as well as models allowing finite execution time can be employed.

6.2.2 Activity Flows

It is usually necessary to combine a number of transforming actions in order to get an overall desired effect. This is accomplished by using activity flows, which are used in order to combine actions that perform primitive (basic) state transformation. Thus, the activity flows represent the means of combining actions and their effects to produce complex transformations and serve to specify the conditions and execution order of the combined actions.

The UML standard supports both control and data (object) flows. In the present material we will concentrate on the control flow, as it represents one of our main targets of the verification and validation efforts of the present research. Control flows are the most conventional and natural way of combining actions. Accordingly, a control flow edge between two actions is understood as starting the execution of the action at the target end of the control flow edge immediately after the executing action at the source of the control flow edge has completed its execution.

In most of the cases, the processes described using activity diagrams require the alteration of the execution flow in various ways, requiring for example conditional branches, loops, etc. In order to support the aforementioned constructs, a number of special control nodes are used including standard control constructs such as *fork* and *join* that are used in a similar manner to those used in the state machine diagrams. Likewise, the control flow can be adjusted by specifying guards (in essence, side effect free boolean predicates) that label the flows, which are used for conditional transfer of control.

6.2.3 Activity Building Blocks

An activity diagram might contain object nodes for capturing corresponding object flows[1]. However, as the control flow aspect represents the point of interest in the behavioral assessment, we discuss in the following paragraphs the corresponding subset of control flow artifacts.

The building blocks of the activity diagram consists of the control flow elements depicted in

[1]Data (object) flows connect actions in a different manner than control flows and are used to connect input and output pins on actions while allowing certain actions to be started before finishing the activity wherefrom the dataflow control stems. Consequently, dataflow dependencies tend to be highly fine-grained when compared to the control flow, as the actions might have multiple inputs and outputs connected in sensitive and intricate ways.

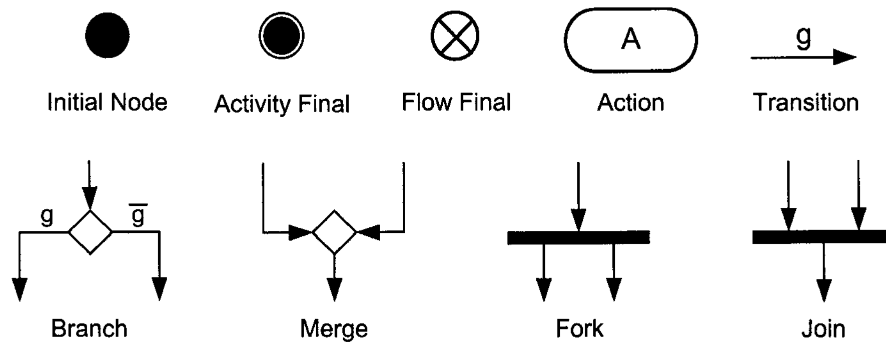

Figure 6.3: Activity Control Flow Artifacts

Figure 6.3:

- Initial, which indicates the beginning (e.g. initial entry point) of the activity diagram.
- Final node, which terminates the execution in the whole activity diagram.
- Flow Final, it stops the execution of actions in the specified flow only.
- Action node, which represents a processing node. Actions can be executed sequential or concurrently (as a result of forking). Furthermore, an action represents an elementary execution step that cannot be decomposed into smaller actions.
- Fork, it is used for concurrent processing in parallel execution paths.
- Join, it is used to synchronize different concurrent execution paths into one execution path.
- Transition, it is used to transfer the control flow in the diagram from an activity node to another. If specified, a guard can be used to control the firing of the transition.

- Branch, it is used to select between different execution flows in the diagram by choosing a specific execution path depending on the truth value of a guard.

- Merge, it is used to merge several alternative paths into a common activity flow.

An aggregation of actions nodes represents an executable block. An executable block is usually conceived as a structured activity node, which corresponds in a general sense to the concept of a block in some structured programming languages. Thus, a structured activity node generally contains several activity nodes and edges allowing for recursive structure definition. Therefore it is possible to construct arbitrarily complex activity node hierarchies.

6.2.4 Activity Diagram Execution

The execution semantics of the UML activity diagram is similar to Petri net token flow. In this respect, according to the specification, each activity action is started as soon it receives a control-flow token. Once an action is completed, subsequent actions receive a control flow token and are triggered immediately while the tokens are consumed by the activity node receiving the control flow.

There are several basic control-flow patterns defining elementary aspects of process control as depicted in Figure 6.4.

- Sequencing, it denotes the ability to execute a series of activities in sequence;

- Parallel split (fork connector), it denotes the ability to split a single thread of control into multiple threads of control which execute in parallel;

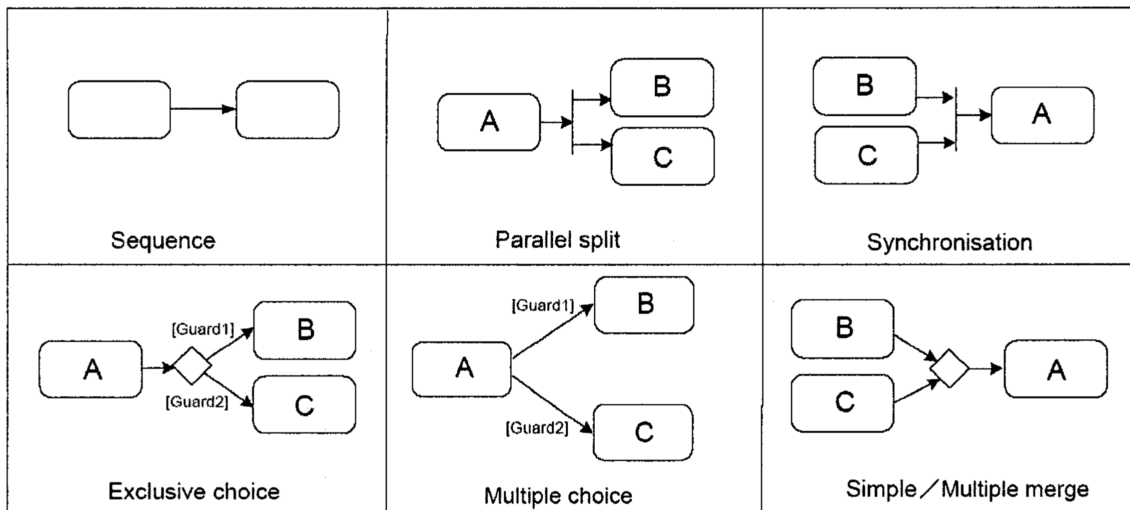

Figure 6.4: Activity Control Flow Patterns

- Synchronization (join connector), it denotes the convergence of multiple parallel subprocesses/activities into a single thread of control thus synchronizing multiple threads;
- Exclusive choice (branch connector), it denotes the ability to choose one of several branches at a decision point in a workflow process;
- Multiple choice, it denotes the ability to split a thread of control into several parallel threads on a selective basis;
- Simple Merge (merge connector), it denotes a point in the workflow process where two or more alternative branches come together without synchronization;
- Multiple merge (merge connector), it denotes the convergence of two or more distinct branches in an unsynchronized manner. If more than one branch is active, the activity following the merge is started for every activation of every incoming branch.

Chapter 7

Verification and Validation of UML Behavioral Diagrams

It is generally accepted that any system that exhibits a dynamics of some kind 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 set of states wherein the system abides at any particular moment. Hence, all the possible configurations summed up by the dynamics of the system and the transition thereof can be coalesced into a configuration transition system (CTS). The basic idea was explored in the work of Eshuis et al. [21] in the context of verifying UML 1.x activity diagrams. However, the CTS concept has a more general nature and can be conveniently adapted for a broad range of behavioral diagrams, including state machine and activity. In essence the CTS is basically a form of automaton and it is characterized by a set of configurations that include a (usually singleton) set of initial configurations and a transition relation

that encodes the dynamic evolution of the CTS from one configuration to another. Moreover, depending on the required level of abstraction, the CTS configuration structure may include more or less of the dynamic elements of the behavioral diagram. Thus, it is exhibiting an abstraction scalability trait that allows for efficient dynamic analysis by adjusting the scope to the desired parameters of interest.

7.1 Configuration Transition System

Given an instance of a behavioral diagram, we can find the corresponding CTS provided that the elements of the diagram are understood and there exist (and is defined) a step relation that enables one to systematically compute the next configuration(s) of the diagram from any given configuration. When the variables of interest, within the dynamic domain of a behavioral diagram, can be abstracted to boolean state variables, each of the enclosed configurations within the CTS can be represented by the set of states that are active[1] simultaneously. Furthermore, the transition relation of the CTS links configuration pairs by a label comprising all those variable values (e.g., events, guards) that are required to trigger the change from the current configuration to the next one. Also, a requirement needed in order to achieve tractability, is that the configuration space must be bounded. That is, we have to assume a finite countable limit for every variable within the dynamic domain of the diagram. Hereafter, we further detail the CTS concept.

Definition 1. (Dynamic domain) The set of heterogenous attributes that characterize the evolution of a behavioral diagram D represents the dynamic domain of that diagram, denoted with D_{Δ}.

[1]Though usually a $true$ boolean value denotes the active status of a state, the $false$ boolean value might similarly be used, as long as the convention is used consistently.

A configuration can be understood as a particular snapshot in the evolution of a set of dynamic elements of a system at a particular point in time and from a particular view.

Definition 2. (Configuration) For an established variable ordering, a configuration c is a particular binding of a set of values to the set of variables in the dynamic domain D_Δ of a behavioral diagram.

Given a diagram D and its corresponding dynamic domain D_Δ, if for every attribute $a_{1..n} \in D_\Delta$, we can find a corresponding positive integer $i_{1..n}$ so that for each projection of the dynamic domain $\pi_{a_k}(D_\Delta), k \in 1..n$, we have $max|\pi_{a_k}(D_\Delta)| < 2^{i_k}$, then a configuration c belonging to the CTS of D needs at most $I = \sum i_k$ bits, while the number of possible CTS configurations is at most 2^I. Notwithstanding, the actual number of configurations is usually much smaller and is restricted to the number of configurations reachable from the initial set of configurations. Moreover, the state attributes are in most of the cases confined to boolean values.

Definition 3. (Configuration Transition System) A CTS is a tuple (C,Λ,$\rightarrow$), where C is a set of configurations taken from the same view, Λ is a set of labels, and $\rightarrow \subseteq C \times \Lambda \times C$ is a ternary relation, called a transition relation. If c_1, $c_2 \in C$ and $l \in \Lambda$, the common representation of the transition relation is: $c_1 \xrightarrow{l} c_2$.

Since the dynamics for a particular diagram is captured by the corresponding CTS, it is then considered as the underlying semantic model. Consequently, the CTS can be used to systematically generate the model checker input.

Moreover, the CTS structure can also provide useful feedback to the designer. Thus, after

the generation of the CTS, it can be graphically visualized using a suitable graph editor such as daVinci [12]. The latter can be used in order to provide an overall visual appraisal of the diagram complexity with respect to the number of nodes and edges. It can also be used as a quick feedback when applying corrective measures giving some insights about the resulting increasing or decreasing of the diagram's behavioral complexity.

7.2 Model Checking of Configuration Transition Systems

The following paragraphs detail the back-end processing required for the model checking procedure of the CTS model. The chosen model checker is NuSMV [15], an improved version of the original SMV [36]. The way to encode a transition system in the NuSMV input language basically involves a grouping in at least three main syntactic declarative divisions[2] as follows. First we need a syntactic block wherein the state variables are defined along with their type and range. Secondly, we have to specify an initialization block, wherein the state variables are given their corresponding initial values or a range of possible initial values. Third, we have to describe the dynamics of the transition system in a so called *next* clause block, wherein the logic governing the evolution of the state variables is specified. Based on this, the state variables can be updated in every next step based on logical valuation done at the current step.

The CTS can be used to systematically and automatically generate its corresponding encoding into the model checker input language by constructing the three declarative divisions mentioned

[2]If appropriate, one might use various other convenient constructs while encoding a transition system in NuSMV as well as various levels of hierarchy where a main module is referring several other sub-modules due to the modular aspect that some particular transition systems might exhibit. However, this has no semantic impact with respect to the considered declarative divisions.

above. As presented in the foregoing section, the CTS dynamics is given in the form of pairwise configuration transition relations. Hence, any given CTS transition links a source configuration to a destination configuration. Consequently, it is conceivable that we might have the possibility to encode each configuration as a distinct entity in the NuSMV model. However, one can note that in a given CTS, the number of configurations may be significantly higher than the number of states that are members of different configurations. Also, the properties to be verified ought to be expressed on states and not on configurations. It follows that in order to encode the CTS representation into the model checker language in a compact and meaningful way, we need to use as dynamic entities, the configuration states rather than the configurations themselves. This will be reflected accordingly in all three declarative blocks.

Thus, after the establishment of the dynamic entities to be defined, one can proceed with the compilation of the three code blocks. The first one consists of enumerating the labels that are associated with each dynamic entity along with its type and range. The second one is compiled by using the initial configuration of the CTS in order to specify the initial values. The third one is more laborious in nature and consists in analyzing the CTS transitions in order to determine its state based evolution from its configuration based one.

More precisely, for every state s in any given destination configuration that is part of one or more transition relations of the CTS, we need to specify those conditions that are required for the activation of s, for each destination configuration while specifying that in the absence of such conditions, s would be deactivated. The aforementioned activation conditions can be expressed for every destination configuration as boolean predicates in the form of conjunctions over the active

status belonging to each state in the corresponding source configuration along with the test term for the transition trigger if it is the case. However, in the more general case where the source configuration elements might contain both multiple value and boolean state variables, the activation condition predicates would also include value test terms for the corresponding multivalued variables. Consequently, for each state[3] variable in the configurations of the CTS model, we have to specify what we would henceforth denote as transition candidates. Specifically, a candidate for each state s, represents the disjunctive combination over the activation conditions of all the destination configurations that have s as a member.

In mathematical terms, under the convention that the $true$ boolean value represents for each state its active status and given the structures:

- S, the set of all states in the CTS configurations;
- C, the set of all configurations in the CTS;
- Λ, the set of all trigger event labels in the CTS;
- $\rightarrow \ \subseteq (src : C \times lbl : \Lambda \times dst : C)$, the transition relation;
- $e \in \Lambda$, trigger event label.

we can determine the following:

$\forall t \in \rightarrow .c = \pi_{dst}(t).A_c = \bigwedge(\pi_{src}(t) \wedge e \equiv \pi_{lbl}(t))$

$\forall s \in S.\forall c \in \{C|\exists t \in \rightarrow .s \in C = \pi_{dst}(t)\}.A_s = \bigvee A_c$

where:

[3]In the presented context, a state should be understood as any boolean or multivalued variable that is part of one or more CTS configurations.

- A_c is the set of CTS configuration activation conditions;
- A_s is the activation condition set of the states in the CTS configurations.

Given that A_s contains the transition candidates for each state, we can use it in order to compile in the *next* clause block the corresponding evolution logic for each state in the CTS configurations. Thus, the dynamics of the CTS is encoded at state level by specifying that each state is activated at the next step whenever the transition candidate for a state is satisfied (true) in the current step and conversely deactivated if not.

7.3 Property Specification using CTL

The verification process by means of model checking requires the precise specification of properties in order to unfold the potential benefits of this technique. The NuSMV model checker uses the CTL (Computational Tree Logic) [19] temporal logic for this purpose. This logic has interesting features and great expressivity. It can be used to express general safety and liveness as well as more advanced properties like conditional reachability, deadlock freedom, sequencing, precedence, etc.

In the following paragraphs we briefly introduce the CTL logic and its operators. CTL is used to reason about computation trees that are unfolded from state transition graphs. CTL properties refer to the computation tree derived from the transition graph. The paths of a computation tree represent all possible computations of its corresponding model. Moreover, CTL is classified as a branching-time logic due to the fact that it has operators that describe properties on the branching structure of the computation tree.

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. The temporal operators are interpreted in the context of an implicit current state. 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). Table 7.1 presents the syntax of CTL formulas while Table 7.2 explains the underlaying meaning of the temporal modalities.

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

Table 7.1: CTL Syntax

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 that q holds and q must eventually hold

Table 7.2: CTL Modalities

7.4 Program Analysis of Configuration Transition Systems

In the following, we discuss the use of program analysis techniques (data and control flow), on our semantic model, namely the Configuration Transition System. This techniques can potentially improve the effectiveness of the model checking procedure by narrowing the scope of the verification to what we might call semantic projections of the transition system. Our goal is to identify

and extract those parts of the CTS that exhibit properties that can be used in order to simplify the transition system that is supplied to the model checker. The aspects that we are interested in are the data and control flow. The former is applied by basically searching for the presence of invariants (e.g. specific variable values or relations) whereas the latter can be used in order to detect control flow dependencies among various parts of the transition system. Consequently, the CTS may be sliced into smaller independent subgraphs that can be individually subjected to the model checking procedure.

Though it might be possible to specify some properties that could span across more than one subgraph of the original CTS, the slicing can be safely done under the following conditions:

1. The properties to be verified fall into liveness or safety category;

2. No property specification should involve sequences or execution traces that require the presence of the initial state more than once.

It must be noted that the second constraint does not represent a major hindrance for the verification potential. In this respect, the presence of invariants is assuring that revisiting the initial state or entering it for the first time is equivalent with respect to the dynamics of the transition system.

It must be mentioned however that even though some of the configuration subgraphs derived might be rather simple, it is nevertheless required for the model checking procedure that one specifies all the elements of the original model for each transition system that is fed to the model checker. This must be done in order to preserve the original elements of the transition system while assuring that the underlying dynamics is captured by the particular configuration subgraph in question. Moreover, due to the fact that the 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 automated and virtually transparent to the front-end of the verification framework.

In order to illustrate more effectively how data and control flow analysis can be applied on the CTS, we will provide an edifying example in chapter 8 that follows hereafter, where we detail the verification and validation procedure of the State machine diagram and show the application of program analysis techniques in section 8.2.

Chapter 8

Verification and Validation of UML State Machine Diagram

In this chapter, we describe the verification and validation procedure of the State machine diagram by means of model checking. A state machine is a specification that thoroughly describes all the possible behaviors of some dynamic model. The diagram representation contains hierarchically organized states that are related with transitions labeled with events and guards.

The state machine evolves in response to events that trigger the corresponding transitions provided that the source state is active, the transition has the highest priority, and the guard on the transition is true. If transitions have conflict, priorities are assigned to decide which transition will fire. Higher priority is assigned to those transitions whose source states are the deeper nested in the containment hierarchy.

8.1 Derivation of the State Machine Diagram Semantic Model

The hierarchical structure of the state machine diagram can be represented as a tree, where the root is the top state, the basic states are the leaves and all the other nodes are composite states. The tree structure can be used to identify the Least Common Ancestor (LCA) of a source and a target state of a transition. This is useful in identifying the states that will be deactivated and those who will be activated after firing a transition. An appropriate labeling (encoding) of the states is required in order to capture the hierarchical containment relation among them. That is, we have *"is ancestor of"/"is descendant of"* relations within the set of states. Moreover, each state down the hierarchy is labeled in the same manner as the table of contents of a book (e.g. 1., 1.1., 1.2., 1.2.1., ...). The labeling procedure consists in assigning Dewey positions and is presented in Algorithm 1, where the operator $+_c$ denotes string concatenation (with implicit operand type conversion). To that effect, the labeling is done by executing Algorithm 1 on top state with label "1." thus recursively labeling all states. The information encoded in the label of each state can be used to evaluate the relation among the states: For any two states s and z of a state machine where s_l and z_l represent their respective labeles, (s *"is ancestor of"* z) holds if s_l is a proper prefix of z_l (e.g. $s_l = "1.1.", z_l = "1.1.2."$). Conversely, ($s$ *"is descendant of"* z) holds if z_l is a proper prefix of s_l. The state labeling is used in order to find the LCA state of any pair of states under the top state[1] by identifying the common prefix. The later represents the label of the LCA state and can be more formally expressed in the following way:

For any pair of states (s, z), $s_l \neq "1." \neq z_l, \exists! lp \neq \epsilon$ such that lp is the greatest (longest) proper

[1]The LCA of any two states is the closest state in the containment hierarchy that is an ancestor of both of them.

prefix of both s_l and z_l. Consequently, $\exists! lcaState$ =LCA(s, z) such that $lcaState_l = lp$. While for any pair of states under the top, there is an unique LCA, it is also possible to have states that do not share *"is ancestor of"/"is descendant of"* relations (e.g. $s_l = "1.1.1."$, $z_l = "1.2.1."$).

Algorithm 1 Hierarchical State Labeling

$labelState$(State s, Label l)
  $s_l \leftarrow l$
  for all substate k in s **do**
    $labelState(k, l +_c indexof(k) +_c ".");$
  end for

A configuration is the set of states of the state machine where the true value is bound to active states and the false value to inactive ones. To avoid redundancy, for every configuration we only need to specify the states that are active. However, to support a mechanism whereby all the configurations of a state machine can be generated, we keep in each configuration two additional lists, one containing the value of all the guards for that particular configuration and the other containing a so called join pattern list for that configuration. The join pattern list terminology is borrowed from [24] and it is used to record various synchronization points that may be reached in the evolution of the state machine from one configuration to another.

In the following, we explain the procedure used for the generation of the CTS, presented by Algorithm 2. The CTS is obtained by a breadth-first search iterative procedure. The main idea consists in exploring for each iteration all the new configurations reachable from the current configuration, identified as *CurrentConf.* Moreover, three main lists are maintained. One denoted by *FoundConfList*, is recording the so far identified and explored configurations. The second one is holding the newly found but unexplored configurations and is denoted by *CTSConfList*. Finally,

the third list is used to record the identified transitions from one configuration to another and is denoted by *CTSTransList*. Additionally, we have a container list, *CTScontainer* that holds all the state and guard elements of the state machine along with an initially empty join pattern list placeholder.

The iterative procedure starts with *CTSConfList* containing only the initial configuration of the state machine, denoted by *initialConf*. In each iteration, a configuration is popped from *CTSConfList* and represents the value of *CurrentConf* for the current iteration. From *CurrentConf*, the three subsumed lists (*crtStateList*, *crtGList* and *crtJoinPatList*) are extracted. In order to be able to properly evaluate the value of the guards before firing the transitions, the *crtGList* is inspected to check if it contains an unspecified ("any") guard value. If so, then two new configurations are added to *CTSConfList* wherein the unspecified guard value is assigned the $true$ and $false$ value respectively and the next iteration immediately starts. Otherwise, if *FoundConfList* does not contain *CurrentConf* then the latter is added to *FoundConfList*.

Based on a list of possible incoming events referred to *EventList*, we pick each element one by one and dispatch it, each time restoring the state machine to the current configuration by setting the latter to the *CTScontainer*. The dispatching operation is a generic procedure that is responsible with the event processing and it is using the state containment hierarchy labeling in order to properly move the state machine from the current configuration to the next. Thus, the corresponding enabled transitions that are labeled with the dispatched event are triggered respecting their priorities and if a previously unidentified configuration is discovered, it is added to *CTSConfList*. Whenever the next found configuration is different from the current one, a new transition between *CurrentConf* and the next found configuration is formed and added to *CTSTransList* if not present. After adding all

Algorithm 2 Generation of the State Machine CTS

```
FoundConfList = ∅
CTSConfList = { initialConf }
CTSTransList = ∅
CTScontainer = {DiagramStateList,guardValueList,∅}
while CTSConfList is not empty do
  CurrentConf = pop(CTSConfList)
  crtStateList = get(CurrentConf,0)
  crtGList = get(CurrentConf,1)
  crtJoinPatList = get(CurrentConf,2)
  if crtGList containsValue "any" then
    splitIndex = getPosition(crtGList, "any")
    crtGList[splitIndex] = true
    CTSConfList = CTSConfList ∪ { crtStateList, crtGList, crtJoinPatList }
    crtGList[splitIndex] = false
    CTSConfList = CTSConfList ∪ { crtStateList, crtGList, crtJoinPatList }
    continue
  end if
  if FoundConfList not contains CurrentConf then
    FoundConfList = FoundConfList ∪ CurrentConf
  end if
  for each event e in EventList do
    setConf(CTScontainer, CurrentConf)
    dispatch(CTScontainer, e)
    nextConf = getConf(CTScontainer)
    if nextConf not equals CurrentConf then
      CTSConfList = CTSConfList ∪ nextConf
      crtTrans = {CurrentConf, e, nextConf}
      if CTSTransList not contains crtTrans then
        CTSTransList = CTSTransList ∪ {crtTrans}
      end if
    end if
  end for
end while
```

the new possible successor (next) configurations of the current configuration to the *CTSConfList*, the next iteration starts. The procedure stops when no elements can be found in *CTSConfList*. Thus, by applying the foregoing algorithm, we obtain the CTS corresponding to a given state machine.

8.2 Case Study

In the following, we present a case study related to a UML 2.0-based design describing an Automated Teller Machine (ATM). We perform V&V of the design with respect to predefined properties and requirements. We present hereafter the behavioral view of the design captured by a state machine diagram.

The ATM interacts with a potential customer (user) via a specific interface and communicates with the bank over an appropriate communication link. A user that requests a service from the ATM has to insert an ATM card and enter a personal identification number (PIN). Both pieces of information need to be sent to the bank for validation. If the credentials of the customer are not valid, the card will be ejected. Otherwise, the customer will be able to perform one or more transactions (e.g. cash advance or bill payment). The card will be retained in the machine during the customer interaction until the customer wishes no further service. Figure 8.1 shows the UML 2.0 state machine diagram of the Automated Teller Machine (ATM) system. The model is based on a hypothetical behavior and is meant only as an example. It is a syntactically-enriched extension of a similar example that was presented in one of our publications [4]. We intendedly modeled some flaws in the design to outline the success of our approach in discovering problems in the behavioral model. The diagram has several states that are going to be presented in accordance to the diagram

containment hierarchy. The top container state ATM encloses four substates: IDLE, VERIFY, EJECT and OPERATION. The IDLE state, wherein the system waits for a potential ATM user, is the default initial substate of the top state. The VERIFY state represents the verification of the card validness and authorization. The EJECT state depicts the phase of termination of the user transaction. The OPERATION state is a composite state that includes the states that capture several functions related to banking operations. These are the SELACCOUNT, PAYMENT and TRANSAC.

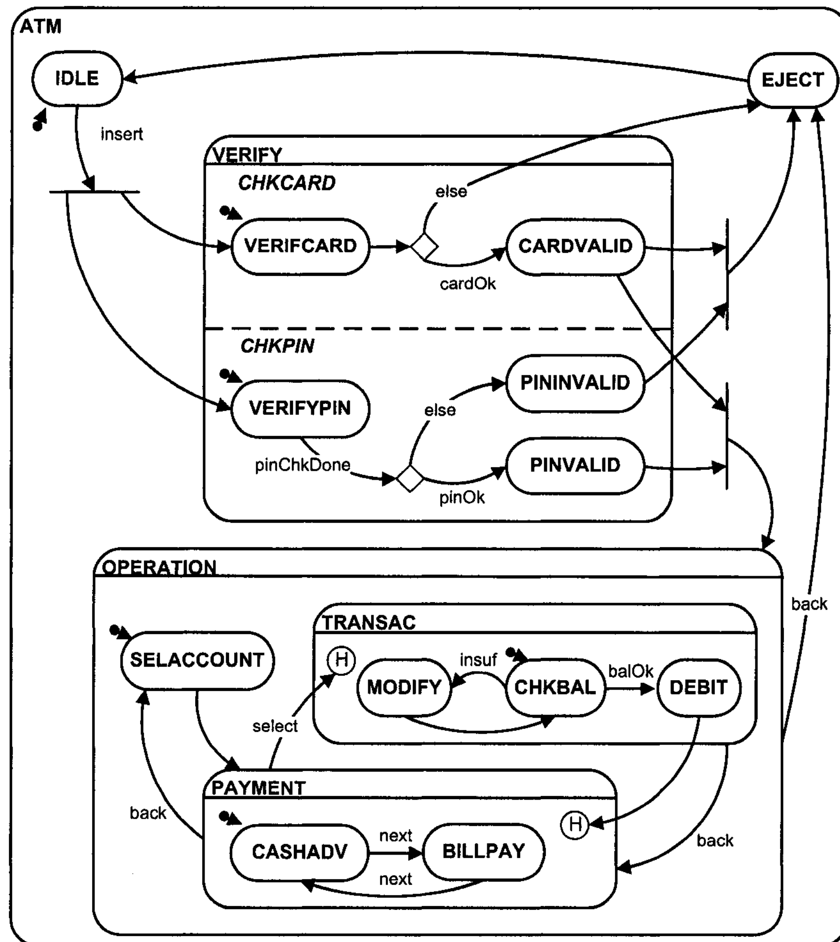

Figure 8.1: ATM State Machine Diagram Example

The `SELACCOUNT` state is where an account, belonging to the proprietor 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 `PAYMENT` state 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 for checking the balance (`CHKBAL`), modifying the amount if necessary (`MODIFY`) and debiting the account (`DEBIT`) respectively. Each one of the states `PAYMENT` and `TRANSAC` contains a shallow history pseudostate. If a transition targeting a shallow history pseudostate is fired, the activated state is the most recent active substate in the composite state containing the history connector.

When applying formal analysis to assess the presented state machine diagram, the steps are as follows. We first convert the diagram to its corresponding semantic model (CTS), as depicted in Figure 8.2. Each element is represented by a set (possibly singleton) of states and variable values of the state machine diagram. Thereafter, we automatically specify deadlock and reachability properties for every state. Furthermore, we also provide user defined specification in both macro and CTL notation.

After finishing the model checking procedure, the results that have been obtained pinpointed some interesting problems in the ATM state machine design.

The model checker determined that the `OPERATION` state exhibits deadlock, meaning that once entered, it is never left. This was found to be caused by the fact that in UML state machine diagrams, the transitions that have the same trigger are given higher priority when the source state is deeper in the containment hierarchy. Moreover, the transitions that have no event are fired as

soon as the state machine reaches a stable configuration that is containing the corresponding source state. This is precisely the case with the transition from `SELACCOUNT` to `PAYMENT`. Thus, there is no configuration that allows the operation state to be exited. This can also be seen by looking at the corresponding configuration system where we can notice that once a configuration that contains the `operation` state is reached, there is no transition to a configuration that does not contain it.

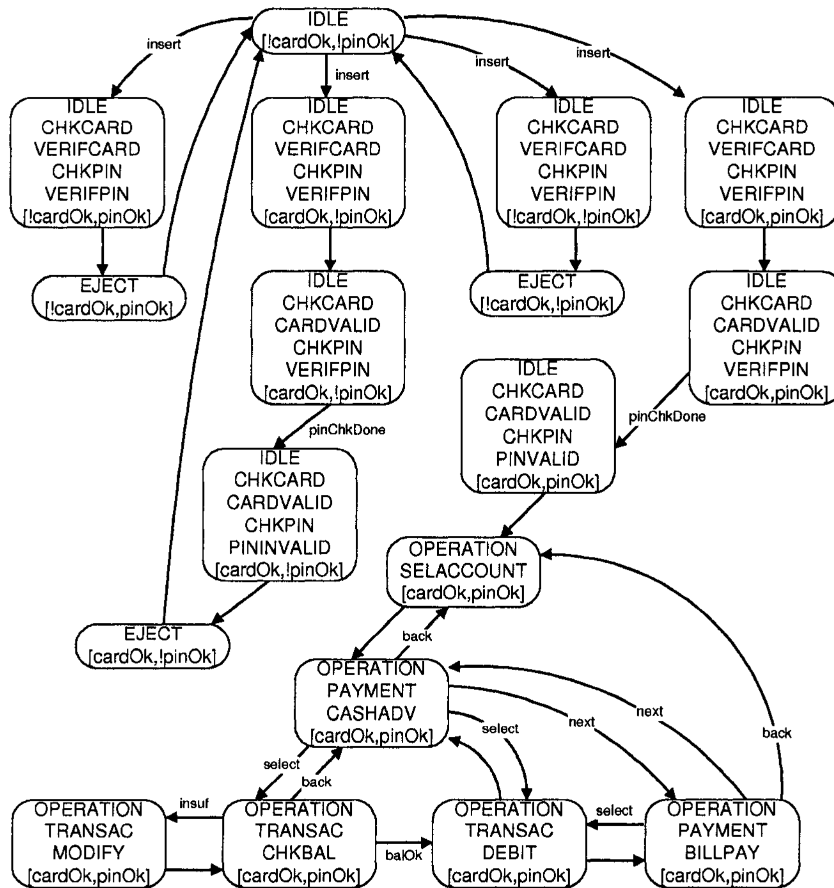

Figure 8.2: CTS of the ATM State Machine Example

In addition to the automatically generated properties, we present some relevant user-defined

properties described in both macro and CTL notations.

The first one asserts that it is always the case that if `VERIFY` state is reached then from that point, `OPERATION` state should be reachable:

`ALWAYS VERIFY` → `MAYREACH OPERATION`

CTL: `AG((VERIFY` → `(E[!(IDLE) U OPERATION])))`

The next property asserts that it is always the case that after reaching state `OPERATION` it should be inevitable to reach state `EJECT` at a later point:

`ALWAYS OPERATION` → `INEVIT EJECT`

CTL: `AG((OPERATION` → `(A[!(IDLE) U EJECT])))`

The last one states that `CHKBAL` must precede state `DEBIT`:

`CHKBAL PRECEDE DEBIT`

CTL: `(!E[!(CHKBAL) U DEBIT])))`

The first manually input specification turned out to be satisfied when running the model checker. However, the last two properties failed. The first failed property was not unexpected as, from the automatic specifications, we noticed that state `operation` is never left once entered (it exhibits deadlock) and does not have state `eject` as a substate.

The failure of the last property was accompanied by a trace provided by the model checker, depicted by Table 8.3. 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.

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].

Table 8.3: State Machine Counter-Example

The foregoing counterexample is represented by a series of configurations (semicolon separated). Moreover, whenever two or more states are present in a given configuration, a comma is separating them in the notation. Additionally, we have for each configuration the variable values enclosed in square brackets.

The failure in this case is due to the presence of a transition from 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 can help the designer to infer the necessary changes that will fix the identified problems. The first modification consists in adding a trigger such as `select` to the transition from the state `SELACCOUNT` to the state `PAYMENT`. This will fix the deadlock problem and the second user-defined property. The second modification corrects the problem related to the last unsatisfied specification. It consists in removing the history connector of the state `TRANSAC` and changing the incoming transition from this target directly to the state `TRANSAC`. After re-executing the V&V process for the fixed diagram, all the specifications, both automatic and user-defined properties were satisfied.

8.3 Program Analysis Example of the State Machine Diagram

We will use the configuration transition system of the state machine case study presented in Section 8.2. For the corresponding CTS, presented in Figure 8.2, in every configuration there are various values for the variables `cardOk` and `pinOk`. Whenever we have an exclamation mark preceding a variable in a particular configuration, the meaning is that the variable is false in that configuration.

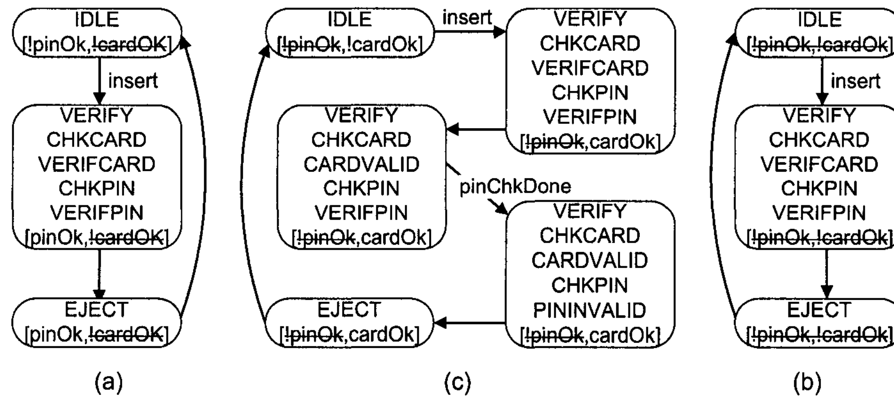

Figure 8.3: Data Flow Sub-Graphs

There are several subgraphs where some invariants hold. Figure 8.3 presents these subgraphs, each having invariants that can be abstracted. In Figure 8.3.a, we have a subgraph where the invariant `!cardOk` can be noticed. Similarly, Figure 8.3.b, shows another subgraph where the invariant `!pinOk` holds. In the subgraph shown by Figure 8.3.c, we have both `!cardOk` and `!pinOk` invariants. Additionally, Figure 8.4 depicts a subgraph that is independent from the control flow perspective. To that effect, once the control is transferred to this subgraph, it is never transferred outside of it.

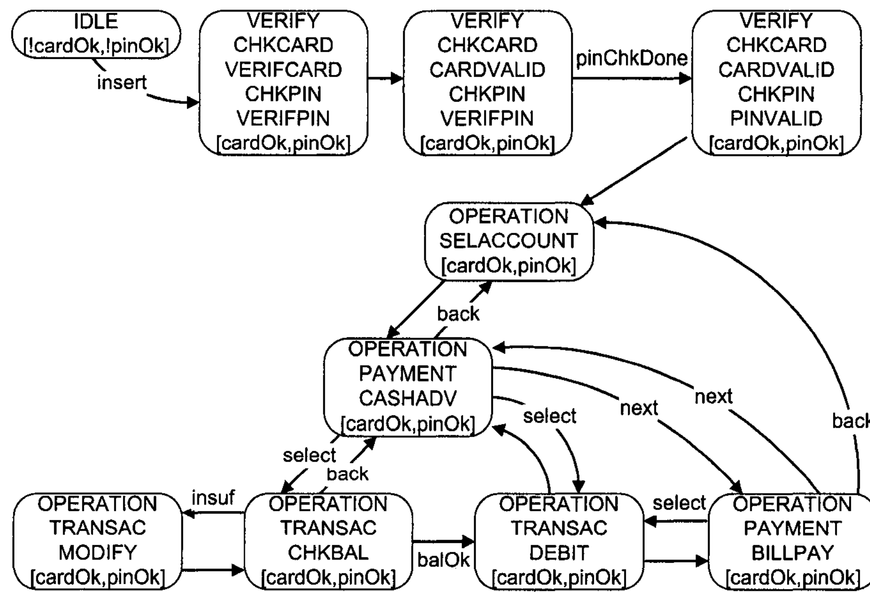

Figure 8.4: Control Flow Subgraph

The subgraphs identified in the foregoing paragraph represent the basis that enable us to slice (decompose) the initial model into several independent parts that can be analyzed separately. Obviously, the subgraphs have reduced complexity when compared to the original model. Accordingly, for each of them, the corresponding transition system that is going to be subjected to model checking requires fewer resources in terms of memory space and computation time. In order to emphasize the benefits of the slicing procedure, we subsequently give some statistics.

Since the model checker uses binary decision diagrams (BDDs) in order to store the generated state space of the model in a highly compact manner, this represents an eloquent comparison parameter. Thus, 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 follows. For the graphs in Figure 8.3.a and 8.3.c the

number of BDD nodes was around 4 thousand. Furthermore, the graph in Figure 8.3.b required around 8 thousand BDD nodes whereas the graph in Figure 8.4 required from about 28 to 33 thousand nodes.

Chapter 9

Verification and Validation of UML Activity Diagram

The UML activity diagram is basically inheriting the structured development concept of flowchart, essentially being the object-oriented equivalent thereof. As such, it can be used for business process modeling, for modeling various usage scenarios, or for capturing the detailed logic of a complex operation. It must be noted that the activity and state machine diagrams are related to some extent. However, while a state machine diagram is focusing on the state of a given object as it is undergoing a process (or on a particular process that captures the object state), an activity diagram is focusing on the flow of activities that are involved in a particular process or operation that may involve one or more interacting objects. Specifically, the activity diagram shows the nature of the relations established among the activities involved in carrying out a process or operation, typically including relations such as sequencing, conditional dependancy, synchronization, etc.

9.1 Derivation of the Activity Diagram Semantic Model

The semantic model derivation for the activity diagram inherits an idea that stems from the work of Eshuis et al. [21] and consists in encoding the activity diagram dynamics by generating its reachable configurations. In this respect, it resembles the approach presented in the state machine related sections where the same basic idea was employed.

Thus, in a similar manner to that presented in the case of the state machine, the activity diagram is converted to its corresponding CTS. Accordingly, each configuration is represented by the set of states of the activity diagram that are active concurrently (the true value is bound to its active states). Likewise, in order to generate all the reachable configurations of the activity diagram, we keep in each configuration the two additional lists, one corresponding to the value of all the guards for that configuration and respectively the second containing the join pattern list for that configuration. The latter is required due to the fact that the activity diagram allows for forking and joining activity flows. Thus, as in the case of the state machine, the join pattern list is used in order to record various synchronization points that may be reached while generating the configurations of the activity diagram.

Consequently, the procedure used for the CTS generation in the case of the activity diagram is just a variation of the state machine CTS generation algorithm presented in Section 8.1. The main difference consists in the fact that instead of generating the CTS configurations by using a list of possible incoming events, we track each activity flow associated with each concurrent activity state that is member of every new identified configuration. The procedure is presented by Algorithm 3 where the modification consists in picking each state in *crtStateList* and computing

all the possible next configurations that are reachable by any control transfer to a successor state in the same activity flow.

9.2 Activity Diagram Case Study

The selected case study for the activity diagram is aiming at presenting a compound usage scenario for the UML 2.0-based ATM design, whose state machine diagram was presented in Section 8.2. The usage scenario is likewise hypothetical and reflects a typical cash withdrawal operation that a potential customer (user) may perform. In the following paragraphs, we detail the intended operation captured by the activity diagram along with some relevant properties. Figure 9.1 shows the UML 2.0 activity diagram of the ATM cash withdrawal operation.

The operation begins with the `Insert Card` activity. Thereafter, the `Read Card` and `Enter Pin` activities are forked. The activity flow starting with `Read Card` continues with the `Authorize Card` activity while the one starting with `Enter Pin` continues with `Authorize Pin`.

Both `Authorize Card` and `Authorize Pin` activities are followed by corresponding test branching points. In the case where both the user card and PIN number check out, the two activity flows are joined together and the `Initiate transaction` activity is begun, followed in order by the `Select amount` and `Check Balance` activities. The latter is followed by a test branching point that if satisfied, forks two new activity flows. The first one begins with the `Debit account` activity and continues with the `Record Transaction` activity. The second one forks anew to `Dispense Cash` and `Print Receipt` activities. The three activity flows that

Algorithm 3 Generation of the Activity CTS (reusing part of Algorithm 2)

```
FoundConfList = {}
CTSConfList = { initialConf }
CTSTransList = ∅
CTScontainer = {DiagramStateList,guardValueList,∅}
while CTSConfList is not empty do
  CurrentConf = pop(CTSConfList)
  crtStateList = get(CurrentConf,0)
  crtGList = get(CurrentConf,1)
  crtJoinPatList = get(CurrentConf,2)
  if crtGList containsValue "any" then
    splitIndex = getPosition(crtGList, "any")
    crtGList[splitIndex] = true
    CTSConfList = CTSConfList ∪ { crtStateList, crtGList, crtJoinPatList }
    crtGList[splitIndex] = false
    CTSConfList = CTSConfList ∪ { crtStateList, crtGList, crtJoinPatList }
    continue
  end if
  if FoundConfList not contains CurrentConf then
    FoundConfList = FoundConfList ∪ CurrentConf
  end if
  for each state s in crtStateList do
    setConf(CTScontainer, CurrentConf)
    execute(s)
    nextConf = getConf(CTScontainer)
    if nextConf not equals CurrentConf then
      CTSConfList = CTSConfList ∪ nextConf
      crtTrans = {CurrentConf, nextConf}
      if CTSTransList not contains crtTrans then
        CTSTransList = CTSTransList ∪ {crtTrans}
      end if
    end if
  end for
end while
```

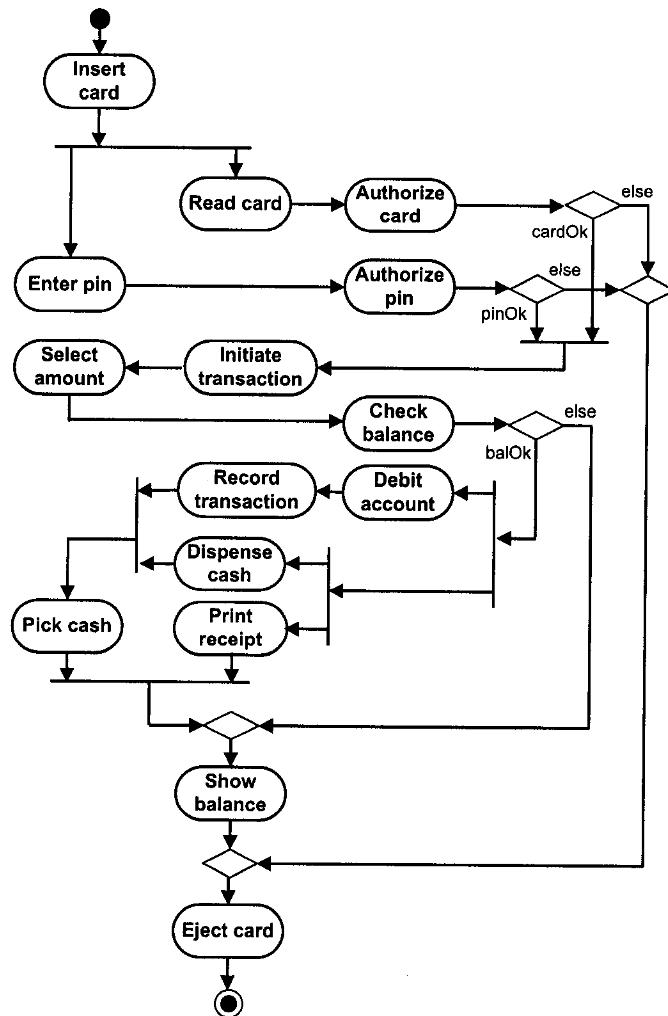

Figure 9.1: ATM Activity Diagram Example

are executing at this point are cross synchronizing as follows. The `Record Transaction` and `Dispense Cash` activities are joined in a single activity flow that begins executing the `Pick Cash` activity. This activity flow is then joined with the remaining one that was executing the `Print Receipt` activity. The control is then transferred in order to the `Show Balance` and `Eject Card` activities, the latter finishing the whole operation.

For the cases where the authorization test branching points are not satisfied, the control flow is transferred to the `Eject Card` activity whenever the card or the PIN number do not check out while in the case of the balance test branching point, the control is transferred to the `Show Balance`.

In order to outline the benefit of the model checking procedure, the presented case study intendedly contains a number of issues that are going to be identified during the verification procedure. Furthermore, in order to subject the activity diagram to the model checker, the following steps are required. First, the diagram is converted to its corresponding CTS depicted in Figure 9.2, which represents its semantic model wherein each element is represented by a set (possibly singleton) of activity nodes.

Second, we automatically specify deadlock and reachability properties for every activity node. Third, user defined specifications, intended to capture the desired behavior, are presented in both macro and CTL notation:

Property number one asserts that executing the `InsertCard` activity implies that it is inevitable to reach at a later point the `EjectCard` activity.

`Insert_Card` → `INEVIT Eject_Card`

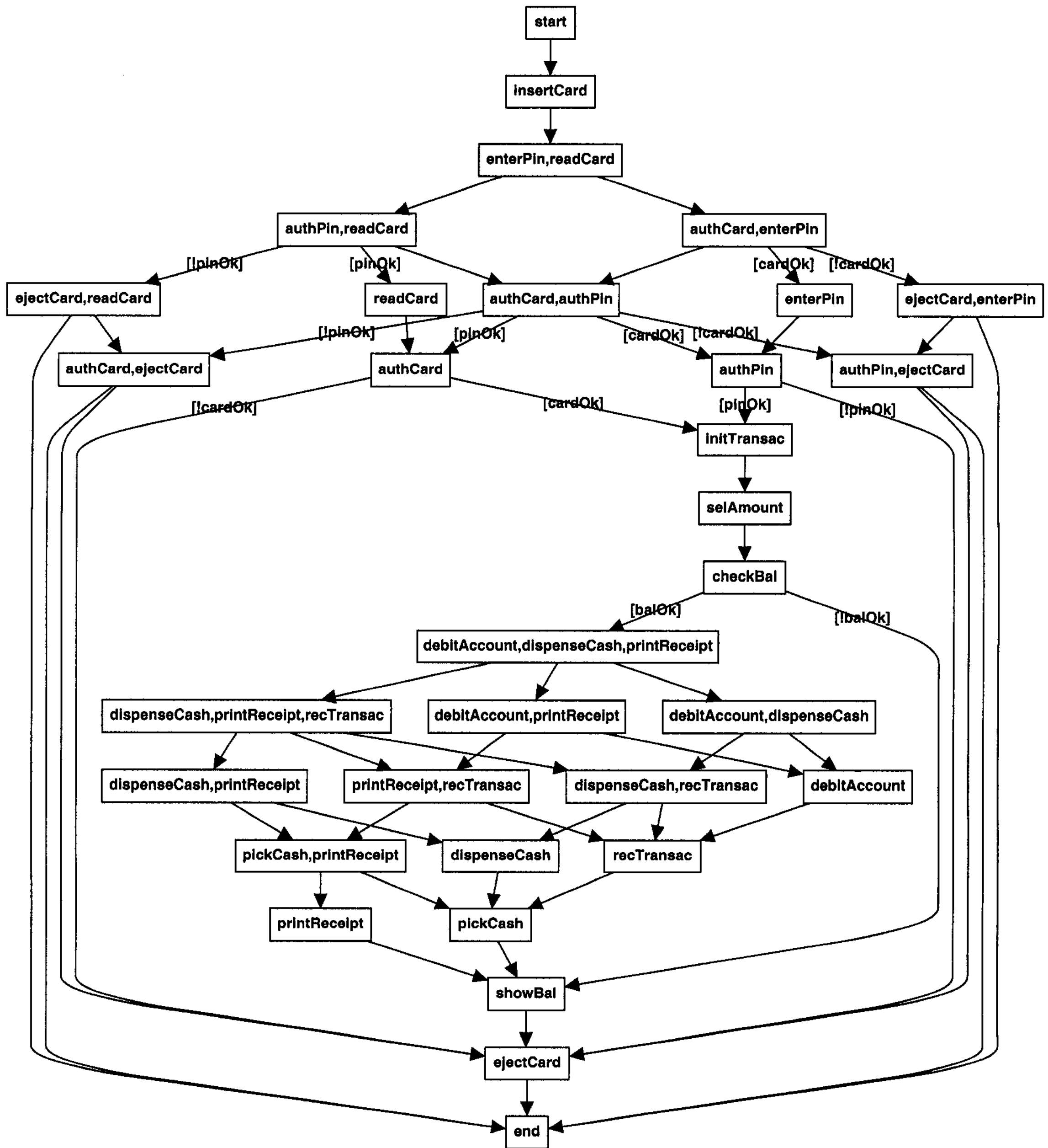

Figure 9.2: CTS of the ATM Cash Withdrawal Activity Diagram

CTL: `Insert_Card` → `A[!(end) U Eject_Card]`

Property number two asserts that it is always the case that executing the `InsertCard` activity implies that `AuthCard` activity precedes the `EjectCard` activity.

`ALWAYS Insert_Card → Auth_Card PRECEDE Eject_Card`

CTL: `AG((Insert_Card → (!E[!(Auth_Card) U Eject_Card])))`

Property number three asserts that it is always the case that executing the `Init_Transac` activity implies that `PickCash` activity may be reachable at a later point.

`ALWAYS Init_Transac → MAYREACH Pick_Cash`

CTL: `AG((Init_Transac → (E[!(end) U Pick_Cash])))`

The fourth property asserts that `DebitAccount` should precede `DispenseCash`.

`Debit_Account PRECEDE Dispense_Cash`

CTL: `(!E[!(Debit_Account) U Dispense_Cash])))`

Property number five asserts that the `EjectCard` activity should never be followed by other activity.

`NEVER (Eject_Card & POSSIB !end)`

CTL: `!EF(Eject_Card & EX !end)`

After running the model checker, the results that have been obtained indicate that no unreachable or deadlock states were detected while for the manual specifications, properties with the numbers one and three were satisfied. However, the properties with numbers two and four and five failed.

The failure of the properties with the numbers two and four was accompanied by couterexamples provided by the model checker and presented in Table 9.4 and respectively Table 9.5.

Insert_Card;
Enter_Pin,Read_Card;
Authorize_Card,Enter_Pin;
Eject_Card,Enter_Pin;

Table 9.4: Activity Diagram Counter-Example for Property Number Two

Insert_Card;
Enter_Pin,Read_Card;
Authorize_Pin,Read_Card;
Authorize_Card,Authorize_Pin;
Authorize_Pin;
Init_Transac;
Sel_Amount;
Check_Bal;
Debit_Account,Dispense_Cash,Print_Receipt;
Debit_Account,Print_Receipt;

Table 9.5: Activity Diagram Counter-example for Property Number Four

Property number five failed since there are reachable configurations that contain the `Eject_Card` activity together with another activity such as `Read_Card`.

The cash withdrawal operation requires a number of changes in order to have all of the specified properties pass. Figure 9.3 presents the corrected version of the activity diagram.

In order to fix the identified problems, several corrections were performed. Thus, the authorization test branching points are cascaded in sequence rather than concurrently after joining the activity flows forked for reading the card and entering the PIN. Moreover, the activities `Dispense_Cash` and `Record_Transaction` are swapped in order to enforce the sequencing of the former after the `Debit_Account` activity.

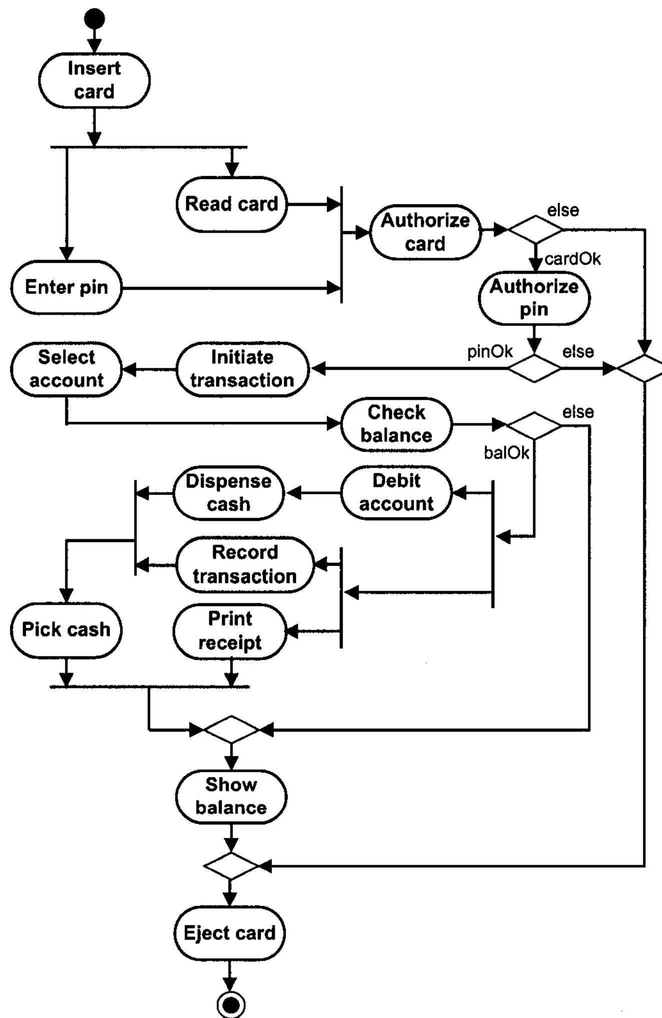

Figure 9.3: Fixed ATM Activity Diagram Example

Chapter 10

Performance Analysis of Time Constrained SysML Activity Diagrams

Many modern systems are now being developed by aggregating other subsystems and components that may have different expected though not exactly determined characteristics and features. As such, this kind of systems may exhibit features such concurrency and probabilistic behavior.

In order to address the need for an appropriate standardized modeling language that can support the Model Driven Architecture (MDA) [47] approach for the design and development of modern systems engineering products, OMG officially released the System Modeling Language (SysML) specification [25]. SysML is based on UML 2.0 but it is extending and adapting it to fit specific systems engineering modeling requirements.

Among SysML diagrams, activity diagram [8] represents a highly interesting one due to its suitability for functional flow modeling and similarity to the Extended Functional Flow Block

Diagrams (EFFBDs) [9] commonly used by systems engineers. The SysML specification has redefined and widely extended activity diagrams using the profiling mechanism of UML. The main extensions concern the support of continuous and probabilistic systems modeling.

10.1 Time Annotated SysML Activity Diagrams

Activity modeling is used for coordinating behaviors in the system being modeled. Particularly, these behaviors may require time duration to execute and terminate. Thus, we need to specify such constraints in order to be able to verify time-related properties for quantitative analysis.

The features related to probabilistic systems modeling are mainly used on the edges outgoing form decision nodes where probabilities can be assigned on the transitions emanating from the same decision node according to a specified distribution. Accordingly, the assigned values ought to sum up to unity as illustrated in Figure 10.4.

Since the execution time of a behavior may depend on different parameters such as resource availability and rates of incoming dataflows, this may lead to a variable termination time of a behavior. Consequently, if an action may terminate within a bounded time interval, then a probability distribution for terminating the action can be established with respect to the corresponding execution time interval. Consequently, we propose the use of an appropriate time annotation showing the execution duration of the actions in the activity diagram.

Thus, we denote by $TimeExp \subset \mathbb{N}$ the set of value specifications of time, where each value is expressed as a multiple of unit time and $\mathbb{N}$ is the set of natural numbers. The time reference is maintained by a global clock C against which all the time values are evaluated. A clock valuation

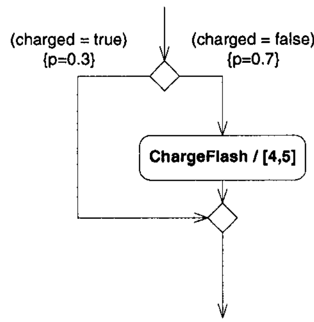

Figure 10.4: Probability and Execution Duration Annotation

v assigns to a clock a value in the domain of $TimeExp$.

For an action node, the *activation time* represents the duration of time wherein it is active. When an activity diagram starts, C is reset to zero value. Since we consider that a transition taken on an activity edge is timeless, time annotations are specified only for action nodes in the form of a time interval $I = [a, b]$, where $a, b \in TimeExp$ are evaluated relatively to the start of the activation time of the corresponding action. Time value a represents the earliest time for the execution completion and b is the latest. For instance, as depicted in Figure 10.4, action `ChargeFlash` terminates within 4 to 5 units of time. However, some actions may need a fixed time value to complete execution, i.e. $a = b$ and in that case, only a single time value may be shown. Also, the time annotation can be omitted if the activation time of an action is negligible compared to other actions.

10.2 Modeling Time Constrained Activity Diagrams

We introduced in Chapter 7 the concept of Configuration Transition System (CTS) that can be used in order to model the dynamics of various behavioral diagrams. In essence the CTS is a

form of automaton characterized by a set of configurations that include a (usually singleton) set of initial configurations and a transition relation that encodes the dynamic evolution of the CTS from one configuration to another. However, the CTS model assumes a background computation of some sort, that is responsible for the change in the set of dynamic parameters, a computation which is abstracted to a possible transition from a configuration to another. While this abstraction can be suitable in many cases, it might need more refinement in the cases where features such as the duration of the computation and/or the likelihood of a decision must be taken into account in relation to dynamic elements of the model. Thus, it becomes apparent that an enriched model is required in order to capture the aforementioned features. Since the most important elements of interest in the dynamics corresponding to various behavioral diagrams are usually represented by state variables or action blocks, we will consider these artifacts as dynamic elements.

When modelling a system with a network of automata, communication can be used in order to achieve synchronization. Also, time quantization techniques can be used in order to capture the dynamics of the communicating automata into a compact computable model suitable for automatic verification purposes. Thus, the approach consists in mapping the SysML activity diagram into a corresponding aggregation of Discrete Time Markov Chain (DTMC) modules representing a discrete-time transition system with discrete probability distributions. Then, the resulting model is encoded into the input language of the Probabilistic Symbolic Model Checker (PRISM)[1] [38] which can be used for performance analysis of the model[2] as illustrated in Figure 10.5.

[1]PRISM is probabilistic model checker developed at the University of Birmingham.

[2]The actual values for the probability of a given behavior occurrence can be determined by PRISM.

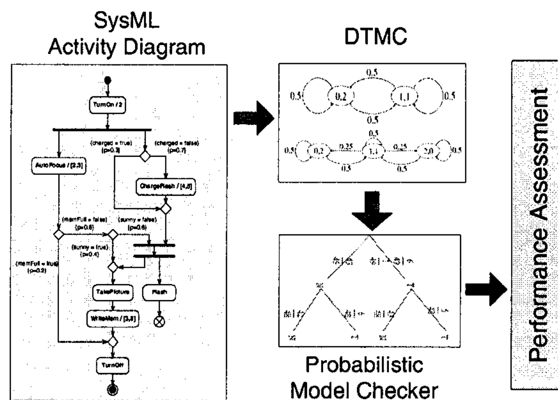

Figure 10.5: Proposed Approach for Assesing SysML Activity Diagrams

Moreover, since counterexamples are presently not provided by PRISM, we use the state space that it computes in order to construct the reachability graph that can be used to graphically explore the behavior of the model.

PRISM is based on reactive modules [5], where each module represents a system component. It targets the assessment of probabilistic systems such as various communication protocols, security related systems, etc. The modules describing the system are composed in parallel and contain variables as well as commands labeled with actions for synchronization between modules, hidden actions, guards and probabilities. As the objective is to capture the dynamics of time annotated activity diagram into a compact computable model, we selected DTMC as its interpreted semantics since it is suitable to model the coordination of the behavior, action execution duration, synchronizations as well as probabilistic path selection. Moreover, it represents a discrete-time transition system with discrete probability distributions that can efficiently capture the intended behavior of the activity diagram. Furthermore, it is lightweight compared to the other probabilistic models

supported by PRISM[3]. For convenience, we henceforth assume an equal probability for action termination anytime within the annotated time interval. However, this has no restrictive impact with respect to our approach. The tradeoff involved in selecting a discrete time model consists in either having a more fine grained representation at the expense of higher costs in terms of verification feasibility or a less fine grained one that can in turn significantly boost the verification performance thus allowing for more elaborated models to be verified.

A concise definition of the DTMC model [54] is as follows:

Definition 4. A discrete-time Markov chain (DTMC) is a tuple $\mathcal{D} = (S, \overline{s}, \mathbf{P})$ where S is a finite set of states, $\overline{s} \in S$ is the initial state, and $\mathbf{P} : S \times S \rightarrow [0, 1]$ is a transition probability matrix, such that: $\sum_{s' \in S} \mathcal{P}(s, s') = 1$ for all $s \in S$ where $\mathcal{P}(s, s')$ is the probability of making a transition from a state s to a state s'.

For the specification of quantitative analysis properties, PRISM incorporates two property specification languages: Probabilistic Computation Tree Logic (PCTL) used for DTMC and MDP and Continuous Stochastic Logic (CSL) used for CTMC. PCTL [14] is an extension of CTL [16] mainly with probability operator and according to [14] its syntax is as follows:

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

$$\psi \ ::= \ \phi_1 \, \mathcal{U}^t \, \phi_2 \mid \phi_1 \, \mathcal{U} \, \phi_2 \mid \mathcal{X} \phi$$

where a is an atomic proposition, $t \in \mathbb{N}$, $p \in [0, 1] \subset \mathbb{R}$, and $\bowtie \in \{ >, \geq, <, \leq \}$. PCTL formulas are of two types, state (ϕ) and path (ψ). The former is evaluated over a state and the latter over a

[3]Relating to the other probabilistic models supported by PRISM, CTMC can be viewed as DTMC with an infinitesimally small time step whereas MDP is extending DTMC with non-determinism [38].

path. For example, a state s of the DTMC satisfies the formula $\mathcal{P}_{\bowtie p}[\psi]$ if the probability of taking a path from s satisfying ψ is in the interval defined by $\bowtie p$.

10.3 Mapping SysML Activity to DTMC

The first step in transforming SysML activity to DTMC consists in identifying different communicating modules corresponding to the synchronizing threads in the diagram. In order to be able to delimit the existing individual threads, we have to explore the different execution flows of the activity diagram. Special constructs such as $fork$ and $join$ represent respectively spawning and synchronization points. At each $fork$, there are as many new threads as outgoing edges and at each $join$, there is a waiting point for all the incoming threads to synchronize. Accordingly, we allocate a module for each identified thread.

In the case where some execution paths are intersecting with each other we chose to merge the corresponding modules. Also, when a thread reaches a join construct and terminates after synchronization, the corresponding module can be reused for the subsequent thread. This is done for the sake of optimizing the PRISM code in order to keep it easily manageable. Algorithm 4 presents the generic procedure for converting an activity diagram to its corresponding DTMC.

The synchronization mechanism among the modules allows for two or more concurrent activity flows or execution threads to "experience" the same passage of time with respect to the time constraints that may be specified for each of them. Consequently, each thread has its own clock variable that is used to track the passage of time in the current state of the thread. The dynamics of the model ensures that all the clock variables are updated (advanced or reset) synchronously. Thus,

Algorithm 4 Mapping Activity Diagrams *AD* To PRISM DTMC

```
CTT = GetStructure(AD) {Encodes the structural artifacts into Control Transfer Table (CTT)}
FEP = generateFEP(CTT) {Explores CTT and generates the list of Flow Execution Paths (FEP)}
SSP = GetSynchropoints(CTT,FEP) {Generates a list of States Synchronization POINTs (SSP)}
for all item in FEP do
  LCT= Compact(FEP) {Compacts the overlapping parts of FEP into LCT (list of connecting
  threads)}
end for
for all item in LCT do
  CreateNEWModule(item) {encode each thread in LCT along with its SSP as a separate
  PRISM module}
  PRISMModulesList.add(newPRISM) {generate PRISM code}
end for
```

the clock variable of each thread is advanced as a result of a self transition to its current state or reset whenever the current state is left and another one is entered. Furthermore, whenever the clock variable of a thread falls within the time constraint interval of the current state, this means that the control can either remain in the current state or be transferred to another state that can be reached by a transition from the current state. This amounts to the use of a probability distribution based on which the thread will remain in the current state or exit it. Though the choice for such a distribution may depend on the actual system being modeled, by default we use a uniform distribution over the time constraint interval.

10.4 Performance Evaluation Case Study

In this section, we present a relevant case study involving a systems engineering product for which a SysML activity diagram with time constraints is analyzed. The system represents a hypothetical model of a digital photo-camera while the given activity diagram is meant to capture the functional aspect of taking a picture as depicted by Figure 10.6. The selected properties aim to measure and

verify certain performance characteristics exhibited by the model. The presented activity diagram is intentionally not very laborious as to serve its instructive purpose. However, as it will become apparent later, a simple model does not preclude a highly dynamic behavior.

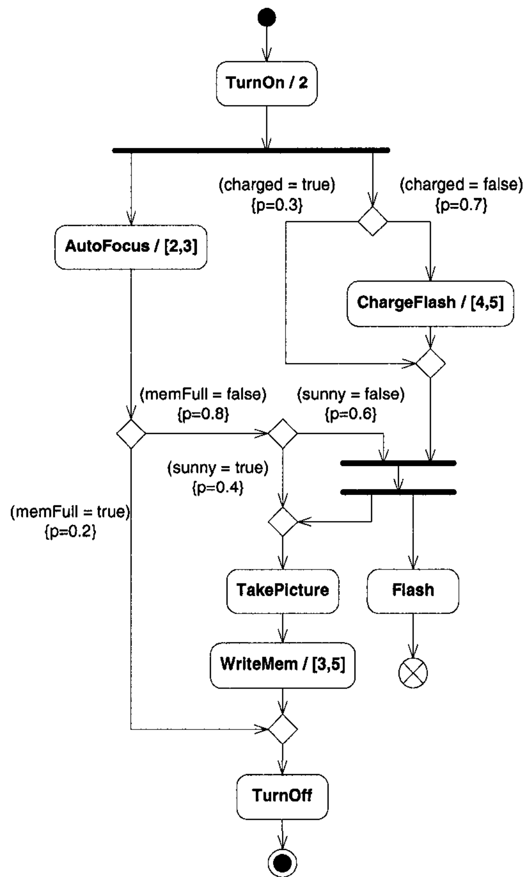

Figure 10.6: Digital Camera SysML Activity Diagram Example

For our example, the selection of an appropriate unit of time from the sequencing and performance perspectives has to be relevant for the user of the product. Accordingly, the activity diagram nodes that may require one or more units of time to complete are augmented with corresponding

values for the interval denoting the expected time to completion. In contrast, the activity nodes that take less than one unit of time to complete have no such augmentation.

The process captured by our activity diagram example starts by turning on the camera (TurnOn). Subsequently, two threads are spawned. The first one commences by auto focusing (AutoFocus) followed by a decision point that checks whether the memory is full (memFull guard) or not. In the latter case (memFull = false), a second decision point is reached, where depending on the ambient lighting conditions (sunny or not) taking a picture (TakePicture) is executed. TakePicture may be reached either directly (sunny = true) or after synchronizing with the second thread (sunny = false). Thereafter, the picture is stored in the memory (WriteMem). The second thread begins by checking whether the flash is charged or not. If not, the control is transferred to charging the flash (ChargeFlash). Subsequently or if the flash is already charged, the second thread waits in order to synchronize (join node) with the first thread. After synchronization, the flashing is triggered (Flash). The activity diagram execution is ended with turning off the camera (TurnOff).

To assess the activity diagram of the digital camera, we generated the corresponding DTMC according to the procedure depicted in Algorithm 4 and encoded it into PRISM model checker input language.The corresponding reachability graph[4] is presented in Figure 10.7. As illustrated, each edge of the reachability graph is annotated with its corresponding probability of selection. Furthermore, the graph shows a highly dynamic behavior resulting from the concurrency of the threads in the activity diagram in conjunction with the overlapping completion intervals of various activity nodes.

In the following paragraphs, we present the results that are obtained by verifying a number of

[4]The reachability graph was derived from the state and transition matrices computed by PRISM version 3.0.

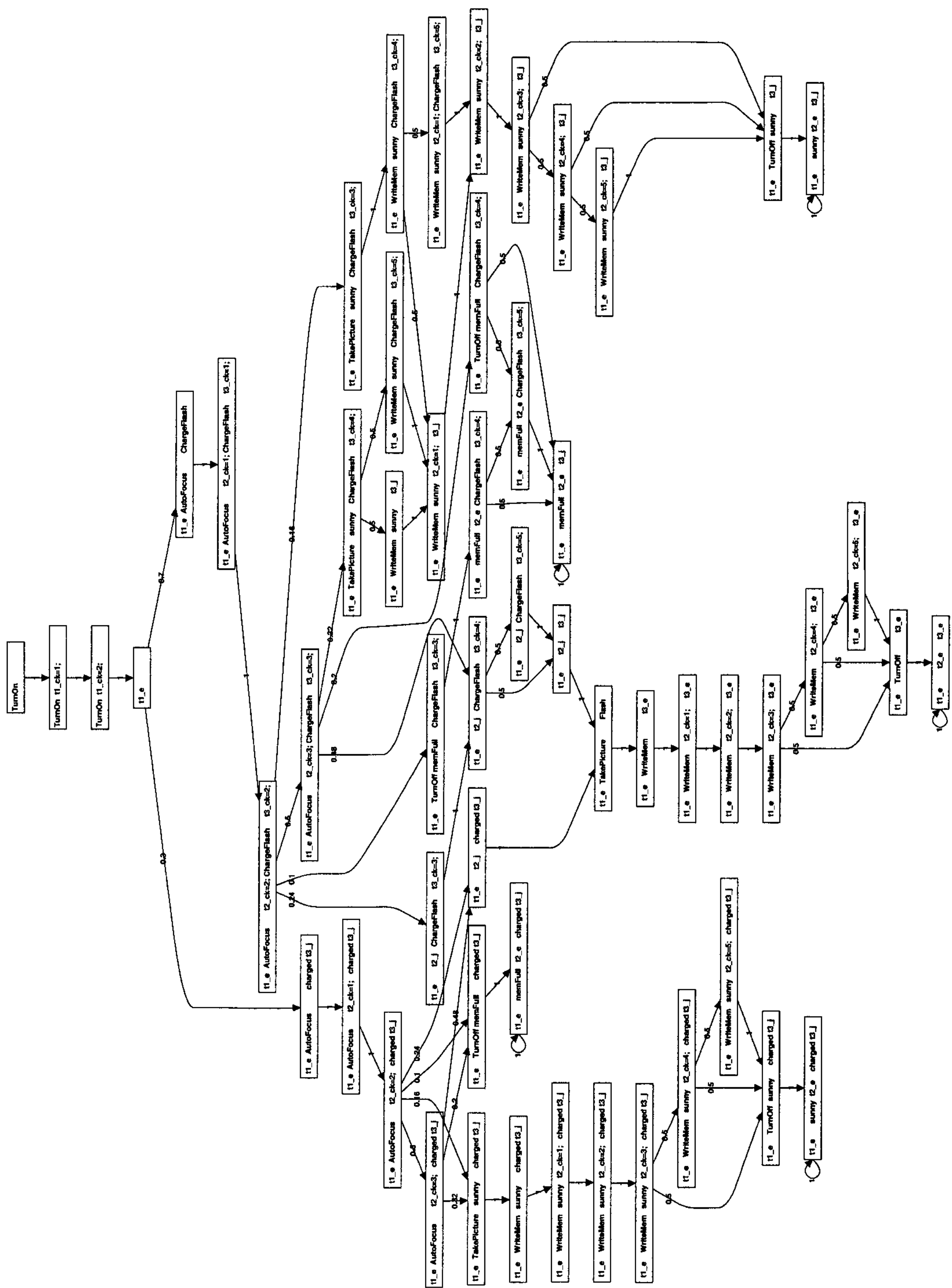

Figure 10.7: Reachability Graph of the Digital Camera Activity Diagram Example

interesting properties expressed in the syntax of the temporal logic defined by PRISM and based on PCTL. This highlights the benefits of assessing probabilistic behavior in the presence of time constraints.

The first property states that the probability to take a picture after turning on the camera should be 0.75 or greater. It is expressed in PRISM as follows:

$$\texttt{TurnOn} \Rightarrow \texttt{P} \geq 0.75\,[\,\texttt{true}\ \mathrm{U}\ \texttt{TakePicture}\,]$$

When analyzing this property over the DTMC of the activity diagram, PRISM reported that it turns out to be satisfied. This reflects that the model meets the minimum required level of reliability. The second property is building on the previous one and aims to check whether in the case of poor lighting conditions, the probability of taking a picture using the flash after turning on the camera is at least 0.5. It is expressed in PRISM as follows:

$$\texttt{TurnOn} \Rightarrow \texttt{P} \geq 0.5\,[\,\texttt{true}\ \mathrm{U}\ !\texttt{sunny}\ \&\ \texttt{TakePicture}\ \&\ \texttt{Flash}\,]$$

Though the model checker determined that the property fails for a probability value greater or equal to 0.5, it passes for a value less or equal to 0.48. The actual value of the probability can be determined by using a specific PRISM construct that queries for the probability value. This illustrates the quantitative analysis feature provided by PRISM. The formula is as follows:

$$\texttt{P} = ?\,[\,\texttt{true}\ \mathrm{U}\ !\texttt{sunny}\ \&\ \texttt{TakePicture}\ \&\ \texttt{Flash}\ \{\texttt{TurnOn}\}\,]$$

Apart from verifying properties that concern the probability of reaching a particular situation, we also consider time-bounded reachability verification which consists in measuring the probability of reaching a certain situation within a time bound. For instance, the following property is measuring the probability of taking a picture using the flash within 6 units of time after turning on the camera:

$$\texttt{P} = ?\,[\,\texttt{true}\,\texttt{U}\ \leq 6+2\ \texttt{TakePicture}\ \&\texttt{Flash}\ \{\texttt{TurnOn}\ \&\ \texttt{t1_ck} = \texttt{0}\}\,]$$

One can note that 2 supplementary units of time are added in the property formula. The reason behind this is due to the fact that when reaching a synchronization point (join node), each thread needs to enter a waiting state wherein it awaits for the other synchronizing threads to complete their respective activities. When all the synchronizing threads reach the synchronization point, they require one unit of time to leave their waiting states and one unit of time to perform the synchronization and proceed further. Thus, for this kind of properties involving synchronizing threads, one has to take this fact into account.

The obtained result indicates a severe performance issue for the considered time constraint requirement since the computed value of the probability is 0.072. This means that the likelihood to take a picture within 6 units of time is less than $1/10$. Nevertheless, considering the fact that charging the flash may take a significant amount of the 6 unit of time, the obtained result reflects either a bad design or that this particular performance requirement is not mandatory but rather optional. However, the model checker has determined higher probability values for more relaxed time constraints. Thus, a value of 0.144 was identified for a time interval of 7 units, 0.312 for 8 units and 0.48 for 9 units.

Chapter 11

Conclusion

The increased difficulties posed by modern system development, especially in the area of software intensive systems design led to the emergence of systems modeling languages such as UML 2.0 and SysML. Organizations such as INCOSE and OMG represent very active drivers in the development of the aforementioned modeling languages that are meant to accommodate a broad range of systems engineering aspects.

Systems engineering aims to the successful design and development of complex systems by providing the means to cope with the increased complexity exhibited by modern system design allowing for the development of structured and efficient approaches for solving complex engineering problems.

In this research initiative, we presented a unified paradigm for the automated verification and validation of software and systems engineering design models expressed in UML and SysML mainly based on model checking techniques and intended as a rigorous augmentation for conventional techniques involving testing and simulation.

The appropriate techniques for the proposed approach were gradually introduced in the context of the presented verification and validation framework embodying the proposed methodology. The thesis also details the artifacts and specification of the targeted behavioral diagrams, namely the state machine and activity diagrams while introducing a unified semantic model that is appropriate for capturing the semantics of such behavioral models. The procedure involved in applying the model checking technique is also described along with model property specification in the context of the CTL temporal logic. Moreover, the feasibility of the methodology is demonstrated on relevant case studies. Additionally, the synergy emerging from using program analysis techniques is advocated as means to tackle the state explosion problem that is characteristic to the model checking procedure. Furthermore, it details a transformation procedure of SysML activity diagram annotated with time constraints and probability artifacts to discrete time Markov chains that can be analyzed by a probabilistic model checker in order to assess various performance parameters.

The significance of the presented research initiative gained scientific visibility in several publications [2–4, 33]. Also, the proposed methodology is cost effective and can be used in order to diminish and even eliminate the variances between design intent and actual design quality and performance. Moreover, by using it during the early stages of the design, the common penalities associated with late maintenance, debugging and error fixing can be significantly alleviated. Furthermore, due to its automated characteristic, one can also use such a verification and validation paradigm for the assessment of already existing software and system design models. Consequently, this can help in discovering potential heretofore unidentified erroneous or undesired behavior as well as in detecting concealed performance issues.

Bibliography

[1] National Aeronautics and Space Administration. 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 20771, JUNE 1995.

[2] Luay Alawneh, Mourad Debbabi, Fawzi Hassane, Yosr Jarraya, Payam Shahi, and Andrei Soeanu. Towards a Unified Paradigm for Verification and Validation of Systems Engineering Design Models. In *SE'06: Proceedings of the IASTED International Conference on Software Engineering*, pages 282–287. ACTA Press, 2006.

[3] Luay Alawneh, Mourad Debbabi, Fawzi Hassane, and Andrei Soeanu. On the Verification and Validation of UML Structural and Behavioral Diagrams. In *ACST'06: Proceedings of the 2nd IASTED international conference on Advances in computer science and technology*, pages 304–309. ACTA Press, 2006.

[4] Luay Alawneh, Mourad Debbabi, Yosr Jarraya, Andrei Soeanu, and Fawzi Hassayne. A Unified Approach for Verification and Validation of Systems and Software Engineering Models.

In *ECBS '06: Proceedings of the 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems (ECBS'06)*, pages 409–418, 2006.

[5] Rajeev Alur and Thomas A. Henzinger. Reactive Modules. *Form. Methods Syst. Des.*, 15(1):7–48, 1999.

[6] M. Bartley, D. Galpin, and T. Blackmore. A Comparison of Three Verification Techniques: Directed Testing, Pseudo-Random Testing and Property Checking.

[7] Shoham Ben-David, Tamir Heyman, Orna Grumberg, and Assaf Schuster. Scalable Distributed On-the-Fly Symbolic Model Checking. In *Formal Methods in Computer-Aided Design*, pages 390–404, 2000.

[8] C. Bock. Systems Engineering in the Product Lifecycle. *Int. Journal Product Development*, 2:123137, 2005.

[9] C. Bock. SysML and UML 2 support for Activity Modeling. *Syst. Eng.*, 9(2):160–186, 2006.

[10] G. Booch. *Object-Oriented Analysis and Design with Applications*. Addison-Wesley, second edition, 1997.

[11] Egon Börger, Alessandra Cavarra, and Elvinia Riccobene. An ASM Semantics for UML Activity Diagrams. In *AMAST*, pages 293–308, 2000.

[12] Universitat Bremen. udraw(graph)tool. `http://www.informatik.uni-bremen.de/uDrawGraph/en/index.html`.

[13] C. Canevet, S. Gilmore, J. Hillston, L. Kloul, and P. Stevens. Analysing UML 2.0 Activity Diagrams in the Software Performance Engineering Process. In *WOSP '04: Proceedings of the 4th international workshop on Software and performance*, pages 74–78, New York, USA, 2004. ACM Press.

[14] Frank Ciesinski and Marcus Größer. On Probabilistic Computation Tree Logic. In *Validation of Stochastic Systems*, pages 147–188, 2004.

[15] Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. NuSMV: A New Symbolic Model Verifier. In *cav*, 1999.

[16] Edmund M. Clarke and E. Allen Emerson. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, 1982. Springer-Verlag.

[17] Communications Committee. The International Council on Systems Engineering INCOSE. `http://www.incose.org/practice/whatissystemseng.aspx`.

[18] Werner Damm, Bernhard Josko, Amir Pnueli, and Angelika Votintseva. Understanding UML: A Formal Semantics of Concurrency and Communication in Real-Time UML. In Frank de Boer, Marcello Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Proceedings of the 1st Symposium on Formal Methods for Components and Objects (FMCO 2002)*, volume 2852 of *LNCS Tutorials*, pages 70–98, 2003.

[19] Pallab Dasgupta, Arindam Chakrabarti, and P. P. Chakrabarti. Open Computation Tree Logic for Formal Verification of Modules. In *ASP-DAC '02: Proceedings of the 2002 conference on*

Asia South Pacific design automation/VLSI Design, page 735, Washington, DC, USA, 2002. IEEE Computer Society.

[20] Clarence A. Ellis and Gary J. Nutt. Modeling and Enactment of Workflow Systems. In *Proceedings of the 14th International Conference on Application and Theory of Petri Nets*, pages 1–16. Springer-Verlag, 1993.

[21] R. Eshuis and R. Wieringa. Tool Support for Verifying UML Activity Diagrams. *IEEE Trans. Softw. Eng.*, 30(7):437–447, 2004.

[22] Rik Eshuis. *Semantics and Verification of UML Activity Diagrams for Workflow Modelling*. PhD thesis, University of Twente, 2002.

[23] Harald Fecher, Marcel Kyas, and Jens Schönborn. Semantic Issues in UML 2.0 State Machines. Technical Report 0507, Christian-Albrechts-Universität zu Kiel, 2005.

[24] C. Fournet and G. Gonthier. The Join Calculus: A Language for Distributed Mobile Programming. In *Proceedings of the Applied Semantics Summer School (APPSEM), Caminha, September 2000.*

[25] Object Management G. *OMG Systems Modeling Language (OMG SysML) Specification*, May 2006. Final Adopted Specification of Systems Modeling Language (SysML).

[26] Marcela Genero, Mario Piattini, Esperanza Manso, and Giovanni Cantone. Building UML Class Diagram Maintainability Prediction Models Based on Early Metrics. In *Proceedings of the 9th International Symposium on Software Metrics*, page 263. IEEE Computer Society, 2003.

[27] Richard Gronback. Model Validation: Applying Audits and Metrics to UML Models. In *BorCon 2004 Proceedings*, 2004.

[28] Object Management Group. UML for Systems Engineering. `http://syseng.omg.org/UML_for_SE_RFP.htm`, 2003.

[29] Esther Guerra and Juan de Lara. A Framework for the Verification of UML Models. Examples Using Petri Nets. In *JISBD*, pages 325–334, 2003.

[30] Gerard J. Holzmann. The Model Checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[31] IEEE. *IEEE Std 610.12-1990, IEEE Standard for Software Verification and Validation*, 1990.

[32] Averant Inc. Static Functional Verification with Solidify, a New Low-Risk Methodology for Faster Debug of ASICs and Programmable Parts. Technical report, Averant, Inc., 2001.

[33] Yosr Jarraya, Andrei Soeanu, Mourad Debbabi, and Fawzi Hassaine. Automatic Verification and Performance Analysis of Time-Constrained Sysml Activity Diagrams. In *ECBS '07: Proceedings of the 14th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems (ECBS'07)*, pages 515–522, 2007.

[34] K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*, volume 1 of *Monographs in Theoretical Computer Science*. Springer-Verlag, 1997.

[35] 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 *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 1–33, Washington, D.C., 1990. IEEE Computer Society Press.

[36] K.L. McMillan. The SMV system. Technical Report CMU-CS-92-131, 1992.

[37] D. Kroening. Application Specific Higher Order Logic Theorem Proving. In *S. Autexier and H. Mantel, editors, Proc. of the Verification Workshop - VERIFY'02, pages 5–15*, 2002.

[38] 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.

[39] Gihwon Kwon. Rewrite rules and Operational Semantics for Model Checking UML Statecharts. In Andy Evans, Stuart Kent, and Bran Selic, editors, *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000, Proceedings*, volume 1939 of *LNCS*, pages 528–540. Springer, 2000.

[40] Unified Modeling Language. `http://www.uml.org/`.

[41] Diego Latella, István Majzik, and Mieke Massink. Automatic Verification of a Behavioural Subset of UML Statechart diagrams using the spin model-checker. *Formal Asp. Comput.*, 11(6):637–664, 1999.

[42] Diego Latella, Istvan Majzik, and Mieke Massink. Towards a Formal Operational Semantics of UML Statechart Diagrams. In *Proceedings of the IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, page 465. Kluwer, B.V., 1999.

[43] W. Li and S. Henry. Maintenance metrics for the object oriented paradigm. In *First International Software Metrics Symp.*, pages 52–60, 1993.

[44] Johan Lilius and Ivan Porres Paltor. vUML: a Tool for Verifying UML Models. Technical Report TUCS-TR-272, 18, 1999.

[45] S. Mehta, S. Ahmed, S. Al-Ashari, Dennis Chen, Dev Chen, S. Cokmez, P. Desai, R. Eltejaein, P. Fu, J. Gee, T. Granvold, A. Iyer, K. Lin, G. Maturana, D. McConn, H. Mohammed, J. Moudgal, S. Nori, N. Parveen, G. Peterson, M. Splain, and T. Yu. Verification of the Ultrasparc Microprocessor. In *40th IEEE Computer Society International Conference (COMPCON'95)*, pages 452–461, 1995.

[46] Erich Mikk, Yassine Lakhnech, Michael Siegel, and Gerard J. Holzmann. Implementing Statecharts in PROMELA/SPIN. In *WIFT '98: Proceedings of the Second IEEE Workshop on Industrial Strength Formal Specification Techniques*, page 90. IEEE Computer Society, 1998.

[47] J. Miller and J. Mukerji. MDA Guide Version 1.0. *OMG Document.*, May 2003.

[48] Birger Moller-Pedersen and Dagbjorn Nogva. Scalable and Object Oriented SDL State (Chart). (IFIP TC6/WG6.1):59–74, 1999.

[49] F. Nielson, H. Nielson, and C. Hankin. *Principles of Program Analysis.* Springer Verlag, 1999.

[50] SysML Partners. System Modeling Language: SysML. `http://www.sysml.org/`, 2004.

[51] Doron Peled. Combining partial order reductions with on-the-fly model-checking. In *CAV '94: Proceedings of the 6th International Conference on Computer Aided Verification*, pages 377–390, London, UK, 1994. Springer-Verlag.

[52] Wolfgang Reisig. *Petri Nets, An Introduction.* Springer-Verlag, Berlin, 1985.

[53] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual Second Edition.* Addison-Wesley, 2005.

[54] J. Rutten, M. Kwiatkowska, G. Norman, and D. Parker. *Mathematical Techniques for Analyzing Concurrent and Probabilistic Systems*, volume 23 of *CRM Monograph Series.* American Mathematical Society, 2004.

[55] Johann Schumann. Automated Theorem Proving in High-Quality Software Design. In *Intellectics and Computational Logic*, pages 295–312, 2000.

[56] Bran Selic. On the Semantic Foundations of Standard UML 2.0. In *SFM*, pages 181–199, 2004.

[57] 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 Ninth International Symposium of the International Council on Systems Engineering (INCOSE 99)*, Brighton, U.K., 1999.

[58] Wil M. P. van der Aalst. The Application of Petri Nets to Workflow Management. *Journal of Circuits, Systems, and Computers*, 8(1):21–66, 1998.

[59] Valdis Vitolins and Audris Kalnins. Semantics of UML 2.0 Activity Diagram for Business Modeling by Means of Virtual Machine. In *EDOC*, pages 181–194, 2005.

[60] Michael von der Beeck. A Structured Operational Semantics for UML-Statecharts. *Software and System Modeling*, 1(2):130–141, 2002.

[61] What Is OMG-UML and Why Is It Important? `http://www.omg.org/news/pr97/umlprimer.html`.

[62] Xuede Zhan and Huaikou Miao. An Approach to Formalizing the Semantics of UML Statecharts. In *Conceptual Modeling - ER 2004, 23rd International Conference on Conceptual Modeling, Shanghai, China, November 2004, Proceedings*, pages 753–765, 2004.