3-26-2009

Formal verification and testing of software architectural models

Gonzalo Argote Garcia
Florida International University

FLORIDA INTERNATIONAL UNIVERSITY

Miami, Florida

FORMAL VERIFICATION AND TESTING OF SOFTWARE ARCHITECTURAL MODELS

A dissertation submitted in partial fulfillment of the

requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Gonzalo Argote Garcia

2009

To: Dean Amir Mirmiran
College of Engineering and Computing

This dissertation, written by Gonzalo Argote Garcia, and entitled Formal Verification and Testing of Software Architectural Models, having been approved in respect to style and intellectual content, is referred to you for judgment.

We have read this dissertation and recommend that it be approved.

Yi Deng

Peter J. Clarke

Ronald M. Lee

S. Masoud Sadjadi

Xudong He, Major Professor

Date of Defense: March 26, 2009

The dissertation of Gonzalo Argote Garcia is approved.

Dean Amir Mirmiran
College of Engineering and Computing

Dean George Walker
University Graduate School

Florida International University, 2009

DEDICATION

I dedicate this dissertation to my parents, wife and son.

ACKNOWLEDGMENTS

First of all, I would like to extend my infinite gratitude to Prof. Xudong He, my advisor. He has been supportive, both academically and financially, throughout all these years pursuing my Ph.D. at Florida International University (FIU). His wisdom and advice have helped to shape a better person and professional in me. I am deeply thankful for all that he has done.

Dr. Peter J. Clarke also deserves lots of gratitude, it has been a pleasure to work on research papers with him, his drive to excel and enthusiasm are great motivators. He encouraged me repetitive times to complete my Ph.D. work and gave me advice when I needed. I would also like to thank all the members in my committee, to Dr. Ronald M. Lee for giving time to this dissertation in his busy schedule, to Dr. Yi Deng for being a great guidance in all these years at FIU and to Dr. S. Masoud Sadjadi for sharing his enthusiasm and knowledge with me on several occasions.

My wife, Ivanka C. Medrano, and son, Joshua G. Argote, deserve special thanks for their understanding when I had to spend my time at work and doing my dissertation and, as a result, not being available for them. They were the pillars for giving the final push to my work.

I would also like to thank all the members in the University Graduate School (UGS) at FIU. Special thanks to Mrs. Graciela Laforest and her family, great friends whatsoever. I could not forget Mr. Ruben Jaen, former vice-dean of the UGS, who willingly accepted me as a Graduate Assistant during my first semesters at FIU. I can not name all the people that one way or another have influenced me at FIU, Mr. Louis Farnsworth, Dr. Douglas Wartzok, it was a great pleasure to work with you. To all the members of the Graduate Admissions Office and the Undergraduate Admissions Office who received me with open heart, thank you for being so kind to my family and me.

My friends and peers in the School of Computing and Information Sciences deserve special consideration. Especially Richard Whittaker who has been such a great support in classroom and outside of it, he is in all aspects a true friend. Ariel Cary helped me a lot in the final stages of my dissertation, coordinating with Ricardo Koller and dealing with the logistics of the dissertation while I was away. To the members of the CADSE group, the ones that already finished, Drs. Yujian Fu, Zhijiang Dong, Weixiang Sun and Tianjun Shi, the ones that are still there, Lily Chang, among others, thank you for your fruitful discussions and friendship.

My brothers and sisters, Dunia, Gustavo, Gilda, Yuri and Nelo who have influenced me since I was a kid, this achievement is theirs too. I would like to thank my sister Dunia Sanzetenea and her family for hosting me at their home when I first came to the United States to pursue my graduate studies.

To my parents, Vicente Argote Covarrubias and Rosario García Zambrana, there are no words to express what I feel, this achievement is yours.

In summary, I want to thank all the people who directly or indirectly were responsible for the completion of my graduate studies. I know that I am leaving FIU with a treasure which is all of them.

This dissertation research was partially supported by the NSF awards HDR-0317692 and IIP-0534428.

ABSTRACT OF THE DISSERTATION

FORMAL VERIFICATION AND TESTING OF SOFTWARE ARCHITECTURAL MODELS

by

Gonzalo Argote Garcia

Florida International University, 2009

Miami, Florida

Professor Xudong He, Major Professor

Ensuring the correctness of software has been the major motivation in software research, constituting a Grand Challenge. Due to its impact in the final implementation, one critical aspect of software is its architectural design. By guaranteeing a correct architectural design, major and costly flaws can be caught early on in the development cycle. Software architecture design has received a lot of attention in the past years, with several methods, techniques and tools developed. However, there is still more to be done, such as providing adequate formal analysis of software architectures. On these regards, a framework to ensure system dependability from design to implementation has been developed at FIU (Florida International University). This framework is based on SAM (Software Architecture Model), an ADL (Architecture Description Language), that allows hierarchical compositions of components and connectors, defines an architectural modeling language for the behavior of components and connectors, and provides a specification language for the behavioral properties. The behavioral model of a SAM model is expressed in the form of Petri nets and the properties in first order linear temporal logic.

This dissertation presents a formal verification and testing approach to guarantee the correctness of Software Architectures. The Software Architectures studied are expressed in SAM. For the formal verification approach, the technique applied was

model checking and the model checker of choice was Spin. As part of the approach, a SAM model is formally translated to a model in the input language of Spin and verified for its correctness with respect to temporal properties. In terms of testing, a testing approach for SAM architectures was defined which includes the evaluation of test cases based on Petri net testing theory to be used in the testing process at the design level. Additionally, the information at the design level is used to derive test cases for the implementation level. Finally, a modeling and analysis tool (SAM tool) was implemented to help support the design and analysis of SAM models. The results show the applicability of the approach to testing and verification of SAM models with the aid of the SAM tool.

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE PAGE

CHAPTER 1

INTRODUCTION

Ensuring software correctness "*has long been the goal in Computer Science*" ([42] and [41]). As software systems become more complex and are integrated into almost every aspect of people's lives, there is an urgent need to guarantee their correctness. Moreover, there are software systems that are "*mission and safety critical and thus need to be highly dependable*" [36]. Even though people have learned to live with failures in software systems [62], there is the need to ensure they do not happen.

Since software involves a construction process, it needs to be guaranteed that from requirements to implementation the software meets the required specifications. Getting closer to a correct software allows to have better confidence that the software program performs as expected, hence increasing its dependability. Deciding that software in general is correct, is unfeasible; nevertheless, several approaches have been defined to provide approximations to this ideal situation.

In order to guide the transformation of software from the user requirements to the final implementation, different Software Development Processes have been proposed. Each one embracing some methodology, methods, techniques and tools that will guide and help the developer produce different software products. One way of guaranteeing that such processes lead to correct software is to make the different steps and iterations and the different products produced as precise as possible. Formal methods have come in our way to help solve this problem. Formal methods, as stated by Wing [67], are mathematically based techniques that can be used for the systematically specification, construction and verification of software systems.

Formal methods can be applied in different situations throughout the development of a software program. Usually, some of the most critical parts of a system will make use of some formal methods technique. One area in formal methods that has been heavily studied in the last years is Software Design. Since design is usually at a higher level of abstraction and since its impact in the final implementation is big, it is ideal for formal modeling and analysis. In conclusion, Software Architecture plays a major role in Software Development.

Software Architecture Research has been one of the most active areas in software research and several ADLs (Architecture Description Languages) and tools have been proposed [55]. Yet, there is still a lot to be done, for there have been several weaknesses detected in software architecture research such as "*inadequate formal analysis of architecture designs and the lack of assurance of correct implementations of architecture designs*" [36]. A formal approach to architectural design and its realization to an implementation will help achieve the goal of correct software. Formal methods analysis techniques such as Model Checking ([44] and [19]) suffer from the state explosion problem, as a result, they can be complemented with Testing and Runtime Verification techniques.

Other important aspect to help devise the correct software is by making use of modeling and analysis tools and environments. For ADLs, several support tools have been developed, such as the tool environment for Acme, an ADL, from Carnegie Mellon University ([29]), among other efforts. Tools can automatically detect inconsistencies in the model, they can be used to generate intermediate models or implementation skeletons, they can also help translate a model in one language to another model in a different language, and they can provide analysis features.

By looking at all these past results, Hoare et. al. ([42] and [41]) presented the idea of the Grand Challenge, a program verifier that would do an automatic "*check of the correctness of the programs submitted to it*" [42]. This program checker will also be supported by other programs for constructing and analyzing software. The idea is to use all the achievements in terms of techniques, tools and theory that have been studied and that are being studied, and embark in this visionary project. Modeling and analysis tools and environments will play an important role in this vision.

Along these lines, He et. al. have proposed a "*Framework to ensure system dependability in SAM*" [36]. The Software Architecture Model (SAM) is a software architecture model based on Petri nets and Temporal Logic [37]. The elements in the framework can be seen in Figure 1.1 ([36]). This dissertation incorporates an approach for combining verification and testing, and a tool component for the modeling and analysis of Software Architectures in SAM.

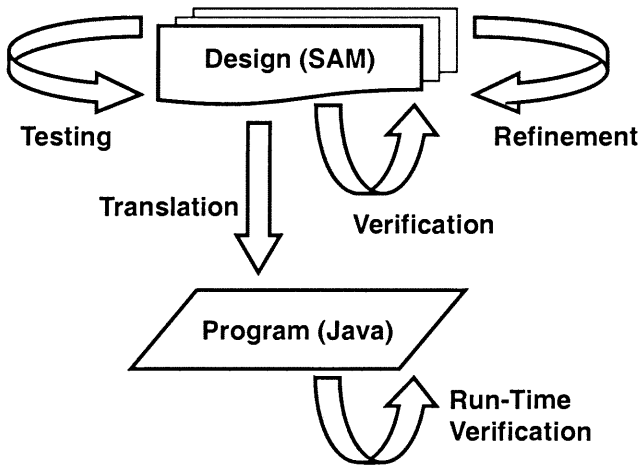

Figure 1.1: Framework to Ensure Dependability from Design to Implementation.

The framework based in SAM includes, first the modeling of an architectural design, and next its refinement, testing, formal verification, translation to other models

(e.g. Java code) and run-time verification. Two critical aspects of a framework are whether the framework is able to provide the elements to model a software system, and whether the artifacts developed within the framework are able to be analyzed. The framework in SAM contains those attributes. An architectural model in SAM consists of components and connectors organized in hierarchies, i.e. a connector or component can be refined into another layer of components and connectors. The behavior of a component or connector is expressed in Petri Nets and properties are defined in First Order Linear Temporal Logic. These elements do not only allow a SAM architecture to be precisely defined but they also lend it to be formally analyzed.

1.1 Motivation

As it was previously discussed, software correctness is a major challenge in software development. One critical aspect in software development is software architecture design, because it has a profound impact in the final software product. As defined in [8] Software Architecture encompasses the structure or structures of a system, containing software elements that are related and exhibit external visible properties. The goal is then to assure that those software elements are correctly defined and exhibit the desired properties and are in concordance with the specification of the software. One way to achieve this is by defining a formal architecture of the system and proving its correctness with respect to desired properties. SAM is an ADL that enables the formal definition of software architectures and allows it to be analyzed by various techniques. SAM exhibits a dynamic behavior, defined by the underlying Petri nets, and this dynamic behavior lends itself to the verification and testing processes.

In addition, the implementation of a system depends on the architectural design, and usually the implementation of the system is tested for potential bugs. Thus, it is

important to relate the architecture of the system to the implementation so that the information at the design level can be used in the testing process at the implementation level.

On another hand, there are several well-known proven techniques for the analysis of software products. One such technique is model checking. Model checking explores the state space of a finite state transition system and verifies the model against properties of interest expressed in Linear Temporal Logic. Spin ([44]) is one example of a tool that implements the model checking technique and it is widely used. As a result, one important aspect in software research is the integration of such techniques and tools into the software architecture research.

Finally, tools and environments for modeling and analysis of software products have proven useful in software development. Hence, for software architectures specified in a given ADL, the availability of support tools not only helps in the ease of design and analysis of an architecture but also in the adoption of the tool and the ADL as useful instruments for software development.

1.2 Research Problems

There has been extensive work in Software Architecture, both in terms of modeling and analysis. ADLs have been proposed and tools for supporting them have been implemented.

The main problems investigated in this dissertation are the formal verification and testing of architectural models. More specifically, the formal verification and testing of SAM models:

- Verification of SAM models. There have been studies done in the verification of SAM models and other studies in the verification of Petri nets ([70], [37] and [40]). But there is yet an approach to formally analyze SAM architectures using model checking in terms of the model checker Spin, this includes how to formally prove the translation from SAM to a model in the model checker, and how to deal with first order LTL when the model checker supports propositional LTL.

- Testing of SAM models. There has also been studies done in the testing of SAM and PrT net models ([72] and [16]). Most of that work set the theoretical grounds for future testing approaches for PrT nets. However, a practical approach to select and measure the adequacy criteria of test cases for complex systems is still missing. Finally, there is yet work to be done in deriving test cases for the implementation based on the verification and testing procedures at the design level in a SAM model. This is an important aspect of software development, given the high impact of design in the final implementation.

Other problem investigated in this dissertation is related to support tools for the modeling and analysis of SAM models. One important element is the reuse of proven technologies not only at the theoretical level, but also at the practical one. For the purposes of a tool support for SAM, there is no graphical interface that allowed the modeling and analysis of architectures in SAM. Most of the modeling in SAM was done by hand in text files and the generation or translation to other models was also manually performed. As models become more complex, they become unmanageable and keeping the consistency between different elements becomes harder. Hence, the chances of getting the wrong design increase.

1.3 Approach Proposed

This dissertation proposes an integrated Formal Verification and Testing approach for Software Architectures defined in SAM. SAM can use any kind of Petri Net, and for this work Predicate Transition Nets (PrT nets) are used. For the Formal Verification of SAM Architectures, the approach translates a SAM model to the input language of the model checker Spin. For the testing approach a procedure to test SAM models and to evaluate the adequacy of test cases at the design level is defined.

1.4 Contributions

The integrated approach combining formal verification and testing has the following contributions:

- The verification of SAM models using model checking. As part of this component, a formal approach for translating a SAM model to a PROMELA program, the input for Spin, was defined.

- A testing approach for testing a SAM model at the Component, Integration and System Level. The testing approach focuses on evaluating the adequacy of test cases in transition oriented and state oriented testing for PrT nets. Additionally, a test case generation based on model checking and on test cases at the design level were studied and defined.

- A modeling and analysis tool to design and analyze SAM models. This tool includes facilities for syntactic verification of First Order Logic formulas and also contains a semiautomatic translation module from a SAM model to PROMELA code. Manual intervention is necessary for the translation of some complex properties.

1.5 Scope and Limitations

There are several formalisms for describing software architectures. In this dissertation the software architectures are limited to architectural models defined within the SAM framework.

Given that a SAM model can have an infinite state space, for the formal verification approach, the SAM models to be analyzed are restricted to finite states ones. This is important since model checking can not handle infinite state space systems. The kinds of properties dealt with in verification are safety as well as liveness; however, for a semiautomatic translation the kinds of properties are also restricted to simple forms. Additionally, the First Order Temporal Logic formulas that define the properties for SAM models are also reduced to propositional ones in the verification process.

1.6 Outline of the Dissertation

The rest of the dissertation is organized as follows. First, the background and related work are discussed on Chapter 2. Chapter 3 presents the verification and testing framework briefly, showing the main elements in the approach and how they relate. Then on Chapter 4, the verification approach is explained in detail, with emphasis in the translation procedure from a SAM model to a PROMELA program. Next on Chapter 5, the testing approach is presented and several concepts are defined for the testing procedure; and also testing at the design and at the implementation level are discussed. On Chapter 6 the most relevant parts of the tool environment are presented. Case studies are detailed on Chapter 7. Finally, the dissertation concludes in Chapter 8 where the conclusions and future work are stated.

CHAPTER 2

LITERATURE REVIEW

The background and the related work are provided in this Chapter. The background introduces three main subjects: architectural design in SAM, formal verification in Spin and testing theory of high level Petri nets. Meanwhile, the related work introduces research that has been done and that is being performed in the areas of testing and verification of architectural models and Petri nets.

2.1 Background

2.1.1 SAM (Software Architecture Model) Framework

This section is based on "*A Framework for Ensuring System Dependability from Design to Implementation*" presented in [36] and "*A Framework for Developing and Analyzing Software Architecture Specifications in SAM*" presented in [37].

SAM is a software architecture framework for specifying and analyzing software architectures. Petri Nets ([57],[59]) and Temporal Logic ([51]) are the underlying formalisms that provide the foundations for SAM. Petri nets are used to describe the behavioral models of components and connectors, while first order temporal logic is used to specify system properties of components and connectors. There are different kinds of Petri nets, from low level to high level Petri nets. These different kinds of Petri Nets can be used as the underlying behavioral model for a SAM architecture. In this dissertation Predicate Transitions Nets (PrT nets) are the kind of Petri nets used.

There are several ADLs proposed in the literature. In [55], Medvidovic e.t al., a study on different ADLs is presented, showing the basics characteristics an ADL

should posses. An ADL needs to present at least three elements in its definition, the notions of Components, Connectors and Ports. SAM is an ADL that defines those elements as part of its specification.

Briefly, a SAM architecture model is defined as a set of compositions, representing different design levels. Each composition is comprised of a set of components, connectors and composition constraints, and each component (or connector) is composed of two elements, a behavior model and a property specification. A component or a connector can be further refined by defining a mapping relation to a composition. Some of these aspects can be observed in Figure 2.1 which shows a graphical view of a simple SAM architecture model. Predicate Transition nets (PrT nets) are used to define the behavior of components and connectors and Linear Temporal Logic (LTL) is used to specify properties for components and connectors ([37] and [40]). The next paragraphs describe SAM and the underlying formalisms in more detail.

SAM model. A SAM model consists of a set of compositions $C = \{C_1, C_2, ..., C_k\}$ with a top level composition $C_l \in C$ representing the top level design. Each composition $C_i = (Cm_i, Cn_i, Cs_i)$ consists of a set of Cm_i components, a set Cn_i of connectors, and a set Cs_i of composition constraints. A component or connector $C_{ij} \in Cm_i \cup Cn_i$ is non-elementary if it is refined by a lower level composition in C; otherwise, it is elementary. Each $C_{ij} = (S_{ij}, B_{ij})$ has a property specification S_{ij} and a behavior model B_{ij}. First Order Linear Temporal Logic and Predicate Transition Nets are used to define the properties and the behavior respectively. The property specification and behavior model for a non-elementary C_{ij} is obtained by merging the behaviors and specifications of the components and connectors of the composition mapped to it (see [37]). In the verification approach followed in this dissertation, each non-elementary component/connector is replaced by the corresponding components

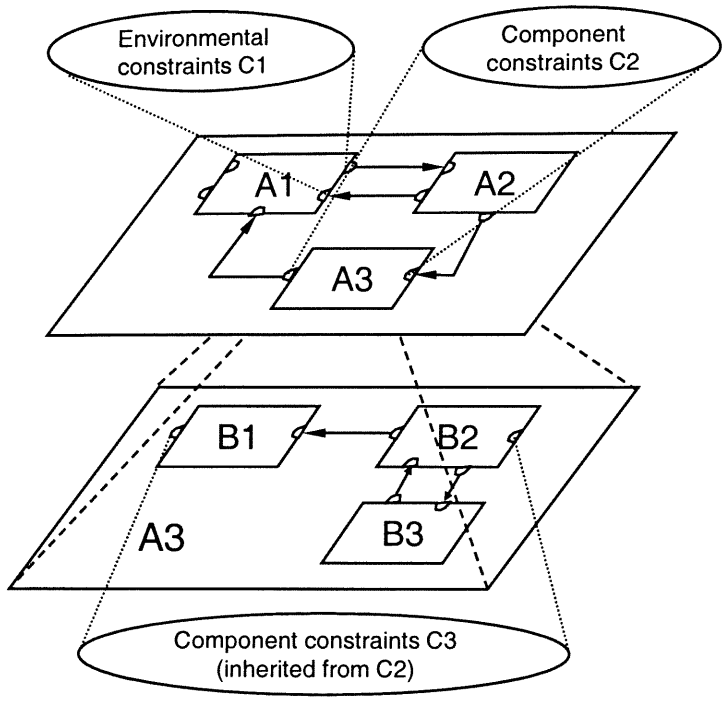

Figure 2.1: SAM Software Architecture.

and connectors in the composition it maps to; as a result, the top level composition will only contain elementary components and connectors.

Predicate Transition Nets (PrT Nets). A PrT Net ([30] and [35]) consists of a net structure (P, T, F), an algebraic specification (S, Op, Eq) and a net inscription (φ, L, R, M_0). The most important aspects to note are that each $p \in P$, where P is the set of predicates (places), is mapped to a sort $s \in S$ $(\varphi(p) = s)$ and contains tokens that are ground terms of its corresponding sort s. T is the set of transitions and R defines for each transition $t \in T$ its precondition and postcondition expressed as first order logic formulas. The arcs in F connect places and transitions, and have labels defined by L which are used in the pre and post conditions in transitions. A transition is enabled if there is a substitution for the variables in the incoming arcs that satisfies its precondition. The substitution is achieved by assigning tokens in

the corresponding place to each variable. A transition is fired if it is enabled, and the postcondition is then satisfied. Finally M_0 represent the initial marking, i.e. the initial tokens contained in the places. In the next paragraphs, PrT nets are formally explained.

Formally, a PrT net consists of: a finite net structure (P, T, F), an algebraic specification $SPEC$, and a net inscription (φ, L, R, M_0). P and T are the set of predicates and transitions, respectively, where $P \cap T = \emptyset$. F is the flow relation where $F \subseteq P \times T \cup T \times P$. $SPEC$ is a meta-language to define the tokens, labels, and constraints of a PrT net. The underlying specification $SPEC = (S, OP, Eq)$ consists of a signature $\Sigma = (S, OP)$ and a set Eq of Σ-equations. S is a set of sorts and OP is a family of sorted operations. Tokens of a PrT net are ground terms of the signature Σ, written $MCON_S$. The set of labels is denoted by $Label_S(X)$, where X is the set of sorted variables disjoint with OP. Each label can be a multiset expression of the form $\{k_1x_1, ..., k_nx_n\}$. Constraints of a PrT net are a subset of first order logic formulas containing the S-terms of sort *bool* over X, denoted as $Term_{OP,bool}(X)$.

The net inscription (φ, L, R, M_0) associates each graphical symbol of the net structure (P, T, F) with an entity in the underlying $SPEC$, defining the static semantics of a PrT net. Each predicate (place) in a PrT net is a data structure and a component of the overall system state. Mapping $\varphi : P \rightarrow \wp(S)$ assigns a subset of sorts to each predicate p in P, which defines its valid values, i.e. proper tokens. Mapping $L : F \rightarrow Label_S(X)$ is a sort-respecting labeling of flows. Mapping $R : T \rightarrow Term_{OP,bool}(X)$ associates each transition t in T with a constraint expressed in a first order logic formula in the underlying algebraic specification. The constraints define pre-conditions and post-conditions for transitions. The pre-condition specifies

the constraints on the incoming arcs and the post-conditions specify the relationships between the variables of the incoming arcs and label variables of the outgoing arcs.

A marking m of a PrT net is a mapping $P \rightarrow MCON_S$ from the set of predicates to multi-sets of tokens. M_0 is the initial marking. A transition is enabled if its pre-set contains enough tokens and its constraint is satisfied with an occurrence mode. The pre-set (${}^\bullet t$) for a transition are the set of input places for that transition. Similarly, the post-set ($t^\bullet$) for a transition are the set of output places for that transition. The firing of an enabled transition consumes the tokens in the pre-set and produces tokens in the post-set. Two transitions (including the same transition with two different occurrence modes) fire concurrently if they are not in conflict; however, in this dissertation, interleaving semantics is assumed. Conflicts are resolved non-deterministically. The firing of an enabled transition is atomic. The behavior of a PrT net is defined as the set of all possible execution sequences E.

Each execution sequence $e \in E$ represents reachable markings from the initial marking, in which a successor marking is obtained through a step (firing of some enabled transitions) from the predecessor marking. An execution is denoted as:

$$e : m_0 \xrightarrow{n_0} m_1 \xrightarrow{n_1} m_2 \xrightarrow{n_2} \ldots \xrightarrow{n_{k-1}} m_k \xrightarrow{n_k} \ldots$$

Where n_i is a set of transitions, M_0 is the initial marking, $m_i, i = 1, 2, \ldots$, are markings such that m_i is obtained from m_{i-1} by firing transition set n_i. The execution sequence e is said to be *flat* if all the n_i's are singletons, otherwise e is said to be *non-flat*. A flat execution sequence can be obtained from a non-flat execution sequence by interleaving the transitions in the non-singleton n_i's. In this dissertation the interleaving flat form of this execution is adopted.

In [72], Zhu and He define M_0 not as the initial marking, but rather as the set of initial markings. This provides useful for testing purposes in their work. However, in this dissertation, M_0 is considered as the initial marking for a PrT net unless stated otherwise.

Figure 2.2 shows the major elements of the PrT net model for the "five dining philosopher problem".

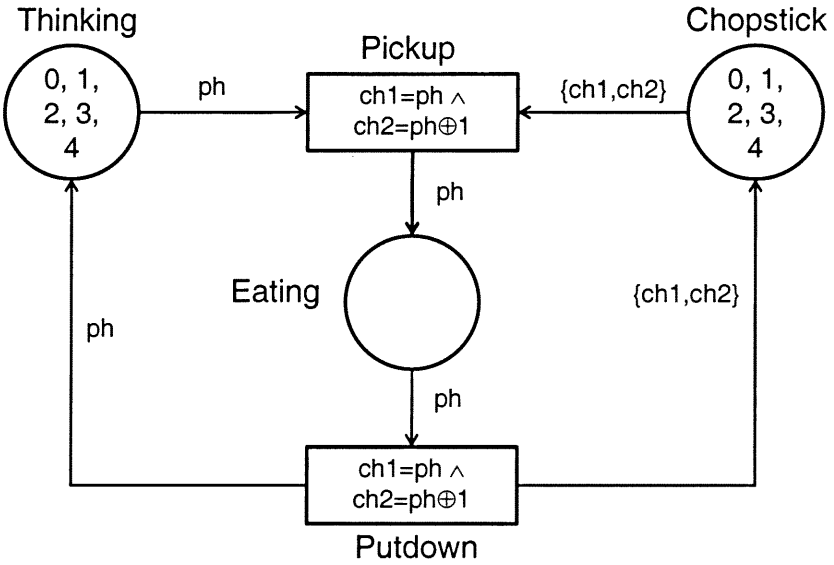

Figure 2.2: PrT net for the five dining philosopher problem.

The model in figure 2.2 consists of three predicates - *Thinking*, *Eating* and *Chopsticks*, and two transitions - *Pickup* and *Putdown*. The flow relation includes variables *ph*, identifying a philosopher, and variables *ch*1 and *ch*2, identifying chopsticks. Operator $\oplus$ defines the modulus 5 operator. The precondition for *Pickup* and the postcondition for *Putdown* are shown in the figure. The ground terms for the sorts of predicates *Thinking*, *Eating* and *Chopstick* are integer numbers in the range [0..4]. When transition *Pickup* fires, one token from *Thinking* and two tokens from *Chopstick* are consumed, and one token representing the philosopher is placed in predicate *Eating*. When *Putdown* fires, one token is consumed and three tokens, one going to

predicate *Thinking* and two going to predicate *Chopstick* are produced. An alternative way of defining the sort of predicate *Eating* is by declaring it as a structured sort, so that tuples of the form $< ph, ch1, ch2 >$ can be stored in *Eating* (each tuple representing the philosopher and the chopsticks being used while eating).

Linear Temporal Logic (LTL). LTL [51] has been widely used to specify properties for software systems. Within the SAM framework, property specifications for components and connectors, and constraints for compositions, are defined in first order LTL. A first order LTL formula contains predicates as terms and can contain universal quantifiers. Due to the fact that the model checker Spin verifies properties defined in propositional LTL, in the approach presented in this dissertation the properties and constraints for SAM models are modified so that the LTL verification power of Spin can be applied to them.

LTL is applied to PrT nets and Petri net concepts are included in its definition. A standard linear time temporal logic (LTL), is a function that maps each time point into the set of propositions that hold at that point. Classic linear temporal logic provides, in addition to the propositional logic operators, the temporal operators $\Box$ (always), $\diamond$(eventually), U (until) and $\circ$ (next). A LTL formula, $\Box(p \rightarrow \diamond q)$ means that the situation that p is true implies eventually q is true, always holds. The semantics of temporal logic is defined on behaviors (infinite sequences of states). The behaviors are obtained from the execution sequences of Petri nets where the last marking of a finite execution sequence is repeated infinitely many times at the end of execution sequence. For example, for an execution sequence $M_0, ..., M_n$, the following behavior $\sigma = \ll M_0, ..., M_n, M_n, ... \gg$ is obtained, where M_i, $i \in [0..n]$, is a marking of the Petri net.

Let $\sigma = \ll M_0, M_1, ... \gg$ be the behavior. The semantics of a temporal formula p in behavior σ and position j is denoted by $(\sigma, j) \models p$:

- For a state formula p, $(\sigma, j) \models p \Leftrightarrow M_j \models p$;
- $(\sigma, j) \models \neg p \Leftrightarrow (\sigma, j)\neg \models p$;
- $(\sigma, j) \models p \vee q \Leftrightarrow (\sigma, j) \models p$ or $(\sigma, j) \models q$;
- $(\sigma, j) \models \Box p \Leftrightarrow (\sigma, i) \models p$ for all $i \geq j$;
- $(\sigma, j) \models \Diamond p \Leftrightarrow (\sigma, i) \models p$ for some $i \geq j$;
- $(\sigma, j) \models pUq \Leftrightarrow \exists i \geq j : (\sigma, i) \models q$, and $\forall j \leq k < i, (\sigma, k) \models p$.

2.1.2 Software Verification

The goal of Software Verification is to examine whether a software system satisfies the specified functional requirements. There are two fundamental approaches to verification: Testing and Formal Verification. Software testing alone cannot prove that a system does not have a certain defect (nor the opposite). Only Formal Verification can prove that a system does not have a certain defect or does have a certain property. One of the most widely used techniques in formal verification is Model Checking. Model checking has the benefit that it can be automated, and this gives it an advantage over Theorem Proving, other formal verification technique, for practical purposes ([33]). However there has been work in combining these two approaches, such as the works in [66] and [10].

A technique that lies in-between testing and formal verification is runtime verification ([15], [34]). It is considered as a light-weight formal method. The difference with respect to testing is that in runtime verification monitoring facilities are added

to the program to observe its behavior during normal operations. During the program execution faults are detected and actions are taken to mitigate those faults. In [18] runtime verification techniques were applied to the SAM framework.

Model Checking Model checking is a method for formally verifying finite-state concurrent systems. Specifications about the system are expressed as temporal logic formulas, and algorithms are used to traverse state space of the model defined by the system and check if the specification holds or not. In [19] states that extremely large state-spaces can often be traversed in minutes during model checking; hence, it provides a good choice for the analysis of concurrent systems.

Model checking techniques face a combinatorial blown up of the state-space, commonly known as the state explosion problem. There are several approaches to cope with this problem, they include symbolic model checking ([12], [54] which makes use of binary decision diagrams ([11]), partial order reduction ([58]) and abstraction. Usually, the models to verify using model checking are hardware designs, but finite state software models are also suitable for model checking. If the state space is too big or infinite, abstraction can be used in order to reduce that state space and make the model checking feasible.

Formally, the model checking problem is stated as follows: given a desired property, expressed as a temporal logic formula p, and a model M, decide if $M \models p$. If M is finite model checking reduces to a graph search.

One of the most widely used model checking tools is Spin Model checker [44], and it is the tool of choice for this dissertation.

Spin model checker. Spin ([44] and [43]) is a model checking tool to formally analyze the logical consistency of distributed systems, which are expressed using the language PROMELA. Spin has three basic roles: (1) As an exhaustive state space analyzer for rigorously proving the validity of user-specified correctness requirements. (2) As a system simulator for rapid prototyping. (3) As a bit-state space analyzer that can validate large protocol systems.

PROMELA is the modeling language for Spin. It has a *C* programming language style. A PROMELA program consists of processes, message channels, and variables. Processes are global objects. Message channels and variables can be declared either globally or locally within a process. Processes specify behavior, channels and global variables define the environment in which processes run. Spin is used in the verification stage as well as the simulation stage in this dissertation. Spin has a simulation capability that is used to drive the coverage evaluation of test cases for PrT nets.

Spin offers three options for performing simulation these include: (1) random, (2) interactive, and (3) guided. The simulation mode in Spin is intended primarily for the debugging of a model. The random simulation option allows a user to monitor the behavior of a model by printing any output produced by the model to the console. Interactive simulation allows a user to resolve non-deterministic choices during the simulation of the model by selecting an option during the simulation process. If there is only one option then Spin selects that option and continues the simulation. Guided simulation uses a specially encoded trail file generated by the verifier, after a correctness violation, to guide the search. The execution sequence stored in the trail file represents the events leading up to the error. This dissertation makes use of the simulation capability of Spin for coverage measurement in the testing of PrT nets.

2.1.3 Software Testing

There are several studies done in Software Testing ([7] and [47]). The main difference between software testing and model checking is that software testing is selective (it will select specific execution paths); whereas, model checking is exhaustive (it will explore the whole state space). This is the reason why model checking is more expensive than testing.

One aspect of testing is measuring the coverage and adequacy of a test set. Beizer [9] defines test coverage as any metric of completeness with respect to a test selection criterion; meanwhile, Zhu et al. [71] explain the notion of test data adequacy criteria by providing three definitions of test data adequacy criteria: (1) test data adequacy as stopping rules, (2) test data adequacy criteria as measurements, and (3) test data adequacy criteria as generators. In [72] they define an observational schema for testing high-level Petri nets. This dissertation is based on this last work.

Testing PrT nets. PrT nets can play two different roles in the development of concurrent systems: as a formal specification, and as an executable model ([72]). These two roles provide the developer with the opportunity to combine both verification and testing of the PrT net model, providing a higher level of confidence in the correctness of the system. The properties of a PrT net allow the application of both specification-based and program-based testing techniques.

In [72] a methodology of testing high-level PrT nets was developed. Four classes of testing strategies were identified: *transition-oriented testing, state-oriented testing, flow-oriented testing*, and *specification-oriented testing.* For each strategy, a set of schemes to observe and record testing results and a set of coverage criteria to

measure test adequacy were defined. The concept of an *observational scheme* for a concurrent system p was defined as the ordered pair $< B, \mu >$ where B is the set of partial orders of events generated by p, and μ represents the mapping from a test set to a non-empty consistent subset of all partial orders for p. Due to non-determinism and concurrency, two or more partial orders may be generated by the same test input for a given p. Unlike test data adequacy criteria (used to measure the adequacy of a test set), an observation scheme determines how to observe and record a system's dynamic behavior during test executions. This dissertation considers State and Transition Oriented Testing which are discussed below.

Transition coverage (Transition-oriented testing) It is the ratio of transitions fired during an execution of a PrT net to the total number of transitions.

$$TransitionCoverage(N, E) = \frac{\left| \bigcup_{e \in E} Firing(e) \right|}{|T_N|}$$

Where $Firing(e) = \bigcup_{i=0,1,...n_i}$, $e = m_0 \xrightarrow{n_0} m_1 \xrightarrow{n_1} m_2 \xrightarrow{n_2} ... \xrightarrow{n_{k-1}} m_k \xrightarrow{n_k} ...$, and n_i are non-empty subsets of transitions that are not in conflict with each other. T_N is the set of transitions of N. Note that a $Trace$ of an execution e is defined as $Trace(e) = < n_0, n_1, ...n_k, ... >$. For this dissertation the flattened execution sequence is assumed.

K-concurrency length-L trace coverage over a collection of executions E of a PrT net is defined as the existence of at least one $e \in E$ covered by the transition trace q with length less than or equal to L and concurrency degree less than or equal to K (K and L are natural numbers greater than zero). The concurrency degree of a transition trace is the maximum number of transitions that may fire between any successive markings in that trace.

Interleaving length-L transition sequence coverage over a collection of executions E of a PrT net is defined as: given any feasible transition sequence q with length less than or equal to L there is at least one $e \in E$ the logically covers q. A execution e logically covers sequence q if a flattening of e contains q as a consecutive subsequence of transition firings. To obtain the best results it is better to chose the longest possible feasible transition sequence q since the longer sequence coverage subsumes the shorter sequence coverage.

All transition trace coverage requires that there is at least on $e \in E$ that covers any feasible transition trace q. In general, achieving all transition trace coverage is unfeasible.

State coverage (State-oriented testing) It is the ratio of of the reachable markings associated with abstract states during an execution to the finite set of abstract states for the PrT net. An *abstract state* (AS) is one of a finite set of states that is reachable in a PrT net given an initial marking:

$$StateCoverage(N, E) = \frac{\left| State_N \left(\bigcup_{e \in E} Markings(e) \right) \right|}{|AS_N|}$$

Where $State_N : Mark(N) \rightarrow AS_N$ defines how markings are associated with states. AS_N is a finite set of abstract states of N.

State transition coverage over a collection E of test executions is satisfied, if for all feasible state transitions $< s_1, s_2 >$ there is at least one execution e in E such that e covers that state transition.

State transition path coverage. State transition path coverage, more specifically *length-k state transition path coverage*, is defined over a set of execution E and is satisfied, if and only if for any feasible state transition path q of length less than or equal to k there is an execution e in E such that e covers path q. Assume k is a natural number greater than 1. A state transition path of length k is defined as a sequence of states $< s_1, s_2, ..., s_k >$. In general, it is not practical to handle state transition path coverage, however, the restriction of specifying a length of the path makes it more practical.

2.2 Related Work

2.2.1 Model Checking and Software Architecture

The advantage of model checking over theorem proving is that the former can be automated while the latter needs manual intervention ([3]). There have been some previous works in applying model checking to software specification, such as the "system for alert and collision avoidance system for air traffic control" ([14]). In order to accomplish such effort, abstraction had to be used. Within the SAM framework, a formal approach to analyze software architectures and the application of symbolic model checking was presented in [38]. Other effort at applying model checking to large systems is the work by Wing et. al. [68] which depicts a case study of the application of the model checking technique to software systems using the SMV ([52]) model checker.

2.2.2 Model Checking and Petri nets

There has been work done in model checking low level Petri Nets [21] mainly using the unfolding technique developed by McMillan ([53]). Also, there has been some work done in model checking high level Petri nets as the work by Schroter et.

al. [60] for parallel model checking based on unfoldings. In [26], Spin was used to model check low level Petri nets. Other efforts in model checking Petri nets include the work in [49] for model checking modular Petri nets and [48] for model checking high level Petri nets. In [32] Spin is integrated into the PEP (Programming Environment based on Petri nets [31]) tool for the verification of Petri net models.

2.2.3 Test Case Generation

Several studies for the generation of test cases have been undertaken. To generate inputs for test cases, even for complex systems, is straight-forward, but the oracles are the elements that pose challenges [3]. One major goal in testing is the automatic generation of test cases.

Test generation based on temporal properties has been a subject of research in the past years ([20], [45] and [24]). Model Checkers have been used in the generation of test cases, because counterexamples from model checkers are potential useful test cases. For example, Ammann et.al. ([3], [2] and [4]) use the model checker SMV to generate test cases following a mutation analysis approach. Other research works using model checking include the efforts by Gargantini et. al. [27], where model checkers are used to generate tests from requirements specifications, and [28], where the model checker Spin is applied to the generation of test cases for ASM specifications. More recently the generation of test requirements for aspectual use cases based on aspect oriented Petri nets was studied in [69]. SMV performs a breadth first exploration of the state space, producing counterexamples that tend to be short ([3]).

In addition, there is work in combining Runtime Verification and test case generation, as is the work in [6]. In this work, the authors present a framework for the

automatic generation of test input and properties and the runtime verification of each of the properties on each of the inputs. Test input is generated based on the structure of the input and the input pre-conditions. Automatic generation of test input and properties are tailored specifically to the application to be studied. Once the input and the properties are defined (generated), the input is fed into the program to be verified, the program executes generating an execution trace. This execution trace is checked against the set of properties for that input. The first step, generating the input and properties is called the test case generation, and the second step is the runtime verification of each of the test cases and the properties relevant to it.

2.2.4 Property Oriented Testing

There is work done in property coverage criteria in software testing. For example the work by Tan et. al. [64]. That work was inspired the in initial work done in CTL property coverage in [2]. In [64], they realized that a property defined in LTL needs to hold in all possible paths, with the number of paths possibly infinite and each path also being an infinite sequence. They define a mutation testing approach to property in which, a property is mutated to another one and then test cases that identify these mutations as undesired behaviors are taken into account and provide the required coverage on the property being tested. In their work, Li Tan et. al. consider black-box and white-box testing, whereas in this dissertation only white box testing is defined. Another work, [45] uses a temporal approach for the generation of test cases. Other relevant works combining testing and temporal logic properties can be found in [23], [50].

Ammann and Black ([2]) concentrate on specifications, mutants of these specifications and test cases that cover them. Computational Tree Logic, CTL ([56]), is used

in their work. The system's transition relation is encoded in CTL and test cases derived from them are encoded in finite state machines that when executed will provide a measure of the coverage of the CTL formula mutants. In a more concrete manner, given an SMV model of the system, the transition relations in SMV are used to generate CTL formulas that express the given relations. This is called by the authors reflection, in the sense that the CTL formulas reflect the logic of the transitions relations. Applying diverse techniques and heuristics, mutants can be constructed out of this specification. Given a set of test cases (that can be automatically generated from the specification), those are encoded as constrained finite state machine (CFSM). The final step is when the test cases are executed (the CFMs are executed) and then they are measured in whether they were able to cover (satisfy or falsify) the mutants for the specification.

CHAPTER 3

VERIFICATION AND TESTING FRAMEWORK

This chapter introduces the Verification and Testing Approach for Architectural Models in SAM. Figure 3.1 shows the major components of the approach. Given a SAM model, a Verification process is applied to it, as well as a Testing one. Based on the information at the design level, both in terms of Verification and Testing, a Test Case Generation process produces test cases for the implementation level. In addition, Tool Support aids in the modeling and analysis of the SAM model.

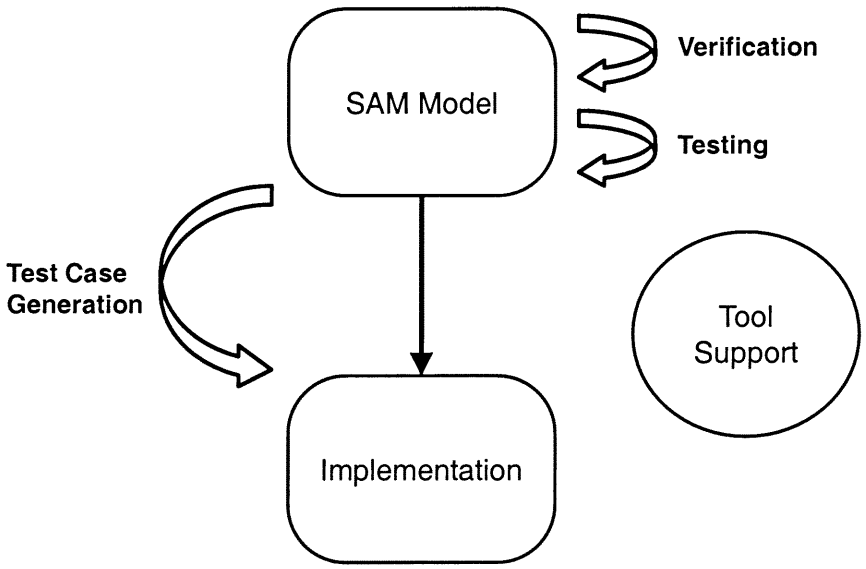

Figure 3.1: The verification and testing approach.

At the design level, the architecture of the system is expressed as a SAM model. A SAM model has a dynamic behavior, which results from the aggregation of the underlying behavioral models of its components and connectors. The behavioral models are defined in Petri nets and for the purposes of this dissertation the kind of Petri nets used are Predicate Transition Nets or PrT nets ([30] and [39]).

On one hand, for the verification process, a model checking approach is applied at the design level. In order to accomplish that, a restricted finite state form of a SAM model is defined. On the other hand, a PrT net is an executable model, and hence it can be tested as programs. As part of this dissertation, a testing approach at the design level is detailed with emphasis on the evaluation of test cases for PrT nets.

The design is related to the implementation by the generation of test cases based on the information at the design level resulting from the verification and testing processes. This last step is important in software development, since there is the need to reflect properties from the design into the implementation.

As systems become more complex, the need of modeling and analysis tools applied to their construction grows. For this dissertation, a modeling and analysis tool for SAM models was implemented and the case studies presented in this document were created and analyzed with the aid of the tool.

3.1 Verification

At the design level, a formal verification process is applied to a SAM model. This verification process translates a SAM model to a PROMELA program, and then uses Spin to verify properties of interest (PROMELA is the input language for Spin [44]). In order to accomplish the translation from a SAM model to a PROMELA program, the SAM model is restricted in terms of the state space of its underlying behavioral model. This behavioral model is expressed in PrT nets, and as such the PrT nets are restricted to be bounded and the sorts for the predicates (places) are also restricted to contain only finite number of ground terms. The properties for the SAM model are converted from first order linear temporal logic formulas to propositional linear

temporal logic formulas. This is required since the model checker Spin handles propositional linear temporal logic formulas.

With this approach, the support tool implements an automatic translation of the SAM model to a PROMELA program, except for the properties. The properties are semi-automatically translated and parts of them need to be handcrafted.

3.2 Testing

Testing at the design level involves the evaluation of the tests sets which are based on the initial marking of the PrT nets expressing the behavioral aspect of a SAM model. This evaluation is based on the theory of testing high level Petri nets in [72] and includes transition and state oriented testing.

Once test cases are evaluated, they can be used for testing the design itself, or for deriving test cases for the implementation level. In the last scenario, since the implementation has more detailed information, the test cases at the design level serve as abstract test cases, that need to be mapped to specific constructs in the implementation level. Besides test cases based on the initial marking of PrT nets, positive test cases are generated by model checking the negation of properties of interest.

In addition to testing the behavior itself, an approach to test the SAM model at the system, integration and unit levels is also introduced.

3.3 Tool support

In order to design and analyze complex scenarios, a tool support is required. This dissertation includes a tool support for the modeling and analysis of SAM models.

One of the main features of the tool is the automatic translation of a SAM model into a program written PROMELA, the input language of the model checker Spin.

The tool provides several modeling facilities. Two useful elements are the editors for transition constraints in the underlying PrT net models and for properties defined in first order linear temporal logic. With the aid of the editors, a syntactic check of the formulas can be done, leveraging the work of the designer.

In the next chapters, the verification approach, the testing approach and the tool support are detailed. Case studies showing the applicability of the framework and the tool are presented as well.

CHAPTER 4

VERIFICATION OF ARCHITECTURAL MODELS

Verifying that a Software Architecture complies with properties of interest is of primary importance. In this chapter the approach at formally verifying architectural models in SAM [37] using modeling checking [19] is detailed. The model checker of choice is Spin [44]. Predicate Transition Nets [39] (PrT nets) are used to define the underlying behavioral models of the SAM architecture. Hence, in order to apply model checking, restrictions are imposed to the kind of SAM and PrT net models and the properties to verify: SAM and PrT nets models have finite state spaces and properties in first order logic form are reduced to propositional ones. This approach corresponds to the Verification component of the "Verification and Testing Framework" proposed in Chapter 3.

To verify a SAM model, first the SAM model is translated to a program in PROMELA, the input language of Spin. Next, the PROMELA program is loaded in Spin and executed. If Spin finds any errors it generates counterexamples that are traced back to the original SAM model in order to detect design flaws. See Figure 4.1.

The main piece in the verification of SAM models relies in the formal translation of a SAM model to a program in PROMELA. The translation takes as input a SAM model and generates as output a PROMELA program. The translated PROMELA program needs to reflect all the elements in the SAM model (complete) and it has to preserve the semantics of the SAM model (consistent) for the translation approach to be considered correct. The completeness and consistency of the translation are shown here. Some of these aspects are discussed in [5].

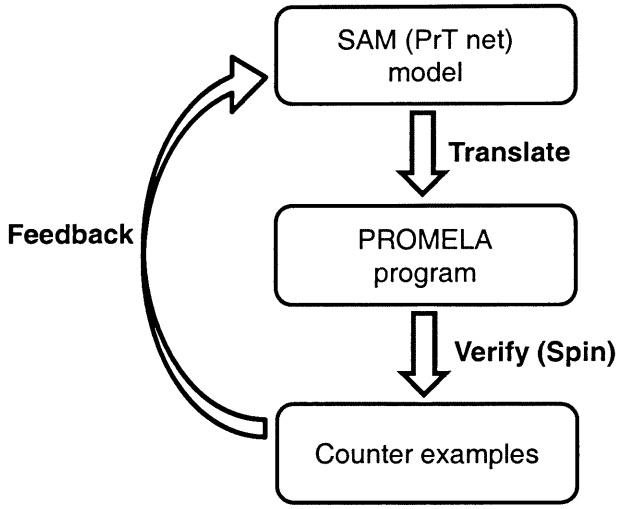

Figure 4.1: Verification of SAM models.

In the next sections, first the restrictions on SAM and PrT net models are detailed (4.1). These restrictions allow providing a sound translation approach . Next, the translation approach is defined and the mapping between elements in the SAM model and the PROMELA program (4.2) is detailed. In addition, a discussion on the correctness of the translation is also provided (4.3). A brief discussion describes alternative translation approaches to the one provided here (4.5). Finally, the summary of the chapter is presented (4.6).

4.1 Restricted SAM and PrT net models

There are some restrictions placed on the SAM and PrT net models in order to make them suitable to be analyzed by a model checker. The most important one is to define a finite state space SAM model, so as to make it feasible to convert it to a finite state PROMELA model without relying on abstraction. Additionally, the kinds of properties that can be verified are restricted. These restrictions also allow a semiautomatic translation of a SAM model to PROMELA and the corresponding verification process in Spin.

Restrictions on ports. A port can be mapped to multiple ports in the lower layers within the hierarchical composition view of a SAM model. A one to one mapping needs to be established and as a result the following restriction is defined:

One-to-one port mapping between compositions: Given two compositions C_i and C_k, where C_k is the refinement of one component or connector C_{ji} in C_i, a port in C_{ji} can only be mapped to one port in C_k.

The state space of a SAM model is defined by its behavioral model, a PrT net model; hence, the PrT net model needs to have a finite state space. Not only do the sorts are restricted but also the number of tokens each predicate (place) in the PrT net can have is set to have a limit. First the restrictions on sorts are discussed.

Restrictions on Sorts. There are four aspects to consider: (1) Finite number of ground terms, (2) real numbers, (3) strings and (4) derived sorts. Each one is elaborated in some detail below.

(1) *Finite number of ground terms.* The basic sorts available for the restricted version of PrT Nets are defined by:

$$s_{basic} ::= string|bit|bool|byte|short|int|unsigned$$

One requirement is that the number of ground terms for each basic type be finite. Each basic type is restricted to be the same as its PROMELA counterpart, except for type string. Sort *string* is mapped to a *short* integer type in PROMELA.

(2) *Real numbers.* Even though type *double* was not included as a basic sort, to work with finite real numbers they have to be mapped to integer values. It is assumed that this mapping is done explicitly in the model built.

(3) *Strings.* Sort *string*, which represents any string of finite length, is reduced to represent a limited number of strings. Type *string* in SAM is mapped to type *short* in PROMELA, and in the translation process each string ground term is mapped to a unique integer number in a sequential fashion (the strings encoding). The only operations available on strings are assignment and comparison, which can be realized in the mapped integer values. Other operations such as concatenation and sub strings, are not available (given that the encodings become cumbersome). Finally, this restriction also means that all the string constants in the model need to be defined explicitly; otherwise, they are not encoded.

(4) *Derived sorts.* The derived sorts are the Cross Product and the Powerset, so derived sorts of the form $s_1 \times s_2 ... \times s_n$ and $\wp(s)$, with $s, s_1, ..., s_n$ being sorts, are defined. Since the underlying basic sorts have finite ground terms, the resulting cross products and powersets are also finite. However, the number of ground terms might explode. For instance, for sort *short*, there are 65536 possible values for a variable of that type (if the size of an element of that type is 2 bytes). If the powerset is considered $\wp(short)$, now that accounts for 2^{65536} possible ground terms for that new type. This means that a limit needs to be set to the number of elements each subset can contain.

The set S of sorts for a PrT Net contains elements defined by:

$$s ::= s_{basic} | s(\times s)^+ | \wp(s)$$

Bounded Nets. Each $p \in P$, with P being the set of places in the behavioral model, is bounded.

If a designer builds an infinite state space SAM model, in order for the approach to work, it is assumed that prior to the translation to PROMELA the designer makes the explicit mapping to a finite state space by using abstraction and by directly defining bounded values for all the places in the behavioral model and mapping the basic sorts to basic PROMELA types.

Property Specification. The studies are restricted to a finite set of properties.

- *Liveness properties.* In order to specify properties of the form $\forall x \cdot (\Box(P1(x) \rightarrow \Diamond P2(x)))$, the SAM model does not have to contain cycles that may lead to an infinite sequence of repetitive states.

- *FOL-LTL formulas.* First Order Logic properties are instantiated, according to the initial marking, into Propositional Logic ones. So a formula of the style $\forall x \cdot (\Box(P1(x) \rightarrow \Diamond P2(x)))$, where $P1$ and $P2$ are two predicates and $\varphi(P1) = \varphi(P2) = short \times short$, and where the initial marking is $M_0(P1) = \{< 1,1 >, < 2,2 >\}$ and $M_0(P2) = \{\}$ yield the set of properties: $\{\Box(P1(< 1,1 >) \rightarrow \Diamond P2(< 1,1 >)), \Box(P1(< 2,2 >) \rightarrow \Diamond P2(< 2,2 >))\}$.

The properties studied in this dissertation were three liveness and one safety:

- *Forward*: $\forall x \cdot (\Box(P1(x) \rightarrow \Diamond P2(x)))$. A system that needs to forward a request can define this kind of property. If there is a token in $P1$ (the request arrives at $P1$) then eventually it will show up at $P2$ (the request will arrive at $P2$).

- *Transformational forward* $\forall x \exists y \cdot (\Box(P1(x) \rightarrow \Diamond(P2(y) \wedge y[1] = x[1])))$. This is more generic than the previous one. If there is a request that suffers transfor-

mations, the first element in the tuple can be marked as the id of the request, so that it is always known which request is being handled.

- *Transformational free form*: $\forall x \exists y \cdot (\Box(P1(x) \rightarrow \Diamond(P2(y) \wedge R1(x,y))))$, where $R1(x,y)$ is a relation between x and y. This is even more generic, in that for a token at some place, there is another one in another place, and there is a relation known before hand that relates the latter to the former.

- *Safety*: $\forall x \cdot (\Box((P1(x) \rightarrow R1(x))))$. This safety property states that if a given token is at a place $P1$ it complies with some restriction $R1$. In safety properties, usually, instead of wanting to observe the property is being satisfied, the desire is to observe it is not being complied with. For example, a token c that is in place $P1$ and where $R1(c)$ evaluates to false.

The set of SAM models is defined SAM. And the set of restricted SAM models is defined as R_SAM: $R\_SAM \subset SAM$.

4.2 Translation to SPIN

In this section, the approach at translating the restricted version of a SAM model to a PROMELA model is explained in detail. The first step is to create a flattened version of the SAM model, and the second one is to translate the flattened SAM model to a PROMELA program.

Flattened version of a SAM Model . Given a SAM Model with a set of compositions $C = \{C_1, C_2, ..., C_k\}$, set C is reduced to a set $C' = \{C_l\}$ containing only one composition. This is achieved by replacing each component or connector that is refined with the components and connectors in the composition refining it (a refined component or connector is called non-elementary component or connector re-

spectively). This means that given a non-elementary component or connector C_{ji} in composition C_i, which is refined by a composition C_k, the following steps are followed:

(1) C_{ji} is replaced by the components and connectors Cm_k and Cn_k in C_k.

(2) The property specification S_{ji} for C_{ji} is added to the set of constraints Cs_i for composition C_i.

(3) The set of constraints Cs_k in C_k is added to the set of constraints Cs_i for composition C_i.

In Figure 4.2, an example on the flattening of a SAM model can be seen.

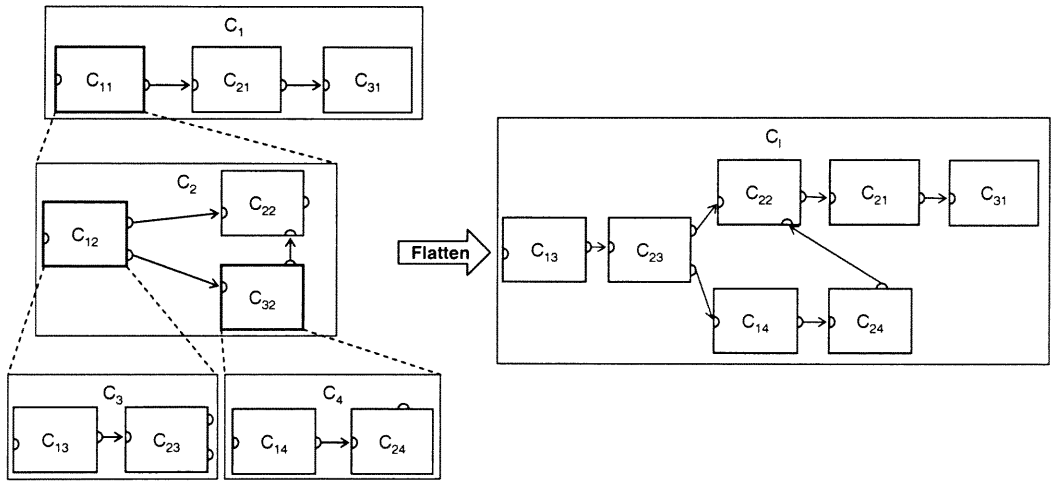

Figure 4.2: Flattening a SAM Model.

Having the flattened version of a SAM model, its integrated behavioral model is extracted next.

Integrated Behavioral Model. Given a SAM model consisting of only one composition $C_i = (Cm_i, Cn_i, Cs_i)$, its integrated behavioral model B_i is created by combining the behavioral models of each $C_{ji} \in Cm_i \cup Cn_i$ with $C_{ji} = (B_{ji}, S_{ji})$:

$$B_i = \bigcup_j B_{ji}$$

Once the flattened version of a SAM model is obtained, the actual translation process (mapping of elements in SAM to elements in PROMELA) is described next. The translation is divided in two parts, one called *Composition Translation* that takes into consideration all the elements except the behavioral models for components and connectors, and the other part referred to as *Behavior Translation* that considers the behavioral models (PrT nets). In the translated PROMELA program, the embedded *C* code feature of PROMELA ([44]) is used to define functions implementing various notions in the SAM model. For example, a function that adds tokens to a place and another function that fires a transition are defined.

Composition Translation

The composition translation is straightforward, a component is translated into a *proctype*. A component's specification is translated into a never claim with some restrictions. Also, the constraints for the top level composition is translated into a never claim. Later in this section, it is explained how the never claims are constructed.

Behavior Translation

A PrT Net consists of three elements: a finite net structure (P, T, F), an algebraic specification (S, Op, Eq) and a net inscription (φ, L, R, M_0). All of them need to be reflected in the target PROMELA code. The three elements are closely related, for example, when a reference to a predicate $p \in P$, means that there is the notion of its sort $(\varphi(p) \in S)$, and its initial marking $(M_0(p))$ among others. The behavior translation process is divided in three parts: sort translation, place translation and transition translation.

(a) **Sort Translation.** Sorts in a SAM model are defined the following way:

$$s ::= s_{basic} | s(\times s)^{+} | \wp(s)$$

$$s_{basic} ::= string | bit | bool | byte | short | int | unsigned$$

Basic sorts. Each basic sort, except sort *string*, has a PROMELA counterpart. The operations for those sorts are the ones available for any integer type and boolean type. Sort *string* is mapped to type *short* in PROMELA. For *string* sort only comparison and assignment operations are allowed, as it was discussed earlier in the chapter.

Cross product. A cross product sort $s = s_1 \times s_2 ... \times s_n$ is translated as:

```
typedef s{
  s1 field1;
  s2 field2;
  . . .
  sn fieldn;
};
```

Equality and assignment operations are defined for this sort using the PROMELA embedded *C* code feature. The prototypes for the functions are:

```
int is_equal_s(s *left, s *right){...}
void assign_s(s *left, s *right){...}
```

Powerset. A powerset sort $s = \wp(s_1)$ is translated as a structure containing a fixed-length array of objects of type s_1 and the number of elements in the array being used:

```
typedef s{
  int num;
  s1 set[max_n];
};
```

Constant *max_n* is the maximum number that a set belonging to the powerset can have. The operations defined for this sort are the usual set operations, plus equality and assignment. These operations are implemented using the embedded *C* code feature in PROMELA. For example, some of the prototypes of the functions implementing the operations are:

```
int is_equal_s(s *left, s *right){...}
void assign_s(s *left, s *right){...}
int count_s(s1 *pval, s *pset){...}
int in_set_s(s1 *pval, s *pset){...}
s set_union_s(s *left, s *right){...}
```

(b) **Place Translation.** Given a place $p \in P$ the following are defined in PROMELA:

Bounded Values. A symbol for the bound value for p:

```
#define BOUND_p maxval
```

Where *maxval* is the maximum number of tokens for that place. By default it is set to 10 ($maxval is 10$).

Tokens Storage. Let $s = \varphi(p)$ be the sort of place p. Two variables for dealing with the tokens at p are defined:

```
s v_p[BOUND_p];
short num_p;
```

Array *v_p* contains the tokens. The maximum number of tokens is given by *bound_p*. And *num_p* is the current number of elements to be considered in the array, i.e. the number of tokens at place p ($0 \leq num\_p \leq BOUND\_p$).

Initial Marking. The initial marking $M_0(p)$ is defined by creating a series of expressions in PROMELA language to assign values to each token and each

field the token has. For example, assuming that $\varphi(p)$ is a cross product of m basic sorts and the initial marking for p has n elements, the code generated is:

```
num_p = n;
v_p[0].field1 = val_1;
...
v_p[num_p-1].fieldm = val_nm;
```

Add and Remove Tokens. Two helper functions are implemented, one to add a token and the other to remove a token from place p:

```
void add_p(short *num_p, s *v_p, s *_tok)
void remove_p(short *num_p, s *v_p, short _idx)
```

Query Tokens in Place. A function to test whether a token is in a place or not is also defined:

```
int in_p(short num_p, s *v_p, s *_tok)
```

(c) **Transition Translation.** An overview of the two basic procedures for a transition, testing the enabledness and firing, is presented. Given a transition $t \in T$, the following constructs are defined in PROMELA:

Enabledness. Testing the enabledness of a transition t involves a substitution on all the variables in the incoming arcs with the corresponding tokens in the incoming places, and then testing the precondition on t. In other words, given ${}^\bullet t$, the set of incoming arcs for t, testing the enabledness of transition t involves for each $p_i \in {}^\bullet t$ a substitution on the variables in $L(p_i, t)$ with tokens in p_i and the testing the precondition in $R(t)$ with those substitutions. A function is written such that given the preset for transition t, it either returns 1 if there is a substitution that satisfies its precondition, or 0 otherwise:

```
int is_enabled_t(preset_t)
```

Firing. When t fires, tokens are removed from ${}^\bullet t$ and tokens are added to $t^\bullet$. The following function prototype is defined and implements the firing notion in C language:

```
void fire_t(preset_t , postset_t)
```

Universal Quantifiers. Functions are added to test the truth value of universally quantified terms whenever $R(t)$, for transition t, includes them. These functions are used in testing the enabledness of transition t. Their prototypes are as follows:

```
int forall_t(params)
int exists_t(params)
```

These functions return either 1, if the term evaluates to true, or 0 if not. The parameters *params* for these functions are lists of all the entities involved in evaluating the term.

Property and Constraint Translation. The set of constraints Cs_i for composition C_i and the property specification S_{ji} for each component/connector $C_{ji} \in C_i$, are expressed in first order LTL. Some aspects on how an LTL formula is translated to PROMELA code are briefly mentioned. Given an first order logic LTL formula f, all predicates (places) are instantiated, each predicate term Pi in f generates a macro:

```
#define Pi c_expr{TokenAtPi()}
```

The body of function $TokenAtPi()$ reflects an instantiation of the predicate to a propositional formula. It returns an integer value of 1 or 0, depending on whether a specific token or set of tokens can be found there, this is defined by the person verifying the model. There are a few approaches on how to select the ground terms for instantiating the predicate, the approach selected in this dissertation considers the

tokens in the initial marking as the base values. Next, the *never* claim is generated by making use of the property automaton generator facility available in Spin. For instance, for liveness property $\forall x \cdot (\Box(P1(x) \rightarrow \Diamond P2(x)))$ the following is generated:

```
#define P1 c_expr{TokenAtP1()}
#define P2 c_expr{TokenAtP2()}
never {    /* !([](P1-><>P2)) */
T0_init:
 if
 :: (! ((P2)) && (P1)) -> goto accept_S4
 :: (1) -> goto T0_init
 fi;
accept_S4:
 if
 :: (! ((P2))) -> goto accept_S4
 fi;
}
```

For this example, if function $TokenAtP1()$ evaluates whether a token c is at place $P1$ or not, then function $TokenAtP2()$ has to evaluate whether the same token c is at place $P2$. If the property holds, these two functions will return 1 at different times (first $TokenAtP1()$ and next $TokenAtP2()$).

PROMELA model for a SAM Model. The previous paragraphs described how each part in a SAM model is translated to a PROMELA construct, but did not mention how they combine together. A brief explanation on how the different parts are combined by looking at the structure of the resulting PROMELA program is presented. Below, each number represents a section in the PROMELA code and the order in which its appears:

(1) Defines all the constant symbols identifying the bounded values for places.

(2) Defines the structured sorts, i.e. cross product and powerset types.

(3) Sorts operations are defined for structured types in embedded *C* code sections.

(4) The port places variable are defined next. They have to be defined globally, since two components that communicate share the same place at the specified port, and they need access to it.

(5) A synchronization global variable is also defined to keep track of the number of component processes that are initialized. Each process updates this synchronization variable whenever it has finished setting up the initial marking of its corresponding PrT net. Only when all the components have defined the initial markings of their corresponding PrT nets, the firing of transitions can begin.

(6) Embedded *C* code with functions to add tokens to places and to remove tokens from places.

(7) The transitions embedded *C* code for testing their enabledness and executing the firing are defined.

(8) The *proctype* definition for each component is defined. This *proctype* defines the variables for the places of its behavioral model that are not related to ports. The places related to ports are defined globally. Next, code for setting the initial marking is added, and then a synchronization point is established to wait until all the other processes have established their initial marking. At last, an infinite loop is created that tests for the enabledness of its transitions and fires transitions that are enabled.

(9) An *init* process available in PROMELA is implemented, the initial marking for places related to ports is established there. It also starts executing all the other processes (the components processes).

(10) Finally, the never claim for the desired property to check is defined. It includes the automaton code generated for the property of interest by the Spin.

In table in 4.3 a summary of the translation is shown. For each section a PROMELA code outline sample and its corresponding relation to a SAM model are presented.

Section	PROMELA code example	Relation to SAM
1	#define BOUND_P1 maxP1	Bound value for place P1.

2	typedef PSET{
	  string set[max];
	  short num;
	};
		Definition of powerset sort PSET to be used to define sets of strings.

3	c_code{
	  int is_equal_PSET(PSET *l, PSET *r){
	  // return 1 if equal, 0 otherwise
	  }
	}
		Operations on sorts. Here, equality testing for two elements of type PSET is defined.

4	PSET v_Port1[BOUND_Port1] short num_Port1	Port1 is a place related to a port.
5	int _proc_num;	The number of processes that have already been initialized.

6	c_code{
	  void add_P1(...){ ... }
	  void remove_P1(...){ ... }
	}
		Add/remove tokens to/from P1 when a transition having P1 as part of its pre/post set fires.

7	c_code{
	  int is_enabled_T1(...){ ... }
	  void fire_T1(...){ ... }
	}
		Testing the enabledness and firing a transition T1.

8	proctype Comp{
	  // initial marking
	  // wait for the other procs to initialize
	  do
	  :: atomic{ c_expr{is_enabled_T1(...)}
	  ->c_code{fire_T1(...)}}
	  od
	}
		Component Comp. Initial marking: Comp process initializes its state, waits for the other processes to initialize. Behavior execution: after initialization, Comp fires its transitions if they are enabled.

9	init{
	  // initial marking for port places
	  // initialize synchronization constants
	  atomic{
	  run Comp();
	  }
	}
		Initial marking: sets the initial marking for places acting as ports. Executes the Component processes for them to start the execution of the Petri nets.

10	#define P1 c_expr{TokenAtP1()}
	never{
	  // LTL automaton definition
	}
		The property to be verified.

Figure 4.3: Overview of the sections in the PROMELA code.

A graphical example of the translation approach showing some of these aspects can be seen in Figure 4.4.

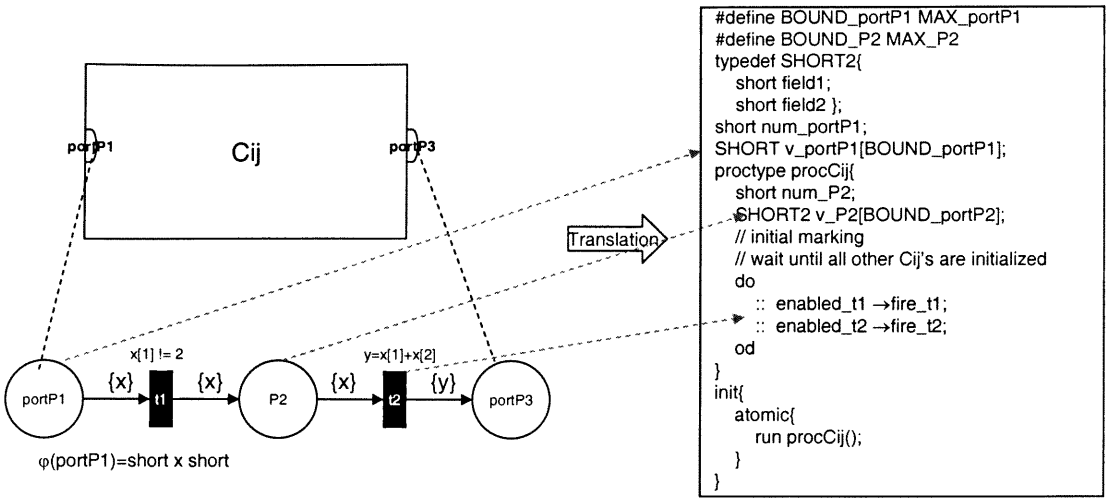

Figure 4.4: A simple outline of the translation process.

4.3 Translation Correctness

This section provides a discussion on the correctness of the translation approach from SAM to PROMELA. It does so by demonstrating the completeness and the consistency of the translation.

Interleaving semantics *Both for the Behavioral model in SAM and for the translated PROMELA model, the interleaving execution semantics is chosen.*

Since there is no true concurrency in PROMELA, this is an important observation that will allow the two models to be compared. For an interesting discussion on interleaving semantics w.r.t true concurrency refer to [44].

Claim 1 ***(Flattened version correctness)*** *The flattened version of a SAM model respects the original model's behavior and specification.*

Proof *Follows directly from the flattening procedure.*

The different compositions can be thought as being different ways of partitioning a system. In the end all those partitions are for the same underlying behavioral model, and the partitioning that is picked is one with a flat structure, i.e. with a single composition.

Completeness

Claim 2 ***(Completeness of the translation)*** *Given a restricted SAM Model, there exists a corresponding PROMELA model that defines all of its elements.*

Proof *Follows directly from the mappings.*

In the translation process, each of the elements in the SAM model is translated into a PROMELA construct. Two elements to pay special attention to, are the property specification and the initial marking. For the first one, it was mentioned how to convert a predicate into a proposition so that properties can be verified in Spin. For the second one, it was explained how to define a series of expressions to build the initial marking for each component/connector and how all the components/connectors are synchronized before actually executing the related code for testing the enabledness and firing of transitions.

Consistency

Claim 3 ***(Initial Marking consistency)*** *Given a restricted SAM model* sam *and its translated PROMELA program* prom*, the initial marking in the underlying behavioral models in* sam *is consistent with the state* prom *is previous to the point where the processes test and fire the translated transitions relations.*

Proof *In SAM model* sam, *the initial marking of its components and connectors is defined as part of the model itself. In PROMELA program* prom, *there is a series of steps that make the variables related to the places take the initial values, and no process executes the enabledness testing and firing of transitions before every other one has initialized its variables. In* prom, *there is a synchronization step before the* do::...::od *main loop in each process. This synchronization step waits for a global variable to reach the number of processes that have been initialized:*

(init_procs == num_procs)

Hence, the initial marking in sam *corresponds to the state in* prom *previous to which each component/connector process starts to execute the enabledness testing and firing of transitions.*

A single step in a PrT net execution can correspond to multiple ones in the translated PROMELA program. For instance, when firing a transition, there are multiple instructions that need to be combined to realize it. This dissertation introduces the notion of abstract execution of a translated PROMELA program.

Translated PROMELA program abstract execution sequence *After initialization, when the initial marking of the model is set, the only executable instructions in the processes are the ones within the do..od construct:*

```
proctype proc_ij(){
 do
  ::atomic{is_enabled_t1 -> fire_t1}
  ::atomic{is_enabled_t2 -> fire_t2}
  ...
  ::atomic{is_enabled_tn -> fire_tn}
 od
}
```

An abstract execution is a sequence $\sigma_0 fire\_t_1 \sigma_1 fire\_t_2 ...$, *where a change of state occurs only when an executable statement* $fire\_t \in \{fire\_t_1, fire\_t_2, ..., fire\_t_n\}$ *to the right of the arrow is executed.*

The enabledness testing and firing of transitions in the translated PROMELA program are atomic constructs; as a result, the firing of transitions in the PROMELA program has the same meaning as the firing of transitions in the PrT model.

Claim 4 ***(Semantic Consistency between a SAM model and its translated PROMELA program)*** *A SAM model* sam *is semantically consistent with its translated PROMELA program* prom, *iff for every execution sequence in* sam *there is a corresponding abstract execution in* prom.

Proof *Follows directly from the definition of abstract execution for a PROMELA program.*

Since in the translated PROMELA program there is a finer granularity, for instance, when firing a transition there are multiple instructions that need to be combined to realize it, this dissertation works on the abstract version of an execution sequence in PROMELA. This abstract sequence contains atomic aggregates of sub steps that correspond to single steps in the PrT net model. When firing a transition the sub steps are part of an aggregate atomic construct, they are uninterrupted and hence can be seen as a single step.

4.4 Automatic Translation

The preciseness of the translation approach from SAM to PROMELA allowed the implementation of a semiautomatic translation procedure as part of the SAM

modeling and analysis tool depicted in Chapter 6. One part that can not be directly automated is the property specification. The kinds of properties that have been studied are safety and liveness (from those guarantee ones are considered), which have general skeletons in terms of the verification automata in Spin. The translation procedure in the tool starts by encoding the internal SAM object structure to XML format. Then the XML structure is translated to PROMELA code. In the tool, class *SAM2SpinTranslator* implements this translation, the top level code of the translation is outlined below:

```
public class SAM2SpinTranslator {
 ... // other definitions
 public String translate(SAMML model){
  // Get the XML version of the SAM model
  XMLExport export = new XMLExport(model);
  String origXml = export.getXML();
  // Reduce the XML version of the SAM model to a unique composition.
  SAMXMLTransformer trans = new SAMXMLTransformer();
  Document doc = trans.reduceToSingleComposition2(origXml);
  Node nodeSAM = doc.getFirstChild();
  // Create the PROMELAModel object
  PROMELAModel prom = new PROMELAModel();
  // Now proceed to transform it.
  // Generate the constants
  generateCons(nodeSAM,prom);
  // Generate the sorts
  generateSorts(nodeSAM,prom);
  // Generate the operations on complex sorts (assignment,equality)
  generateSortsOps(nodeSAM,prom);
  // Generate the global ports
  generatePorts(nodeSAM,prom);
```

```
    // Generate the global vars
    generateGlob(nodeSAM,prom);
    // Generate the transitions
    generateTrans(nodeSAM,prom);
    // Generate the components/connectors processes
    generateProcs(nodeSAM,prom);
    // Generate the initial process
    generateInitProc(nodeSAM,prom);
    //generate the Property
    generateProp(nodeSAM,prom);
    return prom.toString();
  }
  ... // other definitions
}
```

4.5 Discussion

A SAM model encompasses three aspects: *Structure* represented by components and connectors within compositions, *Behavior* described by PrT nets and *Properties* expressed in first order Linear Temporal Logic. The translation mapping presented in this Chapter describes how each of those aspects are translated to PROMELA constructs. This subsection provides some of the alternative ways of doing this translation. It also discusses some aspects of the verification of FO-LTL properties in Spin.

Non-flattened Composition Translation. A non-flattened SAM model m, consists of multiple levels of compositions. One way to translate m to PROMELA is to define each component/connector as a process, preserving the hierarchical structure. A consequence of this is that the process for a refined component will contain other

processes, the processes for the components and connectors in the refining composition. As a result, a flattened version is preferred.

Single integrated behavioral model. An integrated behavioral model for a SAM model can be defined. This leads to another approach in which a single process is defined for the whole system, and the places (and ports) are globally visible. The notion of components and connectors disappear at this level. This can provide useful if there is a need to measure transition coverage and state coverage for the whole system in terms of testing. For purposes of proving the correctness of this translation approach, a similar methodology as explained in this Chapter can be followed.

Transition as process. Each transition in the PrT net model can be defined as a process. Given that interleaving semantics for the execution of the PROMELA code is assumed, the same effect is obtained as if the transition code is part of a process. If the transition is picked to be fired, it will fire without interruption (atomic construct). One reason for not choosing this alternative is that Spin limits the number of processes that can be run, so if there is a model with several transitions, the available processes might get exhausted.

Transition enabling and firing construction. Two important aspects of the translation procedure are how to compute the enabledness and to execute the firing for a transition. When testing the enabledness of a transition t, another approach is to compute all the substitutions for the variables in the incoming arcs. Next the substitutions that enable the transition can be computed and from these enabling substitutions, one transition can be randomly chosen for the actual firing. For each $p \in {}^\bullet t$, a matrix of indexes with $|L(p,t)|$ number of columns and $\frac{|p|}{|p|-|L(p,t)|}$ number of rows is created. Finally, a random selection process is executed on the matrices

corresponding to the incoming places for a transition. Experiments regarding this approach were not performed.

First Order LTL Expansion. A formula can be expanded to include all possible substitutions for the predicates involved in it. For example, given a formula $\forall x \cdot (\Box P1(x) \rightarrow \Diamond P2(x))$, where $P1$ and $P2$ are two predicates and $\varphi(P1) = \varphi(P2) = short \times short$, the complete enumeration is obtained:

$$\Box P1(< 0,0 >) \rightarrow \Diamond P2(< 0,0 >)) \wedge ... \wedge \Box P1(< 0,n >) \rightarrow \Diamond P2(< 0,n >))$$

...

$$\Box P1(< n,0 >) \rightarrow \Diamond P2(< n,0 >)) \wedge ... \wedge \Box P1(< n,n >) \rightarrow \Diamond P2(< n,n >))$$

Where n is the number of constants in *short* type. Overall there are n^2 formulas in the enumeration. The bigger the domain the bigger the formula enumeration. This provides impractical for model checking.

4.6 Summary

This chapter presented a formal approach for verifying that an architectural model in SAM satisfies properties defined in First Order Linear Temporal Logic. To that end, first restrictions on the kinds of SAM models that the approach can be applied to were presented. Next, a translation procedure was detailed in which elements in a SAM model map to elements (constructs) in a PROMELA program. For the translation to be considered correct, the resulting PROMELA code has to reflect every element in the SAM model (completeness) and it has to preserve the semantics of the model (consistency). A proof discussion on the consistency and the completeness of the translation was provided. A quick introduction to the automatic translation of a SAM model to PROMELA was also presented. Finally some alternatives to the translation part of the approach were discussed.

CHAPTER 5

TESTING OF ARCHITECTURAL MODELS

An architectural model that exhibits a dynamic behavior can be analyzed in terms of testing. A SAM architecture defines the behavior of a system in terms of PrT nets. The main focus of this chapter is to introduce the testing process for an architectural SAM model, i.e. design testing. Additionally, it discusses how the design itself is used to guide the testing of the implementation, i.e. implementation testing. Figure 5.1 shows these two kinds of testing.

In design testing, the SAM model is translated to an ArchJava program, which is executed and tested (see [1] for ArchJava details). One interest is how to select test cases based on the design for the implementation. At the implementation level, the design has a corresponding more detailed implementation written in a high level programming language, such as C# and Java, among others. This implementation can be executed and tested based on the information obtained from the design.

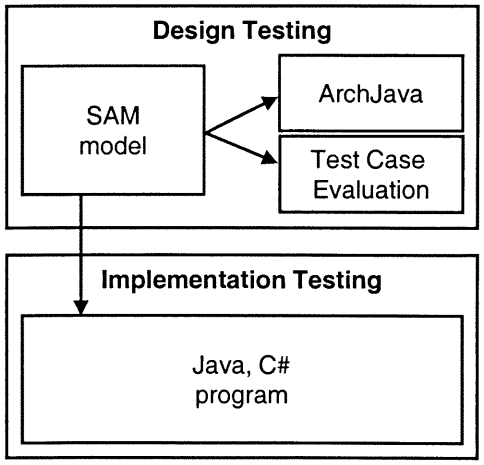

Figure 5.1: Design and Implementation based testing of SAM models.

One aspect of the testing process is the definition of test cases and coverage and adequacy criteria [62]. This dissertation introduces the notions of test cases and coverage and adequacy criteria for design and implementation based testing. In addition, there are levels of testing: unit testing, integration testing and system testing. Those levels are detailed in this chapter as well.

Design Testing

In order to be tested, the element under test has to exhibit a dynamic behavior, so that it can be executed and evaluated. This means that for testing the design of a system, the design needs to include a dynamic behavior resulting from a behavioral model. A SAM design model includes such a behavioral model. The underlying behavioral model of a SAM model is the result of the behavioral model of its components. Each component's behavioral model is expressed as a PrT net. One goal is to test this underlying behavioral model.

There are two elements related to testing the design in SAM, (1) test case selection, and (2) execution and testing of the design itself.

One on hand, for test case selection the interest is in measuring the adequacy of test sets that can be used to test the design and also that can be used at the implementation level. Since the behavioral aspect of SAM designs relies in PrT nets, then it is of primary interest the application of the testing theories presented in [72].

On the other hand, testing of the SAM model itself is achieved by translating the model to an ArchJava program and executing it under a test scenario. ArchJava

includes architectural concepts that makes it amenable for the implementation of architectures. There are previous works at translating a SAM model to ArchJava (see [25]); in consequence, the translation itself is not elaborated in this dissertation. A testing approach for SAM models is critical, specially in cases where model checking is not possible or presents too many restrictions. In addition, a SAM model that was model checked, can be still tested by loosening the restrictions enforced upon it, and the results in the model checking stage can be applied to drive the testing process.

Implementation Testing

A SAM model represents the architecture of the system. The final implementation of the system, which can be done in any high level programming language such as Java, C# or C++, needs to reflect the properties of the architecture. To this end, the interest is in generating test cases based on the architecture that can be applied at the implementation level. Note that there is a difference in the ArchJava program and the final implementation of the system: the final implementation of the system contains much more implementation details than the one provided in the ArchJava program for the SAM model.

In this work, the derivation of test cases for the implementation based on the design is achieved by using: (1) model checking, and (2) the test cases at the design level. In the first case, Model checking is applied to negated properties at the design level, this generates counterexamples (execution sequences) representing desired behaviors that the testing phase in the implementation is interested in observing. In the second case, test cases used at the design level can be reused by translating them to test cases in the implementation.

Before dwelling into the details of the testing approach for architectural models in SAM, some important concepts are discussed in the next section.

5.1 Concepts

This section describes definitions introduced as part of this dissertation. It also incorporates previously defined concepts, specially the ones related to testing coverage for PrT nets in [72].

5.1.1 Test Cases and Petri nets

A Predicate Transition Net (PrT net) is an executable model and as such it can be executed and tested as programs ([72]). In addition, a PrT net can serve as the model for an implementation and it can be used in the testing process for the implementation. Towards a theory of testing PrT nets, in [72] Zhu and He devised a theory for observing the dynamic behavior during testing and evaluating test cases for PrT nets. They defined M_0 as a set of initial markings instead of a single marking, mainly because they have the complete input domain explicitly ([72]); hence, each marking in M_0 constitutes a test case. Given the non-deterministic nature of Petri nets, a test case can be applied multiple times leading to different observations and coverage, so they also define a test set as a multiset of test cases.

This dissertation not only uses the initial marking but also the execution sequences to define test cases for PrT nets. The initial marking is used when the testing process is based upon random executions of the PrT net. Meanwhile, the execution sequences, which include firing sequences, are used when desired paths need to be observed and when the testing process can control the execution of the system under test. The next paragraphs discuss these aspects.

Random execution of a PrT net (non-guided execution)

For testing involving the random execution of a PrT net, the same initial marking can lead to different execution sequences when applied multiple times. Petri nets are concurrent by nature and the execution paths can not be predicted before hand. Following on the notion that M_0 is the initial marking of a PrT net, rather than a set of initial markings (as defined in [72]), M_0 can be seen in two ways:

- As the test input: the initial marking is a test case.
- As the input domain: test cases can be derived by selecting tokens in the initial marking.

Although the semantics of concurrent programs cannot be defined as a partial function from inputs to outputs ([72] and [73]), it is still possible to introduce the notion of input and output for a test case in the testing process for PrT nets. The test input is defined as the initial marking or a subset of it, this also marks the point from where the testing process starts, and the output is defined as the set of final markings where the testing process ends. This yields the generic form of a test case for a PrT net:

Definition 5.1.1. (PrT net test case) Given a PrT net model N, a test case tc for N is a tuple:

$$tc = (M_0, tc_{input}, tc_{output})$$

Where M_0 is the initial marking for N, tc_{input} is the input for the test case and tc_{output} is its output. If M_0 is the input domain, then $tc_{input} \subset M_0$; otherwise, $tc_{input} = M_0$. The output tc_{output} is the set of markings reachable from M_0 up to where the behavior of the system under test is observed.

When testing the implementation of a PrT net, if the input tc_{input} is a subset of M_0, $M_0 - tc_{input}$ defines the set of tokens that are not part of the execution and that can be removed. Since the execution of a PrT net under testing can be stopped at different times, the output tc_{output} is defined as a set of markings reachable from the initial one and marks the points up to where the PrT net was executed under the test case.

The following definition of a test set for Petri nets (definition 5.1.2) corresponds to the notion of test set that introduced in [72]:

Definition 5.1.2. (PrT net test set) Given a PrT net model N, a test set is defined as the multiset:

$$ts = \{(tc, n)\}$$

Where tc is a test case and $n \in \mathbf{N}$ is the number of times the test case is executed in the testing procedure.

Controlled execution of a PrT net (guided execution)

Two challenges in testing concurrent systems are (1) observing the dynamic behavior of the system during testing and more importantly (2) replaying scenarios of interest, as mentioned by Carver and Tai ([13]). In unit testing, the element being tested is executed under a controlled scenario, and as part of the controlled scenario, a replaying facility can be introduced in the module being tested. For Petri nets, the replaying facility translates to being able to control the firing of transitions, as opposed to the original non-deterministic transition firing.

This is of particular interest when testing the implementation of a Petri net against temporal logic properties. At the design level, model checking can be used to generate execution sequences that lead to desired behaviors, e.g. comply with liveness properties. These execution sequences are abstract execution sequences from the point of view of the implementation (a mapping relation between the executions in the implementation and executions in the design is defined). Next, at the implementation level, the component can be guided to execute following those abstract execution sequences. If during the execution, after each step (transition firing), the resulting state in the implementation maps to the corresponding abstract state in the design, then it can be concluded that the component satisfies the property for the specific scenario.

To this end, a test case that includes information on what input is required and what output is expected at every step, is required. This dissertation defines a test case with execution sequence for that matter:

Definition 5.1.3. (PrT net test case with sequence) Given a PrT net model N, a test case tc for N is a tuple:

$$tc = (M_0, tc_{input}, tc_{output})$$

Where M_0 is the initial marking for N, and tc_{input} and tc_{output} are the test case input and test case output respectively. The test case input tc_{input} contains a finite sequence of n transitions (t_{seq}) and a finite sequence of n substitutions (α_{seq}):

$$\begin{aligned} tc_{input} &= (t_{seq}, \alpha_{seq}) \\ t_{seq} &= t_1, t_2, ..., t_n \\ \alpha_{seq} &= \alpha_1, \alpha_2, ..., \alpha_n \end{aligned}$$

The test case output (oracle) tc_{output} contains a finite sequence of n markings:

$$tc_{output} = M_1, M_2, ..., M_n$$

The initial marking M_0 and the elements in tc_{input} and tc_{output} combine so that $M_0\,[t_1/\alpha_1\rangle\,M_1\,[t_2/\alpha_2\rangle\,M_2 ...\,[t_n\alpha_n\rangle\,M_n$ is a valid execution sequence of N.

In the next paragraphs the relationships among these elements are explained in more detail.

Initial State: The initial marking M_0 defines the initial state of the system. For testing purposes, not all the information contained in the initial marking may be used. As mentioned before, M_0 can serve both as part of a test case or as the whole input domain. For this specific controlled scenario it is assumed to be the whole input domain.

Test input: The test input is given by the transition sequence t_{seq} and the substitution sequence α_{seq}. The transition sequence consists of consecutive transitions that fire one after the other, i.e., given sequence $t_{seq} = t_1, t_2, ..., t_n$, transition t_{i+1} fires after t_i ($t_i \in [1..n]$). Each transition t_i in t_{seq} is enabled under substitution α_i in α_{seq} and marking M_{i-1}. Each α_i instantiates typed label variables in $L(p, t_i)$ where $p \in^\bullet t$.

Test output: The oracle values are represented by the markings in the sequence tc_{output}. Each M_i in tc_{output} results from the firing of transition t_i under substitution α_i and marking M_{i-1} ($i \in [1..n]$). It is concluded that M_i is reachable from M_0.

This definition of test case is used later in this chapter when deriving test cases from design to implementation.

5.1.2 Coverage and Adequacy Criteria for PrT nets

In their paper "A methodology of testing high-level Petri nets", Zhu and He ([72]) described a process for testing high-level Petri nets. They applied the work in [73] and investigated testing strategies by using various test adequacy criteria and behavior observation schemes. Two of the strategies and test adequacy criteria described in that work are applied in this dissertation: transition oriented testing (transition coverage) and state-oriented testing (state coverage). In addition to using two of the already defined techniques for testing PrT nets, this dissertation extends the previous works by discussing Property Oriented Testing applied to PrT nets.

The notion of execution sequence, which is the base for a behavior observation scheme, is introduced:

Execution sequence: Given a PrT net N, its initial marking M_0 and its set of transitions T, an execution sequence is:

$$e : M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} ...M_n \xrightarrow{t_n} ...$$

With $t_i \in T$. This is known as interleaving execution sequence of a PrT net.

Transition Oriented Testing

In transition oriented testing, transitions are observed and recorded during the execution of the PrT net. With the transitions that fired recorded, the adequacy of the test is evaluated according to the transitions covered during its execution. Transition coverage is defined next.

Transition Coverage. Given a PrT net N, the set T of transitions for N, the set of executions E of N, $Firing(e)$ the set of transitions in T that were observed during execution e ($e \in E$ being an execution):

$$TransitionCoverage(N, E) = \frac{\left| \bigcup_{e \in E} Firing(e) \right|}{|T|}$$

$TransitionCoverage(N, E)$ equals 1 when all the transitions are observed (covered) during the executions in E. More details on transition coverage and other transition-related coverage can be found in [72].

State Oriented Testing

In state oriented testing, states are observed during execution and the test adequacy is evaluated according to states covered. For testing purposes, instead of working with markings as states, abstract states are defined. For example, in the dining philosophers problem, instead of tracing whether each specific philosopher is eating or thinking, two abstract states can be defined to trace whether any philosopher is eating or nobody is eating. Next, state coverage is defined.

State Coverage. Given a PrT net N, the set of abstract states AS_N for N, the relation $State_N(m)$ that maps marking m in the reachable markings of N from M_0 to an abstract state $AS_i \in AS_N$ and the relation $Markings(e)$ that defines the set of markings observed during execution $e \in E$:

$$StateCoverage(N, E) = \frac{\left| State_N \left(\bigcup_{e \in E} Markings(e) \right) \right|}{|AS_N|}$$

Similarly to transition coverage metric, the value of 1 for state coverage means that all abstract states where observed during the executions in E. A More detailed discussion on state coverage is available in [72].

Property Oriented Testing

Two kinds of properties are dealt with in this dissertation: liveness and safety (refer to Chapter 4 for a discussion on the form of the properties studied). In general, for safety properties the interest is in finding sequences that lead to undesired behavior; meanwhile, for liveness properties the interest is in finding desired execution sequences. Transition Oriented Testing and State Oriented Testing are more suitable for safety properties, in which the intention is to cover the transitions and states while still satisfying given properties. However, for liveness properties, those coverage criteria do not provide suitable: transitions and states might be covered but that does not necessarily mean that the property has been satisfied. In this section property oriented testing is discussed and its applicability to SAM models and PrT nets is explored as well.

A property for a PrT net is defined as a FO-LTL (First Order LTL) formula; thus, for testing, the expansion of the formula to propositional LTL ones is required. The expansion of a FOL-LTL formula F is defined as:

$$Expansion(F) = \bigwedge_{i=1..n} F_i, \ (F_i \text{ being an instantiation of } F)$$

Each F_i is a propositional LTL with the variables substituted by the corresponding values in their domains. Depending on the domain of the variables in F, the expansion can lead to an infinite number of elements $n \rightarrow \infty$ for infinite state space systems.

The set of expanded terms is:

$$Expanded(F) = \{F_i | F_i \text{ is an instantiated form of } F.\}$$

In testing, only part of the whole behavior of a system is observed, and as such, the possibly infinite property expansion is reduced to a finite one. In general terms, to achieve this expansion is a combinatorial process. Nevertheless, in practical terms, the expansion is highly controlled by the domain of the variables, the sorts of the predicates in the PrT net and the initial marking. For example, for the five dining philosopher problem, the property that eventually every philosopher who is thinking gets a chance to eat, is expanded as:

$$\bigwedge_{i=1..5} \Box(Thinking(i) \rightarrow \diamond Eating(i))$$

There has been work related to Property Oriented Testing, both in generation of test cases based on temporal properties (e.g. [50] and [22]) and testing of the property itself (e.g. [64] and [2]). Their work is basically focused on propositional LTLs, and it can be applied to the propositional formulas resulting from the expansion of a first order LTL.

There is a difference in how safety and liveness properties are handled in terms of testing, and this is something that needs to be explored.

Safety Properties. Given the following safety property for the five dining philosopher problem "two adjacent philosophers can not be eating at the same time":

$$\forall i \cdot \Box(\neg(Eating(i) \wedge Eating(i \oplus 1)))$$

The formula is expanded as:

$$\Box(\neg(Eating(0) \wedge Eating(1))) \wedge$$
$$\Box(\neg(Eating(1) \wedge Eating(2))) \wedge$$
$$\ldots$$
$$\Box(\neg(Eating(4) \wedge Eating(0)))$$

In testing, the interest is looking for a state in which any pair ($Eating(i)$,$Eating(i \oplus 1)$) for $i \in [1..4]$ is true, hence violating the property. There are several approaches on how to evaluate test cases, especially based on program mutants. In this dissertation, there is not an elaboration on specific techniques for testing safety properties. Given a test case for transition or state oriented testing, the test cases are executed and the expanded formula is evaluated at each time. One alternative looked at was the testing of the negation of the safety property which results in a liveness property.

Liveness Properties. A liveness property is one which the testing procedure needs to comply with, i.e. every execution in the testing process needs to satisfy the property. The form of the liveness properties that was handled is: $\forall x \cdot \Box(P1(x) \rightarrow \Diamond P2(x))$.

For example, in the previous sections, a liveness property for the five dining philosopher problem was detailed, as well as its expansion. Obtaining a test set that observes all the propositional properties in the expanded formula can provide unfeasible. As a result, the testing process needs to look at a specific subset of the expansion or at individual terms instead. For example, there might be the need to observe that philosopher 1 while thinking eventually gets the chance to eat: $\Box(Thinking(1) \rightarrow \Diamond Eating(1))$.

In order to observe this, there has to be a sequence of states leading from the philosopher thinking to the philosopher eating. It can be observed that the basic sequence involves three abstract states: $S_0 = \neg Thinking(1)$, $S_1 = Thinking(1)$ and $S_2 = \neg Eating(1)$. Thus, with the markings in the five dining philosopher problem mapped to those three abstract states, an adequate test set needs to observe the state sequence $S = S_0S_1S_2$. In [72], they define a test coverage criteria for state transition path coverage that could be applied here.

Following on the previous example, for each formula $f \in Expanded(F)$ for PrT net N, a set of abstract states AS_{fN} and the sequence $ASeq_{fN}$ satisfying it are defined during the testing process. An execution trace $MarkingTrace(e)$, containing the sequence of markings in e, maps to a sequence of states in AS_{fN}. If the sequence is the same as $ASeq_{fN}$ then the trace satisfies the property $(MarkingTrace(e) \vDash f)$.

Introducing a new coverage criteria based on the previous observation is not possible, because the search for an execution path that satisfies a property is undecidable.

5.1.3 Test Cases and SAM models

At the composition level, a SAM model consists of components and connectors sharing ports. So whenever, a token is observed at an input/output port, the related components/connectors are said to be interacting. The testing effort at the composition level relies in observing the interactions between different components. A test case for a composition is defined by the tokens at the input places and tokens at output places. Since a port maps to a place in the underlying behavioral model, each state for a component is an abstract state which is defined by the input and output ports (places) it is related to.

5.1.4 Coverage and Adequacy Criteria for SAM models

Because ports are mapped to predicates (places) in the petri nets, state coverage can be applied at this level.

Port Coverage. For a composition C a test set E needs to observe tokens at all the ports. This coverage is just an be extension to the state coverage defined for PrT nets, the abstract states are defined based on the tokens available at the ports.

Property coverage can be studied at this level, given that ports directly relate to properties of the component and the model itself. Refer to 5.1.1 for a detailed discussion on property coverage for PrT nets.

5.2 Levels of Testing for a SAM model

The hierarchical structure of a SAM model lends it to be tested at different levels, namely: (1) unit testing, (2) integration testing and (3) system testing. These levels of testing are applied at both the design and the implementation.

Figure 5.2 shows the three levels of testing for a SAM model. First, *unit testing* corresponds to testing an elementary component (one that is not refined further). Next, *integration testing* is applied to compositions (composed of elementary components and refined components). Finally, *system testing* encompasses testing the top level composition.

The procedure defined for testing a SAM model is bottom up:

$$\textit{unit testing} \rightarrow \textit{integration testing} \rightarrow \textit{system testing}$$

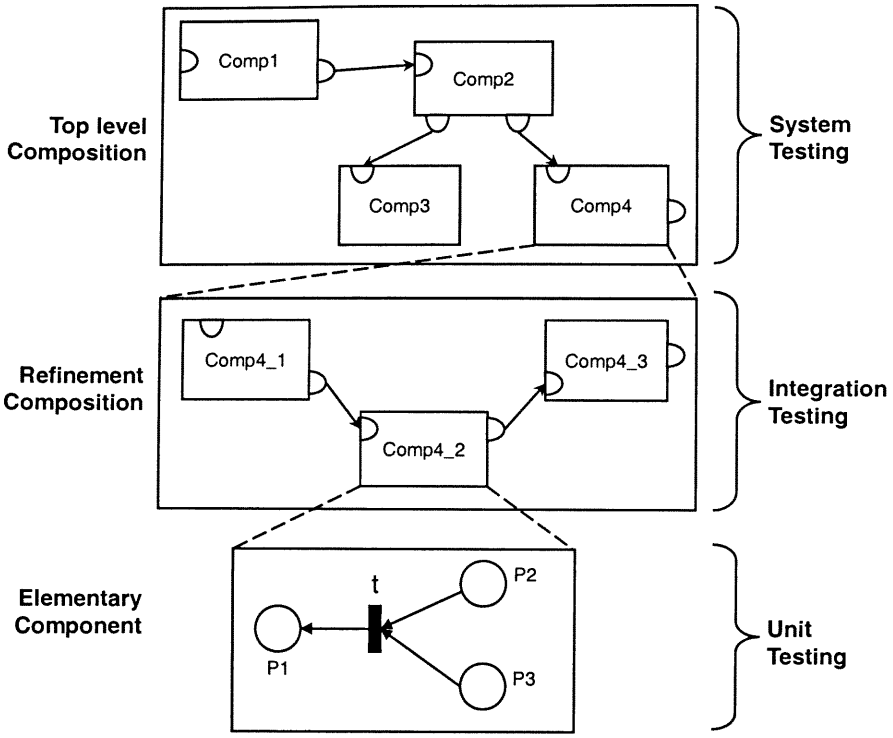

Figure 5.2: Testing Levels of SAM models.

This follows the "V" model for software testing [62]. For each level of testing the coverage and adequacy criteria are defined below.

5.2.1 Top Level Testing

Top level testing, i.e. system testing, is from the point of view of the user of the system. It involves how the SAM model behaves with respect to the environment (environmental constraints); therefore, input ports (input to the system) and environment output ports (output of the system) together the properties are the elements of interest when testing. The testing procedure reduces to check whether for given tokens at input ports there are tokens at output ports. The different configurations of input and output ports are mapped to abstract states in the behavioral model. The test cases selected for exercising the system are ones that are adequate according to certain coverage criteria, in this case state coverage criterion. Given that ports are

related to properties, one important aspect relates to property oriented testing. Refer to section 5.1.4 for a discussion on property coverage.

System testing implies that an integration testing of the top level composition was done (see next section 5.2.2). There is still work to be done in how to integrate the results of previous phases (unit and integration testing) into the top level implementation. The top level composition is a composition, so an integration phase is also defined for it. This is discussed in the next section.

5.2.2 Composition Testing

Testing a composition corresponds to *integration testing*. Since a composition contains components, first those components need to be individually tested:

> Given a composition C_i, each component/connector $C_{ji} \in C_{mi} \cup C_{ni}$ is tested. If C_{ji} is an elementary component or connector, then it is tested using *unit testing* techniques; otherwise, C_{ji} has a refinement composition C_h that is tested using *integration testing* techniques.

Each composition defines a set of constraints, which is basically a set of property specifications involving different components. Also, if the composition is a refinement of a component then the set of component properties are taken into account.

Coverage Criterion. Similarly to system testing, the coverage criterion used is port coverage which maps to state oriented coverage in the behavioral model.

Property oriented testing is also of interest here (refer to 5.1.4 for details on property coverage).

5.2.3 Component Testing

This section deals with testing elementary components, if the component to be tested has a refinement composition, then it is tested using *Integration Testing* (5.2.2).

An elementary component has a behavioral model (PrT net) as well as a property specification (FO-LTL formula). Both elements are used to define the coverage criteria that needs to be applied at this stage. As presented before, in [72] Zhu and He proposed different metrics for testing coverage and adequacy criteria for Petri Nets; also, in [17] and [16] Ding et. al. defined a procedure to measure the adequacy criteria of High Level Petri nets using Spin.

Coverage Criteria. the criteria used at this level to select test cases are (refer to 5.1.4 for a detailed discussion on them):

- Transition coverage.
- State coverage.

Given that an integrated behavioral model can be obtained for a non-elementary component and connector and ultimately for the whole system, then for any level of the testing process the test cases can be selected based on transition and coverage criteria for PrT nets.

5.3 Design Testing

There are two parts in design testing, one is the selection of test cases (the ones that are adequate with respect to certain coverage criteria) and the other is the testing of the design itself.

Test set evaluation

The test set evaluation is related to the test case selection, i.e. based on the evaluation a test set can be chosen for actually testing the model in question. This is largely driven by the initial markings for the PrT nets in the behavioral model. Given a set of test cases, they are measured with respect to transition and state coverage criteria. In [16] the simulation capability of Spin was used in determining the coverage of test cases for the Alternation Bit Protocol. In that work the PrT net is translated to a form in which there is no possibility of using embedded C code, such as in the translation approach in this dissertation. The simulation capability of Spin can not interpret embedded C code for embedded C code is only available in verification mode.

An alternative approach is to use an instrumented version of the ArchJava program corresponding to the model and perform the random executions on inputs and measure the adequacy of the tests.

In any case, there are three aspects that are taken into account:

- *What to observe.* depending on the coverage criteria, either transitions or markings are observed and recorded. During the test execution markings are translated into abstract states.
- *How to record.* Online monitoring, for on-the-fly analysis, or a log file for later analysis can be used.
- *When to stop.* As defined in [16], the conditions for stopping the testing process are: the program terminates, the test criterion is satisfied, the program enters a deadlock state, and the program runs for a predefined amount of time.

Testing the design

For testing the design, the SAM model is translated to an ArchJava program, for this translation the work by Fu et.al. is applied at this point ([25]). One element in the translation is critical, the properties. Properties can be translated two ways: (1) as aspects as in AOP, and (2) as a state automaton code within the test driver program. This is for purposes of monitoring if the property holds (safety) or if it can be satisfied (liveness).

The testing process at this level can be non-guided; however, depending on the information contained in the test cases, it can be guided as well. The ArchJava program can be instrumented to observe any interesting behavior and potentially to evaluate the adequacy of test sets.

5.4 Implementation Testing

On important part for the implementation testing, is deriving the test cases based on the design. In this case, the interest is in deriving test cases based on the information of the SAM model. Figure 5.3 shows the basic approach for generating test cases based on the design.

There are two possible scenarios. First, test cases for the design can be reused and translated for the implementation. Second, the information contained in the design can be used to derive test cases.

For the generation of test cases based on the design, the approach adopted is property oriented, i.e. the design makes use of Model Checking to generate test cases

for liveness properties to test at the implementation stage. The basic idea for the generation of test cases using model checking is shown in Figure 5.4.

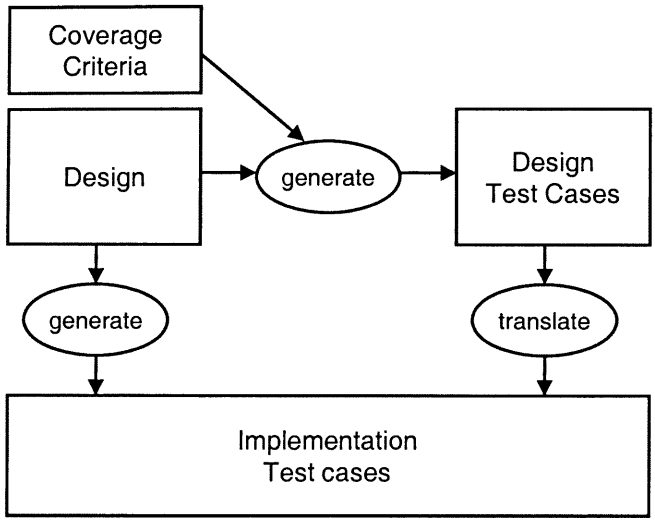

Figure 5.3: Test case generation overview.

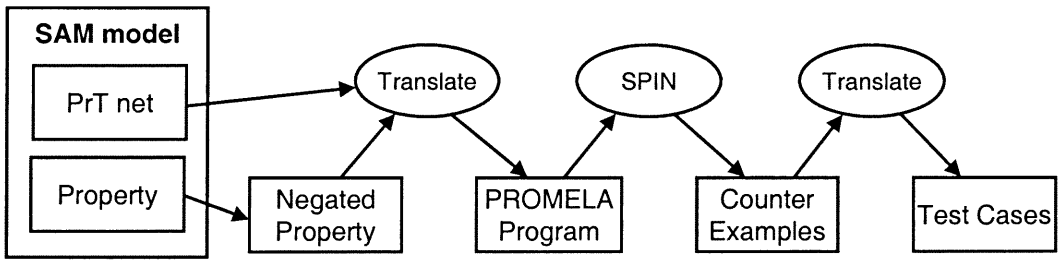

Figure 5.4: Test case generation with model checking.

The procedure to generate test cases using model checking starts by negating the properties of interest and following the model checking approach depicted in Chapter 4. As a result, paths that satisfy the property are encountered. The tests cases produced by this approach correspond to PrT net test cases with sequences defined in 5.1.1. Once the test cases are defined they are translated to specific methods or functions in the implementation and abstract states in the test set are mapped to concrete states in the implementation level.

5.5 Summary

This chapter provided the foundations for testing a SAM model, at the design stage as well as at the implementation one. It started presenting the overall picture of the testing procedure. Next, important concepts for testing PrT nets and SAM models were introduced. Since it serves as the foundation of the behavioral model for a SAM model, testing of PrT nets was discussed in detail. Finally the design and the implementation testing procedures for SAM models were presented.

CHAPTER 6

TOOL ENVIRONMENT

This Chapter introduces the environment tool for the modeling and analysis of SAM models: the SAM tool.

There are tools available to support different methods and techniques in software development. For example, for Petri nets the following tools are available:

- CPN Tools [46]: it is used for editing, simulating and analyzing Coloured Petri Nets. It provides several modeling and visual facilities: such as zoom. It contains a syntax checker and code generator. It has powerful analysis modules.
- PEP tool [31]: PEP (Programming Environment based on Petri Nets) provides similar modeling capabilities as the previous one. It allows to model low-level and high-level Petri nets. It contains components for doing reachability analysis among others. It provides interfaces to SMV and Spin, taking advantage of the latter's model checking capability.

In terms of ADL environments, two prominent examples are:

- Acme: AcmeStudio (Carnegie Mellon University) is a modeling environment for software architecture designs written in the Acme architectural description language. Similarly to the tool we are developing, AcmeStudio is available as a plugin for the Eclipse Environment, which allows creating extensions to it.
- Rapide toolset: developed at Stanford University, supports component-based development of large systems. The modeling process in Rapide starts by defining

a system architecture using a graphical tool, then Rapide's simulator produces an executable that when executed produces a causal event simulation output. This output can be analyzed by various tools.

In this dissertation, a tool to be used in the design and analysis of SAM models was implemented based on the Eclipse GEF Framework ([65]). Some elements of the tool were already available, including the main window with the composition and Petri net views. This dissertation, completed and extended the tool to its final form. The tool elements are depicted in Figure 6.1. This dissertation touched on the *Modeling*, *Design Analysis* and *Data Model* components (showed in gray in the figure); the *Prototype implementation* was implemented separately.

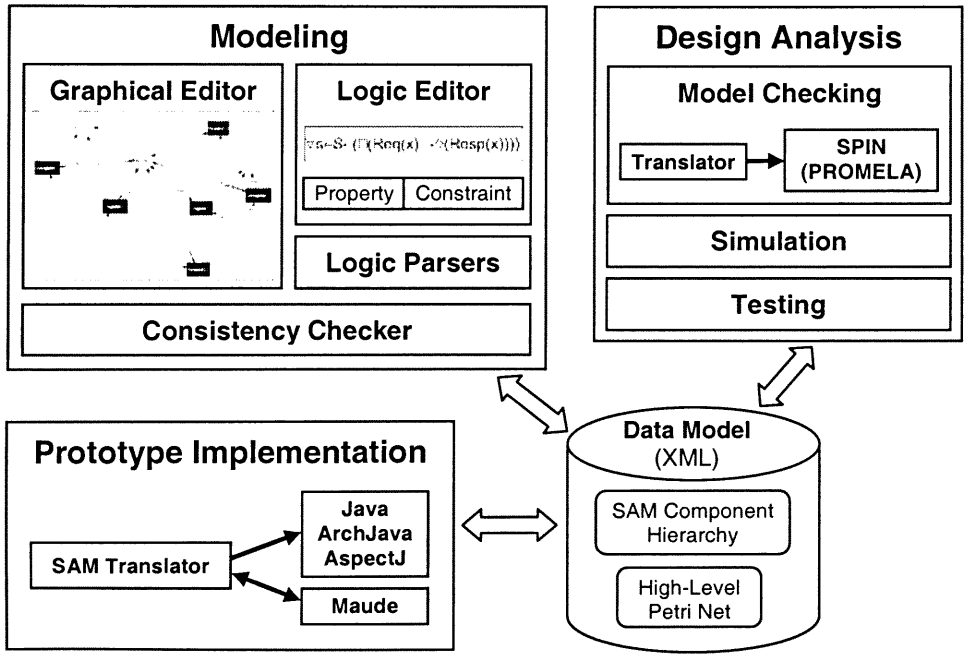

Figure 6.1: SAM Tool elements.

The SAM tool allows the definition of all the elements present in a SAM model, including the behavioral models expressed in PrT nets (only PrT nets are supported).

The formal approach at translating a SAM model to PROMELA made it possible to implement a module to generate PROMELA code automatically (semi automatically for the properties) from a SAM model in the tool. Once the code is generated, Spin can be used to perform the model checking process.

6.1 Editor Window

In Figure 6.2, the editor window displaying a composition can be seen. To the left, the hierarchical structure of the composition is displayed and component *Comp2* is further decomposed into other components. To the right, the palette that allows to add components/connectors and to add ports and connections between them is visible.

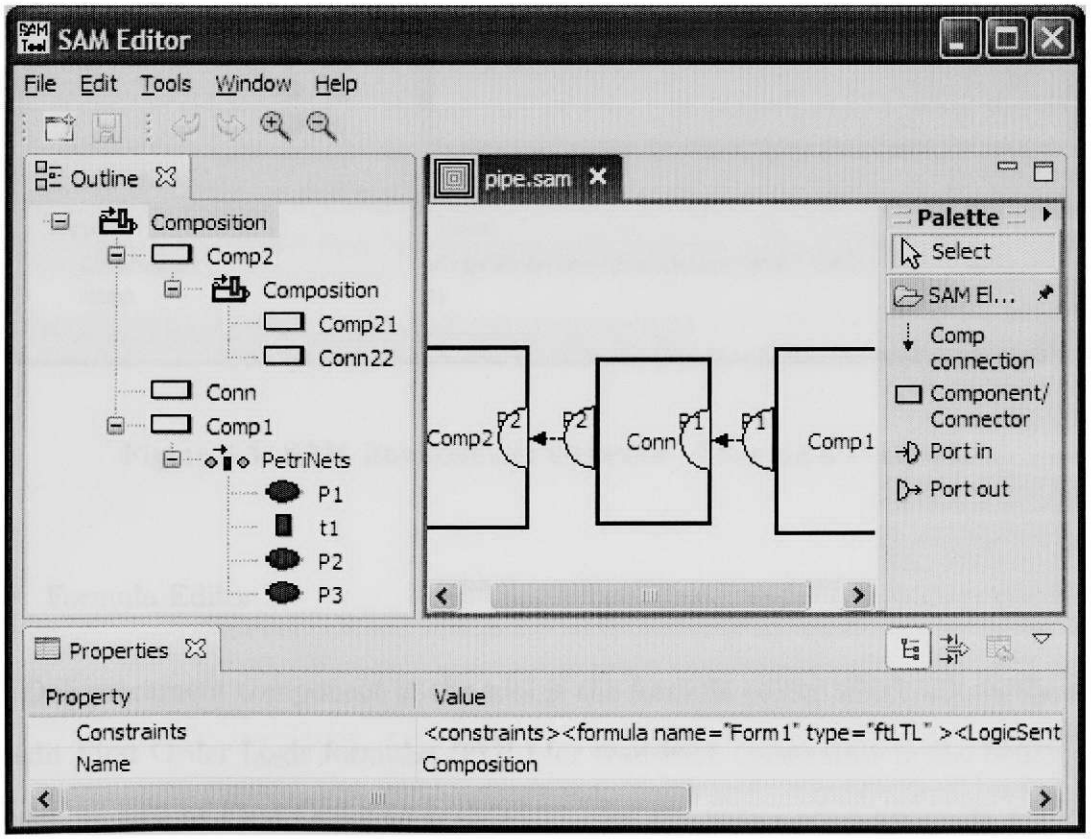

Figure 6.2: SAM Environment Editor - Showing a composition.

In Figure 6.3 the editor shows the behavioral model (i.e. Petri net) for component *Comp1*. To the right, the editor provides with the tools to add places, transitions and arcs to the Petri net model in question.

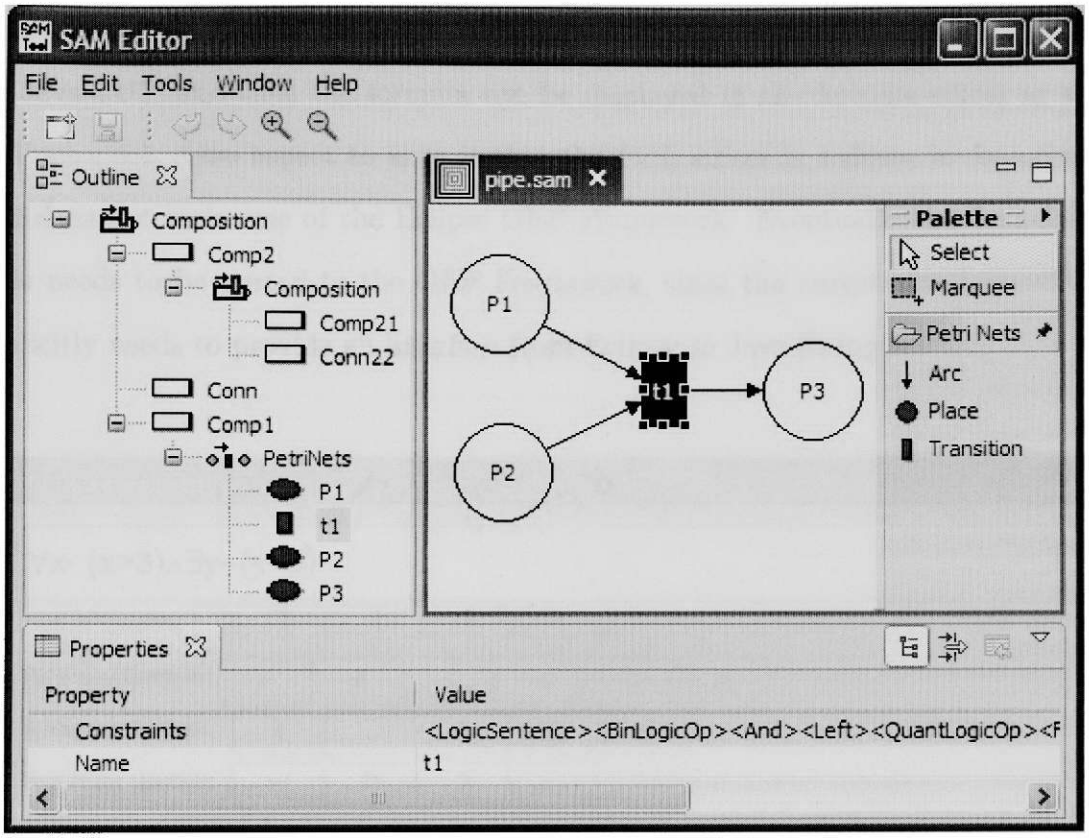

Figure 6.3: SAM Environment elements - Showing a Petri net.

6.2 Formula Editor

One prominent component of the tool is the formula editor which allows the user to edit First Order Logic formulas (FOL) for transition constraints in the behavioral model, as well as First Order LTL (FO-LTL) for properties of components and connectors. It also includes an analyzer to detect syntactically incorrect formulas. There are two formula editors, one for FOL and another for FO-LTL.

FOL Editor

Java-CUP LALR parser ([61]) was used as the engine to parse the First Order Logic (FOL) formulas for transition constraints in the PrT net models. The guard for a transition is stored in XML format, which is parsed into the grammar recognized by Java-CUP, and then the formula can be displayed in the formula editor as seen in Figure 6.4. One aspect to note is that the FOL editor is written in Java-Swing and does not make use of the Eclipse GEF Framework. Eventually the Java-Swing code needs to be ported to the GEF Framework, since the current implementation explicitly needs to provide an interface from Eclipse to Java-Swing.

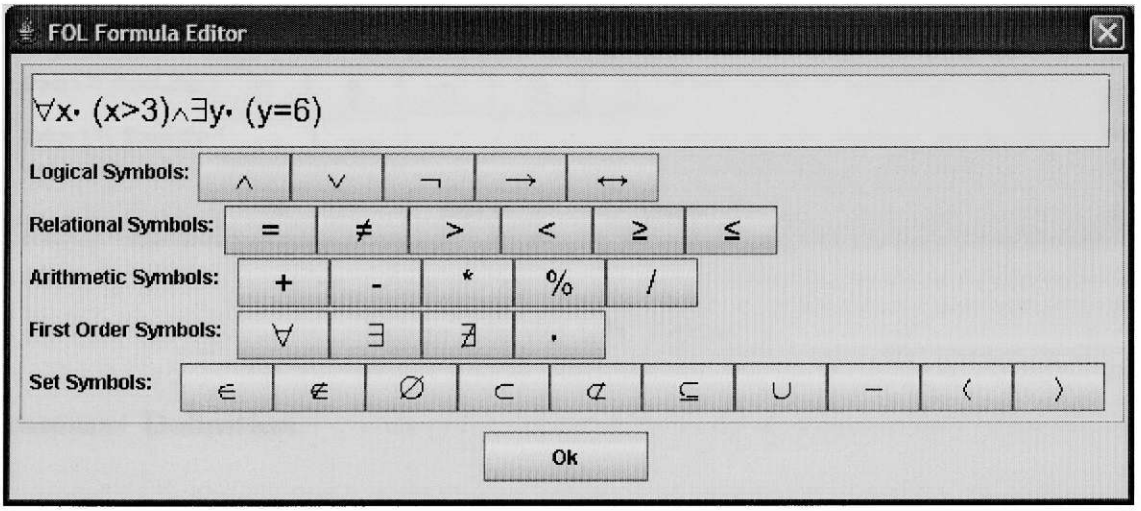

Figure 6.4: FOL Editor.

FO-LTL Editor

Java-CUP LALR parser was also used for parsing FO-LTL (First Order Linear Temporal Logic) formulas. For example in Figure 6.5, one response property, "for every request there is a response", relating two components can be seen.

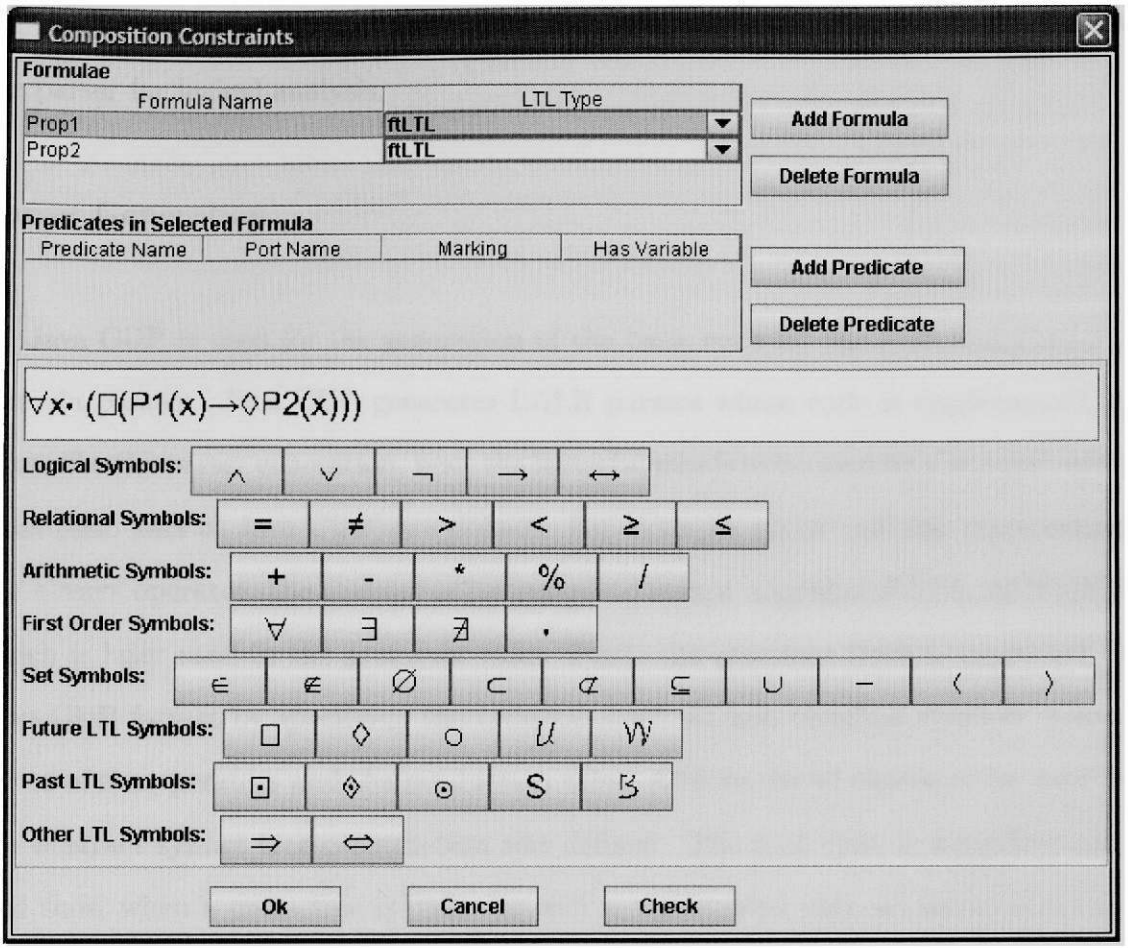

Figure 6.5: FO-LTL Editor.

Grammar Definition

A left recursive grammar was defined for each FO-LTL and FOL in BNF form. This grammar defines the production rules to build terms and expressions as well as the terminal symbols for the tokens. For FOL relational, arithmetic and set operators were included.

Additionally, the tool implemented its own true type font to specifically support the temporal operators. The font encodes each character in Unicode 16 bits format.

A formula is then encoded as a 16 bit character string and it can then be passed to the parser for lexical analysis.

Parser Generation

Java CUP is used for the generation of the basic code for the FOL and FO-LTL formula parsers. Java CUP generates LALR parsers whose code is implemented in Java. For the parser generation, first, the lexical symbols to be used by a scanner using JLex (also part of Java CUP) are defined. For example, given $\backslash uFA60$ representing the always operator, the scanner is instructed to return a symbol $FLTL\_ALWAYS$ which is later used in the grammar rules. Next, the grammar itself is expressed in Java CUP format by specifying the terminal symbols, non terminal symbols, precedence of the operators and the production rules. A hierarchy of classes to be used in the abstract syntax tree construction was defined. The root class is *LogicSentence*, and thus, when a parse tree is built, its root is represented with an instance of this class. The classes available support the different logic constructs such as binary logic operations, arithmetic operations, temporal logic operations, among others. See Appendix D for more details.

Syntax checking

The syntax correctness is ensured by the FOL and FO-LTL parsers. If there is an invalid token, the token analyzer will report it and if there is a syntactic error, the parser will stop and report the corresponding error. For example, given the (syntactically incorrect) formula:

$$\forall\forall\exists \rightarrow a$$

It has valid characters, and as such, valid tokens, so the lexical symbol construction poses no problem. But in the syntax tree construction, the grammar rules define that after each $\forall$ a variable or variable list should appear. This is not the case, and a syntactic error is reported.

Type checking

Type checking is accomplished by relying on the Sorts defined for the PrT net, where each variable and predicate name has an associated sort and hence the formula can be checked for invalid type constructs. For example, given the expression $x[1] + y$, $x[1]$ and y need to be numbers and not other elements such as sets. This semantic check is critical in the translation procedures from SAM to PROMELA and to Java/ArchJava.

CHAPTER 7

CASE STUDIES

This Chapter presents experimental results from the application of the approaches for formal verification and testing of software architectural models in SAM proposed in this dissertation. Three examples are given: the Resource Provider, the User-Centric Communication Middleware and the Alternating Bit Protocol. The Resource Provider example defines an scenario found in distributed and concurrent systems where components request access to resources and a provider handles those requests asynchronously. The User-Centric Communication Middleware defines a middleware that abstracts the heterogeneity of the different communication protocols, separating the logic of the communication from the network. Finally, the Alternating Bit Protocol defines a communications protocol in which messages can get lost but accepted messages are guaranteed to be delivered only once and in order.

All three examples were designed in the SAM tool environment (refer to Chapter 6 for details on the tool environment). Each example made use of the design and analysis features, as well as the code generation facility for model checking, available in the tool. Using the tool environment made it possible to design models that complied with the SAM framework itself, since the graphical composition of the different elements already assured restrictions imposed by the framework. For example, it was enforced that only input ports can connect to output ports and vice versa. The tool also allowed the detection of syntactic errors in formulas for preconditions and postconditions for transitions in the corresponding PrT nets describing the behavior of the system. In addition, the semiautomatic translation from SAM to PROMELA was useful in detecting design flaws; for example, the translation did not succeed if a formula referred a non-existing Predicate in the model.

7.1 Resource Provider

The Resource Provider scenario is defined as follow:

There is a provider component A that needs to serve other client components $C_1, ..C_n$ with resources from component S (see figure 7.1). The interactions among the different components can be synchronous or asynchronous. For example, for the asynchronous case, C_1 submits requests to A and does not wait but returns to continue what it is doing. When A loads the required resources, it notifies C_1 in order for it to access the loaded resources. For a synchronous situation, C_n requests resources to A at different times, but waits for A to deliver them. In general, component A needs to handle the requests from the clients, needs to get the responses from S and deliver them to the clients and, depending on the kind of resources being served, A can cache them for reuse.

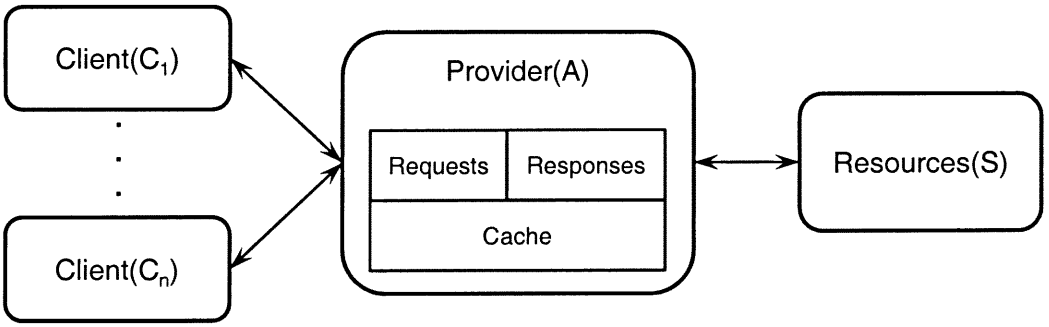

Figure 7.1: Resource Provider example.

These scenarios manifest at different levels in concurrent and distributed where different elements need access to resources. One example is a print server that needs to give the printers available to clients waiting to print within a network environment. Another example is a graphics multithreaded simulation application that has separate threads accessing images representing the objects to be rendered on the screen.

The asynchronous version of the Resource Provider scenario is studied in this section. The top level SAM architecture of the asynchronous Resource Provider example is portrayed in diagram 7.2. The architecture consists of three main components: the *Consumer* (representing the clients), the *ResourceProvider* (representing the provider) and the *SystemResources* (representing the resources to load and access).

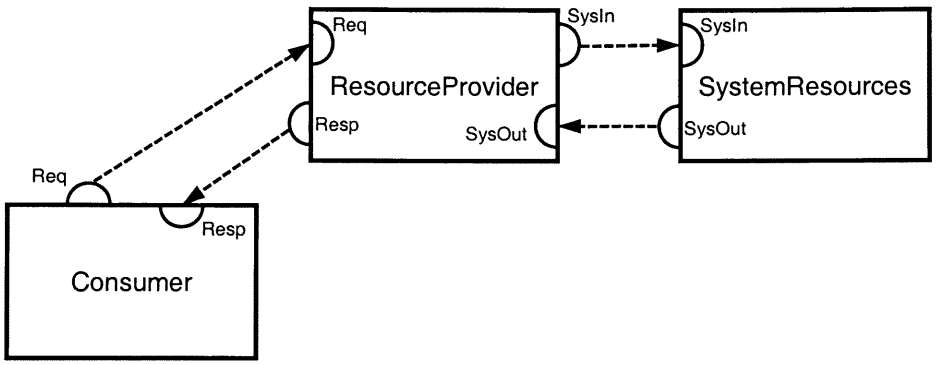

Figure 7.2: Resource Provider SAM Architecture.

The main component that is of interest is *ResourceProvider*; nevertheless, the interactions between *ResourceProvider* and the other components *Consumer* and *SystemResources* are explicitly modeled. There are two connections between component *ResourceProvider* and each one of the other components depending whether the flow is from or to *ResourceProvider*.

Each component has a PrT net describing its behavior. For example, Figure 7.3 shows the PrT net diagram for the *Consumer* component. The consumer is simplified in that it shows the requests being sent and the responses being received: every request that is sent out is kept in a *Pending* place until a response comes back which is kept in *LocalRes* place. Component *Consumer* can have a more complex behavior; for example, if the response received *res* states that the resource requested is not available or that it does not exist, then *Consumer* can try again. This is not

an issue, since in the current model the initial marking can contain sufficient number of requests to simulate this behavior.

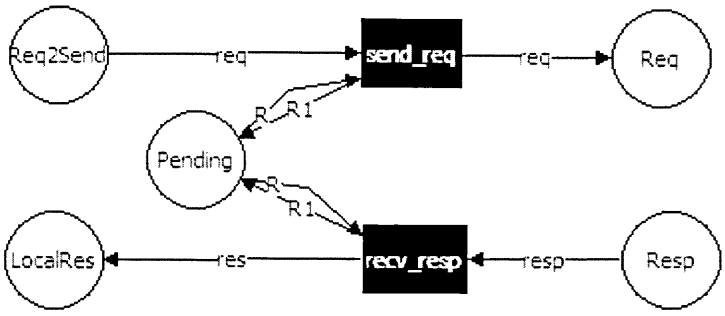

Figure 7.3: PrT net model for *Consumer* component.

Flattened SAM model. For the analysis part, a flattened version of the SAM model for the asynchronous Resource Provider shown in Figure 7.2 is needed. This flattened version of the SAM model can be seen in Figure 7.4. It exposes the sub components of the main component *ResourceProvider* which consists of three elementary components: *RequestHandler*, *Locator* and *Cache*.

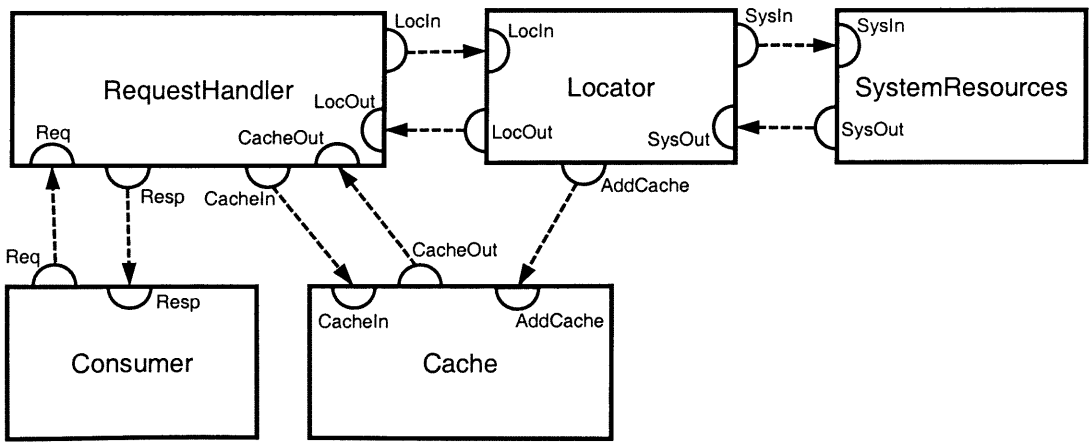

Figure 7.4: Flattened SAM model of the Resource Provider.

In the SAM tool, the flattened version of the SAM model is automatically generated before the model is translated to PROMELA in the formal verification process.

The behavioral models of the different components in the flattened SAM model are expressed in PrT nets. Table 7.1 defines the sorts for the PrT models in the Resource Provider example. Sorts *string* and *int* are basic sorts, and the other structured sorts (powerset and cross product sorts) are based on them. The mechanism for identifying a request and a response to that request is by using a integer *id* field that serves as the identifier for the request.

Sort name	**Definition**
NAME	string
ID	int
ID_RES	int
REQT	ID × NAME
RESPT	REQT × ID_RES
PREQT	$\wp$(REQT)
RESOURT	NAME × ID_RES
PRESOURT	$\wp$(RESOURT)

Table 7.1: *RequestHandler* PrT net model sorts definition.

The specification for component *RequestHandler* is detailed next. Figure 7.5 shows the PrT net model for component *RequestHandler*.

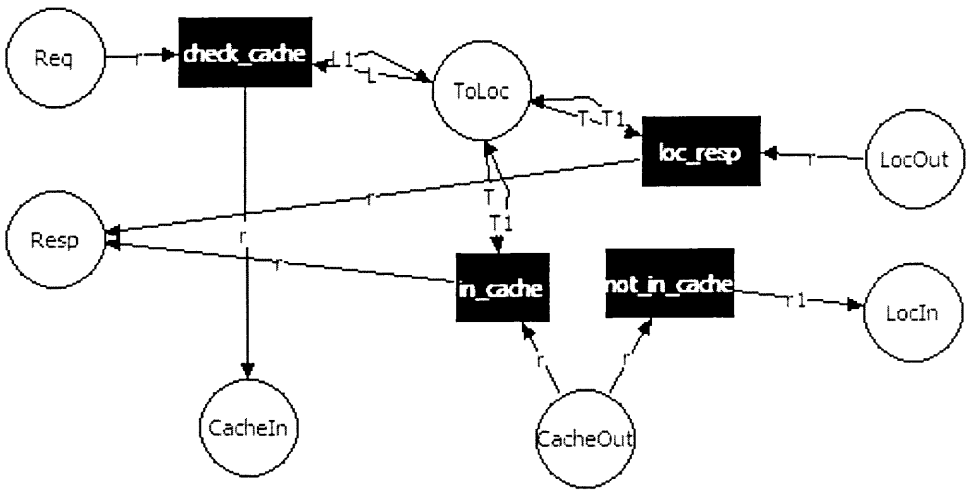

Figure 7.5: Component *RequestHandler* PrT model.

Given the sorts definition in Table 7.1, the sorts for the predicates (places) in the PrT net model for component *RequestHandler* are:

$$\varphi(Req) = \varphi(CacheIn) = \varphi(LocIn) = REQT$$
$$\varphi(Resp) = \varphi(CacheOut) = \varphi(LocOut) = RESPT$$
$$\varphi(ToLoc) = PREQT$$

With respect to the initial marking M_0, predicate (place) $ToLoc$ contains one token, an empty set, which gets populated as requests arrive:

$$M_0(ToLoc) = \{\{\}\}$$

All other predicates do not have tokens initially. However, for testing purposes the initial marking can vary depending on the testing procedure. The constraints for the transitions are defined next:

$$R(check\_cache) : (L1 = L \cup r)$$
$$R(in\_cache) : (r[2]! = 0 \wedge (\exists x \in L \cdot (r[1] = x))) \wedge (L1 = L - \{r[1]\})$$
$$R(not\_in\_cache) : (r[2] = 0) \wedge (r1 = r[1])$$
$$R(load\_resp) : (\exists x \in T \cdot (r[1] = x)) \wedge (T1 = T - \{r[1]\})$$

One of the properties defined for this component is a liveness one (for every request that arrives at this component, eventually there will be a reply for it):

$$\forall x \exists y \cdot \Box(Req(x) \rightarrow \Diamond(Resp(y) \wedge y[1] = x))$$

This is the most important property for component *ResourceProvider* since it needs to give a feedback to any request coming from component *Consumer*.

In the remaining of this section, the following procedures are presented:

- Model Checking: the model was translated to PROMELA with the help of the SAM tool and properties were verified. The results are discussed.
- Testing: given test cases, they were measured according to Transition and State Coverage Criteria for PrT nets.
- Test Case Generation: using model checking on the negation of a property, test cases are generated to be used at the implementation level.

Model Checking. Model checking was applied at the top design level following the procedure explained in Chapter 4. The main property verified was a liveness one: $\forall x \exists y \cdot (\Box(Req(x) \rightarrow \Diamond(Resp(y) \wedge x = y[1])))$. The expanded formula yielded the following propositional LTLs:

$$\Box(Req(< 1,'' resource1'' >) \rightarrow \Diamond Resp(<< 1,'' resource1'' >, c1 >) \wedge$$
$$\Box(Req(< 2,'' resource2'' >) \rightarrow \Diamond Resp(<< 2,'' resource2'' >, c2 >) \wedge$$
$$\Box(Req(< 3,'' resource3'' >) \rightarrow \Diamond Resp(<< 3,'' resource3'' >, c3 >)$$

The expansion is based on the initial marking of the requests coming from the *Consumer* component. Constants $c1$, $c2$, $c3$ are of type *int* and are not taken into consideration when computing the truth value for the property. For each sub formula an automaton is defined in PROMELA.

The automatic code generation using the SAM tool, generated approximately 1700 lines of code that accounted for 49KB of size in disk. In the next paragraphs, the listing of the code generated for component *RequestHandler* can be seen.

```
proctype RequestHandler(){
 /*Net places and initial marking.*/
 PREQT v_ToLoc[10];
 short num_ToLoc;
 num_ToLoc=1;
 v_ToLoc[0].num=0;
 /*Increment the counter and wait for the other processes to start.*/
 _proc_init++;
 ( _proc_init == _proc_num );
 /*Test enabledness and fire.*/
 do
  :: atomic{c_expr{is_enabled_check_cache(now.num_Req, now.v_Req,
           PRequestHandler->num_ToLoc,
           PRequestHandler->v_ToLoc)} ->
       c_code{fire_check_cache(&(now.num_Req), now.v_Req,
           &(PRequestHandler->num_ToLoc),
           PRequestHandler->v_ToLoc, &(now.num_CacheIn),
           now.v_CacheIn);}}
  :: atomic{c_expr{is_enabled_in_cache(PRequestHandler->num_ToLoc,
          PRequestHandler->v_ToLoc, now.num_CacheOut,
          now.v_CacheOut)} ->
       c_code{fire_in_cache(&(PRequestHandler->num_ToLoc),
          PRequestHandler->v_ToLoc, &(now.num_CacheOut),
          now.v_CacheOut, &(now.num_Resp), now.v_Resp);}}
  :: atomic{c_expr{is_enabled_not_in_cache(now.num_CacheOut,
          now.v_CacheOut)} ->
       c_code{fire_not_in_cache(&(now.num_CacheOut), now.v_CacheOut,
          &(now.num_LocIn), now.v_LocIn);}}
  :: atomic{c_expr{is_enabled_loc_resp(PRequestHandler->num_ToLoc,
          PRequestHandler->v_ToLoc, now.num_LocOut,
          now.v_LocOut)} ->
       c_code{fire_loc_resp(&(PRequestHandler->num_ToLoc),
          PRequestHandler->v_ToLoc, &(now.num_LocOut),
          now.v_LocOut, &(now.num_Resp), now.v_Resp);}}
 od
}
```

A generic form of the formula describing the liveness property was generated in PROMELA with the help of the automaton generator in Spin. This generic form is

reused across the different propositional LTL formulas to verify. The listing of the code generated is shown next.

```
#define req c_expr{TokenAtReq()}
#define resp c_expr{TokenAtResp()}
never {    /* !([]  (req -> <> resp)) */
T0_init:
 if
  :: (! ((req)) && (resp)) -> goto accept_S4
  :: (1) -> goto T0_init
 fi;
accept_S4:
 if
  :: (! ((resp))) -> goto accept_S4
 fi;
}
```

Expressions *TokenAtReq* and *TokenAtResp* are two embedded *C* functions that compute whether the token of interest is at the respective place. For formula:

$$\Box(Req(< 1,"resource1" >) \rightarrow \Diamond Resp(<< 1,"resource1" >, c1 >)$$

Expressions *TokenAtReq* and *TokenAtResp* are defined as:

```
c_code{
int TokenAtReq(){
 /*Returns 1 if token <1,"resource"> is at place Req, 0 otherwise*/
 struct REQT _tok;
 _tok.field1 = 1;
 _tok.field2 = 0;/*0 is the encoding for "resource1",
     we can see this from the initial marking*/
 return(in_place_Req(now.num_Req,now.v_Req,&_tok));
}
int TokenAtResp(){
 /*Returns 1 if token <<1,"resource">,c1> is at place Resp, 0 otherwise*/
 struct REQT _tok;
 _tok.field1.field1 = 1;
```

```
_tok.field1.field2 = 0;/*0 is the encoding for "resource1",
    we can see this from the initial marking*/
/*Since we do not want to compare with the second field in _tok then
 we do the iteration rather than calling in_place_Resp*/
int _i;
for (_i=0;_i<num_Resp;_i++){
 if ((v_Resp[_i].field1.field1==_tok.field1.field1)
      && (v_Resp[_i].field1.field2==_tok.field1.field2))
     return 1;
}
return 0;
}
}
```

Similar code was generated for the other properties of interest with the only variation being what tokens to account for at each predicate.

Testing. At the design level, the interest was to measure and evaluate the coverage of test cases with respect to Transition and State Coverage criteria for PrT nets. The PrT net model of the flattened SAM model was studied. This procedure was based on the evaluation of test adequacy coverage of high level Petri nets in Spin explained in [16]. In the testing process, the simulation capability of Spin was used. This implied to manually change the code obtained for the verification stage to not use any embedded *C* code construct in PROMELA (in simulation mode Spin can not interpret the *C* code expressions). The procedure for testing was as follows:

1. Translate the behavioral model to PROMELA without embedded *C* code.
2. Add code for recording the transitions and states.
3. Add code for evaluating the coverage.
4. Define the stop conditions.

For the translation to PROMELA code without embedded C code, one important part was translating operations for sets. In the translation process a set is translated into a bounded array. Tokens can be sets and there was a need to query them for finding out the existence of a specific item. For each sort an *inline* macro was defined. For example, for sort *REQT* the inline code was:

```
/*Checks whether _elem_to_find is in _array of type REQT*/
inline check_elem_in_array_REQT(_array, _arraysize, _elem_to_find,
                _index_found, _found){
 _found = 0;
 _index_found = 0;
 do
  :: (_index_found < _arraysize) ->
  {
   if
    :: compare_REQT(_array[_index_found],_elem_to_find) ->
      _found = 1; break;
    :: else _index_found++;
   fi;
  }
  :: else -> break;
 od
}
```

Where *compare_REQT* is another inline construct to compare two elements of sort *REQT*. This *inline* code was then reused at several places in the code simplifying it. Another inline construct is one removing elements from a set, which meant removing elements from the array representing it. For example, for arrays containing elements of sort *REQT*, the following code was defined:

```
/*Removes element at _index in _array of type REQT*/
int _array_index;
inline remove_elem_array_REQT(_array,_array_num,_index){
 _array_index = _index;
 do
```

```
  :: (_array_index+1 < _array_num) ->
    assign_REQT(_array[_array_index],_array[_array_index+1]);
    _array_index++;
  :: else -> break;
 od;
 _array_num--
}
```

Expression *assign_REQT* assigns an element of type *REQT* to another one of the same type.

The code for testing the enabledness and firing a transition makes use of the above defined macros. For example, for transition *recv_resp* in component *Consumer*, the code was:

```
/*Transition: recv_resp*/

/*To keep the token to be removed from Resp*/
RESPT _recv_resp_resp;

inline precond_recv_resp(){
 /*check in Pending[_index1] whether there is one element complying*/
 /*_index1 will be 0 for this example*/
 check_elem_in_array_REQT(v_Pending[0].set,v_Pending[0].num,
  v_Resp[_index0].field1,_index2,_trans_bEnabled)
}

inline is_enabled_recv_resp(){
 _trans_bEnabled = 0;
 _index0 = 0;
 _index1 = 0;
 _index2 = 0;
 /*loop to iterate through the tokens at the places*/
 do
  :: (_index0 < num_Resp && _trans_bEnabled != 1) ->
  {
   _index1 = 0;
```

```
    do
     :: (_index1 < num_Pending) ->
     {
      /*compute the guard with the current substitution.*/
      precond_recv_resp();
      if
       :: _trans_bEnabled == 1 -> _index0--;break;
       :: else -> _index1++;
      fi;
     }
     :: else -> break;
    od
   }
   _index0++;
   :: else -> break;
  od
}

inline fire_recv_resp(){
 /*These variables contain the indexes of the tokens in the places
   to use for firing*/
 /*_index0(v_Resp), _index1(v_Pending),
   _index2(v_Pending[0])*/

 /*Remove the element from v_Resp*/
 /*Store req token since it is going away*/
 assign_RESPT(_recv_resp_resp,v_Resp[_index0]);
 remove_elem_array_RESPT(v_Resp,num_Resp,_index0);

 /*Remove the element from v_Pending[0]*/
 /*No need to store the token being removed*/
 /*_recv_resp_i1 is 0*/
 /*_recv_resp_i2 contains the element that is to be removed from the set*/
 remove_elem_array_REQT(v_Pending[0].set,v_Pending[0].num,_index2);

 /*Add the element to LocalRes*/
 assign_RESPT(v_LocalRes[num_LocalRes],_recv_resp_resp);
 num_LocalRes++
}
```

Some of the variables set when testing the enabledness of the transition are used at firing it. Since the enabledness and possible firing of a transition are done within an atomic construct, there is no possibility of getting the wrong indexes when firing the transition.

The code for checking the enabledness and firing a transition is within a *do..od* construct in the component process to which the transition belongs. For example, for component *Consumer*, the loop defining the firing of its transition was defined as:

```
do
 :: atomic{tran == 0 -> is_enabled_recv_resp();
   if
    :: _trans_bEnabled -> fire_recv_resp();
     track_trans(Consumer_trans,CONSUMER_CT_TRANS,tran);
    :: else -> skip;
   fi;
   select_tran(tran,tran_n);
  }
 :: atomic{tran == 1 -> is_enabled_send_req();
   if
    :: _trans_bEnabled -> fire_send_req();
     track_trans(Consumer_trans,CONSUMER_CT_TRANS,tran);
    :: else -> skip;
   fi;
   select_tran(tran,tran_n);
  }
od
```

Whenever transition *recv_resp* is enabled, it fires and the following sequence in the code takes place: $is\_enabled\_recv\_resp(); \_trans\_bEnabled \rightarrow fire\_recv\_resp()$. Variable *_trans_bEnabled* is computed in *is_enabled_recv_resp*() and represents the precondition result of the transition. The code for recording the transitions and states, as well as for evaluating the coverage, is done in *track_trans* (this is explained

in the next paragraphs). Another aspect is to control the selection of which transition to fire in simulation mode in Spin (refer to [16] for a discussion on this). Variables *tran*, *tran_n* and inline construct *select_tran* take care of that (*select_tran* sets variable *tran* to an integer representing a transition to be fired based on the number of transitions *tran_n*).

For recording the transitions and states covered, first, for each process, an integer array representing the number of times its transitions have been fired is defined. For example for component *Consumer*:

```
/*Define the array to track the number of times a transition fires
  in a process.*/
#define DEF_ARRAY(_array,_size) int _array[_size]
#define CONSUMER_CT_TRANS 2 /*Number of transitions at Consumer*/
DEF_ARRAY(Consumer_trans,CONSUMER_CT_TRANS);
```

In relation to abstract states, a unique array defining whether the state was covered is defined. This is a feature of the system or component as a whole, not of an individual process. In the studies several abstract states were defined, one of them is:

```
/*State coverage*/
/*Several abstract states can be defined. The one chosen here involves
the caching of resources: State1(NOT_CACHED), State2(CACHED)*/
#define STATE_CT 2 /*Number of states*/
DEF_ARRAY(State_array,STATE_CT); /*array of states*/
```

With these definitions, *track_trans* is:

```
/*Track the transition firing.*/
inline track_trans(_process_trans,_size,_tran_index){
 /*Record the transitions fired*/
 if
  :: (_tran_index < _size && _tran_index >= 0) ->
```

```
    _process_trans[_tran_index]++;
   :: else -> skip
  fi;
  /*Record the states covered*/
  if
   :: (v_ResCache[0].num > 0) -> State_array[1] = 1;
   :: else -> State_array[0] = 1
  fi;
  /*Evaluate depending what kind of coverage was chosen*/
  if
   :: TRANSITION_COVERAGE -> evaluate_trans_coverage();
   :: else -> evaluate_state_coverage()
  fi
}
```

This code records the transitions and states covered in a run. The coverage evaluation was done after each transition firing. When *TRANSITION_COVERAGE* constant is defined as 1, it determines the evaluation of transition coverage, otherwise state coverage evaluation is performed. In the experiments this value was changed in the code before executing the simulations. The code for evaluating transition coverage is defined as:

```
/*Evaluates the transition coverage*/
inline evaluate_trans_array(_trans_array,_size,_covered){
 _index0 = 0;
 do
  :: (_index0 < _size && _covered == 1) ->
  {
   if
    :: (_trans_array[_index0]==0) -> _covered=0; break;
    :: else -> _index0++
   fi
  }
  :: else -> break
 od
}
bool tr_covered;
```

```
inline evaluate_trans_coverage(){
 /*Arrays for evaluating:
   - Consumer_trans (CONSUMER_CT_TRANS)
   - SystemResources_trans (SYS_RES_CT_TRANS)
   - RequestHandler_trans (REQ_HANDLER_CT_TRANS)
   - Cache_trans (CACHE_CT_TRANS)
   - Locator_trans (LOCATOR_CT_TRANS)
 */
 tr_covered = 1;
 evaluate_trans_array(Consumer_trans, CONSUMER_CT_TRANS, tr_covered);
 evaluate_trans_array(SystemResources_trans, SYS_RES_CT_TRANS, tr_covered);
 evaluate_trans_array(RequestHandler_trans, REQ_HANDLER_CT_TRANS, tr_covered);
 evaluate_trans_array(Cache_trans, CACHE_CT_TRANS, tr_covered);
 evaluate_trans_array(Locator_trans, LOCATOR_CT_TRANS, tr_covered);
 /*stop simulation if covered*/
 assert(tr_covered != 1)
}
```

Finally, the code for evaluating state coverage is straightforward:

```
inline evaluate_state_coverage(){
 /*If the number of elements in the cache is 0 state NOT_CACHED is covered.*/
 /*Otherwise state CACHED is covered.*/
 assert(v_ResCache.num>0); /*there has to be a cache*/
 assert(State_array[0] == 0 || State_array[1] == 0)
}
```

Refer to the Appendix A for a complete listing of the code for evaluating the tests cases. When the code is compiled, all *inline* calls are replaced with the body of the *inline* constructs code. The original code was around 1,500 lines of text (~43KB in disk), and the actual *C* code generated in the pre-processing stage in Spin was around 12,500 lines (~340KB in disk).

Full transition and state coverage. In the testing process, full transition and state coverage was achieved. The following test case set accomplished that (only the mark-

ing for predicate *Req2Send* is shown): $M0(Req2Send) = \{< 1,"resource2" >, < 2,"resource1" >, < 3,"resource3" >, < 4,"resourcex" >, < 5,"resource2" >\}$

The initial markings for the other elements remained the same as for the verification part. Refer to Appendix B for details on the models.

Test setup:

- Test cases were run with different seeds in random simulation mode in Spin.
- Tokens arriving at places were handled on a first in first out basis.

Seed value of 0 for the simulation allowed full coverage, whereas a seed of 1 did not. For this specific configuration, when all requests have been responded (received at *Consumer* component) there is no point in waiting for anything to happen in the net, and the simulation is halted, unless there is some internal transition that for example is trying to add an element to the cache. Figure 7.6 shows the system steps performed by each process before reaching the full coverage when seed 0 was selected.

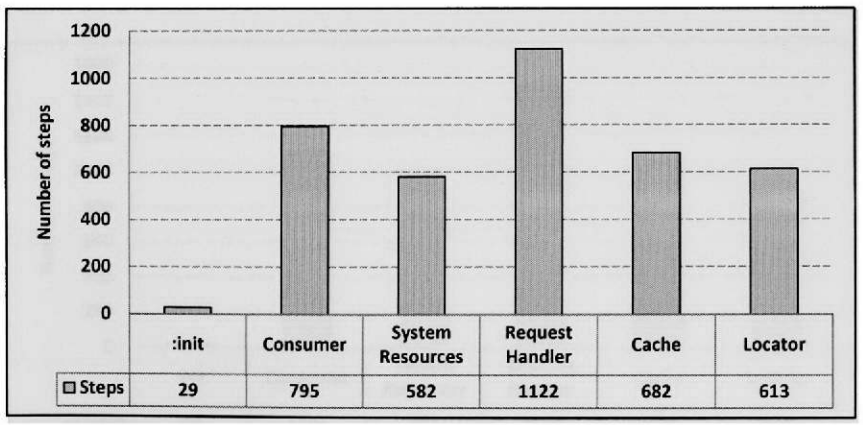

Figure 7.6: System steps during simulation.

Figure 7.7 shows the number of times each transition was fired (covered) during the execution of the tests.

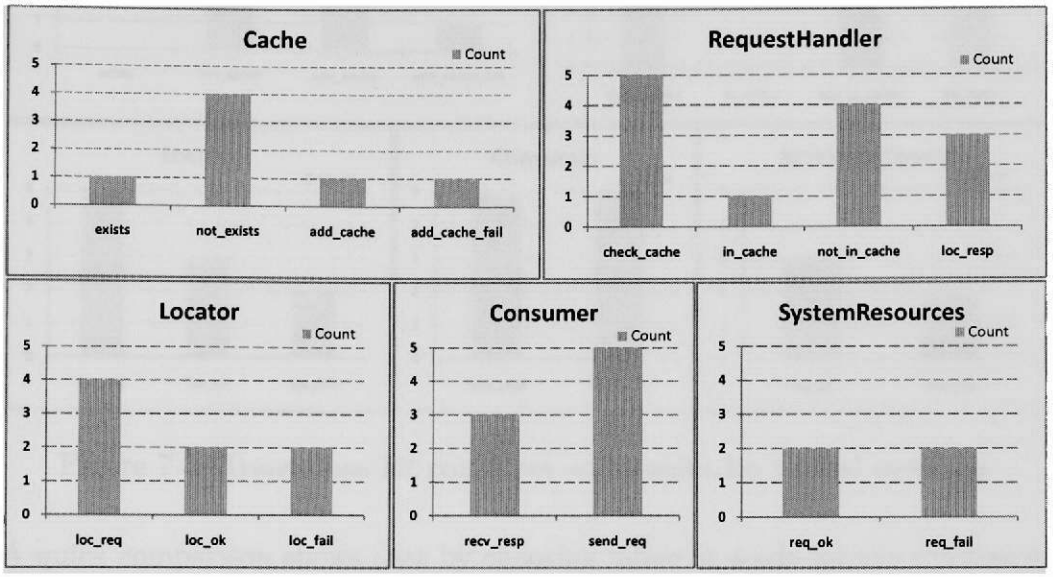

Figure 7.7: Transitions hit count per component for complete coverage.

For partial coverage with the same initial marking, a seed of 1 was used for the simulation and the results can be seen in Figure 7.8 (the number of systems steps performed per each process) and Figure 7.9 (the number of times a transition was fired during the test executions).

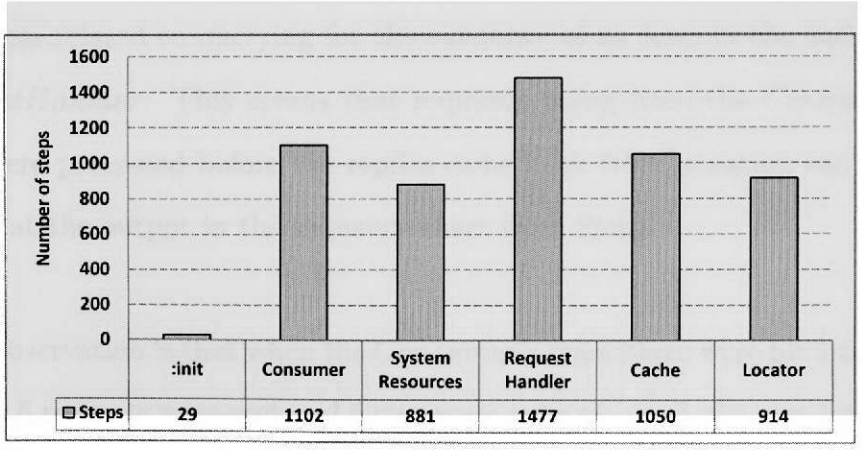

Figure 7.8: Spin system steps for partial coverage.

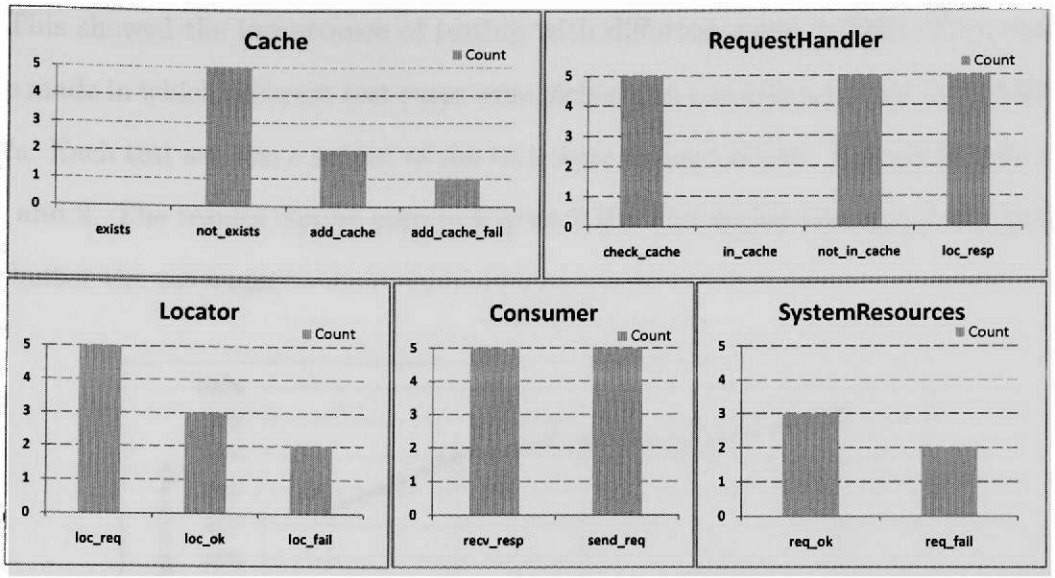

Figure 7.9: Transitions hit count per component for partial coverage.

A quick comparison shows that by choosing different seeds for the random simulation, the coverage of the transitions changes. This does not happen to the states coverage, which the experiments showed 100% coverage. For transition coverage with a seed value of 0, the coverage is 100% (15 out of 15 transitions); whereas with a seed value of 1, the coverage is 73% (11 out of 15 transitions).

For the non full coverage situation, it can be seen that the transitions not covered were the ones related to querying for the existence of an item in the cache made by the *RequestHandler*. This means that requests, going from the *Consumer* to the *Locator*, were processed before the replies came back (this situation can be verified by looking at the output in the sequence chart from Spin).

Other observation is that when the *Consumer*'s transitions were hit 5 times, which means that 5 requests were sent and 5 responses were received, the simulation process was stopped.

This showed the importance of testing with different seeds in Spin. Experiments were made in which different test cases were picked and run several times with different seeds. Each test set was a subset of the test cases defined above. The seeds used were 0, 1 and 2. The results can be seen in Figure 7.10. The higher the size of the test set the better the coverage.

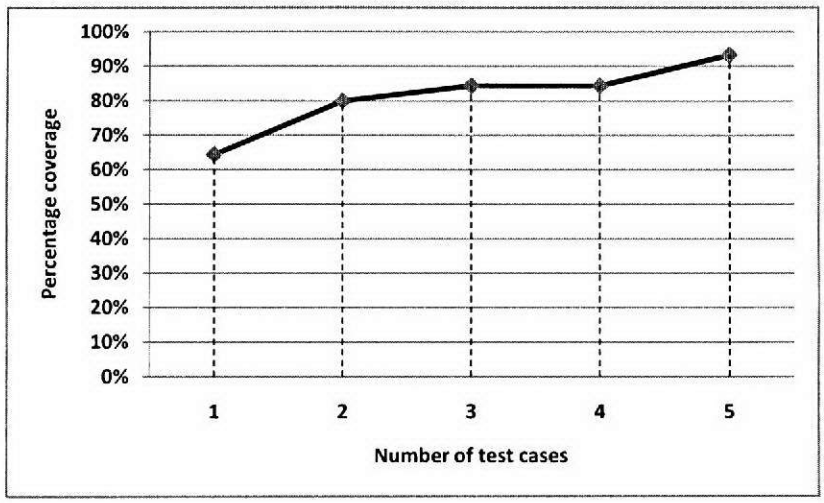

Figure 7.10: Number of test cases to measure transition coverage.

One error in the design not captured at the verification stage was captured at the testing phase. It involved component *Cache* which originally did not have a transition for failing to add elements to the cache. The failing scenario involves adding resources already in the cache.

The results showed the usefulness of the combination between formal verification and testing techniques. With formal verification certain specific properties are verified, but it is likely that not all the "code" is exercised (covered). Testing comes into aid in that transitions and states can be exercised given a set of test cases. The set of test cases can then be used at the implementation level to aid assure the correctness of the software.

Test case generation. The design model is used to generate test cases for the implementation. Property based generation and Coverage based generation were applied.

Property based generation In this dissertation, test cases for liveness properties were derived. Given the following property for the *RequestHandler* component: $\forall x \exists y \cdot \Box(Req(x) \rightarrow \Diamond(Resp(y) \wedge y[1] = x))$. The negation of one of its expanded terms was fed into the model checker, e.g. $\neg(\Box(Req(< 1,''resource1'' >) \rightarrow \Diamond(Resp(<< 1,''resource1'' >, 1001 >)))$. This property did not hold and the model checker generated a sequence of transitions firings along the substitutions used. The PROMELA code was modified to print the state after each firing along the substitutions. This sequence defines the abstract execution sequence for the implementation, and abstract test cases are then generated. This sequence is then converted to the specific method calls at the implementation level in the programming language of choice. The execution of the program was controlled by this sequence.

This is the approach chosen for unit testing. For integration testing and system testing the same approach is followed, the only difference is that the properties to look at relate different components, i.e. the ports that appear in the FO-LTL formulas belong to different components.

Transition/State Coverage based generation The test cases used in the design level are mapped to test cases at the implementation level. Since those test cases provided adequate coverage (transition or state), then based on that information they are said to be adequate for testing the implementation. For this test, random execution at the implementation level was chosen.

The User-Centric Communications Middleware and the Alternating Bit Protocol examples are discussed in terms of model checking.

User-Centric Communications Middleware (UCM)

The UCM abstracts away the complexities of the underlying network protocols. For more details, refer to [63] for the UCM and Chapter 4 and [5] for the model checking approach.

The UCM modeled in SAM consisted of 6 top level components from which two were UCMs at different sites (Figure 7.11). One of the UCMs was refined into 5 components. One of the components, the UCMM (UCM manager), was of special interest, and some properties were verified for it. The integrated behavioral model (PrT net) consisted of 63 places, 140 transitions and 476 arcs. The size of the verification code generated for this case study was around 10,000 lines of code (400KB). One of the critical components of the system is the SIP Manager, and several properties were verified for it.

Alternating Bit Protocol

Alternating Bit Protocol (ABP) is a protocol that consists of a sender, a receiver, and two channels. The transmission on the channels may get corrupted, but no duplication is guaranteed. If there is a corrupted message/acknowledgment, detected by the channels, then the message/acknowledgment is resent([16]). The ABP SAM model design is shown in Figure 7.12.

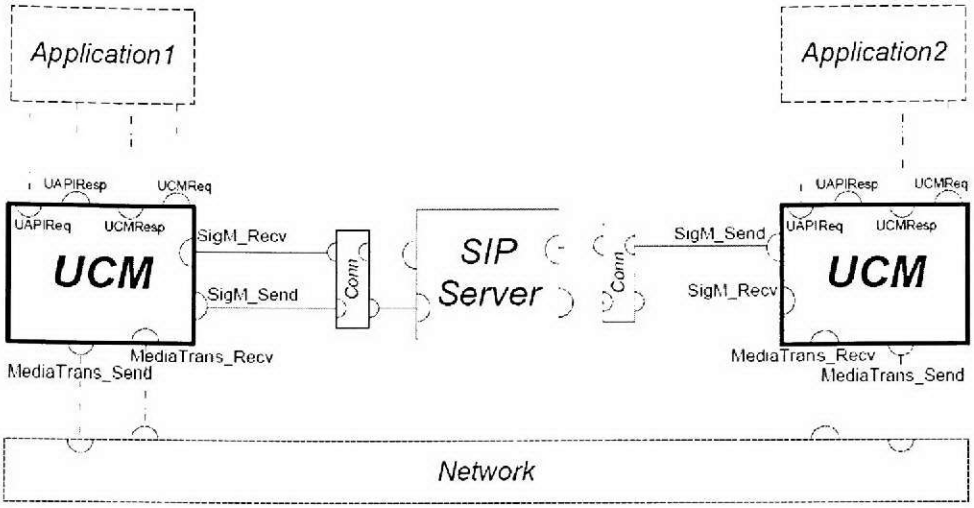

Figure 7.11: UCM system.

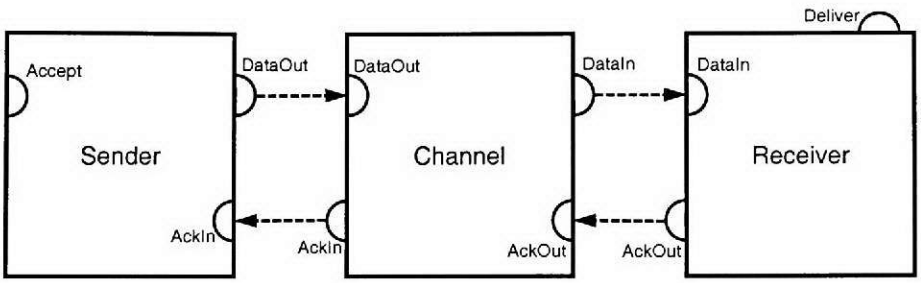

Figure 7.12: Alternating Bit Protocol.

Part of an initial design of the ABP can be seen in Figure 7.13. The specification for its places and transitions is defined next:

$$\varphi(DataIn) = DATA$$

$$\varphi(AckOut) = short$$

$$R(acorrupted) = (c[1] = 2)$$

$$R(resendAck) = (d[1] = 2)$$

$$DATA := short \times MSG$$

$$MSG := string \times ID$$

$$ID := short$$

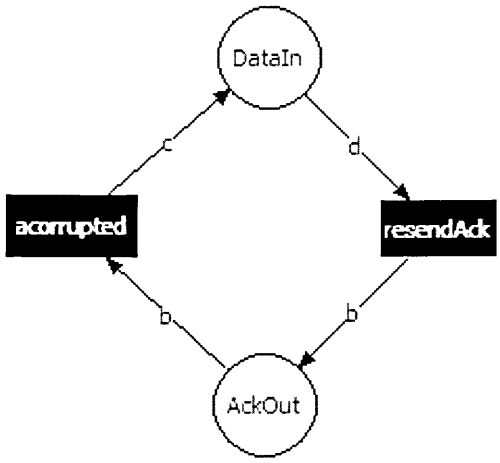

Figure 7.13: Section of the PrT net model for the ABP.

One design flaw was detected with the model checking approach when verifying liveness properties on the initial design of the ABP. It involved part of the PrT net specification of the ABP (see Figure 7.13) and a sequence: *acorrupted*, *resendAck*, *acorrupted*, *resendAck*..., which could go ad infinitum. Once the model was corrected, properties were verified for the model.

The main property that was verified was a liveness property of the form: $\forall x \cdot (\Box(Send(x) \rightarrow \Diamond Recv(x)))$. The formula was instantiated to propositional formulas based on the tokens available in the initial marking. For example, given the initial marking for *Send*, $M_0(Send) = \{\text{“}first\text{”}, \text{“}second\text{”}\}$, first a formula of the form $\Box(Send(\text{“}first\text{”}) \rightarrow \Diamond Recv(\text{“}first\text{”}))$ was verified, and next another formula of the form $\Box(Send(\text{“}second\text{”}) \rightarrow \Diamond Recv(\text{“}second\text{”}))$.

One observation, resulting from the experiments performed, is that for a non-satisfiable liveness property, Spin would not directly state that it was not satisfiable; rather it would report an acceptance cycle. Hence, when verifying a liveness property, the flag for acceptance cycles in the Spin verification environment has to be selected.

7.3 Summary

In this Chapter case studies showing the applicability of the approach were presented. One of the examples, the resource provider, is found in distributed and concurrent systems; a second one is used to define the architecture of a user centric communications middleware and the last one is the alternating bit protocol example. The model checking facility provided useful in determining the correctness of the models and the testing performed determined the adequacy of a test set with respect to different coverage criteria, and also allowed the generation of test cases for testing the implementation. Additionally, the tool environment developed as part of this dissertation was used for the modeling and helped detecting errors in the models.

CHAPTER 8

CONCLUSIONS

The conclusions, a summary of the contributions and the future work are presented in the following sections.

8.1 Conclusions

The dissertation presented in this document has addressed the problem of verification and testing of SAM model designs. The results have shown the applicability of the approach. Case studies were defined in which model checking was successfully applied to liveness properties. The properties verified were the Propositional version of the original First Order temporal properties. The testing approach defined procedures for the design as well as for the implementation. For testing the design, test cases were measured with respect to coverage criteria. The case studies were executed to show the coverage of different test cases for PrT nets based on the initial marking. Also test cases were generated from the counterexamples for use at the implementation level.

Finally, this dissertation has addressed the need of a modeling and analysis tool for the SAM framework by implementing parts of one and by providing different modules for syntactic verification of the model and for semi automatic generation of code from a SAM model to a PROMELA program. The tool was used to model and to generate the PROMELA code for the case studies in this document. It was also used for detecting design flaws, such as incorrect formulas for properties. The tool provided useful in the long sought goal of the correct software.

8.2 Summary of Contributions

There are three main contributions as part of the integrated approach combining formal verification and testing.

First (Verification of SAM models), a formal approach for translating a SAM model to a PROMELA program was defined, this allowed the formal verification of SAM models using the model checker Spin.

Second (Testing of SAM models), the testing approach for SAM models, defined the testing at the component, integration and system level. Within the testing approach, a procedure to evaluate the adequacy of test sets with respect to transition and state coverage for PrT nets was presented. Also, the testing approach defined means to generate test cases from the design by using model checking on the negation of the properties the system needs to comply with.

Third (Tool Environment for SAM), a modeling and analysis tool was implemented which included facilities for syntactic verification of First Order Logic formulas and that also contained a semiautomatic translation module from a SAM model to PROMELA code.

8.3 Future Work

There are three lines of work for the future. First, the incorporation of property oriented testing in test selection and test adequacy criteria. There has been work done in property oriented testing, but there is a need to do more within the SAM framework. Second, other line of work is related to Aspect Oriented system design based on SAM and PrT nets. In this area, the problem of test case generation and

evaluation for Aspect Oriented Systems is of interest. Finally the third line of work is to integrate Spin into the SAM tool, so that the SAM tool can interact with Spin and less manual intervention is required. This is of special interest for automatic test case generation, since results in Spin could be directly translated to the SAM tool for visual display and analysis.

BIBLIOGRAPHY

[1] Jonathan Aldrich, Craig Chambers, and David Notkin. ArchJava: connecting software architecture to implementation. *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 187–197, 2002.

[2] Paul Ammann and Paul E. Black. A specification-based coverage metric to evaluate test sets. In *HASE*, pages 239–248. IEEE Computer Society, 1999.

[3] Paul Ammann, Paul E. Black, and Wei Ding. Model checkers in software testing. Technical Report NIST-IR 6777, National Institute of Standards and Technology, 2002.

[4] Paul E. Ammann, Paul E. Black, and William Majurski. Using model checking to generate tests from specifications. In *In Proceedings of the Second IEEE International Conference on Formal Engineering Methods (ICFEM98*, pages 46–54. IEEE Computer Society, 1998.

[5] Gonzalo Argote-Garcia, Peter J. Clarke, Xudong He, Yujian Fu, and Leyuan Shi. A formal approach for translating a SAM architecture to PROMELA. In *SEKE*, pages 440–447. Knowledge Systems Institute Graduate School, 2008.

[6] Cyrille Artho, Howard Barringer, Allen Goldberg, Klaus Havelund, Sarfraz Khurshid, Michael R. Lowry, Corina S. Pasareanu, Grigore Rosu, Koushik Sen, Willem Visser, and Richard Washington. Combining test case generation and runtime verification. *Theor. Comput. Sci.*, 336(2-3):209–234, 2005.

[7] Luciano Baresi and Mauro Pezzè. An introduction to software testing. *Electronic Notes in Theoretical Computer Science*, 148(1):89–111, February 2006.

[8] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice, Second Edition.* Addison-Wesley Professional, April 2003.

[9] Boris Beizer. *Software Testing Techniques.* Van Nostrand Reinhold, New York, NY, 1990.

[10] Sergey Berezin. *Model Checking and Theorem Proving: a Unified Framework.* PhD thesis, Carnegie Mellon University, January 2002.

[11] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35:677–691, 1986.

[12] 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 *Logic in Computer Science, 1990. LICS '90, Proceedings., Fifth Annual IEEE Symposium on e*, pages 428–439, 1990.

[13] Richard H. Carver and Kuo-Chung Tai. Replay and testing for concurrent programs. *IEEE Software*, 8(2):66–74, 1991.

[14] William Chan, Richard J. Anderson, Paul Beame, Steve Burns, Francesmary Modugno, David Notkin, and Jon Damon Reese. Model checking large software specifications. *IEEE Trans. Software Eng.*, 24(7):498–520, 1998.

[15] Feng Chen and Grigore Rosu. Mop: an efficient and generic runtime verification framework. In Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr., editors, *OOPSLA*, pages 569–588. ACM, 2007.

[16] Junhua Ding, Gonzalo Argote-Garcia, Peter J. Clarke, and Xudong He. Evaluating test adequacy coverage of high level petri nets using Spin. In *AST '08: Proceedings of the 3rd international workshop on Automation of software test*, pages 71–78, New York, NY, USA, 2008. ACM.

[17] Junhua Ding, Peter J. Clarke, Gonzalo Argote-Garcia, and Xudong He. Evaluating test adequacy coverage of high level petri nets using spin. Technical Report FIU-SCIS-2006-05-02, FIU, May 2006.

[18] Zhijiang Dong, Yujian Fu, Yue Fu, and Xudong He. Automated runtime validation of software architecture design. In Goutam Chakraborty, editor, *ICDCIT*, volume 3816 of *Lecture Notes in Computer Science*, pages 446–457. Springer, 2005.

[19] Jr. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 1999.

[20] Jon Edvardsson. A survey on automatic test data generation. In *Proceedings of the Second Conference on Computer Science and Engineering in Linköping*, pages 21–28. ECSEL, October 1999.

[21] Javier Esparza and Keijo Heljanko. Implementing LTL model checking with net unfoldings. In Matthew B. Dwyer, editor, *SPIN*, volume 2057 of *Lecture Notes in Computer Science*, pages 37–56. Springer, 2001.

[22] Jean-Claude Fernandez, Laurent Mounier, and Cyril Pachon. Property oriented test case generation. In Alexandre Petrenko and Andreas Ulrich, editors, *FATES*, volume 2931 of *Lecture Notes in Computer Science*, pages 147–163. Springer, 2003.

[23] Gordon Fraser and Paul Ammann. Reachability and propagation for LTL requirements testing. In Hong Zhu, editor, *QSIC*, pages 189–198. IEEE Computer Society, 2008.

[24] Gordon Fraser and Franz Wotawa. Using model-checkers to generate and analyze property relevant test-cases. *Software Quality Journal*, 16(2):161–183, 2008.

[25] Yujian Fu, Zhijiang Dong, and Xudong He. A method for realizing software architecture design. In *QSIC*, pages 57–64. IEEE Computer Society, 2006.

[26] Gerald C. Gannod and Sunil Gupta. An automated tool for analyzing Petri nets using Spin. *ASE*, 00:404, 2001.

[27] Angelo Gargantini and Constance L. Heitmeyer. Using model checking to generate tests from requirements specifications. In Oscar Nierstrasz and Michel Lemoine, editors, *ESEC / SIGSOFT FSE*, volume 1687 of *Lecture Notes in Computer Science*, pages 146–162. Springer, 1999.

[28] Angelo Gargantini, Elvinia Riccobene, and Salvatore Rinzivillo. Using Spin to generate tests from ASM specifications. In Egon Börger, Angelo Gargantini, and Elvinia Riccobene, editors, *Abstract State Machines*, volume 2589 of *Lecture Notes in Computer Science*, pages 263–277. Springer, 2003.

[29] David Garlan, Robert T. Monroe, and David Wile. Acme: an architecture description interchange language. In J. Howard Johnson, editor, *CASCON*, page 7. IBM, 1997.

[30] Hartmann J. Genrich. Predicate/transition nets. In Wilfried Brauer, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Advances in Petri Nets*, volume 254 of *Lecture Notes in Computer Science*, pages 207–247. Springer, 1986.

[31] Bernd Grahlmann and Eike Best. PEP - more than a Petri net tool. In *TACAs '96: Proceedings of the Second International Workshop on Tools and Algorithms for Construction and Analysis of Systems*, pages 397–401, London, UK, 1996. Springer-Verlag.

[32] Bernd Grahlmann and Carola Pohl. Profiting from Spin in PEP. In *SPIN'98 Workshop*, 1998.

[33] Joseph Y. Halpern and Moshe Y. Vardi. Model checking vs. theorem proving: A manifesto, 1991.

[34] Klaus Havelund, Manuel Núñez, Grigore Rosu, and Burkhart Wolff, editors. *Formal Approaches to Software Testing and Runtime Verification, First Combined International Workshops, FATES 2006 and RV 2006, Seattle, WA, USA, August 15-16, 2006, Revised Selected Papers*, volume 4262 of *Lecture Notes in Computer Science*. Springer, 2006.

[35] X. He and T. Murata. *High-Level Petri Nets - Extensions, Analysis, and Applications*. Electrical Engineering Handbook (ed. Wai-Kai Chen). Elsevier Academic Press, 2005.

[36] Xudong He. A framework for ensuring system dependability from design to implementation. In Ulrich Ultes-Nitsche, Juan Carlos Augusto, and Joseph Barjis, editors, *MSVVEIS*. INSTICC Press INSTICC Press, 2005.

[37] Xudong He and Yi Deng. A framework for developing and analyzing software architecture specifications in SAM. *Comput. J.*, 45(1):111–128, 2002.

[38] Xudong He, Junhua Ding, and Yi Deng. Model checking software architecture specifications in SAM. In *SEKE '02: Proceedings of the 14th international conference on Software engineering and knowledge engineering*, pages 271–274, New York, NY, USA, 2002. ACM.

[39] Xudong He and John A. N. Lee. A methodology for constructing predicate transition net specifications. *Softw., Pract. Exper.*, 21(8):845–875, 1991.

[40] Xudong He, Huiqun Yu, Tianjun Shi, Junhua Ding, and Yi Deng. Formally analyzing software architectural specifications using SAM. *Journal of Systems and Software*, 71(1-2):11–29, 2004.

[41] Tony Hoare. The ideal of program correctness: *Third Computer Journal* lecture. *Comput. J.*, 50(3):254–260, 2007.

[42] Tony Hoare and Jay Misra. Verified software: Theories, tools, experiments vision of a grand challenge project. *Verified Software: Theories, Tools, Experiments: First IFIP TC 2/WG 2.3 Conference, VSTTE 2005, Zurich, Switzerland, October 10-13, 2005, Revised Selected Papers and Discussions*, pages 1–18, 2008.

[43] Gerard J. Holzmann. The model checker SPIN. *IEEE Trans. Softw. Eng.*, 23(5):279–295, 1997.

[44] Gerard J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual.* Addison-Wesley Professional, sixth edition, September 2004.

[45] Hyoung Seok Hong, Insup Lee, Oleg Sokolsky, and Hasan Ural. A temporal logic based theory of test coverage and generation. In Joost-Pieter Katoen and Perdita Stevens, editors, *TACAS*, volume 2280 of *Lecture Notes in Computer Science*, pages 327–341. Springer, 2002.

[46] Kurt Jensen, Lars Kristensen, and Lisa Wells. Coloured Petri nets and CPN tools for modeling and validation of concurrent systems. *International Journal on Software Tools for Technology Transfer (STTT)*, 9(3):213–254, June 2007.

[47] Cem Kaner, Hung Q. Nguyen, and Jack L. Falk. *Testing Computer Software.* John Wiley & Sons, Inc., New York, NY, USA, 1993.

[48] Timo Latvala. Model checking LTL properties of high-level Petri nets with fairness constraints. In José Manuel Colom and Maciej Koutny, editors, *ICATPN*, volume 2075 of *Lecture Notes in Computer Science*, pages 242–262. Springer, 2001.

[49] Timo Latvala and Marko Mäkelä. LTL model checking for modular Petri nets. In Jordi Cortadella and Wolfgang Reisig, editors, *ICATPN*, volume 3099 of *Lecture Notes in Computer Science*, pages 298–311. Springer, 2004.

[50] Patrícia D. L. Machado, Daniel A. Silva, and Alexandre Mota. Towards property oriented testing. *Electr. Notes Theor. Comput. Sci.*, 184:3–19, 2007.

[51] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer-Verlag, 1992.

[52] Ken Mcmillan. *Getting started with SMV*, 1999.

[53] Kenneth L. McMillan. A technique of state space search based on unfolding. *Formal Methods in System Design*, 6(1):45–65, 1995.

[54] Kenneth Lauchlin McMillan. *Symbolic model checking: an approach to the state explosion problem.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1992.

[55] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Trans. Software Eng.*, 26(1):70–93, 2000.

[56] Faron Moller and Alexander Rabinovich. On the expressive power of CTL. In *In Proc. 14th IEEE Symp. Logic in Computer Science (LICS'99*, pages 360–369. IEEE Computer Science Press, 1999.

[57] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[58] 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.

[59] C. A. Petri. *Kommunikation mit Automaten.* PhD thesis, Institut für instrumentelle Mathematik, Bonn, 1962.

[60] Claus Schröter and Victor Khomenko. Parallel LTL-X model checking of high-level Petri nets based on unfoldings. In Rajeev Alur and Doron Peled, editors, *CAV*, volume 3114 of *Lecture Notes in Computer Science*, pages 109–121. Springer, 2004.

[61] Scott Hudson, Georgia Tech. LALR Parser Generator in Java (Java-CUP), 2006. http://www2.cs.tum.edu/projects/cup/.

[62] Ian Sommerville. *Software Engineering.* Addison-Wesley, sixth edition, 2001.

[63] Weixiang Sun, Tianjun Shi, Gonzalo Argote-Garcia, Yi Deng, and Xudong He. Achieving a better middleware design through formal modeling and analysis. In Kang Zhang, George Spanoudakis, and Giuseppe Visaggio, editors, *SEKE*, pages 463–468, 2006.

[64] Li Tan, Oleg Sokolsky, and Insup Lee. Specification-based testing with linear temporal logic. In Du Zhang, Éric Grégoire, and Doug DeGroot, editors, *IRI*, pages 493–498. IEEE Systems, Man, and Cybernetics Society, 2004.

[65] The Eclipse Foundation. Graphical Editing Framework (GEF), 2007. http://www.eclipse.org/gef/.

[66] Toms E. Uribe. Combinations of model checking and theorem proving. In *Proceedings of the Third Intl. Workshop on Frontiers of Combining Systems, volume 1794 of LNCS*, pages 151–170. Springer-Verlag, 2000.

[67] Jeannette M. Wing. A specifier's introduction to formal methods. *IEEE Computer*, 23(9):8–24, 1990.

[68] Jeannette M. Wing and Mandana Vaziri. A case study in model checking software systems. *Sci. Comput. Program.*, 28(2-3):273–299, 1997.

[69] Weifeng Xu and Dianxiang Xu. A model-based approach to test generation for aspect-oriented programs. *First Workshop on Testing Aspect-Oriented Programs (WTAOP'05)*, 2005.

[70] Huiqun Yu, Xudong He, Yi Deng, and Lian Mo. A formal method for analyzing software architecture models in SAM. In *COMPSAC*, pages 645–652. IEEE Computer Society, 2002.

[71] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit testing coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, December 1997.

[72] Hong Zhu and Xudong He. A methodology of testing high-level Petri nets. *Information & Software Technology*, 44(8):473–489, 2002.

[73] Hong Zhu and Xudong He. A theory of behavior observation in software testing. Technical Report CMS-TR-99-05, School of Computing and Mathematical Sciences, Oxford Brookes University, September 1999.

APPENDIX A

SOURCE CODE LISTINGS

The PROMELA code for the Resource Provider example used in the testing procedure is listed below.

```
/*
ResourceProvider.pml
Author: Gonzalo Argote Garcia
Date: March 2009
School of Computing and Information Sciences
Florida International University
Miami FL
*/

/*Bounded values for the places*/
#define DEFAULT_BOUND_PLACE 5
#define BOUND_Req DEFAULT_BOUND_PLACE
#define BOUND_Resp DEFAULT_BOUND_PLACE
#define BOUND_Pending DEFAULT_BOUND_PLACE
#define BOUND_LocalRes DEFAULT_BOUND_PLACE
#define BOUND_Req2Send DEFAULT_BOUND_PLACE
#define BOUND_SysIn DEFAULT_BOUND_PLACE
#define BOUND_SysOut DEFAULT_BOUND_PLACE
#define BOUND_Resources DEFAULT_BOUND_PLACE
#define BOUND_ToLoc DEFAULT_BOUND_PLACE
#define BOUND_CacheIn DEFAULT_BOUND_PLACE
#define BOUND_CacheOut DEFAULT_BOUND_PLACE
#define BOUND_LocIn DEFAULT_BOUND_PLACE
#define BOUND_LocOut DEFAULT_BOUND_PLACE
#define BOUND_AddCache DEFAULT_BOUND_PLACE
#define BOUND_ResCache DEFAULT_BOUND_PLACE

#define DEFAULT_BOUND_PSET 5
/*Types defined in the original SAM model*/
/*
 NAME ::= string
 ID ::= int
 ID_RES ::= int
 REQT ::= ID x NAME
 RESPT ::= REQT x ID_RES
 PREQT ::= P(REQT)
 RESOURT ::= NAME x ID_RES
 PRESOURT ::= P(RESOURT)
*/

/*Type request*/
/*field2 is a short identifying the string*/
typedef REQT{
 int field1;
 short field2;
};

/*Type response*/
typedef RESPT{
 REQT field1;
 int field2;
};

/*Type powerset request*/
typedef PREQT{
 int num;
```

```
 REQT set[DEFAULT_BOUND_PSET];
};

/*Type resource*/
/*field1 is a short identifying the name of the resource*/
typedef RESOURT{
 short field1; /*name*/
 int field2; /*id*/
};

/*Type powerset resource*/
typedef PRESOURT{
 int num;
 RESOURT set[DEFAULT_BOUND_PSET];
};

/*Ports places definitions*/
REQT v_Req[BOUND_Req];
short num_Req;

RESPT v_Resp[BOUND_Resp];
short num_Resp;

REQT v_SysIn[BOUND_SysIn];
short num_SysIn;

RESPT v_SysOut[BOUND_SysOut];
short num_SysOut;

REQT v_CacheIn[BOUND_CacheIn];
short num_CacheIn;

RESPT v_CacheOut[BOUND_CacheOut];
short num_CacheOut;

REQT v_LocIn[BOUND_LocIn];
short num_LocIn;

RESPT v_LocOut[BOUND_LocOut];
short num_LocOut;

RESOURT v_AddCache[BOUND_AddCache];
short num_AddCache;

/*Places in Consumer component*/
PREQT v_Pending[BOUND_Pending];
short num_Pending;

RESPT v_LocalRes[BOUND_LocalRes];
short num_LocalRes;

REQT v_Req2Send[BOUND_Req2Send];
short num_Req2Send;

/*Places in SystemResources component*/
PRESOURT v_Resources[BOUND_Resources];
short num_Resources;

/*Places in RequestHandler component*/
PREQT v_ToLoc[BOUND_ToLoc];
short num_ToLoc;

/*Places in Cache component*/
PRESOURT v_ResCache[BOUND_ResCache];
short num_ResCache;
```

```
/*Variables to coordinate the execution of the processes*/
int _proc_init;
int _proc_num;

/*****************************************************************/
/*Variables to loop and to state whether a given transition is enabled
  or not*/
int _index0; /*variable used for iteration*/
int _index1; /*variable used for iteration*/
int _index2; /*variable used for iteration*/
bool _trans_bEnabled; /*whether a given transition is enabled or not*/

/*Utility macros*/

/*Assigns _right to _left of REQT instances*/
inline assign_REQT(_left,_right){
 _left.field1 = _right.field1;
 _left.field2 = _right.field2
}

/*Compares two REQT instances*/
inline compare_REQT(_e1,_e2){
 (_e1.field1 == _e2.field1 && _e1.field2 == _e2.field2)
}

/*Removes element at _index in _array of type REQT*/
int _array_index;
inline remove_elem_array_REQT(_array,_array_num,_index){
 _array_index = _index;
 do
  :: (_array_index+1 < _array_num) ->
    assign_REQT(_array[_array_index],_array[_array_index+1]);
    _array_index++;
  :: else -> break;
 od;
 _array_num--
}

/*Checks whether _elem_to_find is in _array of type REQT*/
inline check_elem_in_array_REQT(_array,_arraysize,_elem_to_find,_index_found,_found){
 _found = 0;
 _index_found = 0;
 do
  :: (_index_found < _arraysize) ->
  {
   if
    :: compare_REQT(_array[_index_found],_elem_to_find) ->
      _found = 1; break;
    :: else _index_found++;
   fi;
  }
  :: else -> break;
 od
}

/*Assigns _right to _left of type RESPT*/
inline assign_RESPT(_left,_right){
 assign_REQT(_left.field1,_right.field1);
 _left.field2 = _right.field2
}

/*Compares two RESPT instances*/
inline compare_RESPT(_e1,_e2){
 (compare_REQT(_e1.field1,_e2.field1) && _e1.field2 == _e2.field2)
}
```

```
/*Removes element at _index in _array of type RESPT*/
inline remove_elem_array_RESPT(_array,_array_num,_index){
 _array_index = _index;
 do
  :: (_array_index+1 < _array_num) ->
    assign_RESPT(_array[_array_index],_array[_array_index+1]);
    _array_index++;
  :: else -> break;
 od;
 _array_num--
}

/*Assigns _right to _left of type RESOURT*/
inline assign_RESOURT(_left,_right){
 _left.field1 = _right.field1;
 _left.field2 = _right.field2
}

/*Removes element at _index in _array of type RESOURT*/
inline remove_elem_array_RESOURT(_array,_array_num,_index){
 _array_index = _index;
 do
  :: (_array_index+1 < _array_num) ->
    assign_RESOURT(_array[_array_index],_array[_array_index+1]);
    _array_index++;
  :: else -> break;
 od;
 _array_num--
}

/*Compares the names of two RESOURT instances*/
inline compare_name_RESOURT(_e1,_e2){
 (_e1.field1 == _e2.field1)
}

/*Checks whether _elem_to_find has a name in _array of type RESOURT*/
inline check_elem_name_in_array_RESOURT(_array,_arraysize,_elem_to_find,
          _index_found,_found){
 _found = 0;
 _index_found = 0;
 do
  :: (_index_found < _arraysize) ->
  {
   if
    :: compare_name_RESOURT(_array[_index_found],_elem_to_find) ->
      _found = 1; break;
    :: else _index_found++;
   fi;
  }
  :: else -> break;
 od
}


/*****************************************************************/
/*Macros and utilities for testing coverage*/
#define TRANSITION_COVERAGE 1 /*For transition coverage*/
/*#define TRANSITION_COVERAGE 0 For state coverage*/

/*Transition coverage*/
/*Define the array to track the number of times a transition fires
  in a process.*/
#define DEF_ARRAY(_array,_size) int _array[_size]

/*Arrays are defined to keep track of the transitions fired at a given process.*/
#define CONSUMER_CT_TRANS 2 /*Number of transitions at Consumer*/
```

```
DEF_ARRAY(Consumer_trans,CONSUMER_CT_TRANS);

#define SYS_RES_CT_TRANS 2 /*Number of transitions at SystemResources*/
DEF_ARRAY(SystemResources_trans,SYS_RES_CT_TRANS);

#define REQ_HANDLER_CT_TRANS 4 /*Number of transitions at RequestHandler*/
DEF_ARRAY(RequestHandler_trans,REQ_HANDLER_CT_TRANS); /*array of transitions*/

#define CACHE_CT_TRANS 4 /*Number of transitions at Cache*/
DEF_ARRAY(Cache_trans,CACHE_CT_TRANS); /*array of transitions*/

#define LOCATOR_CT_TRANS 3 /*Number of transitions at Locator*/
DEF_ARRAY(Locator_trans,LOCATOR_CT_TRANS); /*array of transitions*/

/*Used in the testing process to select a transition.*/
inline select_tran(_tr,_mod){
 _tr = (_tr+1) % _mod
}

/*Initialize the array of transition fired counters.*/
inline init_array(_array,_size){
 int _idx = 0;
 do
  :: (_idx < _size) -> _array[_idx]=0;
       _idx++;
  :: else -> break;
 od
}

/*Evaluates the transition coverage*/
/*This is specific to this model since we need to know how many processes are*/
inline evaluate_trans_array(_trans_array,_size,_covered){
 _index0 = 0;
 do
  :: (_index0 < _size && _covered == 1) ->
  {
   if
    :: (_trans_array[_index0]==0) -> _covered=0; break;
    :: else -> _index0++
   fi
  }
  :: else -> break
 od
}
bool tr_covered;
inline evaluate_trans_coverage(){
 /*Arrays for evaluating:
   - Consumer_trans (CONSUMER_CT_TRANS)
   - SystemResources_trans (SYS_RES_CT_TRANS)
   - RequestHandler_trans (REQ_HANDLER_CT_TRANS)
   - Cache_trans (CACHE_CT_TRANS)
   - Locator_trans (LOCATOR_CT_TRANS)
 */
 tr_covered = 1;
 evaluate_trans_array(Consumer_trans, CONSUMER_CT_TRANS,tr_covered);
 evaluate_trans_array(SystemResources_trans, SYS_RES_CT_TRANS,tr_covered);
 evaluate_trans_array(RequestHandler_trans, REQ_HANDLER_CT_TRANS,tr_covered);
 evaluate_trans_array(Cache_trans, CACHE_CT_TRANS,tr_covered);
 evaluate_trans_array(Locator_trans, LOCATOR_CT_TRANS,tr_covered);

 /*stop simulation if covered*/
 assert(tr_covered != 1)
}

inline evaluate_state_coverage(){
 /*If the number of elements in the cache is 0 state NOT_CACHED is covered.*/
```

```
 /*Otherwise state CACHED is covered.*/
 assert(v_ResCache.num>0); /*there has to be a cache*/
 assert(State_array[0] == 0 || State_array[1] == 0)
}

/*State coverage*/
/*Several abstract states can be defined. The one chosen here involves
the caching of resources: State1(NOT_CACHED), State2(CACHED)*/
#define STATE_CT 2 /*Number of states*/
DEF_ARRAY(State_array,STATE_CT); /*array of states*/

/*Track the transition firing.*/
inline track_trans(_process_trans,_size,_tran_index){
 /*Record the transitions fired*/
 if
  :: (_tran_index < _size && _tran_index >= 0) ->
   _process_trans[_tran_index]++;
  :: else -> skip
 fi;
 /*Record the states covered*/
 if
  :: (v_ResCache[0].num > 0) -> State_array[1] = 1;
  :: else -> State_array[0] = 1
 fi;
 /*Evaluate depending what kind of coverage was chosen*/
 if
  :: TRANSITION_COVERAGE -> evaluate_trans_coverage();
  :: else -> evaluate_state_coverage()
 fi;

 /*Special situation: if all requests have been sent out and
 Consumer component has received the responses back STOP.
 This is only for this specific test.*/
 assert(num_LocalRes!=5); /*CHECK this value!*/
}

/*****************************************************************/
/*Transitions enabledness testing and firing for Consumer*/

/*Transition: recv_resp*/

/*To keep the token to be removed from Resp*/
RESPT _recv_resp_resp;

inline precond_recv_resp(){
 /*check in Pending[_index1] whether there is one element complying*/
 /*_index1 will be 0 for this example*/
 check_elem_in_array_REQT(v_Pending[0].set,v_Pending[0].num,
  v_Resp[_index0].field1,_index2,_trans_bEnabled)
}

inline is_enabled_recv_resp(){
 _trans_bEnabled = 0;
 _index0 = 0;
 _index1 = 0;
 _index2 = 0;
 /*loop to iterate through the tokens at the places*/
 do
  :: (_index0 < num_Resp && _trans_bEnabled != 1) ->
  {
   _index1 = 0;
   do
    :: (_index1 < num_Pending) ->
    {
     /*compute the guard with the current substitution.*/
     precond_recv_resp();
```

```
      if
       :: _trans_bEnabled == 1 -> _index0--;break;
       :: else -> _index1++;
      fi;
     }
     :: else -> break;
    od
   }
   _index0++;
   :: else -> break;
 od
}

inline fire_recv_resp(){
 /*These variables contain the indexes of the tokens in the places
   to use for firing*/
 /*_index0(v_Resp), _index1(v_Pending),
   _index2(v_Pending[0])*/

 /*Remove the element from v_Resp*/
 /*Store req token since it is going away*/
 assign_RESPT(_recv_resp_resp,v_Resp[_index0]);
 remove_elem_array_RESPT(v_Resp,num_Resp,_index0);

 /*Remove the element from v_Pending[0]*/
 /*No need to store the token being removed*/
 /*_recv_resp_i1 is 0*/
 /*_recv_resp_i2 contains the element that is to be removed from the set*/
 remove_elem_array_REQT(v_Pending[0].set,v_Pending[0].num,_index2);

 /*Add the element to LocalRes*/
 assign_RESPT(v_LocalRes[num_LocalRes],_recv_resp_resp);
 num_LocalRes++ /*check limits*/
}

/*Transition: send_req*/

/*To keep the token to be removed from Req2Send*/
REQT _send_req_req;

inline precond_send_req(){ /*The guard*/
 _trans_bEnabled = 1
}

inline is_enabled_send_req(){
 _trans_bEnabled = 0;
 _index0 = 0;
 _index1 = 0;
 /*loop to iterate through the tokens at the places*/
 do
  :: (_index0 < num_Req2Send && _trans_bEnabled != 1) ->
  {
   _index1 = 0;
   do
    :: (_index1 < num_Pending) ->
    {
     /*compute the guard with the current substitution.*/
     precond_send_req();
     if
      :: _trans_bEnabled == 1 -> _index0--;break;
      :: else -> _index1++;
     fi;
    }
    :: else -> break;
   od
  }
```

```
  _index0++;
  :: else -> break;
 od
}

inline fire_send_req(){
 /*These two variables contain the indexes of the tokens in the places
   to use for firing*/
 /*_send_req_i0(Req2Send), _send_req_i1(Pending)*/

 /*Remove the element from Req2Send*/
 /*Store req token since it is going away*/
 assign_REQT(_send_req_req,v_Req2Send[_index0]);
 remove_elem_array_REQT(v_Req2Send,num_Req2Send,_index0);

 /*Add the element to the first token in Pending and to Req*/
 /*_send_req_i1 is 0*/
 assign_REQT(v_Pending[0].set[v_Pending[0].num],_send_req_req);
 v_Pending[0].num++; /*check limits*/
 assign_REQT(v_Req[num_Req],_send_req_req);
 num_Req++ /*check limits*/
}

/*****************************************************************/
/*Transitions enabledness testing and firing for SystemResources*/

/*Transition: req_ok*/

/*To keep the token to be removed from SysIn*/
REQT _req_ok_req;
RESOURT _req_ok_resource;

inline precond_req_ok(){
 /*check in Resources[_index1] whether there is one element complying*/
 /*_index1 will be 0 for this example*/
 _req_ok_resource.field1 = v_SysIn[_index0].field2;
 check_elem_name_in_array_RESOURT(v_Resources[0].set,
  v_Resources[0].num,_req_ok_resource,_index2,_trans_bEnabled)
}

inline is_enabled_req_ok(){
 _trans_bEnabled = 0;
 _index0 = 0;
 _index1 = 0;
 _index2 = 0;

 /*loop to iterate through the tokens at the places*/
 do
  :: (_index0 < num_SysIn && _trans_bEnabled != 1) ->
  {
   _index1 = 0;
   do
    :: (_index1 < num_Resources) ->
    {
     /*compute the guard with the current substitution.*/
     precond_req_ok();
     if
      :: _trans_bEnabled == 1 -> _index0--; break;
      :: else -> _index1++;
     fi;
    }
    :: else -> break;
   od;
  }
  _index0++;
  :: else -> break;
```

```
 od
}

inline fire_req_ok(){
 /*These variables contain the indexes of the tokens in the places
   to use for firing*/
 /*_index0(v_SysIn), _index1(v_Resources),
   _index2(v_Resources[0])*/

 /*Remove the element from v_SysIn*/
 /*Store req token since it is going away*/
 assign_REQT(_req_ok_req, v_SysIn[_index0]);
 remove_elem_array_REQT(v_SysIn, num_SysIn, _index0);

 /*Nothing to remove from v_Resource[0]*/

 /*Add the response to v_SysOut*/
 assign_REQT(v_SysOut[num_SysOut].field1, _req_ok_req);
 v_SysOut[num_SysOut].field2 = v_Resources[0].set[_index2].field2;
 num_SysOut++ /*check limits*/
}

/*Transition: req_fail*/

/*To keep the token to be removed from SysIn*/
REQT _req_fail_req;
RESOURT _req_fail_resource;

inline precond_req_fail(){
 /*Check in Resources[_index1] whether there is one element complying*/
 /*_index1 will be 0 for this example*/
 /*_index0 is the token to test for*/
 _req_fail_resource.field1 = v_SysIn[_index0].field2;
 check_elem_name_in_array_RESOURT(v_Resources[0].set,
  v_Resources[0].num, _req_fail_resource, _index2, _trans_bEnabled);

 /*If _tran_bEnabled is true the element is in the set, otherwise it is not*/
 /*Change from false to true and vice versa*/
 _trans_bEnabled = !_trans_bEnabled
}

inline is_enabled_req_fail(){
 _trans_bEnabled = 0;
 _index0 = 0;
 _index1 = 0;
 _index2 = 0;
 /*loop to iterate through the tokens at the places*/
 do
  :: (_index0 < num_SysIn && _trans_bEnabled != 1) ->
  {
   _index1 = 0;
   do
    :: (_index1 < num_Resources) ->
    {
     /*compute the guard with the current substitution.*/
     precond_req_fail();
     if
      :: _trans_bEnabled == 1 -> _index0--; break;
      :: else -> _index1++;
     fi;
    }
    :: else -> break;
   od
  }
  _index0++;
  :: else -> break;
```

```
 od
}

inline fire_req_fail(){
 /*These variables contain the indexes of the tokens in the places
   to use for firing*/
 /*_index0(v_SysIn), _index1(v_Resources)*/

 /*Remove the element from SysIn*/
 /*Store req token since it is going away*/
 assign_REQT(_req_fail_req,v_SysIn[_index0]);
 remove_elem_array_REQT(v_SysIn,num_SysIn,_index0);

 /*Nothing to remove from v_Resource[0]*/

 /*Add the response to v_SysOut*/
 assign_REQT(v_SysOut[num_SysOut].field1,_req_fail_req);
 v_SysOut[num_SysOut].field2 = 0; /*no resource!*/
 num_SysOut++ /*check limits*/
}

/*****************************************************************/
/*Transitions enabledness testing and firing for RequestHandler*/

/*Transition: _check_cache*/

/*To keep the token to be removed from Req*/
REQT _check_cache_r;

inline precond_check_cache(){
 _trans_bEnabled = 1
}

inline is_enabled_check_cache(){
 _trans_bEnabled = 0;
 _index0 = 0;
 _index1 = 0;
 /*loop to iterate through the tokens at the places*/
 do
  :: (_index0 < num_Req && _trans_bEnabled != 1) ->
  {
   _index1 = 0;
   do
    :: (_index1 < num_ToLoc) ->
    {
     /*compute the guard with the current substitution.*/
     precond_check_cache();
     if
      :: _trans_bEnabled == 1 -> _index0--; break;
      :: else -> _index1++;
     fi;
    }
    :: else -> break;
   od
  }
  _index0++;
  :: else -> break;
 od
}

inline fire_check_cache(){
 /*These variables contain the indexes of the tokens in the places
   to use for firing*/
 /*_index0(v_SysIn), _index1(v_Resources)*/

 /*Remove the element from v_Req*/
```

```
 /*Store r token since it is going away*/
 assign_REQT(_check_cache_r,v_Req[_index0]);
 remove_elem_array_REQT(v_Req,num_Req,_index0);

 /*Add the element to v_ToLoc[_index1]*/
 /*_index1 is 0 for this example*/
 assign_REQT(v_ToLoc[0].set[v_ToLoc[0].num],_check_cache_r);
 v_ToLoc[0].num++; /*check limits*/

 /*Add the element to v_CacheIn*/
 assign_REQT(v_CacheIn[num_CacheIn],_check_cache_r);
 num_CacheIn++ /*check limits*/
}

/*Transition: in_cache*/

/*To keep the token to be removed from CacheOut*/
RESPT _in_cache_r;

inline precond_in_cache(){
 /*First field of v_CacheOut[_index0] has to be different to 0*/
 if
  :: (v_CacheOut[_index0].field2 != 0 ) ->
   {
   /*Check in ToLoc[_index1] whether there is one element complying*/
   /*_index1 will be 0 for this example*/
   /*_index0 is the token to test for*/
   /*_index2 will contain the index for the matching token in the set*/
   check_elem_in_array_REQT(v_ToLoc[0].set,v_ToLoc[0].num,
    v_CacheOut[_index0].field1,_index2,_trans_bEnabled);
   }
  :: else -> _trans_bEnabled = 0;
 fi
}

inline is_enabled_in_cache(){
 _trans_bEnabled = 0;
 _index0 = 0;
 _index1 = 0;
 _index2 = 0;
 /*loop to iterate through the tokens at the places*/
 do
  :: (_index0 < num_CacheOut) ->
  {
   _index1 = 0;
   do
    :: (_index1 < num_ToLoc && _trans_bEnabled != 1) ->
    {
     /*compute the guard with the current substitution.*/
     precond_in_cache();
     if
      :: _trans_bEnabled == 1 -> _index0--; break;
      :: else -> _index1++;
     fi;
    }
    :: else -> break;
   od
  }
  _index0++;
  :: else -> break;
 od
}
inline fire_in_cache(){
 /*These variables contain the indexes of the tokens in the places
   to use for firing*/
 /*_index0(v_CacheOut), _index1(v_ToLoc),
```

```
    _index2(v_ToLoc[0])*/

  /*Remove the element from v_CacheOut*/
  /*Store r token since it is going away*/
  assign_RESPT(_in_cache_r,v_CacheOut[_index0]);
  remove_elem_array_RESPT(v_CacheOut,num_CacheOut,_index0);

  /*Remove the element from v_ToLoc[_index1]*/
  /*_index1 is 0 for this example*/
  /*_index2 is the element to remove at v_ToLoc[_index1]*/
  remove_elem_array_REQT(v_ToLoc[0].set,v_ToLoc[0].num,_index2);

  /*Add the element to v_Resp*/
  assign_RESPT(v_Resp[num_Resp],_in_cache_r);
  num_Resp++ /*check limits*/
}

/*Transition: not_in_cache*/

/*To keep the token to be removed from CacheOut*/
RESPT _not_in_cache_r;

inline precond_not_in_cache(){
 /*Test for first field of v_CacheOut[_index0]*/
 if
  :: (v_CacheOut[_index0].field2 == 0 ) -> _trans_bEnabled = 1;
  :: else -> _trans_bEnabled = 0;
 fi
}

inline is_enabled_not_in_cache(){
 _trans_bEnabled = 0;
 _index0 = 0;
 _index1 = 0;
 _index2 = 0;
 /*loop to iterate through the tokens at the places*/
 do
  :: (_index0 < num_CacheOut && _trans_bEnabled != 1) ->
  {
   _index1 = 0;
   do
    :: (_index1 < num_ToLoc) ->
    {
     /*compute the guard with the current substitution.*/
     precond_not_in_cache();
     if
      :: _trans_bEnabled == 1 -> _index0--; break;
      :: else -> _index1++;
     fi;
    }
    :: else -> break;
   od
  }
  _index0++;
  :: else -> break;
 od
}

inline fire_not_in_cache(){
 /*These variables contain the indexes of the tokens in the places
   to use for firing*/
 /*_index0(v_CacheOut)*/

 /*Remove the element from v_CacheOut*/
 /*Store r token since it is going away*/
 assign_RESPT(_not_in_cache_r,v_CacheOut[_index0]);
```

```
 remove_elem_array_RESPT(v_CacheOut,num_CacheOut,_index0);

 /*Add the element to v_LocIn*/
 assign_REQT(v_LocIn[num_LocIn],_not_in_cache_r.field1);
 num_LocIn++ /*check limits*/
}


/*Transition: loc_resp*/

/*To keep the token to be removed from LocOut*/
RESPT _loc_resp_r;

inline precond_loc_resp(){
 /*Check in ToLoc[_index1] whether there is one element complying*/
 /*_index1 will be 0 for this example*/
 /*_index0 is the token to test for LocOut[_index0]*/
 /*_index2 will contain the index for the matching token in the set*/
 check_elem_in_array_REQT(v_ToLoc[0].set,v_ToLoc[0].num,
  v_LocOut[_index0].field1,_index2,_trans_bEnabled)
}

inline is_enabled_loc_resp(){
 _trans_bEnabled = 0;
 _index0 = 0;
 _index1 = 0;
 _index2 = 0;
 /*loop to iterate through the tokens at the places*/
 do
  :: (_index0 < num_LocOut && _trans_bEnabled != 1) ->
  {
   _index1 = 0;
   do
    :: (_index1 < num_ToLoc) ->
    {
     /*compute the guard with the current substitution.*/
     precond_loc_resp();
     if
      :: _trans_bEnabled == 1 -> _index0--; break;
      :: else -> _index1++;
     fi;
    }
    :: else -> break;
   od
  }
  _index0++;
  :: else -> break;
 od
}

inline fire_loc_resp(){
 /*These variables contain the indexes of the tokens in the places
   to use for firing*/
 /*_index0(v_LocOut), _index1(v_ToLoc),
   _index2(v_ToLoc[0])*/

 /*Remove the element from v_CacheOut*/
 /*Store r token since it is going away*/
 assign_RESPT(_loc_resp_r,v_LocOut[_index0]);
 remove_elem_array_RESPT(v_LocOut,num_LocOut,_index0);

 /*Remove the element from v_ToLoc[_index1]*/
 /*_index1 is 0 for this example*/
 /*_index2 is the element to remove at v_ToLoc[_index1]*/
 remove_elem_array_REQT(v_ToLoc[0].set,v_ToLoc[0].num,_index2);
```

```
 /*Add the element to v_Resp*/
 assign_RESPT(v_Resp[num_Resp], _loc_resp_r);
 num_Resp++ /*check limits*/
}

/*****************************************************************/
/*Transitions enabledness testing and firing for Cache*/

/*Transition: exists*/

/*To keep the token to be removed from CacheIn*/
REQT _exists_r;
RESOURT _exists_res;

inline precond_exists(){
 /*Check in ResCache[_index1] whether there is one element complying*/
 /*_index1 will be 0 for this example*/
 /*_index0 is the token to test for*/
 /*_index2 will contain the index for the matching token in the set*/
 _exists_res.field1 = v_CacheIn[_index0].field2;
 check_elem_name_in_array_RESOURT(v_ResCache[0].set,
  v_ResCache[0].num, _exists_res, _index2, _trans_bEnabled)
}

inline is_enabled_exists(){
 _trans_bEnabled = 0;
 _index0 = 0;
 _index1 = 0;
 _index2 = 0;
 /*loop to iterate through the tokens at the places*/
 do
  :: (_index0 < num_CacheIn && _trans_bEnabled != 1) ->
  {
   _index1 = 0;
   do
    :: (_index1 < num_ResCache) ->
    {
     /*compute the guard with the current substitution.*/
     precond_exists();
     if
      :: _trans_bEnabled == 1 -> _index0--; break;
      :: else -> _index1++;
     fi;
    }
    :: else -> break;
   od
  }
  _index0++;
  :: else -> break;
 od
}

inline fire_exists(){
 /*These variables contain the indexes of the tokens in the places
   to use for firing*/
 /*_index0(v_CacheIn), _index1(v_ResCache),
   _index2(v_ResCache[0])*/

 /*Remove the element from v_CacheIn*/
 /*Store r token since it is going away*/
 assign_REQT(_exists_r, v_CacheIn[_index0]);
 remove_elem_array_REQT(v_CacheIn, num_CacheIn, _index0);

 /*v_ResCache[_index1] is left untouched*/

 /*Add the element (r1) to v_CacheOut*/
```

```
 assign_REQT(v_CacheOut[num_CacheOut].field1,_exists_r);
 v_CacheOut[num_CacheOut].field2 = v_ResCache[0].set[_index2].field2;
 num_CacheOut++ /*check limits*/
}

/*Transition: _not_exists*/

/*To keep the token to be removed from CacheIn*/
REQT _not_exists_r;
RESOURT _not_exists_res;

inline precond_not_exists(){
 /*Check in ResCache[_index1] whether there is one element complying*/
 /*_index1 will be 0 for this example*/
 /*_index0 is the token to test for*/
 _not_exists_res.field1 = v_CacheIn[_index0].field2;
 check_elem_name_in_array_RESOURT(v_ResCache[0].set,
  v_ResCache[0].num,_not_exists_res,_index2,_trans_bEnabled);

 /*If _tran_bEnabled is true the element is in the set, otherwise it is not*/
 /*Change from false to true and vice versa*/
 _trans_bEnabled = !_trans_bEnabled
}

inline is_enabled_not_exists(){
 _trans_bEnabled = 0;
 _index0 = 0;
 _index1 = 0;
 /*loop to iterate through the tokens at the places*/
 do
  :: (_index0 < num_CacheIn && _trans_bEnabled != 1) ->
  {
   _index1 = 0;
   do
    :: (_index1 < num_ResCache) ->
    {
     /*compute the guard with the current substitution.*/
     precond_not_exists();
     if
      :: _trans_bEnabled == 1 -> _index0--; break;
      :: else -> _index1++;
     fi;
    }
    :: else -> break;
   od
  }
  _index0++;
  :: else -> break;
 od
}

inline fire_not_exists(){
 /*These variables contain the indexes of the tokens in the places
   to use for firing*/
 /*_index0(v_CacheIn), _index1(v_ResCache)*/

 /*Remove the element from v_CacheIn*/
 /*Store r token since it is going away*/
 assign_REQT(_not_exists_r,v_CacheIn[_index0]);
 remove_elem_array_REQT(v_CacheIn,num_CacheIn,_index0);

 /*v_ResCache[_index1] is left untouched*/

 /*Add the element (r1) to v_CacheOut*/
 assign_REQT(v_CacheOut[num_CacheOut].field1,_not_exists_r);
 v_CacheOut[num_CacheOut].field2 = 0;/*is not in the cache*/
```

```
 num_CacheOut++ /*check limits*/
}

/*Transition: _add_cache*/

/*To keep the token to be removed from AddCache*/
RESOURT _add_cache_c;

inline precond_add_cache(){
 /*Add only if there is not already a resource with the same name cached.*/
 /*_index0 (v_AddCache).*/
 /*_index1 is always 0.*/
 check_elem_name_in_array_RESOURT(v_ResCache[0].set,
  v_ResCache[0].num,v_AddCache[_index0],_index2,_trans_bEnabled);
 _trans_bEnabled = !_trans_bEnabled
}

inline is_enabled_add_cache(){
 _trans_bEnabled = 0;
 _index0 = 0;
 _index1 = 0;
 /*loop to iterate through the tokens at the places*/
 do
  :: (_index0 < num_AddCache && _trans_bEnabled != 1) ->
  {
   _index1 = 0;
   do
    :: (_index1 < num_ResCache) ->
    {
     /*compute the guard with the current substitution.*/
     precond_add_cache();
     if
      :: _trans_bEnabled == 1 -> _index0--; break;
      :: else -> _index1++;
     fi;
    }
    :: else -> break;
   od
  }
  _index0++;
  :: else -> break;
 od
}

inline fire_add_cache(){
 /*These variables contain the indexes of the tokens in the places
   to use for firing*/
 /*_index0(v_AddCache)*/

 /*Remove the element from v_AddCache*/
 /*Store c token since it is going away*/
 assign_RESOURT(_add_cache_c,v_AddCache[_index0]);
 remove_elem_array_RESOURT(v_AddCache,num_AddCache,_index0);

 /*v_ResCache[_index1] is left untouched*/

 /*Add the element (r1) to v_ResCache[0]*/
 assign_RESOURT(v_ResCache[0].set[v_ResCache[0].num],_add_cache_c);
 v_ResCache[0].num++ /*check limits*/
}

/*Transition: _add_cache_fail*/

/*To keep the token to be removed from AddCache*/
RESOURT _add_cache_fail_c;
```

```
inline precond_add_cache_fail(){
 /*Add only if there is not already a resource with the same name cached.*/
 /*_index0 (v_AddCache).*/
 /*_index1 is always 0.*/
 check_elem_name_in_array_RESOURT(v_ResCache[0].set,
  v_ResCache[0].num,v_AddCache[_index0],_index2,_trans_bEnabled)
}

inline is_enabled_add_cache_fail(){
 _trans_bEnabled = 0;
 _index0 = 0;
 _index1 = 0;
 /*loop to iterate through the tokens at the places*/
 do
  :: (_index0 < num_AddCache && _trans_bEnabled != 1) ->
  {
   _index1 = 0;
   do
    :: (_index1 < num_ResCache) ->
    {
     /*compute the guard with the current substitution.*/
     precond_add_cache();
     if
      :: _trans_bEnabled == 1 -> _index0--; break;
      :: else -> _index1++;
     fi;
    }
    :: else -> break;
   od
  }
  _index0++;
  :: else -> break;
 od
}

inline fire_add_cache_fail(){
 /*These variables contain the indexes of the tokens in the places
   to use for firing*/
 /*_index0(v_AddCache)*/

 /*Remove the element from v_AddCache*/
 /*Store c token since it is going away*/
 assign_RESOURT(_add_cache_fail_c,v_AddCache[_index0]);
 remove_elem_array_RESOURT(v_AddCache,num_AddCache,_index0);

 /*v_ResCache[_index1] is left untouched*/

 /*The element is lost, no need to add if it is already there*/
}

/*****************************************************************/
/*Transitions enabledness testing and firing for Locator*/

/*Transition: loc_req*/

/*To keep the token to be removed from LocIn*/
REQT _loc_req_r;

inline precond_loc_req(){
 _trans_bEnabled = 1
}

inline is_enabled_loc_req(){
 _trans_bEnabled = 0;
 _index0 = 0;
 /*loop to iterate through the tokens at the places*/
```

```
 do
  :: (_index0 < num_LocIn) ->
  {
   /*compute the guard with the current substitution.*/
   precond_loc_req();
   if
    :: _trans_bEnabled == 1 -> break;
    :: else -> _index0++;
   fi;
  }
  :: else -> break;
 od
}

inline fire_loc_req(){
 /*These variables contain the indexes of the tokens in the places
   to use for firing*/
 /*_index0(v_LocIn)*/

 /*Remove the element from v_LocIn*/
 /*Store r token since it is going away*/
 assign_REQT(_loc_req_r, v_LocIn[_index0]);
 remove_elem_array_REQT(v_LocIn, num_LocIn, _index0);

 /*Add the element (r) to v_SysIn*/
 assign_REQT(v_SysIn[num_SysIn], _loc_req_r);
 num_SysIn++ /*check limits*/
}

/*Transition: loc_ok*/

/*To keep the token to be removed from SysOut*/
RESPT _loc_ok_r;

inline precond_loc_ok(){
 if
  :: (v_SysOut[_index0].field2 != 0) -> _trans_bEnabled = 1;
  :: else -> _trans_bEnabled = 0;
 fi
}

inline is_enabled_loc_ok(){
 _trans_bEnabled = 0;
 _index0 = 0;
 /*loop to iterate through the tokens at the places*/
 do
  :: (_index0 < num_SysOut) ->
  {
   /*compute the guard with the current substitution.*/
   precond_loc_ok();
   if
    :: _trans_bEnabled == 1 -> break;
    :: else -> _index0++;
   fi;
  }
  :: else -> break;
 od
}

inline fire_loc_ok(){
 /*These variables contain the indexes of the tokens in the places
   to use for firing*/
 /*_index0(v_SysOut)*/

 /*Remove the element from v_SysOut*/
 /*Store r token since it is going away*/
```

```
 assign_RESPT(_loc_ok_r,v_SysOut[_index0]);
 remove_elem_array_RESPT(v_SysOut,num_SysOut,_index0);

 /*Add the element (r) to v_LocOut*/
 assign_RESPT(v_LocOut[num_LocOut],_loc_ok_r);
 num_LocOut++; /*check limits*/

 /*Add the resource to the cache*/
 v_AddCache[num_AddCache].field1 = _loc_ok_r.field1.field2;
 v_AddCache[num_AddCache].field2 = _loc_ok_r.field2;
 num_AddCache++ /*check limits*/
}

/*Transition: loc_fail*/

/*To keep the token to be removed from SysOut*/
RESPT _loc_fail_r;

inline precond_loc_fail(){
 /*If the resource id is 0, then failed*/
 if
  :: (v_SysOut[_index0].field2 == 0) -> _trans_bEnabled = 1;
  :: else -> _trans_bEnabled = 0;
 fi
}

inline is_enabled_loc_fail(){
 _trans_bEnabled = 0;
 _index0 = 0;
 /*loop to iterate through the tokens at the places*/
 do
  :: (_index0 < num_SysOut) ->
  {
   /*compute the guard with the current substitution.*/
   precond_loc_fail();
   if
    :: _trans_bEnabled == 1 -> break;
    :: else -> _index0++;
   fi;
  }
  :: else -> break;
 od
}

inline fire_loc_fail(){
 /*These variables contain the indexes of the tokens in the places
   to use for firing*/
 /*_index0(v_SysOut)*/

 /*Remove the element from v_SysOut*/
 /*Store r token since it is going away*/
 assign_RESPT(_loc_fail_r,v_SysOut[_index0]);
 remove_elem_array_RESPT(v_SysOut,num_SysOut,_index0);

 /*Add the element (r) to v_LocOut*/
 assign_RESPT(v_LocOut[num_LocOut],_loc_fail_r);
 num_LocOut++ /*check limits*/
}

/*****************************************************************/
/*Consumer process*/
/* Original SAM spec:
Type(Req2Send) = REQT
Type(Pending) = PREQT
Type(LocalRes) = RESPT
Type(Resp) = RESPT
```

```
Type(Req) = REQT
M0(Req2Send) = {<1,"resource1">,<2,"resource2">,<3,"resource3">}
M0(Pending) = {{}}
M0(LocalRes) = {}
M0(Resp) = {}
M0(Req) = {}
R(send_req)=(true) \and (R1=R \union {req})
R(recv_resp) = (\exists r \in R \dot (resp[1][1]=r[1] \and resp[1][2]=r[2]))
    \and (res=resp \and (R1=R \union {resp[1]}))
*/
proctype Consumer(){
 /*Define the variables for testing.*/
 byte tran = 0;
 byte tran_n = CONSUMER_CT_TRANS;
 init_array(Consumer_trans,CONSUMER_CT_TRANS);
 /*Net places and initial marking.*/
 num_Pending=1;
 v_Pending[0].num=0;
 num_LocalRes=0;
 num_Req2Send=5;
 v_Req2Send[0].field1=1;
 v_Req2Send[0].field2=1; /*"resource2"*/
 v_Req2Send[1].field1=2;
 v_Req2Send[1].field2=0; /*"resource1"*/
 v_Req2Send[2].field1=3;
 v_Req2Send[2].field2=2; /*"resource3"*/
 v_Req2Send[3].field1=4;
 v_Req2Send[3].field2=10; /*"resourcex"*/
 v_Req2Send[4].field1=5;
 v_Req2Send[4].field2=1; /*"resource2"*/
 /*Increment the counter and wait for the other processes to start.*/
 _proc_init++;
 ( _proc_init == _proc_num );
 /*Test enabledness and fire.*/
 do
  :: atomic{tran == 0 -> is_enabled_recv_resp();
     if
      :: _trans_bEnabled -> fire_recv_resp();
       track_trans(Consumer_trans,CONSUMER_CT_TRANS,tran);
      :: else -> skip;
     fi;
     select_tran(tran,tran_n);
    }
  :: atomic{tran == 1 -> is_enabled_send_req();
     if
      :: _trans_bEnabled -> fire_send_req();
       track_trans(Consumer_trans,CONSUMER_CT_TRANS,tran);
      :: else -> skip;
     fi;
     select_tran(tran,tran_n);
    }
 od
}

/*****************************************************************/
/*SystemResources process*/
/* Original SAM spec:
Type(SysIn) = REQT
Type(SysOut) = RESPT
Type(Resources) = PRESOURT
M0(SysIn) = {}
M0(SysOut) = {}
M0(Resources) = {{<"resource2",1002>,<"resource3",1003>,<"resource4",1004>}}
R(req_ok)= (\exists r \in R \dot (req[2]=r[1]))
    \and ( \exists r \in R \dot (req[2]=r[1] \and resp[1]=req \and resp[2]=r[2]))
R(req_fail)= (\notexists r \in R \dot (req[2]=r[1]))
```

```
    \and (resp[1]=req \and resp[2]=0)
*/
proctype SystemResources(){
 /*Define the variables for testing.*/
 byte tran = 0;
 byte tran_n = SYS_RES_CT_TRANS;
 init_array(SystemResources_trans,SYS_RES_CT_TRANS);
 /*Net places and initial marking.*/
 num_Resources=1;
 v_Resources[0].set[0].field1=1;
 v_Resources[0].set[0].field2=1002;
 v_Resources[0].set[1].field1=2;
 v_Resources[0].set[1].field2=1003;
 v_Resources[0].set[2].field1=3;
 v_Resources[0].set[2].field2=1004;
 v_Resources[0].num=3;
 /*Increment the counter and wait for the other processes to start.*/
 _proc_init++;
 ( _proc_init == _proc_num );
 /*Test enabledness and fire.*/
 do
  :: atomic{tran == 0 -> is_enabled_req_ok();
    if
     :: _trans_bEnabled -> fire_req_ok();
      track_trans(SystemResources_trans,SYS_RES_CT_TRANS,tran);
     :: else -> skip;
    fi;
    select_tran(tran,tran_n);
   }
  :: atomic{tran == 1 -> is_enabled_req_fail();
    if
     :: _trans_bEnabled -> fire_req_fail();
      track_trans(SystemResources_trans,SYS_RES_CT_TRANS,tran);
     :: else -> skip;
    fi;
    select_tran(tran,tran_n);
   }
 od
}

/*****************************************************************/
/*RequestHandler process*/
/* Original SAM spec:
Type(Req) = REQT
Type(Resp) = RESPT
Type(CacheIn) = REQT
Type(CacheOut) = RESPT
Type(LocIn) = REQT
Type(LocOut) = RESPT
Type(ToLoc) = PREQT
M0(Req) = {}
M0(Resp) = {}
M0(CacheIn) = {}
M0(CacheOut) = {}
M0(LocIn) = {}
M0(LocOut) = {}
M0(ToLoc) = {{}}
R(check_cache) = true \and (L1=L \union {r})
R(in_cache) = (r[2] \nequal 0 \and (\exists x \in L \dot ((r[1][1]=x[1])
    \and(r[1][2]=x[2])))) \and (L1=L \minus {r[1]})
R(not_in_cache)=(r[2]=0) \and (r1=r[1])
R(loc_resp)= (\exists x \in T \dot ((r[1][1]=x[1]) \and (r[1][2]=x[2])))
    \and (T1=T \minus {r[1]})
*/
proctype RequestHandler(){
 /*Define the variables for testing.*/
```

```
  byte tran = 0;
  byte tran_n = 4;
  init_array(RequestHandler_trans,REQ_HANDLER_CT_TRANS);
  /*Net places and initial marking.*/
  num_ToLoc=1;
  v_ToLoc[0].num=0;
  /*Increment the counter and wait for the other processes to start.*/
  -proc_init++;
  ( _proc_init == _proc_num );
  /*Test enabledness and fire.*/
  do
   :: atomic{tran == 0 -> is_enabled_check_cache();
     if
      :: _trans_bEnabled -> fire_check_cache();
       track_trans(RequestHandler_trans,REQ_HANDLER_CT_TRANS,tran);
      :: else -> skip;
     fi;
     select_tran(tran,tran_n);
    }
   :: atomic{tran == 1 -> is_enabled_in_cache();
     if
      :: _trans_bEnabled -> fire_in_cache();
       track_trans(RequestHandler_trans,REQ_HANDLER_CT_TRANS,tran);
      :: else -> skip;
     fi;
     select_tran(tran,tran_n);
    }
   :: atomic{tran == 2 -> is_enabled_not_in_cache();
     if
      :: _trans_bEnabled -> fire_not_in_cache();
       track_trans(RequestHandler_trans,REQ_HANDLER_CT_TRANS,tran);
      :: else -> skip;
     fi;
     select_tran(tran,tran_n);
    }
   :: atomic{tran == 3 -> is_enabled_loc_resp();
     if
      :: _trans_bEnabled -> fire_loc_resp();
       track_trans(RequestHandler_trans,REQ_HANDLER_CT_TRANS,tran);
      :: else -> skip;
     fi;
     select_tran(tran,tran_n);
    }
  od
}

/*****************************************************************/
/*Cache process*/
/* Original SAM spec:
Type(CacheIn) = REQT
Type(CacheOut) = RESPT
Type(ResCache) = PRESOURT
Type(AddCache) = RESOURT
M0(CacheIn) = {}
M0(CacheOut) = {}
M0(ResCache) = {{}}
M0(AddCache) = {}
R(exists)=(\exists c \in C \dot (r[2]=c[1]))
  \and (\exists c \in C \dot (r[2]=c[1]) \and (r[1]=r) \and (r[2]=c[2]))
R(not_exists)=(\notexists c \in C \dot (r[2]=c[1]))
  \and (r[1]=r \and r[2]=0)
R(add_cache)= (\notexists x \in C \dot (x[1]=c[1] \and x[2]=c[2]))
  \and (C1=C \union {c})
R(add_cache_fail)= (\exists x \in C \dot (x[1]=c[1] \and x[2]=c[2]))
*/
proctype Cache(){
```

```
/*Define the variables for testing.*/
byte tran = 0;
byte tran_n = CACHE_CT_TRANS;
init_array(Cache_trans,CACHE_CT_TRANS);
/*Net places and initial marking.*/
num_ResCache=1;
v_ResCache[0].num=0;
/*Increment the counter and wait for the other processes to start.*/
-proc_init++;
( _proc_init == _proc_num );
/*Test enabledness and fire.*/
do
 :: atomic{tran == 0 -> is_enabled_exists();
   if
    :: _trans_bEnabled -> fire_exists();
     track_trans(Cache_trans,CACHE_CT_TRANS,tran);
    :: else -> skip;
   fi;
   select_tran(tran,tran_n);
  }
 :: atomic{tran == 1 -> is_enabled_not_exists();
   if
    :: _trans_bEnabled -> fire_not_exists();
     track_trans(Cache_trans,CACHE_CT_TRANS,tran);
    :: else -> skip;
   fi;
   select_tran(tran,tran_n);
  }
 :: atomic{tran == 2 -> is_enabled_add_cache();
   if
    :: _trans_bEnabled -> fire_add_cache();
     track_trans(Cache_trans,CACHE_CT_TRANS,tran);
    :: else -> skip;
   fi;
   select_tran(tran,tran_n);
  }
 :: atomic{tran == 3 -> is_enabled_add_cache_fail();
   if
    :: _trans_bEnabled -> fire_add_cache_fail();
     track_trans(Cache_trans,CACHE_CT_TRANS,tran);
    :: else -> skip;
   fi;
   select_tran(tran,tran_n);
  }
od
}

/*****************************************************************/
/*Locator process*/
/* Original SAM spec:
Type(LocIn) = REQT
Type(LocOut) = RESPT
Type(SysIn) = REQT
Type(SysOut) = RESPT
Type(AddCache) = RESOURT
M0(LocIn) = {}
M0(LocOut) = {}
M0(SysIn) = {}
M0(SysOut) = {}
M0(AddCache) = {}
R(loc_ok)= (r[2] \notequal 0)\and (c[1]=r[1][2] \and c[2]=r[2])
R(loc_fail)= (r[2] = 0)
R(loc_req)= true
*/
proctype Locator(){
 /*Define the variables for testing.*/
```

```
 byte tran = 0;
 byte tran_n = LOCATOR_CT_TRANS;
 init_array(Locator_trans,LOCATOR_CT_TRANS);
 /*Net places and initial marking.*/
 /*Increment the counter and wait for the other processes to start.*/
 -proc_init++;
 ( -proc_init == _proc_num );
 /*Test enabledness and fire.*/
 do
  :: atomic{tran == 0 -> is_enabled_loc_req();
    if
     :: _trans_bEnabled -> fire_loc_req();
      track_trans(Locator_trans,LOCATOR_CT_TRANS,tran);
     :: else -> skip;
    fi;
    select_tran(tran,tran_n);
   }
  :: atomic{tran == 1 -> is_enabled_loc_ok();
    if
     :: _trans_bEnabled -> fire_loc_ok();
      track_trans(Locator_trans,LOCATOR_CT_TRANS,tran);
     :: else -> skip;
    fi;
    select_tran(tran,tran_n);
   }
  :: atomic{tran == 2 -> is_enabled_loc_fail();
    if
     :: _trans_bEnabled -> fire_loc_fail();
      track_trans(Locator_trans,LOCATOR_CT_TRANS,tran);
     :: else -> skip;
    fi;
    select_tran(tran,tran_n);
   }
 od
}

/*****************************************************************/
/*init process*/
init{
 /*For testing initialize the state array*/
 init_array(State_array,STATE_CT);
accept_init:
 /*Initialize the port places*/
 atomic{
 num_Req=0;
 num_Resp=0;
 num_SysIn=0;
 num_SysOut=0;
 num_CacheIn=0;
 num_CacheOut=0;
 num_LocIn=0;
 num_LocOut=0;
 num_AddCache=0;
 _proc_num = 5;
 _proc_init = 0;
 }
 atomic{
  run Consumer();
  run SystemResources();
  run RequestHandler();
  run Cache();
  run Locator();
 }
}
```

APPENDIX B

PRT NET MODELS

Figure B.1 shows the top level view of the flattened SAM model for the resource provider.

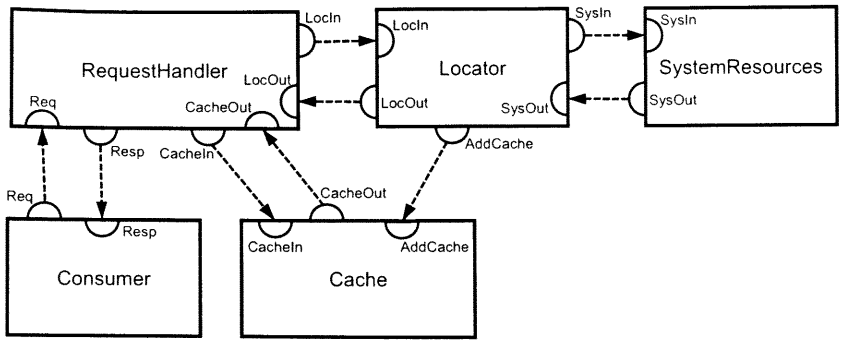

Figure B.1: Top level Resource Provider.

Figure B.2 shows the *Consumer* component PrT net model.

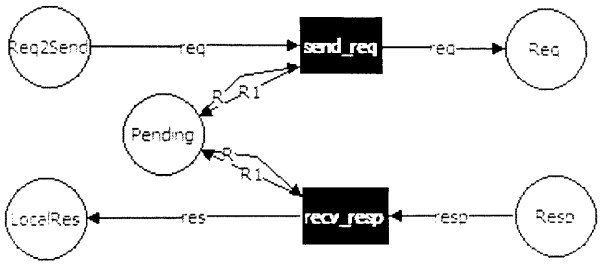

Figure B.2: Consumer component.

The specification for *Consumer* component is:

$\varphi(Req2Send) = REQT$
$\varphi(Pending) = PREQT$
$\varphi(LocalRes) = RESPT$
$\varphi(Resp) = RESPT$
$\varphi(Req) = REQT$
$M0(Req2Send) = \{< 1, "resource1" >, < 2, "resource2" >, < 3, "resource3" >\}$
$M0(Pending) = \{\{\}\}$
$M0(LocalRes) = \{\}$
$M0(Resp) = \{\}$
$M0(Req) = \{\}$
$R(send\_req)\ (true) \wedge (R1 = R \cup \{req\})$
$R(recv\_resp)\ (\exists r \in R \cdot (resp[1][1] = r[1] \wedge resp[1][2] = r[2]))$
$\wedge (res = resp \wedge (R1 = R \cup \{resp[1]\}))$

Figure B.3 shows the *SystemsResources* component PrT net model.

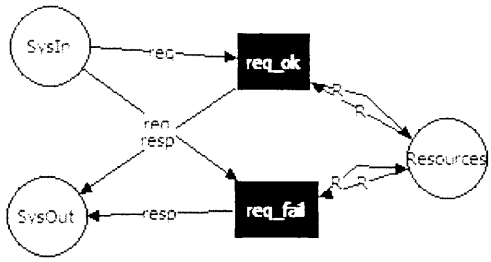

Figure B.3: SystemResources component.

The specification for *SytemResources* component is:

$\varphi(SysIn) = REQT$
$\varphi(SysOut) = RESPT$
$\varphi(Resources) = PRESOURT$
$M0(SysIn) = \{\}$
$M0(SysOut) = \{\}$
$M0(Resources) = \{\{< "resource2", 1002 >, < "resource3", 1003 >, < "resource4", 1004 >\}\}$
$R(req\_ok) = (\exists r \in R \cdot (req[2] = r[1]))$
$\wedge(\exists r \in R \cdot (req[2] = r[1] \wedge resp[1] = req \wedge resp[2] = r[2]))$
$R(req\_fail) = (\neg\exists r \in R \cdot (req[2] = r[1]))$
$\wedge(resp[1] = req \wedge resp[2] = 0)$

Figure B.4 shows *RequestHandler* component PrT net model.

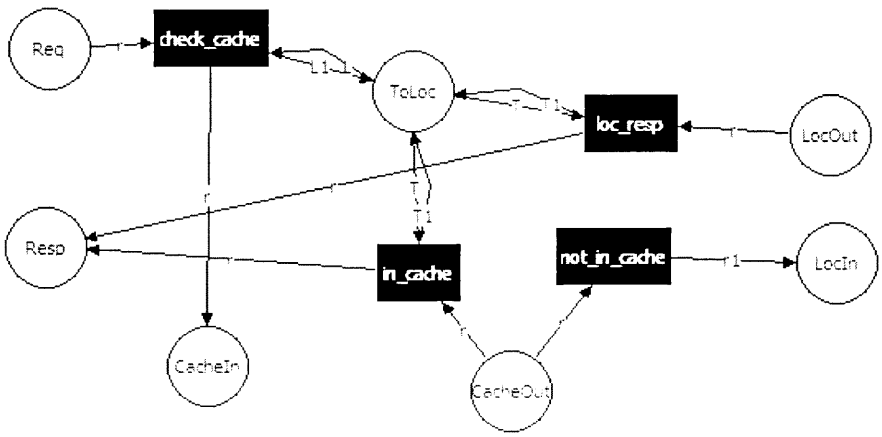

Figure B.4: RequestHandler component.

The specification for *RequestHandler* component is:

$\varphi(Req) = REQT$
$\varphi(Resp) = RESPT$
$\varphi(CacheIn) = REQT$
$\varphi(CacheOut) = RESPT$
$\varphi(LocIn) = REQT$
$\varphi(LocOut) = RESPT$
$\varphi(ToLoc) = PREQT$
$M0(Req) = \{\}$
$M0(Resp) = \{\}$
$M0(CacheIn) = \{\}$
$M0(CacheOut) = \{\}$
$M0(LocIn) = \{\}$
$M0(LocOut) = \{\}$
$M0(ToLoc) = \{\{\}\}$
$R(check_cache) = true \wedge (L1 = L \cup \{r\})$
$R(in_cache) = (r[2] \neq 0 \wedge (\exists x \in L \cdot ((r[1][1] = x[1]) \wedge (r[1][2] = x[2]))))$
$\wedge (L1 = L - \{r[1]\})$
$R(not\_in\_cache) = (r[2] = 0) \wedge (r1 = r[1])$
$R(loc\_resp) = (\exists x \in T \cdot ((r[1][1] = x[1]) \wedge (r[1][2] = x[2])))$
$\wedge (T1 = T - \{r[1]\})$

Figure B.5 shows *Cache* component PrT net model.

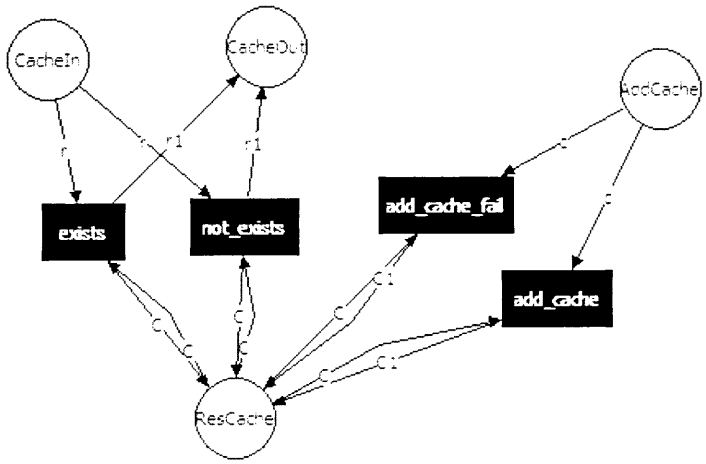

Figure B.5: Cache component.

The specification for *Cache* component is:

$$
\begin{aligned}
&\varphi(CacheIn) = REQT \\
&\varphi(CacheOut) = RESPT \\
&\varphi(ResCache) = PRESOURT \\
&\varphi(AddCache) = RESOURT \\
&M0(CacheIn) = \{\} \\
&M0(CacheOut) = \{\} \\
&M0(ResCache) = \{\{\}\} \\
&M0(AddCache) = \{\} \\
&R(exists) = (\exists c \in C \cdot (r[2] = c[1])) \\
&\wedge(\exists c \in C \cdot (r[2] = c[1]) \wedge (r[1] = r) \wedge (r[2] = c[2])) \\
&R(not_{e}xists) = (\neg\exists c \in C \cdot (r[2] = c[1])) \\
&\wedge(r[1] = r \wedge r[2] = 0) \\
&R(add_{c}ache) = (\neg\exists x \in C \cdot (x[1] = c[1] \wedge x[2] = c[2])) \\
&\wedge(C1 = C \cup c) \\
&R(add_{c}ache_{f}ail) = (\exists x \in C \cdot (x[1] = c[1] \wedge x[2] = c[2]))
\end{aligned}
$$

Figure B.6 shows *Locator* component PrT net model.

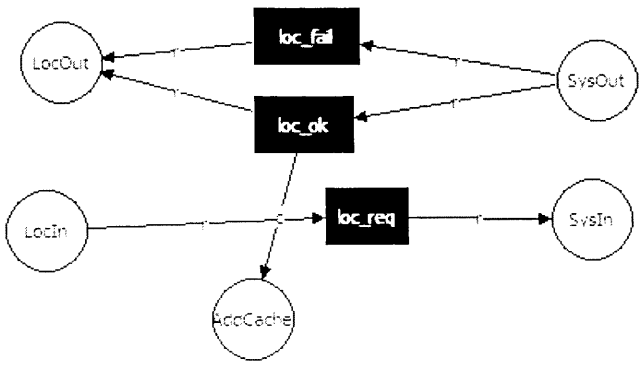

Figure B.6: Locator component.

The specification for *Locator* component is:

$$\varphi(LocIn) = REQT$$
$$\varphi(LocOut) = RESPT$$
$$\varphi(SysIn) = REQT$$
$$\varphi(SysOut) = RESPT$$
$$\varphi(AddCache) = RESOURT$$
$$M0(LocIn) = \{\}$$
$$M0(LocOut) = \{\}$$
$$M0(SysIn) = \{\}$$
$$M0(SysOut) = \{\}$$
$$M0(AddCache) = \{\}$$
$$R(loc\_ok) = (r[2] \neq 0) \wedge (c[1] = r[1][2] \wedge c[2] = r[2])$$
$$R(loc\_fail) = (r[2] = 0)$$
$$R(loc\_req) = true$$

APPENDIX C

PROMELA LANGUAGE

The grammar of the PROMELA language is provided below.

Grammar Rules

```
spec : module [ module ] *

module : proctype /* proctype declaration */
 | init  /* init process       */
 | never  /* never claim        */
 | trace  /* event trace        */
 | utype  /* user defined types */
 | mtype  /* mtype declaration  */
 | decl_lst /* global vars, chans */

proctype: [ active ] PROCTYPE name '(' [ decl_lst ]')'
   [ priority ] [ enabler ] '{' sequence '}'

init : INIT [ priority ] '{' sequence '}'

never : NEVER '{' sequence '}'

trace : TRACE '{' sequence '}'

utype : TYPEDEF name '{' decl_lst '}'

mtype : MTYPE [ '=' ] '{' name [ ',' name ] * '}'

decl_lst: one_decl [ ';' one_decl ] *

one_decl: [ visible ] typename  ivar [',' ivar ] *

typename: BIT | BOOL | BYTE | SHORT | INT | MTYPE | CHAN
 | uname /* user defined type names (see utype) */

active  : ACTIVE [ '[' const ']' ] /* instantiation */

priority: PRIORITY const /* simulation priority */

enabler : PROVIDED '(' expr ')' /* execution constraint */

visible : HIDDEN | SHOW

sequence: step [ ';' step ] *

step    : stmnt [ UNLESS stmnt ]
 | decl_lst
 | XR varref [',' varref ] *
```

```
 | XS varref [ ',' varref ] *

ivar     : name [ '[' const ']' ] [ '=' any_expr | '=' ch_init ]

ch_init : '[' const ']' OF '{' typename [ ',' typename ] * '}'

varref : name [ '[' any_expr ']' ] [ '.' varref ]

send     : varref '!' send_args  /* normal fifo send */
 | varref '!' '!' send_args /* sorted send */

receive : varref '?' recv_args  /* normal receive */
 | varref '?' '?' recv_args /* random receive */
 | varref '?' '<' recv_args '>' /* poll with side-effect */
 | varref '?' '?' '<' recv_args '>' /* ditto */

poll     : varref '?' '[' recv_args ']' /* poll without side-effect */
 | varref '?' '?' '[' recv_args ']' /* ditto */

send_args: arg_lst | any_expr '(' arg_lst ')'

arg_lst   : any_expr [ ',' any_expr ] *

recv_args: recv_arg [ ',' recv_arg ] *  |  recv_arg '(' recv_args ')'

recv_arg : varref | EVAL '(' varref ')' | [ '-' ] const

assign   : varref '=' any_expr /* standard assignment */
 | varref '+' '+' /* increment */
 | varref '-' '-' /* decrement */

stmnt : IF options FI  /* selection */
 | DO options OD  /* iteration */
 | ATOMIC '{' sequence '}' /* atomic sequence */
 | D_STEP '{' sequence '}' /* deterministic atomic */
 | '{' sequence '}' /* normal sequence */
 | send
 | receive
 | assign
 | ELSE   /* used inside options */
 | BREAK   /* used inside iterations */
 | GOTO name
 | name ':' stmnt /* labeled statement */
 | PRINT '(' string [ ',' arg_lst ] ')'
 | ASSERT expr
 | expr   /* condition */
 | c_code '{' ... '}' /* embedded C code */
 | c_expr '{' ... '}'
 | c_decl '{' ... '}'
 | c_track '{' ... '}'
 | c_state '{' ... '}'
```

```
options : ':' ':' sequence [ ':' ':' sequence ] *

andor : '&' '&' | '|' '|'

binarop : '+' | '-' | '*' | '/' | '%' | '&' | '^' | '|'
 | '>' | '<' | '>' '=' | '<' '=' | '=' '=' | '!' '='
 | '<' '<' | '>' '>' | andor

unarop : '~' | '-' | '!'

any_expr: '(' any_expr ')'
 | any_expr binarop any_expr
 | unarop any_expr
 | '(' any_expr '-' '>' any_expr ':' any_expr ')'
 | LEN '(' varref ')' /* nr of messages in chan */
 | poll
 | varref
 | const
 | TIMEOUT
 | NP_   /* non-progress system state */
 | ENABLED '(' any_expr ')'  /* refers to a pid */
 | PC_VALUE '(' any_expr ')'  /* refers to a pid */
 | name '[' any_expr ']' '@' name /* refers to a pid */
 | RUN name '(' [ arg_lst ] ')' [ priority ]

expr : any_expr
 | '(' expr ')'
 | expr andor expr
 | chanpoll '(' varref ')' /* may not be negated */

chanpoll: FULL | EMPTY | NFULL | NEMPTY

string : '"' [ any_ascii_char ] * '"'

uname : name

name : alpha [ alpha | number ] *

const : TRUE | FALSE | SKIP | number [ number ] *

alpha : 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j'
 | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't'
 | 'u' | 'v' | 'w' | 'x' | 'y' | 'z'
 | 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J'
 | 'K' | 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R' | 'S' | 'T'
 | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z'
 | '_'

number : '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
```

APPENDIX D

LOGIC PARSER

Parser for Transition Constraints and Property Specification Formulas

For high level Petri nets, and more specifically for Predicate Transition Nets (PrT nets), transition constraints are defined in first order logic (FOL). In addition to the usual logic operators, we include set, arithmetic and relational operators to be used as part of expressions in the logic. On the other hand, properties specifications are written in first order linear temporal logic (FO-LTL). A FO-LTL bears a similar form as the FOL for transition constraints, except that it includes temporal operators and that predicates (places) in the behavioral model can be part of a formula defining the property. For example, predicate (place) P in the corresponding Petri net can be used to define a term $P(x)$ which evaluates to true if x is a token available at P or false otherwise.

For each of the logics FOL and FO-LTL, we defined a grammar and we used the well known Java CUP parser generator to create a parser for each one. The user inputs the formula through a panel, then the parser checks whether it is syntactically correct; if so, it creates the XML representation based on the resulting parse tree (see Figure D.1).

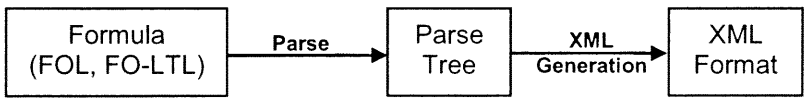

Figure D.1: FOL Parsing.

Grammar Definition

A left recursive grammar was defined for each FO LTL and FOL in BNF form. This grammar defines the production rules to build terms and expressions as well as the terminal symbols for the lexical tokens. For FOL we included relational, arithmetic and set operators. Set operations were included given that data flowing within a high level Petri net include sets.

Also, we implemented our own true type font to specifically support the temporal operators symbols. The font encodes each character in Unicode 16 bits format. A formula is then encoded as a 16 bit character string and it can then be passed to the parser for lexical and syntax analysis.

Parser Generation

Java CUP is used for the generation of the basic code for our FOL and FO-LTL formula parsers. Java CUP generates LALR parsers implemented in Java, allowing an easy integration with existing Java code. For the parser generation we first defined the lexical symbols to be used by a scanner defined using JLex (also part of Java CUP). For example, given "\uFA60" representing the always operator, we instruct the scanner to return a symbol $FLTL\_ALWAYS$ which is later used in the grammar rules.

Next, the grammar itself is expressed in Java CUP format by specifying the terminal symbols, non terminal symbols, precedence of the operators and the production rules. We implemented a hierarchy of classes to be used in the abstract syntax tree construction. The root class is *LogicSentence*, and when a parse tree is built, it is represented with an instance of this class. The classes available support the different logic constructs such as binary logic operations, arithmetic operations, temporal logic operations, among others.

Syntax checking. The syntax correctness is ensured by the FOL and FO-LTL parsers. If there is an invalid character, the token analyzer will report it and if there is a syntactic error, the parser will stop and report the error. For example, given the formula: $\forall\forall\exists a$

It has valid characters, and as such, valid tokens, so the lexical symbol construction poses no problem. But in the syntax tree construction, the grammar rules define that after each $\forall$ a variable list should appear. This is not the case, and a syntactic error is reported.

Type checking. Type checking is accomplished by relying on the Sorts defined for the PrT net, where each variable and predicate name has an associated sort and hence the formula can be checked for invalid type constructs. For example given $x[1] + y$, $x[1]$ and y need to be numbers and not other elements such as sets. This semantic check is critical when we translate our models to PROMELA and to Java implementations.

XML representation. The abstract syntax tree can be easily translated into XML format and vice versa. Each node in the tree will have a corresponding tag in the XML format.

Example. The following liveness formula states that whenever a token x is at $port0$, eventually a token y will show up at place $port2$, and y's third component will be the same as x's one.

$$\forall x \cdot (\exists y \cdot (\Box(port0(x) \rightarrow \Diamond(port2(y)(x[3] = y[3])))))$$

In this example the third components for x and y act as the identifier for a request, stating that y is the result of x. This formula is translated into an internal Unicode 16 bits format of the form:

$$\backslash\text{uFA40 } x \backslash\text{uFA43 (} \backslash\text{uFA41 } y \backslash\text{uFA43 (} \backslash\text{uFA60 (port0(x) . . .}$$

In the lexical analysis, the token construction yields something like:

FORALL ID(x) SCOPE LPAREN EXISTS ID(y) SCOPE LPAREN FLTL . . .

After the lexical analysis is done the parse tree has the form shown in Figure D.2:

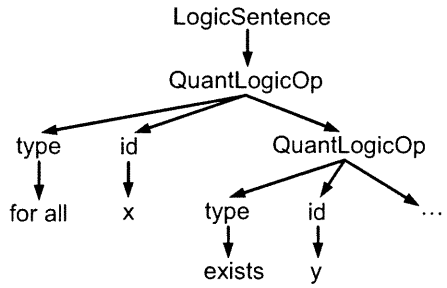

Figure D.2: Logic syntactic tree.

And the final XML is of this form:

```
<Formula name="" type="ftLTL">
<LogicSentence>
  <QuantLogicOp><ForAll>
    <VarList><
     VarListElem><Variable>x</Variable></VarListElem>
    </VarList>
    <QuantLogicOp><Exists>
      <VarList>
       <VarListElem><Variable>y</Variable></VarListElem>
      </VarList>
      <UnLogicOp><FLTLAlways>...</FLTLAlways></UnLogicOp>
    </Exists></QuantLogicOp>
  </ForAll></QuantLogicOp>
</LogicSentence>
</Formula>
```

VITA

GONZALO ARGOTE GARCIA

1997	B.S., Systems Engineering Bolivian Catholic University Cochabamba, BOLIVIA
1997-2002	Researcher and Instructor Bolivian Catholic University Cochabamba, BOLIVIA
2006	M.Sc. Computer Science Florida International University Miami, Florida
2003-2008	Graduate Assistant and Research Assistant Florida International University Miami, Florida
2008-	Software Development Engineer Microsoft Corp. Redmond, Washington
2009	Doctoral Candidate in Computer Science Florida International University Miami, Florida

PUBLICATIONS AND PRESENTATIONS

Gonzalo Argote-García, Peter J. Clarke, Xudong He, Yujian Fu, Leyuan Shi: *A Formal Approach for Translating a SAM Architecture to PROMELA.* SEKE 2008.

Junhua Ding, Gonzalo Argote-García, Peter J. Clarke, Xudong He: *Evaluating Test Adequacy Coverage of High Level Petri Nets Using Spin.* AST 2008.

Richard Whittaker, Gonzalo Argote-García, Peter J. Clarke, Raimund K. Ege: *Decentralized mediation security.* IPDPS 2008: 1-6

Richard Whittaker, Gonzalo Argote-García, Peter J. Clarke, Raimund K. Ege: *Optimizing Secure Collaboration Transactions for Modern Information Systems.* ICONS 2008: 62-68.

Yujian Fu, Zhijiang Dong, Gonzalo Argote-García, Leyuan Shi, Xudong He: *An Approach to Validating Translation Correctness From SAM to Java.* SEKE 2007.

Richard Whittaker, Gonzalo Argote-García, Peter J. Clarke, Raimund K. Ege: *Collaboration Security for Modern Information Systems.* SECRYPT 2006: 363-370

Weixiang Sun, Tianjun Shi, Gonzalo Argote-García, Yi Deng and Xudong He: *Achieving a Better Middleware Design through Formal Modeling and Analysis.* SEKE 2006.

Oscar Antezana, Gonzalo Argote G.: *Desarrollo de Aplicaciones C++ Modificables o Extensibles en Tiempo de Ejecución (Developing C++ Applications that will be modified or extended at runtime)* ACTA NOVA 2001.

Davor A. Pavisic, Gonzalo Argote G., Soraya Ordonez, Oscar Antezana, Ariel Cary, Erick Antezana, Reynaldo Vargas: *Avances en proteómica (Advances in Proteomics)* ACTA NOVA 2000.

Reynaldo Vargas A., Gonzalo Argote G., Ron D. Appel, Denis Hochstrasser, Christian Pellegrini: *Clasificación Automática de Imágenes de Geles de Electrofóresis Bidimensional Mediante la Clasificación Heurística (Two-dimensional electrophoresis gels images automatic classification using Heuristic Clustering)* Science and Technology Simposium Memories 1998.

Gonzalo Argote G.: *Simulación Gráfica por Computadora de Fenómenos Físicos Básicos (Computer graphics simulation of basic physics phenomena)*; B.S. Thesis 1997.