PVS Proof Patterns For UML-Based Verification

by

Yanguo Liu
B.Eng., Harbin Institute of Technology, 1999

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF APPLIED SCIENCE

in the Department of Electrical and Computer Engineering

We accept this thesis as conforming
to the required standard

[redacted]

Dr. Issa Traore, Supervisor (Department of Electrical and Computer Engineering)

[redacted]

Dr. Nikitas J. Dimopoulos, Departmental Member (Department of Electrical and Computer Engineering)

[redacted]

Dr. Hausi Müller, Outside Member (Department of Computer Science)

[redacted]

Dr. Jens H. Jahnke, External Examiner (Department of Computer Science)

University of Victoria

Supervisor: Dr. Issa Traore

ABSTRACT

UML has been established as a practical notation to specify complex software systems and has been adopted by the OMG as an industry standard for object-oriented analysis and design. However, UML lacks of rigor and precision, UML specifications are often incomplete, inconsistent and ambiguous. For critical software systems, the inconsistencies and misconceptions arising from the inherent ambiguities of UML specifications are not only difficult and expensive to correct in the further phases of the software development, but also often lead to safety related failures. Therefore, the early checking of the UML specifications is crucial. In general, validation and Verification of UML diagram are hard to establish. Informal and semi-formal notations such as UML only support a very basic level of assurance, which just impose a structure on the specification that facilitates inspection by domain experts. The work carried in this thesis is part of ongoing effort to develop an automated reasoning framework for rigorous validation and verification of UML specifications. More specifically, we develop in this thesis some formal proof patterns that can be used to automatically verify a subset of UML specifications, which can lead to early detection of misconceptions and inconsistencies in the software development process; meanwhile, we present the implementation of the patterns in the Precise UML Development tool (PrUDE).

[redacted]

Dr. Issa Traore, Supervisor (Department of Electrical and Computer Engineering)

[redacted]

Dr. Nikitas J. Dimopoulos, Departmental Member (Department of Electrical and Computer Engineering)

[redacted]

Dr. Hausi Müller, Outside Member (Department of Computer Science)

[redacted]

Dr. Jens H. Jahnke, External Examiner (Department of Computer Science)

Table of Contents

List of Figures

List of Abbreviations

AI	Artificial Intelligence
API	Application Programming Interface
ASM	Abstract State Machine
BDD	Binary Decision Diagram
CASE	Computer Aided Software Engineering
CTL	Computation Tree Logic
DNF	Disjuctive Normal Form
LTL	Linear Temperal Logic
OCL	Object Constraint Language
OMG	Object Management Group
OMT	Object Modeling Technique
OOSE	Object-Oriented Software Engineering
PROMELA	Pocess Meta Language
PrUDE	Precise UML Development Environment
PVS	Prototype Verification System
PVS-SL	PVS – Specification Language
UML	Unified Modeling Language
V&V	Validation And Verification
XMI	XML Metadata Interchange

Acknowledgement

I appreciate Dr. Issa Traore and my research group members for their help with my studies and research. I appreciate UVIC Engineering Associate, Peter Darling, P. Eng. in reviewing my thesis and providing helpful comments. I appreciate all my committee members for their valuable recommendations in my defense.

I appreciate my lovely wife and my family for their support during my master program.

Chapter 1
Introduction

1.1 Rationale

Software development process encompasses several stages. The typical software development stages are requirement specification and design, coding, testing, and maintenance [1]. The specification stage consists of analyzing the customer requirements leading to a complete specification of the desired software system. In the design stage, a developer decomposes the software system into its actual constituent components on modules with their inputs, outputs and functions. In the coding stage, modules defined during the design stage are implemented using a computer-understandable language (e.g. Java, C, C++ etc.). Software testing is conducted according to the software requirements to uncover and remove "bugs" in the developed software system. If there are no "bugs" detected, then the software is delivered to users.

It has been shown that software errors are most likely to be introduced during the requirement analysis and design stage [2]; and these errors can have a lasting impact on the reliability, cost and safety of a system. From the above software development process, it is expected that the errors occurring in the requirement or design stage are uncovered or removed after the coding stage. Actually this results in a lot of money and time spent in the coding and testing stages. Furthermore, requirement errors are between 10 and 100 times more costly to fix during later stages of the software lifecycle than during the requirements stage [2]. On the other hand, for critical software systems (e.g. mission-critical, business-critical, safety-critical, and security-critical systems) achieving

a high level of dependability (e.g. reliability, availability, safety, and security) is central. Any single failure in critical systems may result in significant economic losses, physical damage or threat to human life. Errors in requirement and design are unacceptable especially for critical software systems [32].

To save money and labor in software development and guarantee a high level of system dependability, it is natural to consider a method to identify and fix errors, inconsistencies and misconceptions at the early stages, for instance, just after the design stage or even before the design stage.

UML is becoming more and more popular for expressing object-oriented models and designs. It is based on an intuitive and friendly diagrammatic notation [3]. More and more software developers are using UML to model their software in the early stages of software development. Since UML is currently the leading notation for requirement and design specification, integrating an efficient verification methodology with the UML would provide a powerful tool for developers. That is the objective sought for in this thesis.

1.2 Problem Statement

In general, it is hard to achieve a rigorous validation and verification of software system based on UML diagrams. Informal methods or semi-formal notations such as the UML only provide a very basic level of assurance, which just impose a structure on the specification that facilitates inspection by domain experts [30]. To improve software reliability and quality, an efficient approach consists of applying formal methods in the verification process. However, many Object Oriented (OO) notations, including UML,

suffer from a lack of a precise semantics. This can lead to confusion and ambiguous interpretations when analyzing a model. Consequently, this decreases the ability to develop tools and guidelines to help the specifier. Designers need a more precise and rigorous model that they can check and validate against customers requirements. Based on this consideration, some UML formalization schemes have been proposed in the literature. Some of these approaches extend the UML notations with formal characteristics and let the user deal with the formal artifacts [33], [38]. Other approaches hide the formal aspects by assigning a formal semantics to UML models [4], [26], [34]. In either case the formal verification process, which is complex and time-consuming, is usually performed in existing formal proof systems involving tedious and intensive user interactions. The ultimate goal of formal methods is to ensure design and programming correctness using formal reasoning (e.g. model-checking, proof-checking etc.). However the esoteric nature of formal methods and the lack of industrial-strength tool support impose significant barriers to their widespread penetration, especially in the industrial world [5]. A prerequisite for any widespread adoption of formal methods is an increased automation of the different steps involved. Model checking, for instance, provides a complete automation solution, however its scope is limited by the state explosion problem. Though proof checking allows the verification of more complex system properties, it requires quite often tedious and intensive user interactions.

1.3 Approach Adopted

The work carried in this thesis is part of ongoing effort to develop an automated reasoning framework for rigorous software validation and verification based on UML specifications. The general framework consists of defining a formal semantic for UML

diagrams and conducting verification and validation activities using that formal semantic. The semantic scheme is defined using the Prototype Verification System (PVS) language. PVS is a formal system that provides a rich semantic foundation and a powerful formal reasoning tool. However, formal reasoning requires strong mathematical background that most software developers don't have. Existing formal verification tools, including the PVS tool, still require intensive and esoteric user interactions that discourage most software developers for the obvious reasons mentioned previously.

In this context, in order to allow the rigorous verification of UML models based on the related formal semantics and eliminate the need for expertise in formal mathematics, we need to make the reasoning process transparent to the user. The approach adopted in this thesis consists of defining some verification patterns that can be used to maximize the automation level of the reasoning process.

UML is a quite complex notation that consists of up to nine standard diagrams. Therefore, we limit the current report to only one such diagram, namely UML Statechart Diagram, which is the most complex diagram to describe dynamic aspects of system behavior. Other colleagues are working on the other important diagrams [7], [9].

The whole framework is implemented as a tool suite named the Precise UML Development Environment (PrUDE) [10]. PrUDE receives as input a UML model (to be verified), and generates automatically a corresponding formal semantic in PVS. Then the PVS formal reasoning tool (proof checker or model checker) is invoked in batch mode to check the expected system properties submitted by the analyst. Our objective in this thesis is to minimize the user interactions in the formal reasoning process as much as possible.

1.4 Contribution

In this thesis, we extend the initial formal semantic defined in [8] for UML statechart in order to identify some proof patterns that can achieve the automation of the complete verification process.

We present an abstraction method called Configuration-Based Abstraction for UML statechart, which serves as basis for the definition of the proof patterns. And we provide the formal semantic for the abstraction method.

We present a set of proof patterns that can be applied to both the PVS theorem prover and the PVS model checker to check system properties.

We present an efficient and systematic way for expressing system properties as PVS formulas by providing suitable templates for liveness and safety properties.

We implement the automatic verification methodology in PrUDE.

1.5 Thesis Outline

The remaining of the thesis is organized as follows:

Chapter two presents some background knowledge, giving an overview of the Unified Modeling Language (UML), the Prototype Verification System (PVS), and the PVS formalization of UML. The PrUDE (Precise UML Development Environment) tool suite is also briefly introduced in this chapter.

Chapter three introduces an abstraction scheme for UML statechart diagram, called Configuration-Based Abstraction, which serves as the basis for our automatic V&V methodology.

Chapter four introduces proof patterns based on the abstraction method and illustrates how the PVS proof system can be used to verify automatically the system properties

using our methodology. We introduce respectively automatic proof approaches for both the PVS theorem prover and the PVS model checker.

In Chapter five the implementation of the automated formal analysis framework in PrUDE is discussed.

Chapter six illustrates our automatic verification methodology by a concrete example.

Chapter seven presents some related work and Chapter eight makes some concluding remarks and discusses future work.

Chapter 2

Background

2.1 UML Overview

The Unified Modeling Language (UML) [3] is a graphical language for visualizing, specifying, constructing and documenting the artifacts of complex software systems, such as telecommunication system, enterprise information system, banking and financial services, distributed web-based services etc. The UML was originally derived from the object modeling notations of three leading object-oriented methods: Booch Method [11], Object Modeling Technique (OMT) [12] and Object-Oriented Software Engineering (OOSE) [13]. It was first added to the list of OMG adopted technologies in 1997, and has since become the industry standard for modeling objects and components.

Using UML, conceptual artifacts, such as business processes and system functions, can be expressed in concrete and visual models, which are more understandable and easy to evolve and communicate between different development teams.

UML is used to model a software system. Normally we can look at a software system from three perspectives: functional, structural and dynamic viewpoints. Functional viewpoint describes the functions carried out by the system. Structural viewpoint focuses on system static structure, while dynamic viewpoint focuses on the dynamic behavior. In UML, we describe the static structure of a system by using Class Diagram, Object Diagram, Component Diagram, and Deployment Diagram. Dynamic aspects are described by Sequence Diagram, Collaboration Diagram, Statechart Diagram, and Activity Diagram. Use Case Diagram describes the functional behavior.

2.2 PVS Overview

The Prototype Verication System (PVS) [14], developed by SRI, is a higher order logic specification and verification environment. It consists of a specification language, a parser, a typechecker, a theorem prover, a model-checker, specification libraries and browsing tools.

PVS provides an interactive environment for writing formal specifications and checking formal proofs. It builds on nearly 20 years experience at SRI in building verification systems. The distinguishing feature of PVS is its synergistic integration of an expressive specification language and powerful theorem proving capabilities.

PVS provides an expressive specification language that augments classical higher-order logic with a sophisticated type system containing predicate subtypes and dependent types, and with parameterized theories and a mechanism for defining abstract datatypes such as lists and trees. The standard PVS types include numbers (reals, rationals, integers, naturals etc.), records, tuples, arrays, functions, sets, sequences, lists, and trees, etc.

The PVS typechecker generates proof obligations for the PVS theorem prover. Most such proof obligations can be discharged automatically. This liberation from purely algorithmic typechecking allows PVS to enforce very strong checks on consistency and other properties (such as preservation of invariants) in an entirely uniform manner.

PVS has a powerful interactive theorem prover/proof checker. The basic deductive steps in PVS are large compared with many other systems: there are atomic commands for induction, quantifier reasoning, automatic conditional rewriting, simplification using arithmetic and equality decision procedures and type information, and propositional simplification using binary decision diagrams. The PVS proof checker manages the proof

construction process by prompting the user for a suitable command for a given subgoal. The execution of the given command can either generate further subgoals or complete a subgoal and move the control over to the next subgoal in a proof. User-defined proof strategies can be used to enhance the automation in the proof checker. PVS's automation successes to prove many straightforward results automatically; for hard proofs, the automation takes care of the details and frees the user to concentrate on directing the key steps.

The PVS model checker provides model-checking capabilities for automatically verifying temporal properties of finite-state systems.

PVS is implemented in Common Lisp with ancillary functions provided in C, Tcl/TK, and LATEX and uses GNU Emacs for its interface. It is Configured for Sun Sparc Workstations and is freely available under license from SRI. PVS has been applied successfully to large and difficult applications in both academic and industrial settings.

2.3 UML Formalization in PVS-SL

In order to make UML amenable to rigorous analysis, in previous work a formal semantic for a subset of UML notations (Class Diagram, Sequence Diagram, Statechart Diagram) has been defined using PVS specification language (PVS-SL) [7], [8], [9]. A PVS specification consists of a collection of theories. A theory consists of type and constant definitions, related axioms, and theorems. Parametric theories, using types and values, are supported as well in PVS. For instance, Figure 1 shows an instance of parametric theory describing stacks [15].

```
Stack [Item: TYPE+]: THEORY
BEGIN

% Type, variable, constant definitions
Stack: TYPE+
b: VAR Stack
i: VAR Item
Empty: Stack
NonEmptyStack?(b): bool = b/=Empty

% Stack operations
push: [Item, Stack -> (NonEmptyStack?)]
pop: [(NonEmptyStack?)-> Stack]

% Axiom definitions
empty_ax: AXIOM pop(push(I, Empty)) = Empty

END Stack
```

Figure 1: Example of PVS Theory

The theory receives as parameter a type named *Item*, which corresponds to the type of the items contained by the Stack. **THEORY, BEGIN** and **END** are the specific keywords that define a theory. Keywords **TYPE, VAR**, and **AXIOM** are used to define respectively types, variables, and axioms. For instance, the axiom labeled *empty_ax* states that the application to an empty stack of a *push* function, followed by a *pop* function will give rise to the same empty stack.

In this thesis, we focus on the verification of UML behavioral diagrams. More specifically, based on the formal semantics of UML statechart defined in [8], we develop some verification patterns that allow the automatic verification of system properties based on UML statechart diagram. The formal semantic assigned to UML statechart in [8] is shown in the following.

In [8], the general formalization approach consists of the following steps:

1. An abstract syntax is provided for the UML statechart diagram using PVS-SL.

2. A set of well-formedness rules are provided under the form of PVS theorems, based on the syntax that is specific to the model considered.
3. The formal semantics of the model in the PVS specification language is generated. The semantic model obtained is a combination of generic formulas and additional formulas provided by the user in order to complete the definitions of specific model features such as elementary states and transitions.

The semantic model obtained serves as basis for verification of the UML model using the PVS proof checker and the system properties specified by the user. Three generic PVS theories are provided for the whole formal semantic of a given UML statechart diagram. Each of them describes different aspect of the model, namely the abstract syntax, the set of well-formedness rules and the formal semantics. The three theories are called respectively *AbstractSyntax, WellFormedness* and *FormalSemantics.* Theories WellFormedness and FormalSemantics are parametric theories that receive as actual parameter the abstract representation of the statechart. The key abstraction of theory AbstractSyntax is the formal representation of a statechart diagram that is given as a record listing the different collections of features (e.g. states, events, transitions etc.) involved in the specific diagram considered. From this definition, we can derive a collection of finite sets representing the basic building blocks of the statechart (e.g. set of states, set of events etc.), that actually describe the domain over which the well-formedness rules provided by theory *WellFormedness* are defined. Theory *FormalSemantics* is defined to provide some operational features of UML statechart. Figure 2, 3, 4 show respectively overview of theories AbstractSyntax, WellFormedness, and FormalSemantics.

```
AbstractSyntax: THEORY

BEGIN

Time: TYPE FROM nat
Vertex: TYPE+
State: TYPE = set[Vertex]

%guard conditions
Condition: TYPE+
……
%actions
Action: TYPE+
……
%transitions
Transition: TYPE+ = [#      source: Vertex,
                            trigger: Event,
                            guard: Condition,
                            effect: Action,
                            target: Vertex
                      #]
……
StateMachine: TYPE+ =
                      [#    State: set[Vertex],
                            StubState: set[Vertex],
                            SynchState: set[Vertex],
                            Initial: set[Vertex],
                            Choice: set[Vertex],
                            ……
                            Root: (State),
                            Context: Context
                      #]
……
sm: VAR StateMachine
ax_pseudo: AXIOM
        subset?(DeepH(sm),PseudoState(sm)) AND
        subset?(DeepH(sm), DeepH) AND
        subset?(ShallowH(sm),PseudoState(sm)) AND
        subset?(ShallowH(sm),ShallowH) AND
        subset?(Choice(sm),PseudoState(sm)) ANDEND AbstractSyntax
……
END AbstractSyntax
```

Figure 2: Theory AbstractSyntax

```
WellFormedness
      [(IMPORTING AbstractSyntax) sm: StateMachine] : THEORY

  BEGIN
   prudelib: LIBRARY = "~/prude/semantic/lib/prudelib"
   IMPORTING AbstractSyntax

   x,y: VAR Vertex
   A: VAR set[Vertex]
   ......
   %auxilliary functions
   atleast2?(A): bool = A /= emptyset AND NOT singleton?(A)
   atmost1?(A): bool = A = emptyset OR singleton?(A)
   atleast2?(T): bool = T /= emptyset AND NOT singleton?(T)
   atmost1?(T): bool = T = emptyset OR singleton?(T)
   ......
   %%%%%%%%%%%%%%%%%Well-formedness rules%%%%%%%%%%%%%%%%%%
   %%%%%%%%%%%%%%%%%Composite state%%%%%%%%%%%
   %can have at most one initial vertex
   wf1: AXIOM
      (member(S1, State(sm)) AND member(S1,State(sm) AND
       compositeState?(S) AND  compositeState?(S1)) =>
         atmost1?(intersection(Initial(sm),dsubvertex(S)))
   ......
   %%%%%%%%%%%%%%%%%%Transitions%%%%%%%%%%%%%%
   %A fork segment should not have guards or triggers
   wf5: AXIOM member(source(tr),Fork(sm)) =>
              (guard(tr)=EmptyC AND trigger(tr) = EmptyE)
   ......

END WellFormedness
```

Figure 3: Theory WellFormedness

```
FormalSemantics
  [(IMPORTING AbstractSyntax) sm: StateMachine,V: TYPE ]: THEORY

  BEGIN
    lib: LIBRARY = "~/prude/semantic/lib/prudelib"
    IMPORTING finite_sequences[(Event(sm))]
    IMPORTING WellFormedness[sm]
    IMPORTING basic_defs[V]
    %auxilliary variables declaration
    r, s, s1, s2, x, y, z: VAR Vertex
    q: VAR PRED[V]
    ......
    EventStatus: TYPE = {pending,received,dispatched,consumed}
    %relate event instance to actual event type
    EventInstance: TYPE+
    ......
    %define a type that encapsulates both the current and
    % next values of the variables involved.
    VC: TYPE = [#current: V, next: V#]
    vc: VAR VC
    ......
    %Predicates associated to states, conditions, and actions
    pred: [Vertex -> PRED[V]]
    pred: [Condition ->PRED[V]]
    pred: [Action -> PRED[VC]]
    ......
    %two transitions are said to conflict if the intersection of
    %the set of states they exit is non-empty
    conflict(tr, tx): bool =
        (tr /= tx) AND
        {s| (s= mainSource(tr)
            OR member(s,substate(mainSource(tr)))) AND
        (s= mainSource(tx)
            OR member(s,substate(mainSource(tx))))} /= emptyset
    ......

END FormalSemantics
```

Figure 4: Theory FormalSemantics

2.4 Formal Verification Using PVS

The automation of mathematical reasoning coincides with the emergence of the field of Artificial Intelligence (AI), whose early pioneers embarked on a program to (mechanically) simulate human problem solving. The underlying logic of automated reasoning includes Temporal Logic, First Order Logic, Higher Order Logic, Computation Tree Logic (CTL), etc. Formal verification aims to prove, using automated reasoning, that the system described meets its complete specification. In other words, system properties (also called putative theorems) are checked mathematically in formal verification process. Formal verification helps to highlight the gaps, errors, and inadequacies in the highlevel functional descriptions of a system. Two formal verification techniques provided in the PVS system are introduced, which are respectively *theorem proving* and *model checking*.

2.4.1 Theorem Proving

Theorem proving [16], also named automated deduction, can be thought of as game playing where there are very precise rules, initial positions and goals; you win if you reach the goal from the initial position by correctly following the rules. More precisely, theorem proving is the mechanization of deductive reasoning that provides a foundation for reasoning about finite or infinite-state system. Proof techniques underlying theorem proving includes resolution, equation or rewriting, constructive type theory methods and a variety of other methods loosely characterizable as interactive. A theorem proving system generally consists of a language whose sentences (called formula) are used to state goals and axioms, plus some rules of inference for deriving new sentences from old

ones. The essential interest of theorem proving involves Guaranteeing type correctness, challenging underlying assumptions, confirming key properties and invariants, etc.

Theorem proving is based on *proof theory*. More specifically, a set of axioms, together with all the theorems derivable from it, is called a *theory*. A proof of a theorem of a theory is simply a series of transformations that confirm to some inference rules. The symbol ⊢ (called "turnstile") is used to express the notion of proof. For example, ⊢ φ, which reads "φ is provable", means that φ is a theorem in the given logic and φ is provable using the given axioms without further assumptions. Generally, a proof of a sentence φ is expressed using φ and a set of sentences, γ0, …, γn, for instance, *γ0, ..., γn* ⊢ *φ*, meaning that φ is provable under γ0, …, γn, where γi is either an axiom, an additional assumption, or a previously proved theorem.

Theorem proving supports more varied and more abstract models, expressive specification, varied properties, and reusable verifications than finite state verification techniques (eg. Model checking).

The main challenge of theorem proving is to mechanize routine manipulations and to reduce low-level interaction and repetitive tedium involved in large proofs.

PVS supports effective theorem proving [17]. In order to make proofs easier to develop, the PVS theorem prover provides a collection of powerful proof commands to carry out propositional, equality, and arithmetic reasoning with the use of definitions and lemmas. These proof commands can be combined to form proof strategies. A proof strategy is intended to capture patterns of inference steps. A defined proof strategy can be applied in a single atomic step so that only the final effect of the strategy is visible and the intermediate steps are hidden from the user.

The PVS theorem prover is interactive, but also supports a batch mode in which proofs can be easily rerun. The proofs are maintained in a proof tree, each node of the proof tree is a proof goal. Each proof goal is a *sequent* consisting of a sequence of formulas called *antecedents* and a sequence of formulas called *consequents.* The template of a sequent in PVS is displayed in Figure 5.

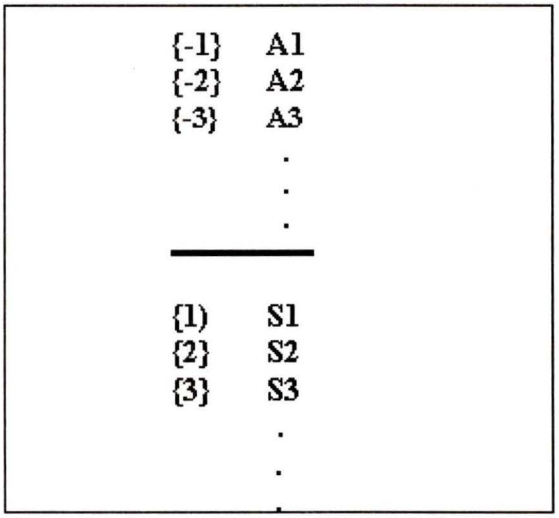

Figure 5: A PVS Proof Sequent

Where the *Ai* and *Sj* are PVS formulas: the *Ais* are the antecedent and the *Sjs* are the consequent. The intuitive interpretation of a sequent is that the conjunction of the antecedents implies the disjunction of the consequents, i.e., $(A1 \wedge A2 \wedge A3 \ldots) \supset (S1 \vee S2 \vee S3 \ldots)$. The proof tree begins with a root node of the form $\vdash A$, where A is the theorem to be proved. PVS prover builds a proof tree by adding subtrees to leaf nodes as directed by the proof commands. Once a sequent is recognized as *true*, that branch of the proof tree is terminated. Once all the branches of a proof tree are terminated, the proof is complete.

For instance, let's consider a PVS specification that defines recursively the sum of the first n natural numbers (see Figure 6).

```
sum: THEORY

 BEGIN

      n: VAR nat
      sum(n): RECURSIVE nat =
            (IF n = 0 THEN 0 ELSE n + sum(n-1) ENDIF)
            MEASURE (LAMBDA n: n)
      formula_form: THEOREM sum(n) = (n*(n+1))/2

 END sum
```

Figure 6: Theory sum

The *sum* theory has no parameters and contains three declarations. The first declares *n* to be a variable of type *nat* that is the built-in type of natural numbers. The next declaration is a recursive definition of the function *sum(n)* whose value is the sum of the first n natural numbers. A *measure* function following the MEASURE keyword is associated with the recursive definition. The final declaration is a theorem definition that gives the formula form of the *sum* function. For the recursive definition *sum(n)* to be acceptable, the theorem *formula_form* has to be true.

The proof script displayed in Figure 7 shows the interactions between the user and the PVS theorem prover on proving the theorem *formula_form*. The user inputs (proof commands) are shown in boldface.

The *(induct "n")* command directs PVS prover to use induction on *n*. Two subgoals are generated corresponding to the base and induction cases. In the base case, the *(expand "sum")* command expands the *sum(n)* definition in the subgoal and generates a trivially true proof sequent that is discharged through the use of the *(assert)* command. In the

induction case, the *(skolem!)* command followed by the *(flatten)* command results in a sequent containing the induction hypothesis in its antecedent and the conclusion in its consequent part. The *(skolem!)* command is used to produce the witnesses for the sum function. The *(flatten)* command eliminates the disjunctive connectives in the formula so as to flatten the formula out into the sequent. Then the *(expand "sum" +)* command expands the *sum(n)* definition in the consequent of the sequent, and the *(assert)* command finishes the proof of the induction case. The Q.E.D message has been generated to indicate a successful proof; the run time and real time of the proof process are shown at the end of the proof. Since we direct the proof in a proper way, the proof is finished in several seconds. In practice, even for such simple proof, the proving process may take hours or even days, considering all the errors and trials involved in finding the appropriate proof commands.

```
formula_form :
 |-------
{1}  FORALL (n: nat): sum(n) = (n * (n + 1)) / 2
Rule? (induct "n")
Inducting on n on formula 1,
this yields  2 subgoals:
formula_form.1 :
 |-------
{1}  sum(0) = (0 * (0 + 1)) / 2
Rule? (expand "sum")
Expanding the definition of sum,
this simplifies to:
formula_form.1 :
 |-------
{1}  0 = 0 / 2
Rule? (assert)
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of formula_form.1.
formula_form.2 :
 |-------
{1}  FORALL j:
       sum(j) = (j * (j + 1)) / 2 IMPLIES
        sum(j + 1) = ((j + 1) * (j + 1 + 1)) / 2
Rule? (skolem!)
Skolemizing,
this simplifies to:
formula_form.2 :
 |-------
{1}  sum(j!1) = (j!1 * (j!1 + 1)) / 2 IMPLIES
      sum(j!1 + 1) = ((j!1 + 1) * (j!1 + 1 + 1)) / 2
Rule? (flatten)
Applying disjunctive simplification to flatten sequent,
this simplifies to:
formula_form.2 :
{-1}  sum(j!1) = (j!1 * (j!1 + 1)) / 2
 |-------
{1}  sum(j!1 + 1) = ((j!1 + 1) * (j!1 + 1 + 1)) / 2
Rule? (expand "sum" +)
Expanding the definition of sum,
this simplifies to:
formula_form.2 :
[-1]  sum(j!1) = (j!1 * (j!1 + 1)) / 2
 |-------
{1}  1 + sum(j!1) + j!1 = (2 + j!1 + (j!1 * j!1 + 2 * j!1)) / 2
Rule? (assert)
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of formula_form.2.
Q.E.D.
Run time  = 0.43 secs.
Real time = 206.58 secs.
```

Figure 7: Proof Script of sum Theorem

2.4.2 Model Checking

Model checking is a technique for verifying state-based systems [18]; it is also called finite state verification technique in distinction with automated deduction technique (theorem proving). The underlying logics used in model checking includes LTL (Linear Temporal Logic), CTL (Computation Tree Logic), propositional mu-calculus, etc. To apply model checking to the verification of a system, it is common to describe the behavior of the system with some state based formalism. In the formalism the behavior of the system is described in terms of local state changes or events. The global behavior of the system is given as the state-space generated from the system description. Model checking offers powerful, automated procedures for checking temporal properties of finite state systems.

Model checking is based on *Model Theory* that is the study of the interpretations of formal systems. More specifically, a model is an interpretation in which all the axioms of the formal system are true. In *Model Theory,* the concepts of logical consequence, validity, completeness, and soundness are important. For instance, let I be a set of interpretations for a calculus and φ be a sentence of the calculus. φ is satisfiable (under I) if and only if at least one interpretation of I valuates φ to true. φ is (universally) valid, written $\models \varphi$, if and only if every interpretation in I valuates φ to true.

Model checking has a number of advantages over traditional approaches that are based on simulation, testing, and deductive reasoning. In particular, model checking is automatic and usually quite fast. Also, if the design contains an error, model checking will produce a counter-example that can be used to pinpoint the source of the error. The method has

been used successfully in practice to verify real industrial designs, and companies are beginning to market commercial model checkers.

Since model checking is less effective on large, unbounded state spaces, the main challenge in model checking is dealing with the state space explosion problem, which occurs in systems with many components that can interact with each other or systems with data structures that can assume many different values. In such cases the number of global states can be enormous, deductive methods (theorem proving) are more appropriate.

The model checker of PVS is based on the branching time temporal logic called Computation Tree Logic (CTL) [19]. CTL formulas consist of atomic propositions, propositional combinations, universal path quantified formulas (denoted by **AG**f -"along every computation path, always f", **AF**f -"along every path, eventually f"), existential path quanlified formulas (**EG**f – "along some path, always f", **EF**f – "along some path eventually f"). In the PVS model-checker, an invariant property in CTL is specified in the form *AG(trans, prop)(s)* where *s* is the transition source state. The *AG* operator then means that the property *prop* must be true of all states that can be reached from *s* by the transition *trans*. The concept of state considered here is that of computation state which is equivalent to the notion of global states [8]. The notion of computation state is specified by a parametric theory *state*, provided in the PVS prelude, which receives a state (e.g. computation or global state) as parameter (See Figure 8).

```
state[ state: TYPE] : THEORY
BEGIN
IMPORTING sequences[state]
statepred: TYPE = PRED[state] %assertions
Action: TYPE = PRED[[state,state]] % transition relation
computation: TYPE = sequence[state]

pp: VAR statepred
action: VAR Action
aa, bb, cc: VAR computation

Init(pp)(aa): bool = pp(aa(0))
.....

END state
```

Figure 8: Theory state

A program represents a set of computation, and can be characterized by an initialization assertion *Init* (defining the initial state) and a binary transition relation (e.g action) which constraints the allowable transitions of a computation. A computation is defined as a sequence of states and an assertion is modeled as a predicate on states.

2.5 The PrUDE Tool Suite

PrUDE (Precise UML Development Environment) is being developed by the ISOT group (http://www.isot.ece.uvic.ca). The purpose of the tool suite is to assist the user in the software validation and verification process. The PrUDE platform consists of an integrated V&V environment that supports consistency-checking, proof-checking, model-checking and testing [10]. Consistency-checking is based on well-formedness rules defined for UML semantics. Model-checking and proof-checking are based on the PVS toolkit. The specification-based testing component of PrUDE consists of a test case generator and a test execution tool [20]. The PrUDE platform is indepent of any UML

tool vendor since it imports UML model in XMI format that is exported by most UML tools. PrUDE's main strength is that it allows users to deal with models in graphical notations that are user friendly and easy to learn and use. All formal specifications in PVS are processed at the back end.

Chapter 3

An Abstraction Pattern For UML Statechart – Configuration-Based Abstraction

In order to establish the conformance of the UML specification with the customer requirements, we express these requirements as system properties and check them against the formal model derived form the UML specification. The verification is performed by invoking the PVS proof system. In order to reduce user interactions with the prover, we' d like to automate as much as possible the verification process. In this chapter, we define an abstraction scheme for UML statechart diagram, called Configuration-Based Abstraction, which is the first step towards that goal.

3.1 UML Statechart Extension

A UML statechart describes dynamic aspects of an object, a subsystem, or a whole system [3]. It specifies the sequences of states an object goes through during its lifetime in response to events, together with its responses to those events.

The basic features of a UML statechart are state, transition, event, action and guard. A *state* is a condition or situation during the life of an object. An *event* is a specification of a significant occurrence that has a location in time and space, which can trigger a state transition. A *transition* is a relationship between two states indicating that an object in the first state will perform some actions and enter the second state when a specified event

occurs and specified guards are satisfied. I discuss briefly some of these notations in the following.

State: A state is a condition or situation in the life of an object during which it satisfies some condition, performs some activitiy, or waits for some event. A state has several parts: name, entry/exit actions, internal transitions, substates and deferred events. Entry/exit actions denote the actions executed on entering and leaving the state. Substates refer to a nested structure of a state, involving sequentially active or concurrently active substates. Generally, a state is represented by a box with a rounded corner. If a state does not include a substate, then this state is called a simple state; otherwise the state is called a composite state. Composite states are of two types. Concurrent composite states are divided into several parallel regions. Each region includes a set of states; regions are separated by dotted lines. Sequential composite states involve a set of sequential states that are related by an exclusive-or relationship. A pseudostate is an abstraction used to connect multiple transitions into more complex state transition paths. There are various kinds of pseudostates, namely initial, deepHistory, shallowHistory, join vertices, fork vertices, junction vertices and choice vertices. For instance, shallowHistory and deepHistory which are some of the most used, are shorthand notations used to depict historical information.

Transition: a transition is represented as an arc linking two states in a statechart diagram. It consists of five elements: source state, trigger, guard condition, action and target state. A source state denotes the departure state of the transition. A trigger denotes an event which activates the transition. In a program, events correspond to the signals on method calls. A guard condition is a boolean predicate which must be true when the trigger

occurs for the transition to be enabled. An action denotes an executable computation that is executed when a transition is fired. A target state becomes active after the transition is fired. Figure 9 illustrates the notations for state and transition in UML statechart diagram.

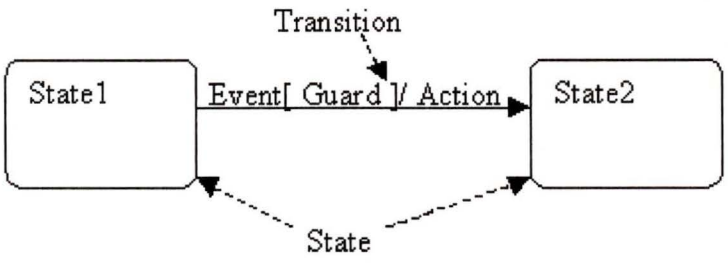

Figure 9: State and Transition

Generally, formal methods require more concrete information that the basic UML paradigm does not support. We propose an extended UML statechart that provides necessary information for formal verification.

For UML statechart diagram to be meaningful, we need to extend it by providing complementary semantic definitions, particularly for the states, actions and conditions involved.

More precisely, we first define the state variables involved in the statechart; actually, these variables correspond to the attributes associated to the class described by the statechart. A state can be either active or inactive during execution. We represent the active status of a state by defining a relevant predicate function of the state variables. A state variable can be either a basic data type (integer, boolean etc.), or a complex data type that the specifier defines. For instance, let us suppose that a statechart involves two

integer state variables A and B. For a simple state in this statechart, we might define the following predicate, say *pred(S)(A,B) = A>60 AND B>A* , meaning that given A and B, state S is active if and only if *pred(S)(A,B)* is true.

The definition of a transition in the UML statechart requires the definition of two predicates corresponding respectively to the guard condition and the action. We don't define a predicate for the event component in a transition since we just consider events as operation invocations triggered by the environment. The predicate of a guard can be given in the same pattern as states predicates. The predicate of an action, however must take into account the state values both in the source state and target state. Therefore it is function of the current state values and the next state values; it corresponds actually to the action postcondition. Figure 10 shows a simple extension of UML statechart, involving corresponding predicates.

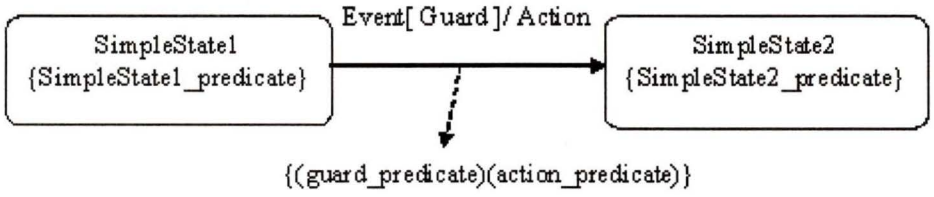

Figure 10: Statechart Extension

Generally speaking, after defining the predicates for all simple states and transitions in a statechart, we obtain an extended UML statechart that contains the necessary information required by a formal verification system. The predicates can be expressed either directly in PVS or using the Object Constraint Language (OCL) [6], which is the standard

assertion language associated to UML. In the future version of PrUDE, we intend to implement an OCL-to-PVS translator; so far the translation is done manually.

3.2 Notion Of Configuration In UML Statechart

In practice, when a state is active, several other states may be active at the same time. More specifically, all the states that contain directly or indirectly that state are active. If the state is a concurrent state, all its regions are active, and if it is a sequential state, exactly one of its direct substates is active. The set of all the states active simultaneously within a statechart diagram is called the active *state configuration*.

3.3 Configuration Graph of A UML Statechart

When we follow the execution path of a UML statechart, we find that any UML statechart can be represented by a configuration graph. Each node in the graph corresponds to a configuration in the original statechart. These configurations are exclusive to each other. The arcs in the graph correspond to transitions in the original state diagram. However, it is not necessary that every transition in the UML statechart becomes an arc in the configuration graph. Only those transitions that connect non-pseudo states are represented as an arc in a configuration graph; we give the name meaningful transitions to such kind of transitions. Transitions that come out of or go into pseudo states are ignored or considered as special cases. Figure 11 illustrates a simple extended sequential statechart and its corresponding configuration graph. Figure 12 illustrates a simple extended concurrent statechart and its corresponding configuration graph.

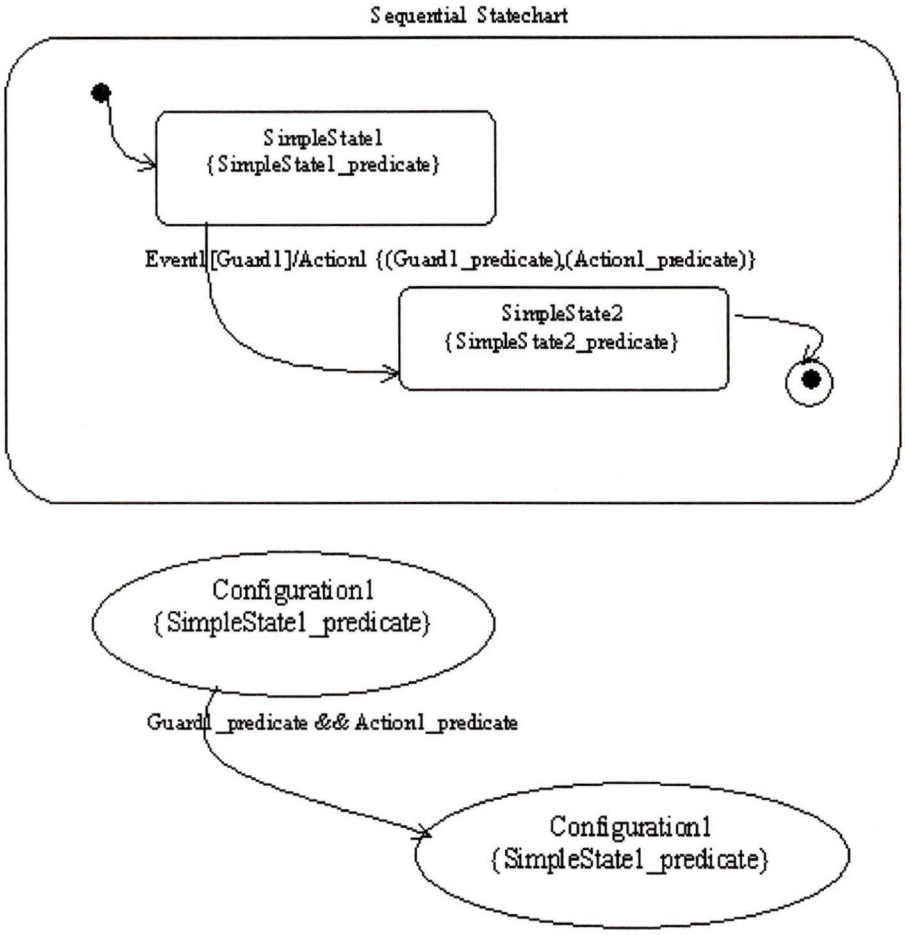

Figure 11: A Sequential Statechart and Its Configuration Graph

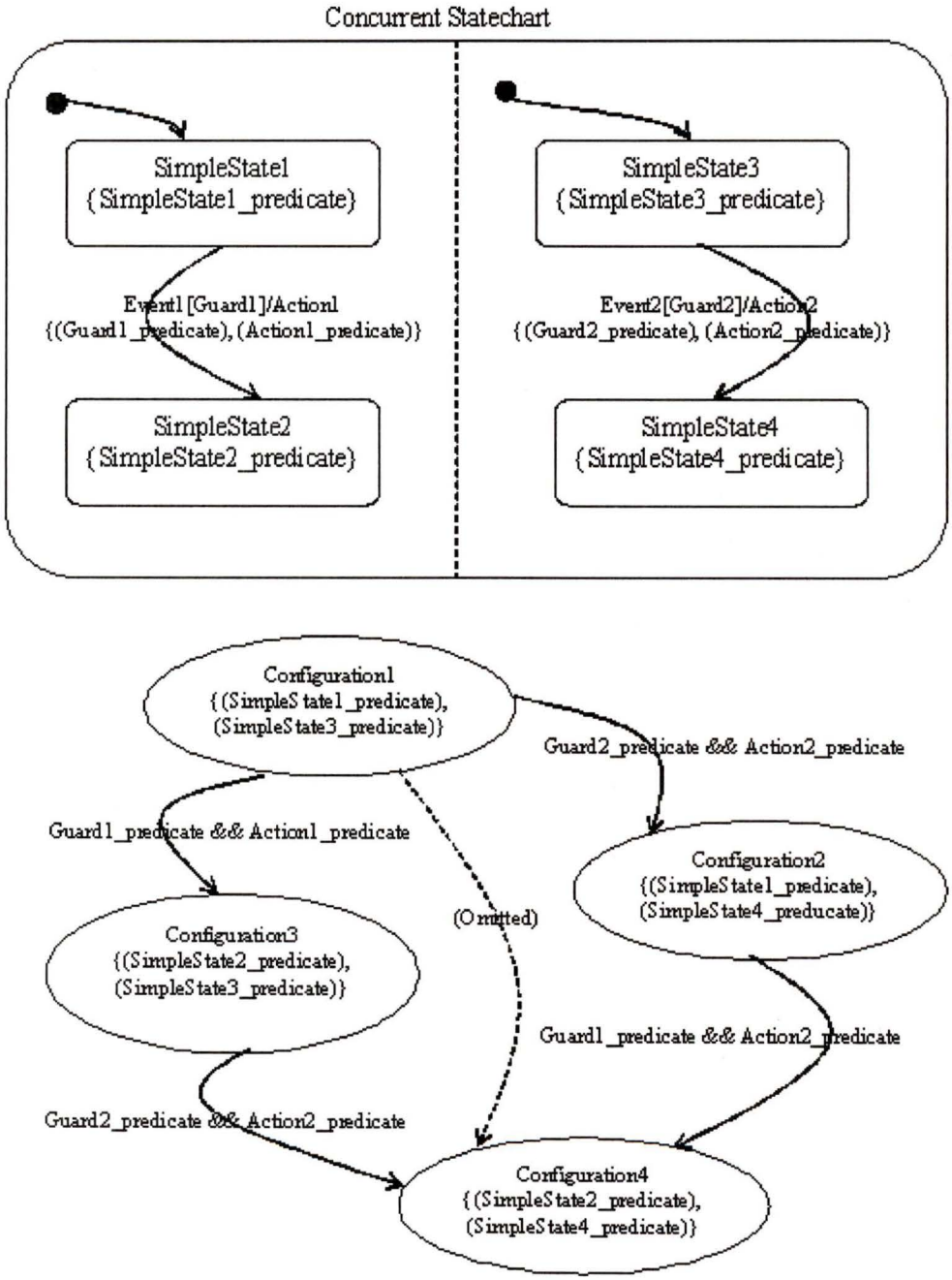

Figure 12: A Concurrent Statechart and Its Configuration Graph

We also notice that a configuration graph can be decomposed into a finite number of configuration pairs equal to the number of arcs in the configuration graph. For example, in Figure 12, we have a configuration graph with 4 arcs (the arc between configuration 1 and configuration 4 is omitted because we assume that the events are different), thus we can derive 4 configuration pairs. Figure 13 shows the configuration pairs derived from Figure 12.

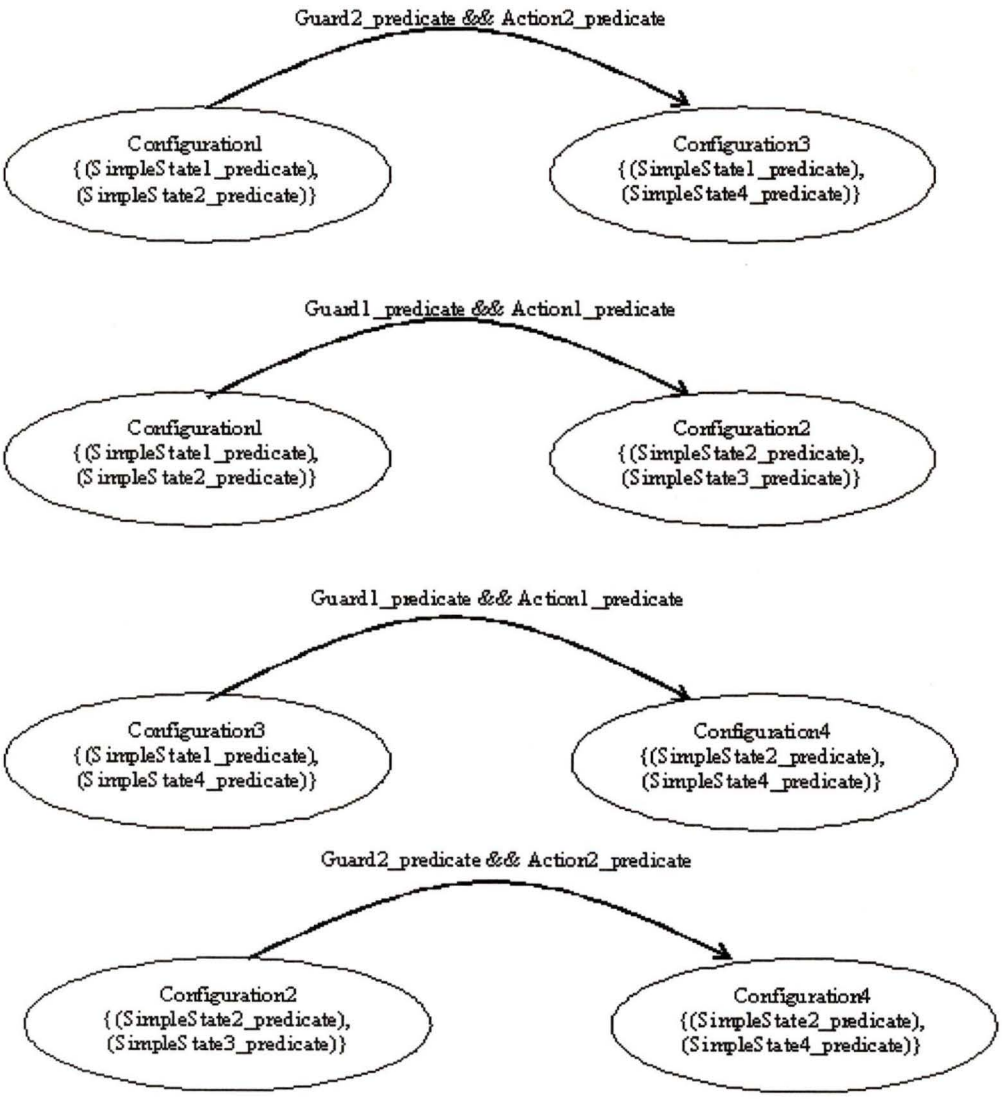

Figure 13: Configuration Pairs of Figure 12

The configuration-based abstraction consists of transforming a statechart diagram by identifying corresponding finite configuration pairs. We use, in practice, the configuration pairs obtained as verification units for the statechart diagram.

3.4 Formalization Of Configuration-Based Abstraction Scheme

As we mentioned before, the arcs in a configuration graph correspond to the meaningful transitions of the statechart diagram; each meaningful transition connects at least one configuration pair. Therefore, it is natural to think about an efficient algorithm based on those meaningful transitions. First, we present the original PVS semantics defined for the notion of configuration in [8]; then we introduce the formalization of our abstraction mechanism in PVS.

In [8], the system global state is represented by a PVS record type named *V* whose fields consist of the system variables *x1,...,xn* of respective types *T1,...,Tn*. Though *T1,...,Tn* can be any kind of types, for the sake of simplicity, we define them in the sequel as PVS uninterpreted types.

```
T1: TYPE
...
Tn: TYPE
V: TYPE = [#x1: T1,  x2: T2, ..., xn: Tn #]
v: VAR V
```

The predicates associated to simple states are defined directly by the user as complementary semantics. The predicate of a sequential state is the disjunction of the predicates of its direct substates. The predicate of a concurrent state is the conjunction of the predicates associated to its regions.

```
Vertex: TYPE+
State: set[Vertex]
pred: [Vertex - > PRED [V]]
```

A transition is defined as a PVS record type whose fields consists of a source state, a target state, an activation event, a guard condition, and an associated action.

```
Transition: TYPE+ = [# source : Vertex,
                       trigger : Event,
                       guard : Condition,
                       effect : Action
                       target : Vertex  #]
```

Based on that definition, we formally characterize a meaningful transition by defining the following PVS predicate, which states that a meaningful transition is a transition whose source and target vertices are not pseudo states.

```
IsMeaningful?(tr: Transition):  bool =
            (Not member(source(tr), PseudoState(sm))) AND
            (Not member(target(tr), PseudoState(sm)))
```

Within a sequential state, each meaningful transition connects two configurations; there is a one-to-one mapping between configuration pairs and meaningful transitions. Therefore, we can derive configuration pairs directly by considering meaningful transitions. However, it is much more complex to identify configuration pairs from a concurrent state. A meaningful transition may connect one or more configuration pairs as shown by Figure 13. Neither the source nor the target of a transition contained in a concurrent state could represent a complete configuration. Based on that, we introduce the concepts of container and default configuration predicate.

A container represents, in UML statechart, the lowest (in the state hierarchy) composite state that contains directly a given vertex. We define the notion of container as the lowest concurrent state that contains a given state either directly or transitively.

Container(x: Vertex): Vertex

If a state vertex is not contained in any concurrent state, its container is itself. Figure 14 illustrates the concept of concurrent-container.

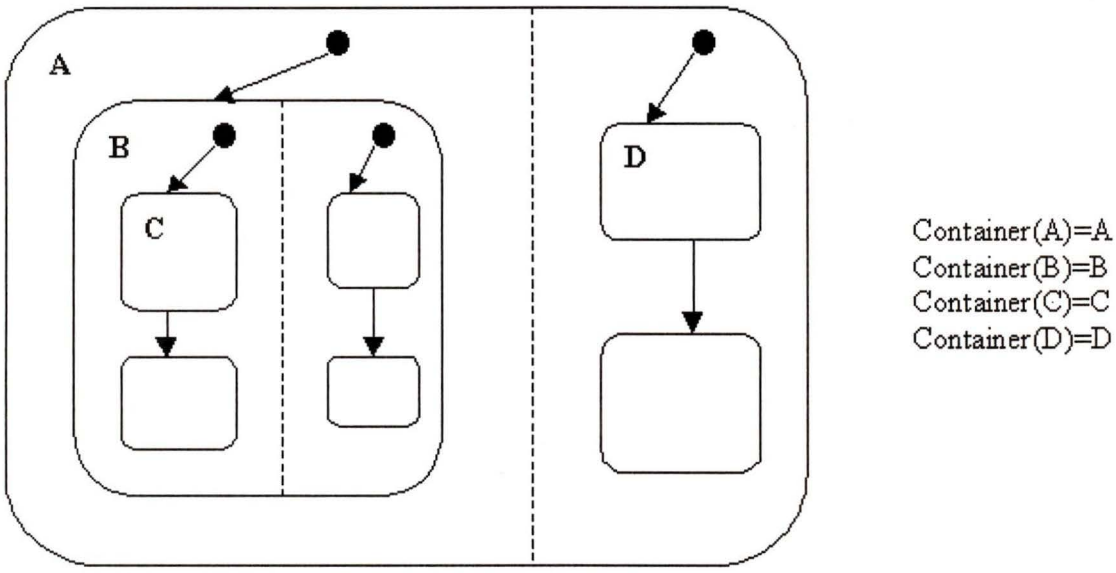

Figure 14: Container Concept

Every state has a default entry that corresponds actually to a simple state. For a simple state, the default entry leads to the state itself; for a composite state, default entries lead to default states. We define the notion of *default configuration predicate* as a predicate describing the activation status of default entries.

DefaultConfigurationPredicate(x: Vertex): PRED[V]

For a simple state, the default configuration predicate corresponds to the activation predicate as defined previously. The default configuration predicate for a sequential state is the predicate of its default state. The default configuration predicate of a concurrent state is the conjunction of its regions' default configuration predicates.

Based on the previous definitions, we provide in PVS the following formal definition for the notion of configuration pair that is applicable to both concurrent and sequential states.

```
ConfigurationPair(v1:V, v2:V):bool =
        EXISTS (tr:{tr: Transition | SmTransition(tr) AND IsMeaningful(tr)}) :
                (pred(source(tr))(v1) AND pred(container(source(tr)))(v1) AND
                 pred(guard(tr))(v1) AND
                 pred(container(target(tr)))(v2) AND
                 DefaultConfigurationPredicate(target(tr))(v2) AND
                 (pred(effect(tr))(vc) WHERE vc=(#current:=v1, next:=v2#)))
```

Where *v1* and *v2* correspond to two configurations in the statechart. Tuple *(v1,v2)* is a configuration pair if there is a meaningful transition whose source state predicate is satisfied by *v1* and target state predicate is satisfied by *v2*; in addition of that, the guard condition of the meaningful transition must be true for *v1,* and *v1* and *v2* must satisfy the associated action postcondition

3.5 Notion of PseudoStates

A pseudostate is an abstraction that encompasses different types of transient vertices in the state machine graph. They are used, typically, to connect multiple transitions into more complex state transitions paths. In UML statechart notation, several kinds of pseudostate are defined. An *initial* pseudostate represents a vertex that is the source for a single transition to the *initial inner* state of a composite state. There can be at most one initial vertex in a composite state. A *deepHistory* is used as a shorthand notation that represents the most recent active configuration of the composite state that directly contains this *deepHistory* vertex; a composite state can have at most one deep history vertex. A *shallowHistory* is a shorthand notation that represents the most recent active adjacent substate of its containing state. A composite state can have at most one *shallow history* vertex. A transition coming into the shallow history vertex is equivalent to a

transition coming into the most recent active substate of a state. A *join* vertex serves to merge several transitions emanating from source vertices in different orthogonal regions. The transitions entering a join vertex cannot have guards. A *fork* vertex serves to split an incoming transition into two or more transitions terminating on orthogonal target vertices. The segments outgoing from a fork vertex must not have guards. J*unction* vertices are semantic-free vertices that are used to chain together multiple transitions. They are used to construct compound transition paths between states. A *choice* is a vertex which, when reached, results in the dynamic evaluation of the guards of its outgoing transitions. It allows splitting of transitions into multiple outgoing paths such that the decision on which path to take may be a function of the results of prior actions performed in the same run-to-completion step. If more than one of the guards evaluates to true, an arbitrary one is selected. If none of the guards evaluates to true, then the model is considered ill-formed.

In our configuration-based abstraction scheme, the configuration pairs of a UML statechart are derived from the *meaningful* transitions of the statechart. As mentioned in section 3.4, the *meaningful* transitions are transitions that connect non-pseudo states. In our methodology, we handle some pseudo states by transforming them into several *meaningful* transitions based on their corresponding interpretations. For instance, *choice* vertices are performed by splitting them into several meaningful transitions with different guard expressions. The pseudo states transformation preserves their original meanings, therefore, we don't miss or ignore the configurations expressed by some pseudo states of the original statechart.

Chapter 4

Automatic Verification Using PVS Theorem Prover and Model Checker

The main objective of our formalization framework is to provide a ground for rigorous verification of system properties. A well-designed model should meet the properties expected in the original customer requirement. In this chapter, we will illustrate how the PVS proof system can be used to verify the system properties with the abstraction scheme presented in the previous chapter. We introduce respectively automatic proof approaches for both the PVS theorem prover and the PVS model checker.

4.1 Expressing System Properties

In our approach, a system property can be expressed using predicate function of the state variables as follows:

System_property(v:V) : bool

As we know, a system property can be either a safety property or a liveness property, or a combination of these two forms [21]. A safety property is a property that must be satisfied in any state configuration; a liveness property is a property that will eventually be satisfied in some state configurations. Having defined a property under the form of a predicate and knowing its type (e.g. safety or liveness), the corresponding verification formula of the system property can be generated without user guidance.

Based on our abstract scheme, expressing the verification formula of a safety property consists of ensuring that the property is satisfied for every configuration pairs, as follows:

Safety_property_theorem: THEOREM
FORALL (v1, v2: V):
ConfigurationPair (v1,v2) ⇒
System_property(v1) AND System_property(v2)

For the verification formula of a liveness property, we need to make sure it is satisfied for some configuration of the statechart. It can be defined as the follows:

Liveness_property_theorem: THEOREM
EXISTS (v1,v2: V):
ConfigurationPair(v1,v2) ⇒
System_property(v1) OR System_property(v2)

4.2 A Proof Strategy for PVS Theorem Prover

In general, deductive methods using theorem proving requires a considerable amount of manual guidance. Hopefully the PVS proof system provides a simple language that allows the construction of powerful proof strategies based on available primitive proof rules [15], [17]. That allows the treatment of a complex proof in a single atomic step, hiding at the same time the tedious intermediary steps to the user. But for a proof strategy to be useful it should be able to capture patterns of inference steps applicable to a large number of properties.

There are four basic forms for constructing strategies: recursion, let, backtracking, and the conditional form. A strategy can be expressed directly at the *Rule?* prompt in a proof process, or users can save their defined strategies in a file called *pvs-strategies.* PVS loads the strategies in these files from both the user's home directory and the current

context directory in which the PVS tool is invoked. A strategy is defined using the following template:

```
(defstep name
        ( required-parameters &optional optional-parameters )
         strategy-expression
         documentation-string
)
```

where *defstep* is the key word to specify a strategy definition. As the above template shows, the complete strategy definition consists of name definition, parameter definition, strategy expression, and strategy documentation. Strategy parameters can be specified as required parameters or optional parameters or rest parameters. Required parameters must be provided when using the strategy. Optional parameters are used for special purpose and are defined after the key word of *&optional*. The parameter definition and strategy documentation are optional in the strategy definition.

Since our target properties are based on a general template (see section 4.1), it is possible to identify similar or related patterns in the proof steps involved in their verification process. We have succeeded in doing so by defining a proof strategy, which consists of primitive proof steps.

```
(defstep property-proof-strategy
        (then * (auto-rewrite " user_defined_ axiom1" "user_defined_ axiom2"...)
                (skosimp)
                (expand "ConfigurationPair ")
                (grind)
        )
)
```

Our strategy name is *property-proof-strategy*; it is defined using existing PVS proof commands. The predicates defined as complementary semantics of the features involved

in a statechart diagram (e.g. state, action, condition etc.) represent assumptions on the system behavior. So we translate them in PVS as axioms. These user-defined axioms are collected and installed in the proof system as auto-rewrite rules using *auto-rewrite* command, so that the PVS theorem prover is able to search these axioms automatically during the proof process. *Skosimp* command replaces universal quanlifications in the target fomula with constants. *Expand* command expands our *ConfigurationPair* definition. *Grind* command is a catch-all strategy that is frequently used to automatically complete a proof branch or to apply all the obvious simplifications till they no longer apply, it first installs the rewrite rules along with all relevent definitions in the given subgoal, then carries out all the equality replacements.

When this strategy is used to prove a fomula (see section 4.1), the theorem prover actually goes through every possible configuration involved in valid configuration pairs, then take advantage of *grind* command to check the property. It will return unproved subgoals (counter examples), when the PVS prover can not successfully verify the system property on all the possible configurations, in which case it helps identifying the design errors.

4.3 Automatic Verification Using PVS Model Checker

4.3.1 State Space Explosion Problem

The state space of a system can be loosely defined as the full range of values assumed by the state variables of the program or specification that describes it. The behaviors that the system can exhibit can then be enumerated in terms of this range of values. If the state space is finite and reasonably small, it is possible to systematically enumerate all possible

behaviors of the system. However, few interesting systems have tractable state spaces, which make model checking less effective. It is generally called state space explosion problem. Furthermore, the state space of a formal specification can be infinite, for example, if it uses mathematical integers as values for state variables, it definitely have no bound for the global states; on the other hand, when asynchronous systems are modeled, the concurrency among their components cause their global state spaces to increase exponentially which make them unfeasible for model checking.

Nevertheless, there are various techniques for "downscaling" or reducing the state space of a system, while preserving its essential properties. For instance, OBDDs (Ordered Binary Decision Diagrams) [39] provides a canonical form for boolean formulas that is often substantially more compact than conjunctive or disjunctive normal form, and very efficient algorithms have been developed for manipulating them; Partial Order reduction [40] exploit the independence of concurrently executed events, which decreases significantly the state space of the system. Other techniques like Abstraction [41], Symmetry [42], and Induction [43], are also important techniques to reduce the complexity of model checking.

In our approach, we adopted an abstraction methodology developed for PVS model checker, which solves the state space explosion problem caused by the range of values of state variables.

4.3.2 Addressing State Space Explosion Using Boolean Abstraction

As we mentioned in previous chapters, in our formalization scheme the system global state is represented by a PVS record type named V whose fields consists of the system variables *x1,...,xn* of respective types *T1,...,Tn*, where *T1,...Tn* can be any kind of types

(e.g. boolean, integer, etc.). For an infinite state system (e.g. the system variables contain an integer type), our global state definition can lead to state space explosion, which is a big barrier to using model checking. Hopefully, a boolean abstraction thechnique has been developed in the PVS proof system that helps in breaking the complexity of inifinite state problems.

Boolean abstraction is a simple abstraction scheme defined in [35] that consists of using predicates over concrete variables as boolean abstract variables. In an abstract version of the infinite-state transition system, the set *{B 1, ..., B k}* of abstract variables is a set of boolean variables corresponding to predicates over the concrete variables. An abstract state in this transition system is therefore a truth assignment to these boolean variables. Since the set of boolean values is finite, so is the set of abstract states. For instance, let us consider the following state type:

state : TYPE = [# pc : control ; y1,y2: nat #]

where control is an enumerated type: *control : TYPE = {idle, wait, enter, exit}*. Considering a boolean abstraction defined with the predicate *lamda(s:state): y1(s)=y2(s)*, the abstract state type is defined as follows:

abs_state : TYPE = [# pc : control; B_1 : boolean #]

The assertion *pc(state1)= idle & state2= state1 with [pc := wait ; y1 := y2(state1)+1]* is abstracted to *(pc(abs_state1))=idle & pc(abs_state2)=wait and not B_1(abs_state_2)* The PVS command *abstract-and-mc* implements an abstract compiler for PVS assertions based on boolean abstractions. Given a state type *STATE* defined as a record type and a set of state predicates *(pred_1 ... pred_k),* an abstract state type *ABS_STATE* is defined. It

consists of a record type with k boolean components and the remaining components of *STATE* that are not referenced in the predicates *(pred_1 ... pred_k)*. The command translates any PVS assertion over variables of type *STATE* to a stronger assertion over variables of type *ABS_STATE*, and then uses PVS model checker to verify the system properties.

4.3.3 Using Configuration Predicates For Boolean Abstraction

We have provided, in previous chapters, a formal definition of the UML statechart and have defined a transition function, *ConfigurationPair(v1:V, v2:V):bool*, based on configuration-based abstraction of UML statechart. Since the number of configurations of a UML statechart is finite, it is natural to use an abstract state type to represent the state configuration. Thus, we can use the abstract state type with the PVS model checker instead of using system variable type *V* which corresponds to the concrete system state.

We note that the semantic of the notion of *Configuration* involves assertions over system variables that come from the predicates of simple state, guards, and actions. These predicates are composed of logical units that are connected by logical connectors (e.g. AND, OR, etc.). Each logical unit is a boolean expression over system variables. If we use Boolean variables *B1, B2, ..., Bk* to represent respectively the logical units, a configuration thus corresponds to a truth assignment of the Boolean variables.

In our approach, we implement, using JAVA, a software component that can take a complex logical expression and parse it into logical units. And all the unduplicated logical units derived from the configuration predicates of a UML statechart are used as inputs of the PVS *abstract-and-mc* command.

Therefore, the theorems of a safety property and a liveness property may be defined using *CTL* formulas (see section 2.4.2), as follows:

Safety_property_theorem: THEOREM
*init(v) => **AG**(ConfigurationPair, System_property)(v)*

Liveness_property_theorem: THEOREM
*init(v) => **EF**(ConfigurationPair, System_property)(v)*

The init(v) defines an initial configuration. The PVS model checker can then verify the above theorems using the derived logical units, as follows:

(ABSTRACT-AND –MC "logical_unit_1" "logical_unit_2" ... "logical_unit_k")

Where *ABSTRACT-AND –MC* is the actual abstraction and model checking command. The command first abstracts the infinite state type *V* into a model checkable form (*Abstract_V)*, based on the PVS assertions (logical units) provided. Then, the *init* predicate, the *ConfigurationPair* definition and the *System_property* definition are redefined according to the abstract state type. After that, the model checker expands the initialization predicate *Abstract_init,* the transition relation *Abstract_ConfigurationPair,* and rewrites the CTL operations. The formula is translated into the Boolean mu-calculus [19] and then checked using a BDD-based validity checker.

Chapter 5

Implementation Of The Automated Formal Analysis Framework In PrUDE

Tools are needed to assist the formal specification and verification process. The PrUDE tool suite is being developed in this respect. In this chapter, the implementation of the automated formal analysis framework in PrUDE will be discussed, including the PrUDE tool framework, its main functionalities and underlying implementation issues.

5.1 PrUDE Framework

PrUDE (Precise UML Development Environment) is being developed by the ISOT group (http://www.isot.ece.uvic.ca). The purpose of the tool suite is to assist the user in the software validation and verification process. Figure 15 shows the general framework of the PrUDE tool suite.

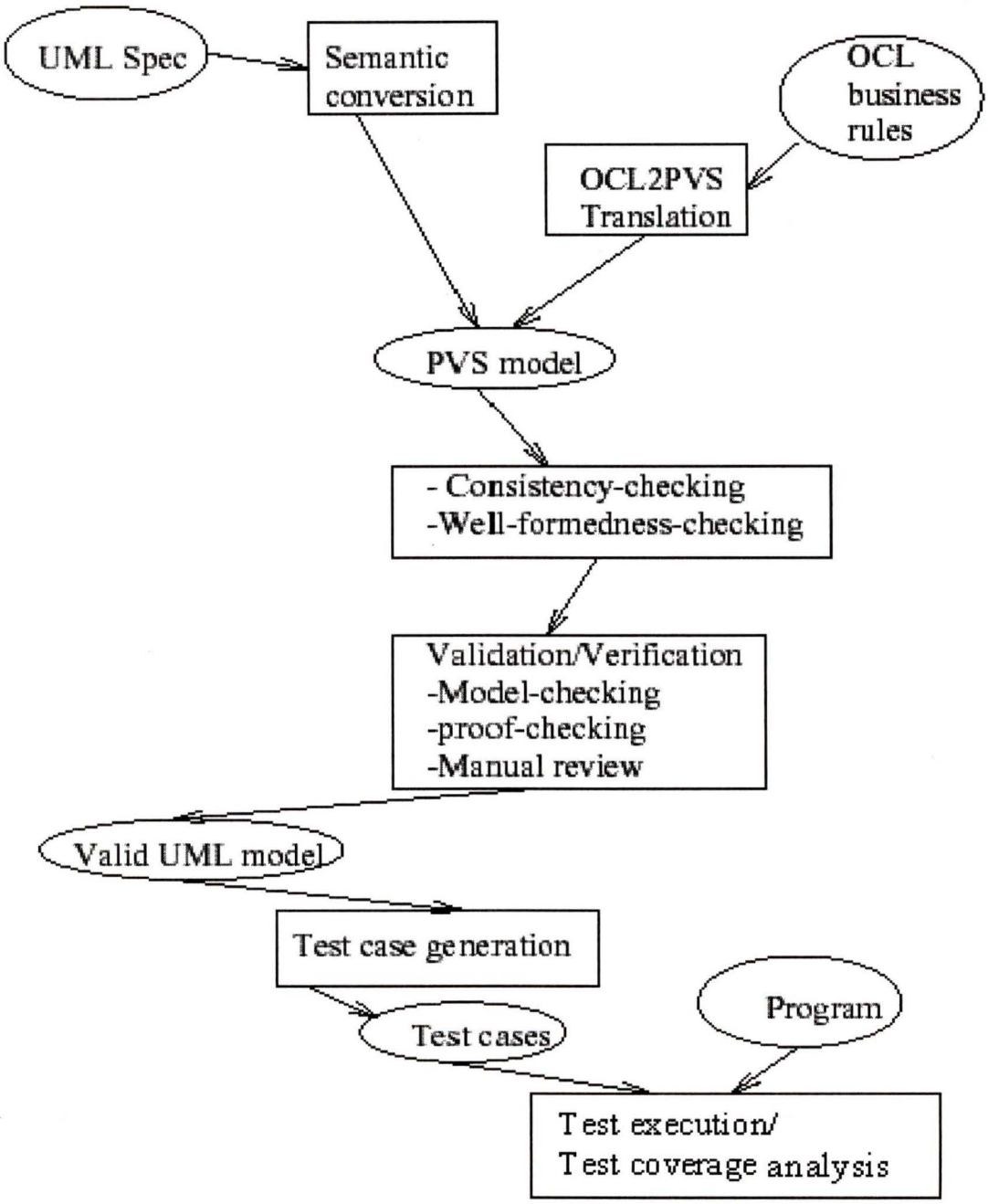

Figure 15: PrUDE Framework

In practice, the user builds his UML model using his favorite tool (e.g. Rational rose, Argo UML etc.), and then submits the model to the PrUDE tool which generates automatically at the back-end a formal semantic in PVS. For formal verification purpose, the user needs to complete his UML model by specifying additional constraints such as invariants, preconditions, post-conditions, system properties, etc. The constraints can be expressed using the Object Constraint language (OCL) [6] and then translated into PVS and integrated with the semantic model in PrUDE. PrUDE provides a property editor that allows users to capture the constraints directly in PrUDE. The PVS semantic model obtained may be checked for well-formedness and consistency against the rules defined in the UML standard [3]. Well-formedness checking and consistency checking mainly focus on syntax correctness of the UML models. In the next step, the model is verified against the system properties. Invoking the PVS model checker and theorem prover in the batch mode carries out the automatic verification of these properties. The verification results are forwarded to PrUDE, which displays them to the user. The user is able to trace the errors in the UML models by analyzing the verification messages. In principle, if a problem is discovered, the user goes back to the OCL business constraints and/or UML models to fix the error. The above process is iterated until a valid UML model is obtained. The specification model obtained after the previous validation step can serve as basis for test case generation; the test cases are derived from various constraints related to the model, e.g. invariants, preconditions, and post-conditions. And the test cases generated are systematically submitted to a built-in test execution component, which can test corresponding implementation.

5.2 Main Functionalities Of PrUDE

The main functionalities of PrUDE are carried by four key components that are respectively *Specifier, Analyzer, Generator and Tester.* Each of the four components implements some aspects of the formal analysis framework. We briefly illustrate the roles of these components in PrUDE.

5.2.1 Specifier Component

As the component name indicates, the *Specifier* component handles the generation of the PVS formal specification for the UML model. A XMI file, which contains the UML model, is input to the *Specifier* component. The component parses the XMI file to get UML model elements with its structural information, then the corresponding PVS semantic is generated automatically based on the translation rules defined in PrUDE. The *Specifier* component also provides an editor that allows users to augment the UML mode. The *Specifier* integrates the complementary information provided into the generated PVS semantic. Users are also able to modify or redefine their model constraints easily using the *Specifier*.

5.2.2 Analyzer Component

The *Analyzer* is the key component for verification activities. It re-expresses the system properties into suitable PVS formulas prior to verification using corresponding templates. During a property verification activity, the *Analyzer* generates dynamically a proof strategy for the PVS theorem prover (or a proof fomula for the PVS model checker), based on the PVS semantic provided by the *Specifier* component. Then the PVS proof system is invoked and runs at the back end, and the verification results are displayed to the user who can trace back and modify the UML model if needed.

5.2.3 Generator Component

The *Generator* component is used to generate test cases for program, based on the UML specification. The actual testing approach used in the component is a specification–based method. Having obtained a valid UML model after the formal verification step, the component can generate test cases from the various constraints associated to the model elements, e.g. classes, states, and operations, and the test cases are generated automatically based on strategies implemented in the component, e.g. a relational testing strategy for class diagram, a transition strategy for statechart diagram [36], etc. In practice, test constraints and assertions used by the *Generator* component are decomposed into disjuctive normal form (DNF), yielding elementary expressions. The expressions are refined into executable expressions, and then suitable test values can be defined using the domain test matrix technique [22].

5.2.4 Tester Component

The *Tester* component implements a test execution environment for JAVA programs. It is designed with the objective of hiding the inherent complexity of testing procedure by increasing the level of automation of the testing process. The *Tester* can explore the inner structure of the Java classes using the Java reflection API, and executes testing in the method level. The component also provides test report in PrUDE that shows the testing result for each test cases.

5.3 Implementation and Operational Aspects of PrUDE

5.3.1 PrUDE Main Interface

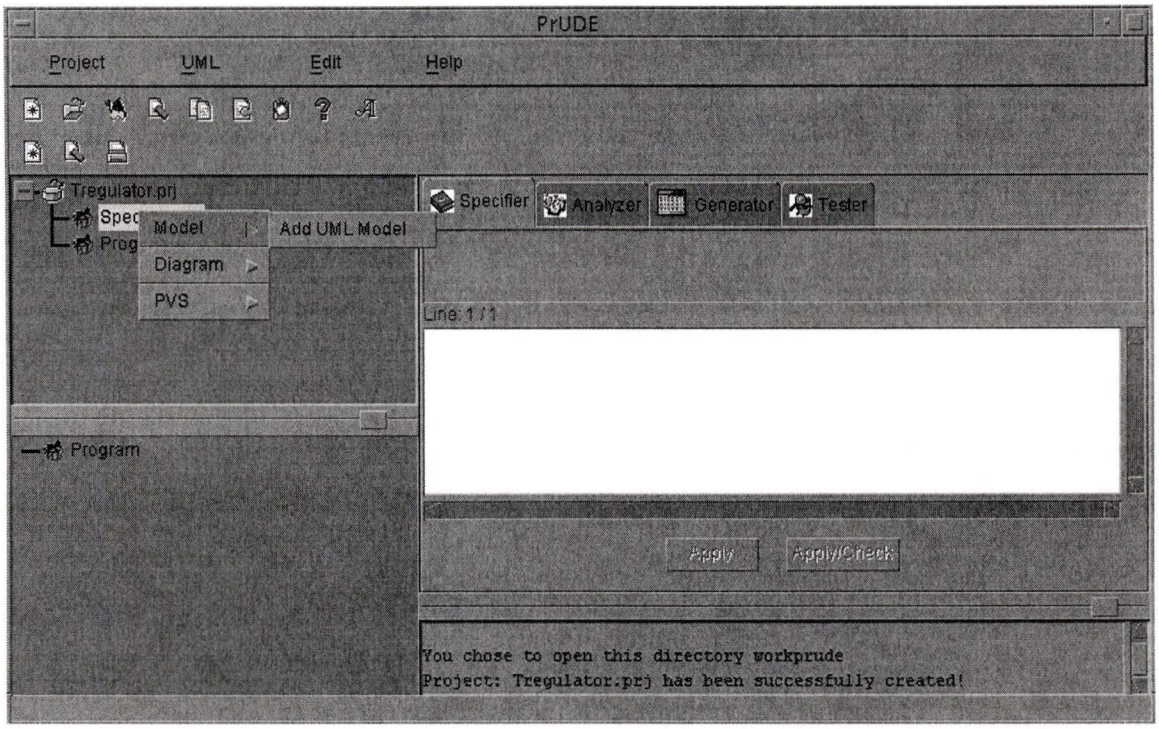

Figure 16: PrUDE Main Interface

Figure 16 shows the main interface of the PrUDE tool suite. At the top there is a menu bar providing various options. In the project menu you can create a new model or open an exsiting model instead. Several other commands can be invoked directly from the menu bar. The upper left part of PrUDE main window shows a browser that displays the structure of specifications and programs submitted to PrUDE. The lower left part of PrUDE shows another browser that is used for testing purpose, an inner structure of a tested component (e.g. class, package) can be browsed in this tree. Users can verify specifications or test programs using a popup menu which appears by right-clicking on the correponding node. The right part of the main window contains a tabbed pane, a text area, and a log window. Users can choose to work on different domain of the project by selecting the corresponding tab and the text area shows the related information for a specific domain. The log window in the lower part shows PrUDE run-time status.

5.3.2 Access XMI Information

Figure 17 shows our concrete implementation on how to retrieve necessary information from the XMI file.

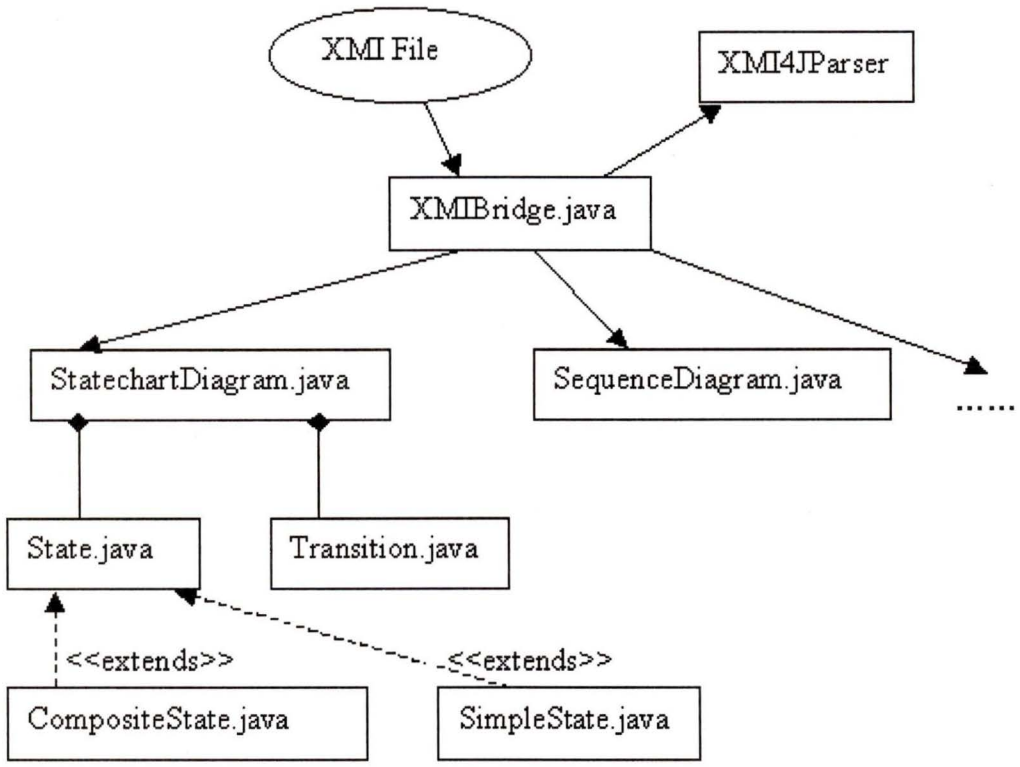

Figure 17: Access XMI Information

XMI4JParser [31] is Java API (Application Programming Interface) package that is used to access XMI file repositories, from which all the information about the user model can be recovered. *XMIBridge.java* is the key component who is in charge of collecting the valuable model information and creating the corresponding UML model object. For instance, the model object of statechart diagram is composed of a set of *State* objects and a set of *Transition* objects.

Figure 18 shows how a UML model is displayed in PrUDE after importing corresponding XMI file.

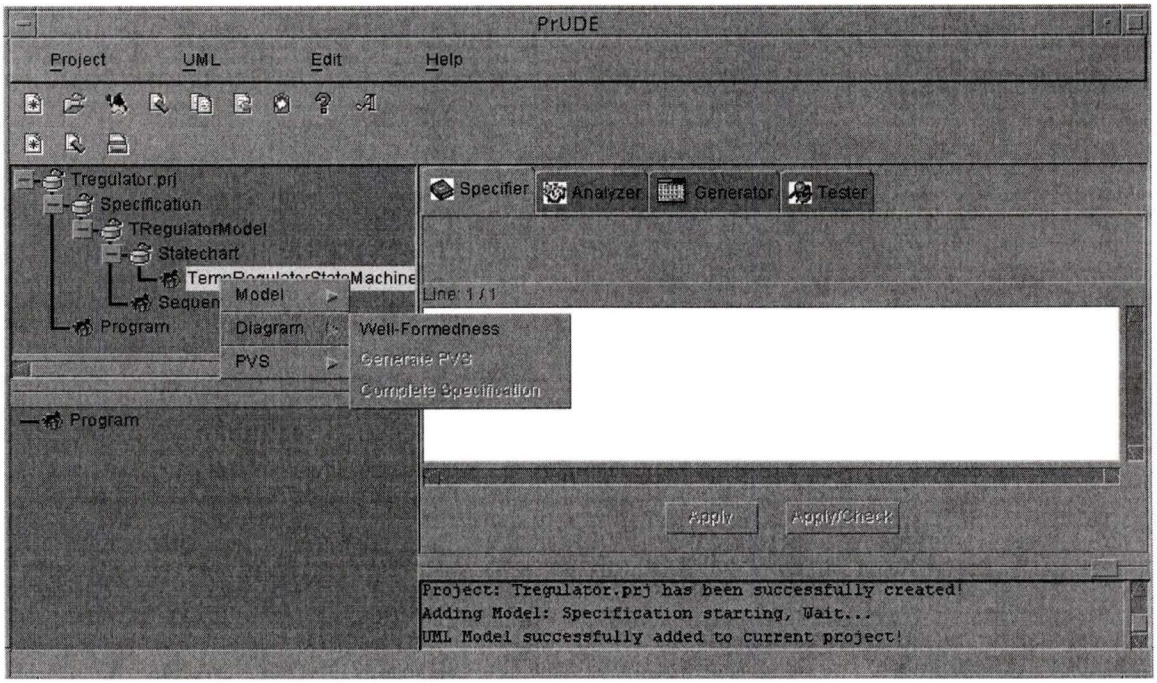

Figure 18: Import a UML Model

5.3.3 Generate And Complete PVS Specification

Since some concrete information required by our verification method is not exported in the XMI files of existing UML CASE tools, we divide the generation of the PVS specification into two steps.

First, a skeleton of PVS specification is generated automatically based on the model information retrieved from the XMI file. A software component, called XMI2PVS, is implemented for this purpose. It takes a model object that contains the various UML diagram objects and then generates corresponding PVS theories, as Figure 19 illustrates.

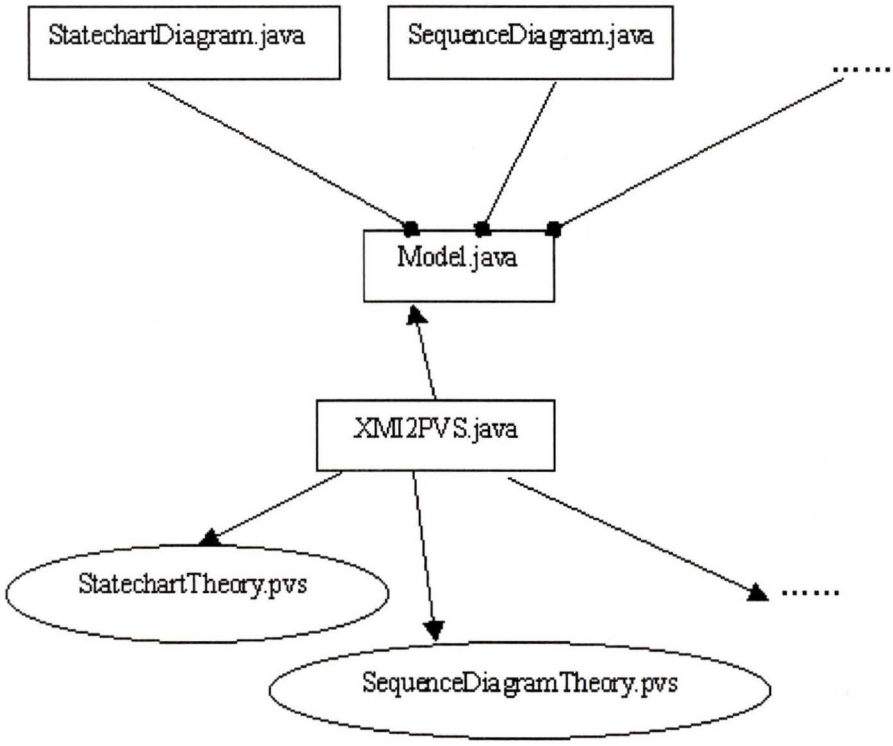

Figure 19: Generate PVS Specification

Second, the user completes the generated semantics directly in PrUDE. Figure 20 shows the interface from where the constraints of state elements can be added to the generated PVS specification. Users, firstly, select an element (e.g. a simple state, an action, etc.) in the tabbed pane, then type the description of the corresponding constraints in the text area and press the *Apply* button. After all the constraints are applied, a more complete PVS semantic is generated for the UML specification.

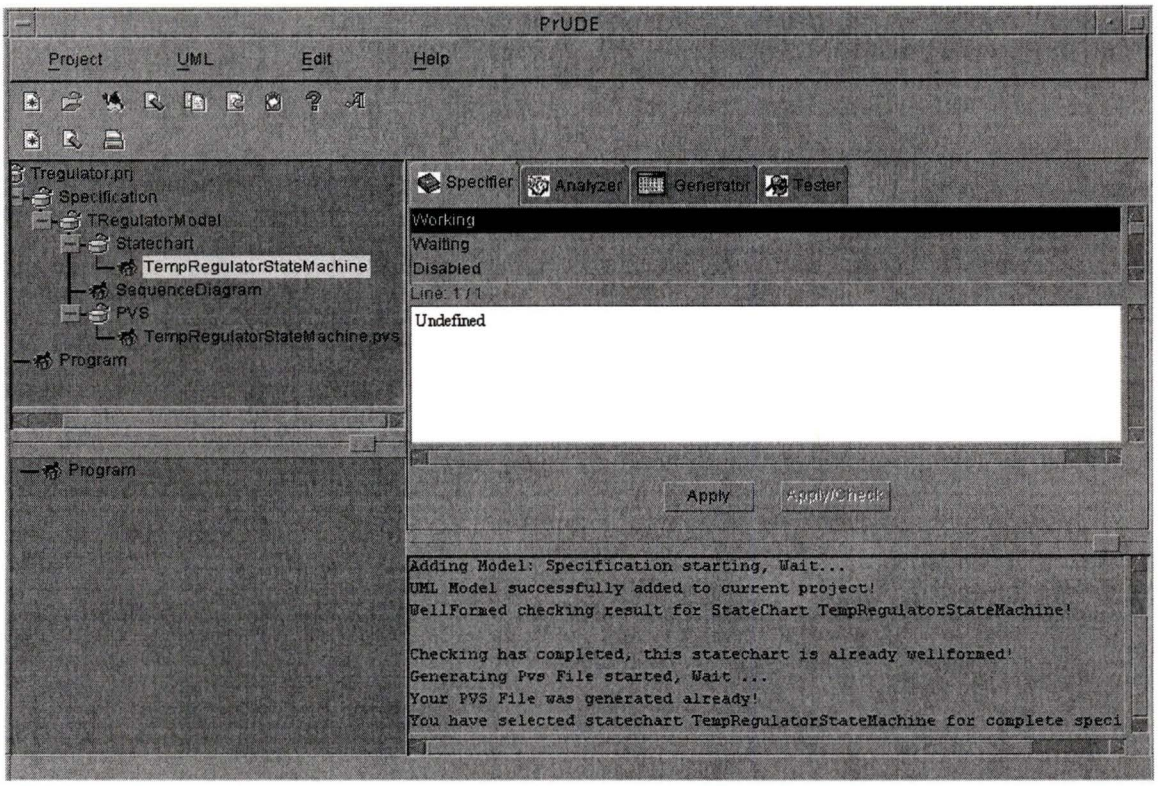

Figure 20: Complete Specification

5.3.4 Verify System Properties

System properties can be verified either using theorem-proving or model-checking. In practice, the user selects a system property in the project browser, and then chooses a proof command by using the pop up menu. Since we don't expect users to interact with the PVS proof system, the PVS system is invoked in batch mode. In order to automate the verification process, a *PVSInvoker* component is implemented. The component generates a PVS batch file and a strategy file based on the generated PVS specification. The batch file contains run-time information including the PVS system initialization, the target specification name, the proof command, etc. The strategy file is used by the PVS proof system. The PVS theorem prover contains a proof strategy specific to the target specification. The PVS model checker contains the abstraction predicates information.

The *PVSInvoker* component executes the batch file in an independent thread and redirect the verification result into PrUDE. Figure 21 shows how the *PVSInvoker* component works with the PVS system.

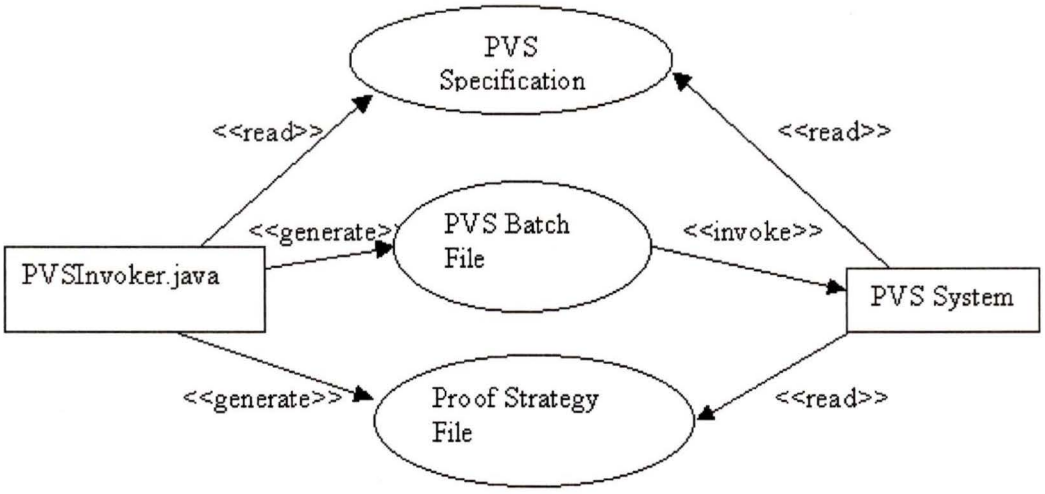

Figure 21: PVSInvoker and PVS System

System properties can be input directly in PrUDE as Figure 22 shows.

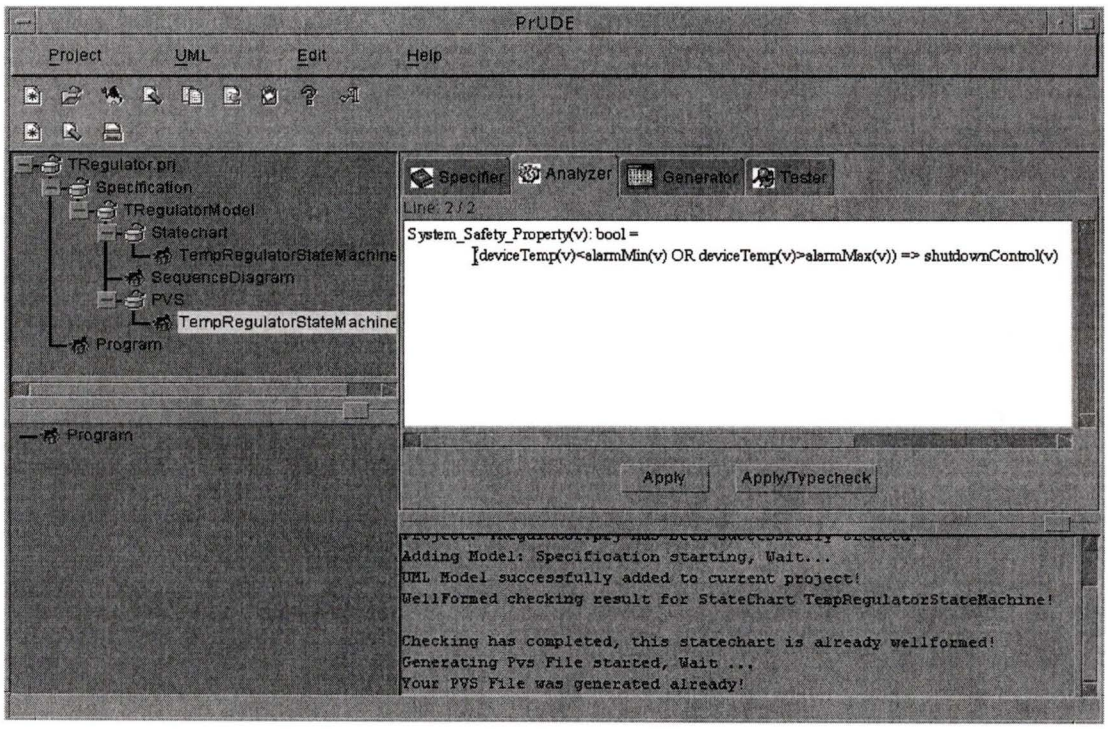

Figure 22: Add System Property

Chapter 6

Case Study: A Temperature Regulator System

In this chapter, we illustrate by a concrete example how our methodology can be used for rigorous model analysis.

6.1 Requirement Specification of A Temperature Regulator Software Component

6.1.1 Summary Of The Requirement

We consider a temperature regulator software component involving a piece of software that is intended for use in devices requiring temperature control such as heaters, hot plates etc [23]. The main purpose of the regulator is to ensure that the device temperature is constant and stays within set safety limits. The temperature regulation process involves, as depicted by Figure 23, a temperature detecting unit (e.g. sensors), a temperature control unit, and some actuators. The device temperature detected by the sensors is sent to the control unit, which compares them to the desired temperature. If the measured temperature is different from the desired one, the control unit computes suitable correction value that is applied to the device via the actuators. In addition if the device temperature is out of the set safety limits; the user is notified by an alarm. The desired operating temperatures are selected by the users, and then checked by the system for range and consistency; the operating ranges for the temperatures involved in the component are assumed to be [0, 500]. The operational parameters consist of three kinds of values: the desired temperature denoted by *desiredTemp*, the lower limit for alarm

generation called *alarmMin*, and the upper limit for alarm generation called *alarmMax*. The device temperature as measured by the sensors is denoted by *deviceTemp*. In order to be able to control the devices (for safety purpose) the system generates the following output signals:

- A signal for on/off control of a device denoted by *deviceControl.*
- An additional control signal to disable software/hardware in the event of an unsafe condition called *shutdownControl.*

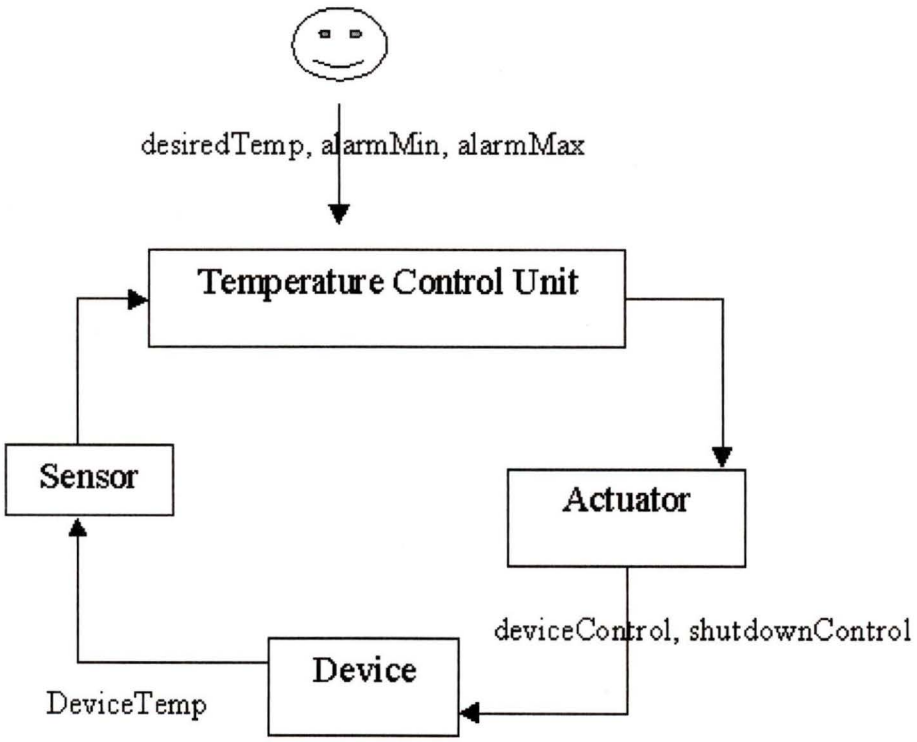

Figure 23: Temperature Regulator Process

6.1.2 Overview Of The Requirements Specification

The UML specification consists of various diagrams each describing specific view of the system. We present in the following some of these diagrams.

Figure 24 describes the use cases involved in the system. Each use case corresponds to specific functionality performed by the system. Each involves one or more scenarios, each corresponding to specific interaction patterns. There are three kinds of external actors that interact with the system, namely the operator or user, the sensors and the actuators; an actor is represented in UML as a stickman. We consider 4 use cases represented by ovals, namely a *Setup* use case, which sets and validates operational parameters on behalf of the user, a *Display parameters* use case, which displays the operational values, a *Temperature regulation* use case, which performs the control operations, and a *Check safety* use case which performs the safety checking activities.

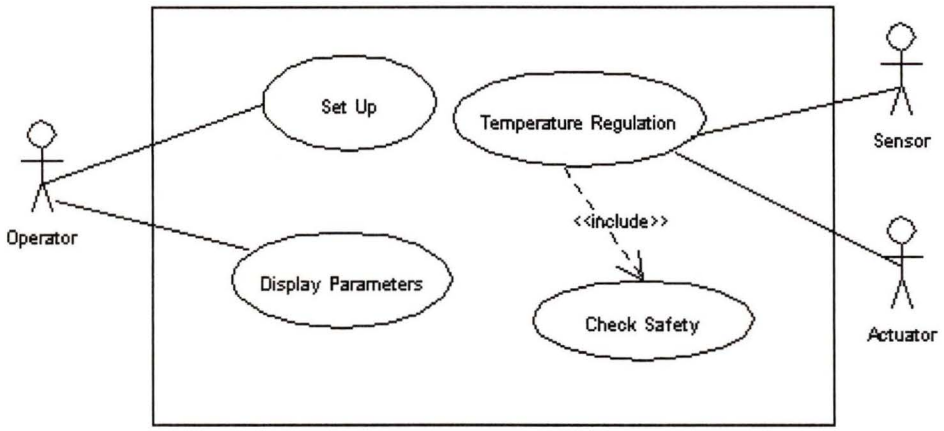

Figure 24: Use Case Diagram

We describe the scenarios involved in each use case by providing UML sequence diagrams. Figure 25 describes the basic scenario underlying the temperature regulation process. It interacts with two other classes denoted. *TempSensor* and *DeviceController* encapsulating respectively computations related to the sensors and the device controller. Figure 26 shows a class diagram describing the mentioned classes and their relationships.

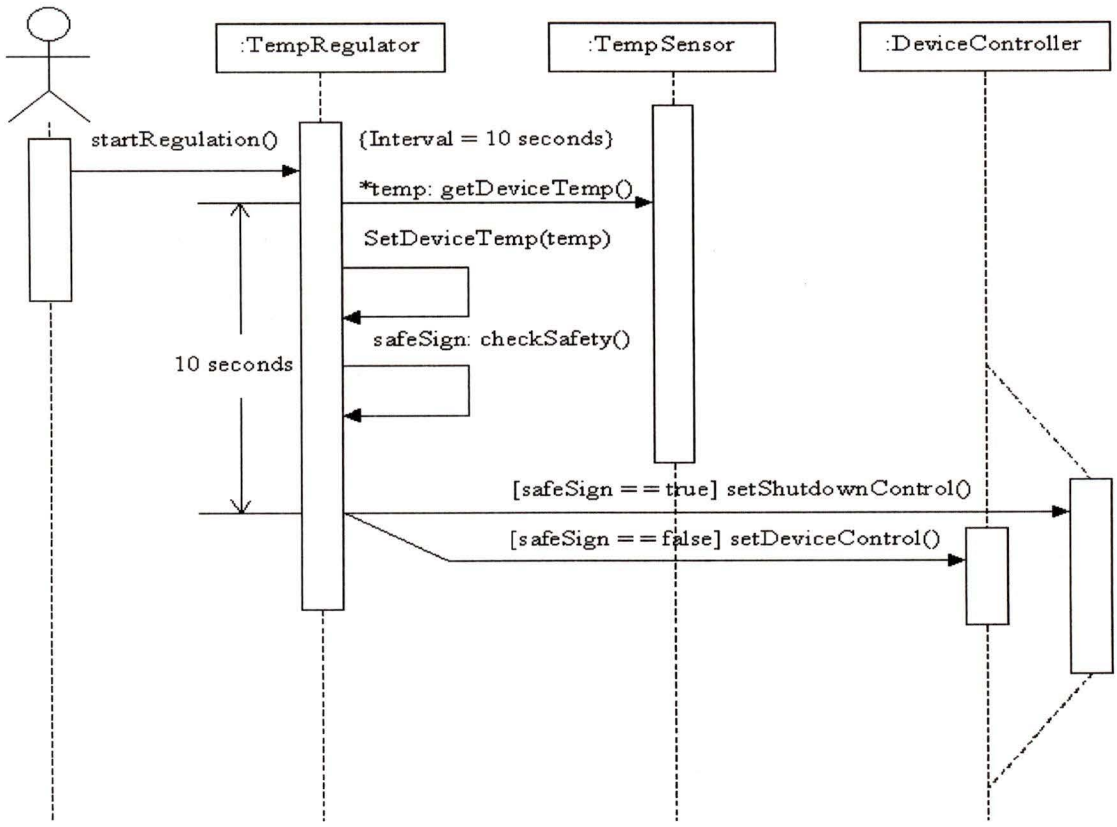

Figure 25: Scenario Describing Temperature Regulation Process

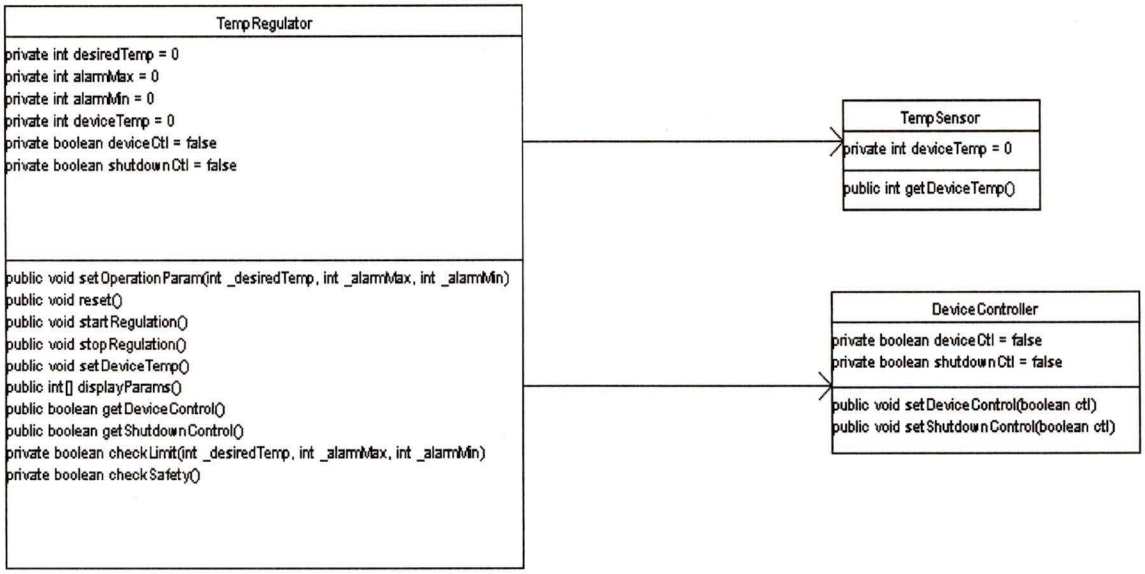

Figure 26: Class Diagram

We describe the dynamic behavior of the class *TempRegulator* by providing the statechart diagram depicted by Figure 27. Initially the *TempRegulator* object is in a disabled state denoted by *Disabled*. When the operator sets the operational parameters the object moves to a state called *Enabled* where it is ready to start working. State *Enabled* is divided into two direct substates denoted by *Waiting* and *Working*. When state Enabled is entered, the object is first in an idle state denoted by substate *Waiting*. Then when the user clicks on the start button, substate *Working* becomes active. If the user doesn't press the stop button, the regulator will keep monitoring the temperature and reset the output signal if necessary. Anytime when the reset button is pressed, the regulator object will go back to *Disabled* state and restore all default operational parameter values.

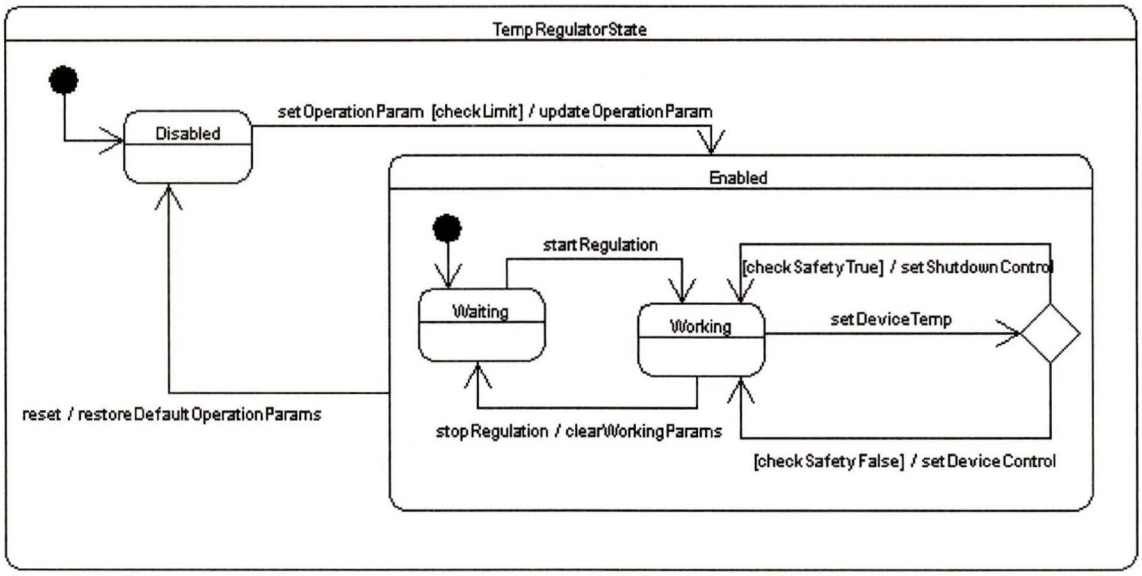

Figure 27: Temperature Regulator Statechart Diagram

6.2 Formal Analysis of The Temperature Regulator Statechart

6.2.1 Formal Semantic of The Statechart Diagram

The PrUDE tool can generate the PVS semantic of the temperature regulator statechart automatically, based on the formalization work presented in [8]. Actually, a PVS theory, which contains the application-specific statechart semantic, is generated. The generic semantics are provided as PVS library. The application-specific PVS theory thereby imports systematically the generic statechart semantics from the library and defines application-specific semantics provided in the UML statechart model. Figure 28 shows an overview of the PVS semantic (application-specific) generated for the temperature regulation statechart using PrUDE.

```
TempRegulatorStateMachine: THEORY

BEGIN
  %Import theory AbstractSyntax from the library
  prudelib: LIBRARY = "~/prude/semantic/lib/prudelib"
  IMPORTING prudelib@AbstractSyntax

  %Instance variables
  V:TYPE
  % v corresponding to the current state and v1 to the next one
  v, v1, v2: VAR V
  x: VAR Vertex
  defaultState(x): Vertex
  %StubState definition
  SmStub: finite_set[Vertex] = emptyset
  ... ...
  %State definitions
  Working: Vertex
  Waiting: Vertex
  Disabled: Vertex
  ... ...
  %SignalEvent definitions
  stopRegulation: Event
  ......
  %Action definitions
  setShutdownControl: Action
  ......
  %Guard definitions
  checkSafetyTrue: Condition
  ......
  %Transition definitions
  t6: Transition=(#source:=Working, trigger:=setDeviceTemp,
                  guard:=checkSafetyTrue,
                  effect:=setShutdownControl, target:=Waiting#)
  ......
  %State machine definition
  sm: StateMachine
  ......
  %import theory WellFormedness and FormalSemantics
  IMPORTING prudelib@WellFormedness[sm]
  IMPORTING prudelib@FormalSemantics[sm,V]
  IMPORTING prudelib@state[V]
  ......
  END TempRegulatorStateMachine
```

Figure 28: PVS Semantic of Temperature Regulator Statechart

6.2.2 Complementary Semantics for the Regulator Statechart

As noted earlier the UML graphical model is not enough to provide a precise and complete specification for the system. We need to provide additional definitions describing in a meaningful way the graphical constructs used. We provide, for instance, for the constructs involved in a statechart diagram, additional semantic definitions under the form of predicates. We describe in the sequel some of the predicates associated to the statechart diagram of Figure 27. The predicates are integrated directly by the designer to the generated PVS semantic using PrUDE (see Figure 20). The predicate expressions are function of the global state type V that we have introduced previously as a record whose fields correspond to the state variables involved. For instance, we define below the specific type V corresponding to our statechart example.

```
V: TYPE = [# desiredTemp : int,
            alarmMax : int,
            alarmMin : int,
            tempLimit : int,
            deviceTemp : int,
            deviceControl : bool,
            shutdownControl : bool#]
v: VAR V
```

We need to provide predicates only for simple states, guards and actions; predicates corresponding to composite states are derived by composing (e.g. conjunction, disjunction) the predicates of their substates.

For instance, in state *Disabled* all the state variables are set to their default values. The predicate of state *Disabled* can be expressed as follows:

pred(Disabled)(v) = (desiredTemp(v) =0) AND
(alarmMax(v) = 0) AND
(alarmMin(v) = 0) AND
(deviceTemp(v) =0) AND
(deviceControl (v)= FALSE) AND
(shutdownControl(v) = FALSE)

Guard condition *checkSafetyFalse* ensures that the device temperature detected is within the safety range defined initially. Its predicate is as follows:

pred(checkSafetyFalse) *= (deviceTemp(v) < alarmMax(v)) AND*
(deviceTemp(v) > alarmMin(v))

The predicates associated to actions are function of the current and next state values described in our work by the record type VC.

VC: TYPE = [#current: V, next: V#]
vc: VAR VC

Action *setDeviceControl* sets the output signal *deviceControl* after comparing the detected device temperature (e.g. *deviceTemp*) to the desired temperature (e.g. *desiredTemp*) as set initially. If *deviceTemp* is less than *desiredTemp*, then output *deviceControl* is set to true and the device heats up. Otherwise, *deviceControl* is set to false and the device cools down.

pred*(setDeviceControl) (vc)= IF(deviceTemp(current(vc)) >= desiredTemp(current(vc)))*
DeviceControl(next(vc)) = false
THEN
DeviceControl(next(vc)) = true
END IF

A complete formal semantic for the temperature regulator statechart is shown in the Appendix A. It contains all the predicates defined for the statechart and their corresponding explanations.

6.2.3 V & V Activities

Having generated a formal representation of the temperature regulator statechart, we can check whether the designed model satisfies the system requirement using the automated framework presented in the previous chapters.

6.2.3.1 Identifying and Expressing System Properties

We describe the system requirements as system properties. The properties definitions can be input directly in PrUDE (see Figure 20 for a snapshot). Let's consider for instance the following safety requirement.

Requirement: *If the detected device temperature is outside the range determined by safety parameters, then the device should be shutdown.*

We have defined, in the state type *V*, the safety parameters using *alarmMin*, and *alarmMax*, and we assign *truth* to *shutdownControl* to represent a shutdown operation. Thus, a corresponding safety property can be defined as follows:

System_Safety_Property(v): bool =
(deviceTemp(v) < alarmMin(v) OR deviceTemp(v) > alarmMax(v))
=> shutdownControl(v)

Let's consider another example of important requirement as follows:

Requirement: *When the temperature regulator is in working situation, the detected device temperature will vary around the desired temperature. Accordingly, two situations should eventually happen; one is that the device heats up when the detected temperature is less than the desired temperature, the other is that the device cools down when the detected temperature is greater than the desired temperature.*

Assigning truth to deviceControl variable represents the heating up operation of the device. Since this requirement is only forced in working status, we define a corresponding liveness property as follows:

System_Liveness_Property(v): bool =
deviceTemp(v) < desiredTemp(v) <=> deviceControl(v)

6.2.3.2 Checking System Properties

PrUDE can check whether the designed model satisfies a system property by invoking the PVS proof system in batch mode and applying our proof patterns. No human interaction is needed in the verification process and the verification results are redirected from the PVS proof system to PrUDE.

For instance, if the user selects proof checking for the properties identified in section 6.2.3.1, then PrUDE will generate the following theorems dynamically and check them using the corresponding strategy defined in section 4.2.

safety_property_theorem: *THEOREM FORALL (v1,v2:V):*
ConfigurationPair (v1, v2) => System_Safety_Property(v1) AND
System_Safety_Property(v2)

liveness_property_theorem: *THEOREM EXISTS (v1,v2:V):*
ConfigurationPair (v1, v2) => System_Liveness_Property(v1) OR
System_Liveness_Property(v2)

If the user selects the PVS model checker, CTL formulas are generated automatically by PrUDE and checked either for *safety* and *liveness*, as follows:

safety_property_theorem: *THEOREM*
pred(Disabled)(v) => AG(ConfigurationPair, System_Safety_Property)(v)

liveness_property_theorem: *THEOREM*
pred(Disabled)(v) => EF(ConfigurationPair, System_property)(v)

Figure 29 shows proof checking result for the safety property using PrUDE, the property has been automatically checked in 36.46 seconds.

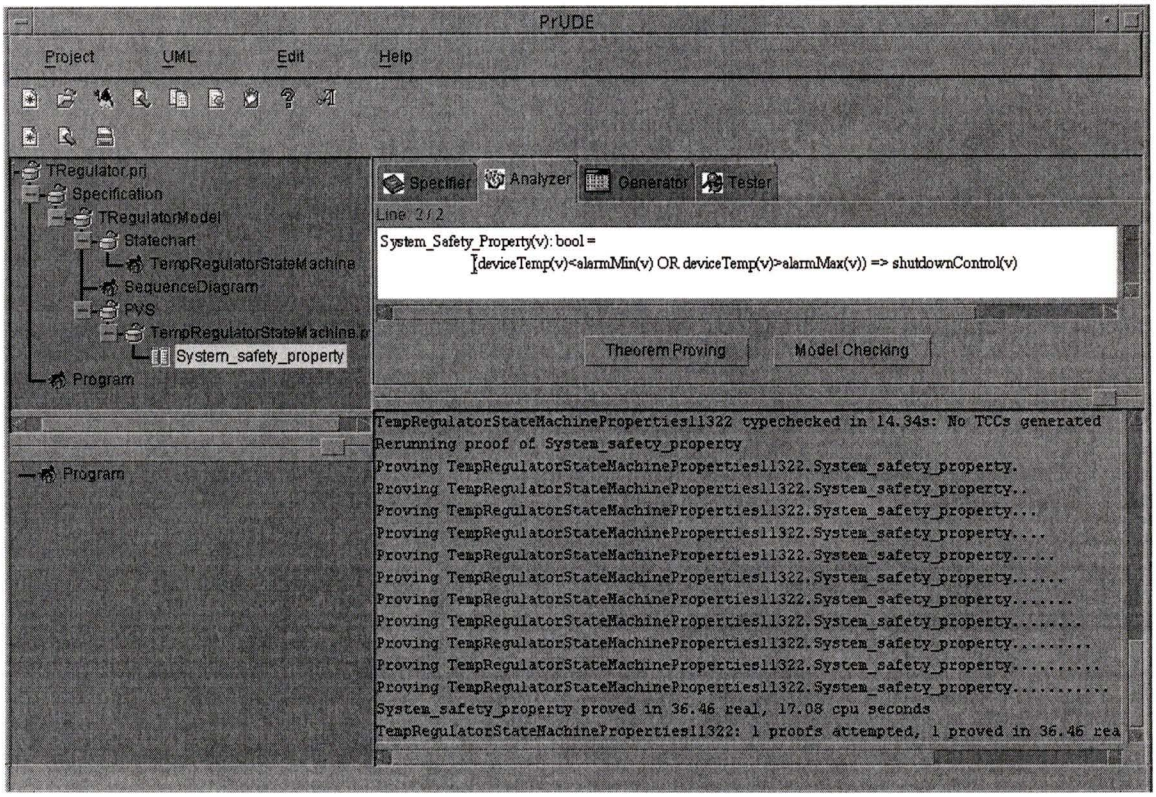

Figure 29: Proving Safety Property

Without using our automated verification framework, the property theorems can't be expressed using a general PVS formula. And if we check this property directly using the PVS tool, we have to manually direct the proof for each subgoal in the proof tree and it takes much more time and user interactions to complete the proof as Figure 30 shows. On the other hand, without using a proof strategy based on proof patterns, we can't be sure whether the theorem is not provable or the proof guidance is incorrect if users fail a proof in the interactive proof mode.

```
System_safety_property :
  |-------
{1}   FORALL (v1, v2: V):
      ConfigurationPair(v1, v2) =>
       System_Safety_Property(v1) AND System_Safety_Property(v2)
Rule? (skosimp)
Skolemizing and flattening,
this simplifies to:
System_safety_property :
{-1}  ConfigurationPair(v1!1, v2!1)
  |-------
{1}   System_Safety_Property(v1!1) AND System_Safety_Property(v2!1)
Rule? (expand "ConfigurationPair")
Expanding the definition of ConfigurationPair,
.....
Rule? (grind)
Trying repeated skolemization, instantiation, and if-lifting,
this yields  24 subgoals:
System_safety_property.1 :
{-1}  tr!1 =
     (# source := Working,
        trigger := setDeviceTemp,
        guard := checkSafetyTrue,
        effect := setShutdownControl,
        target := Waiting #)
{-2}  pred(Working)(v1!1)
{-3}  pred(checkSafetyTrue)(v1!1)
{-4}  pred(defaultState(Waiting))(v2!1)
{-5}  pred(setShutdownControl)((# current := v1!1, next := v2!1 #))
{-6}  deviceTemp(v2!1) < alarmMin(v2!1)
  |-------
{1}   shutdownControl(v2!1)
Rule? (lemma "ax_pred_Working")
.....
Rule? (grind)
Trying repeated skolemization, instantiation, and if-lifting,
.....
This completes the proof of System_safety_property.1.
System_safety_property.2 :
.....
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of System_safety_property.24.
Q.E.D.
Run time  = 49.25 secs.
Real time = 1188.31 secs.
```

Figure 30: Snapshot of Interactive proof

Figure 31 shows model checking result for the liveness property using PrUDE, the property has been automatically checked in 38.09 seconds. Without our Boolean

Abstraction scheme (see 4.3.2), it is hard to use model checking on the temperature regulator statechart that actually contains infinite global states.

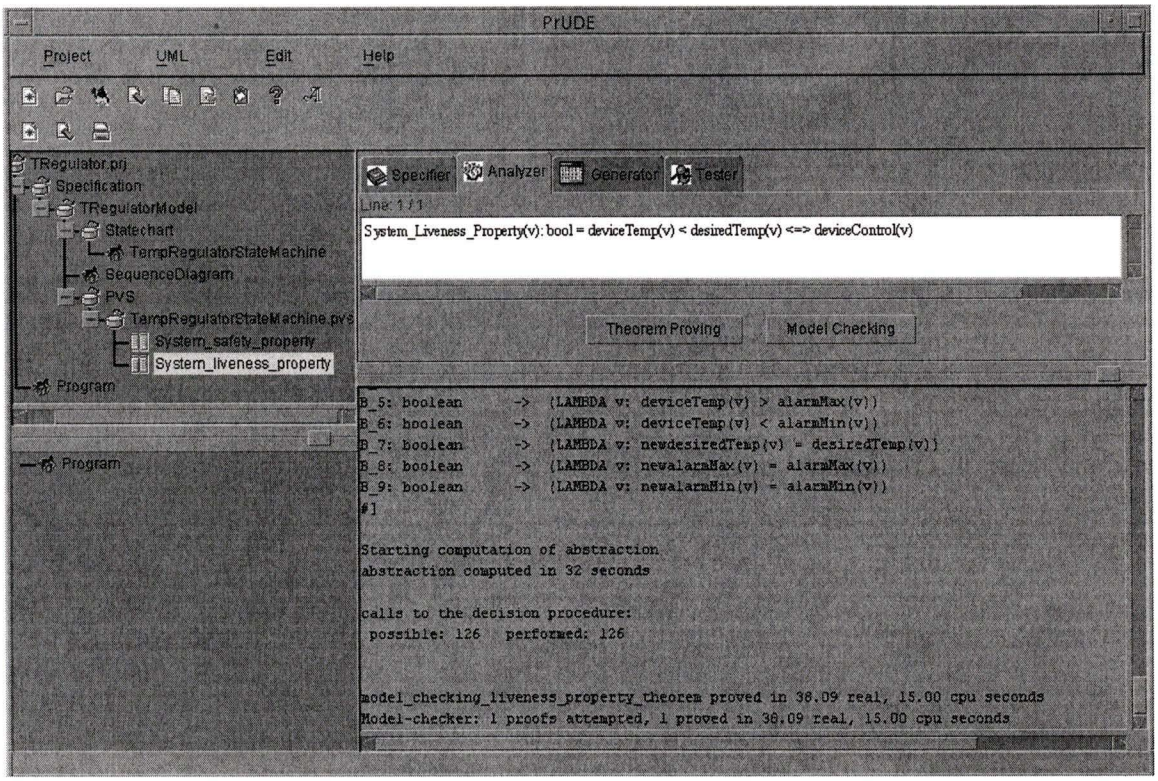

Figure 31: Model Checking Liveness Property

6.2.3.3 Counterexample Analysis

If the PVS proof system fails to prove a property, it will return unproved subgoals which can serve as counterexamples. Since we use the *ConfigurationPair* definition of UML statechart in the actual proof process, the unproved counterexamples actually specify the configuration pairs that violate the specific property, using predicate expressions. Based on the counterexample, it is easy to trace back the errors in the original UML model. For instance, let's consider the following system requirement:

Requirement: *After the temperature regulator detects a dangerous situation and shutdowns the device, the regulator should terminate the regulation process and wait for the operator's instruction.*

Since all the operational parameters of the regulator have to be default values after a *shutdown* operation, the corresponding system property can be defined as follows:

System_Safety_Property(v): bool =
ShutdownControl(v) => (desiredTemp(v)<alarmMax(v) AND
desiredTemp(v)>alarmMin(v) AND
deviceTemp(v)=0 AND
deviceControl(v)=FALSE AND
tempLimit(v)=550)

However, the PVS proof system fails to prove the above property as Figure 32 shows. The PVS proof system is invoked directly in PrUDE in order to show unproved subgoals.

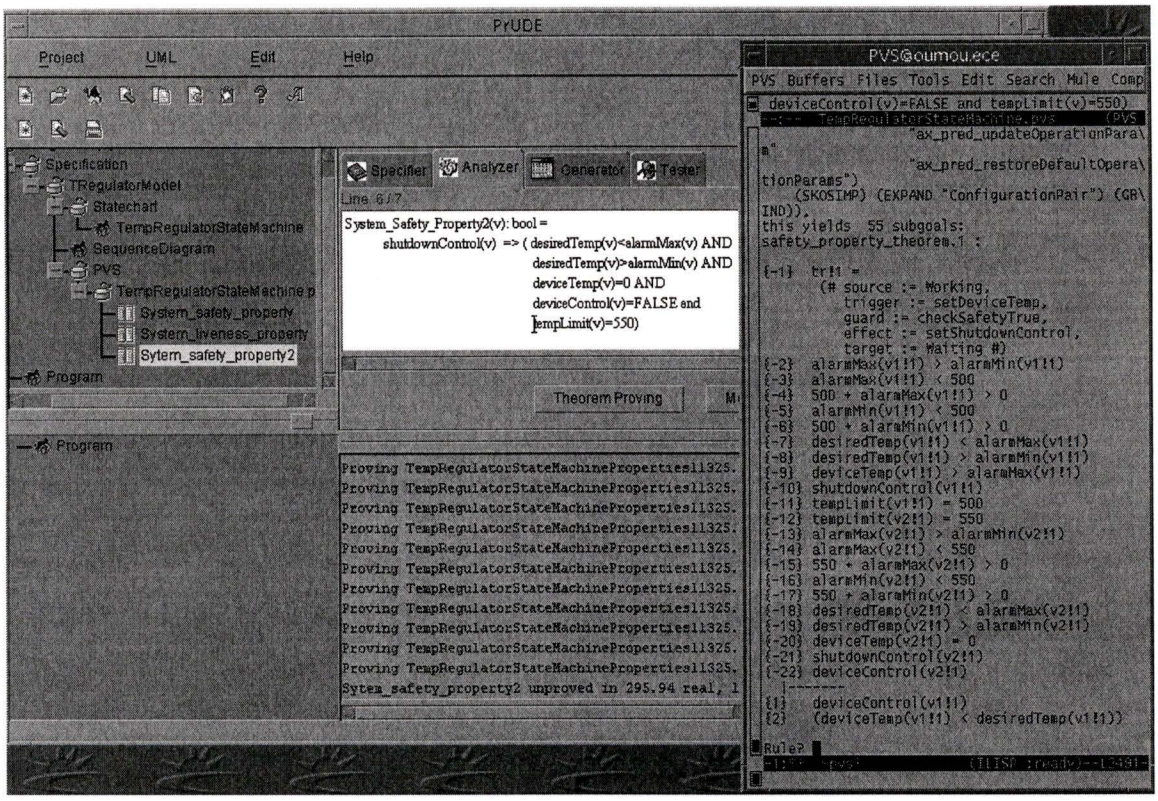

Figure 32: Unsuccessful Proving

Currently, the counterexamples are still expressed in PVS semantic. Users have to analyze the unproved subgoals in order to trace back the errors. Some student in our group is working on PVS-OCL transformation that will help to translate the

counterexamples into more understandable form. The predicates of the unproved subgoal shows that the *deviceTemp* isn't in its default value (zero) in some state when the *shutdownControl* is assigned a *truth* value. Since the device temperature varies only in the *Working* state of the UML statechart, we found that the problem is caused by the transition which performs the *setShutdownControl* action (see Figure 27). This transition makes the system to stay in the *Working* state after an emergency shutdown operation, which is a design error. Thus, the original statechart diagram is redefined as follows:

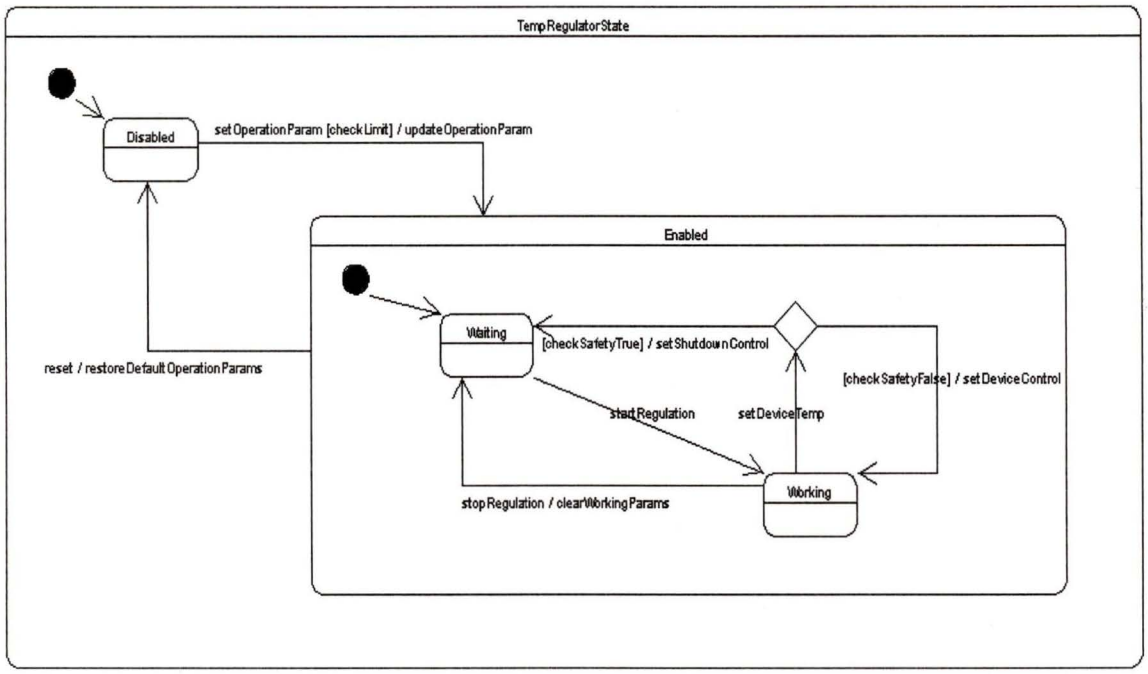

Figure 33: The Modified UML Statechart of Temperature Regulator

The transition, which performs the *setShutdownControl* action, is redirected to the *Waiting* state. The formal model for the updated UML statechart diagram was regenerated and the system properties were proved successfully.

Chapter 7

Related Work

Since UML have become more and more popular in the industry, and UML notations are widely used in systems design, several approaches and mechanisms for UML verification have been proposed in the literature. We provide some of these approaches in this chapter and discuss how they relate to our work.

7.1 Management And Verification Of The Consistency Among UML Models

Atsushi Ohnishi proposed a method for verifying consistency among UML models using a knowledge-based approach [24]. Their method enables efficient verification by setting the order of the verification step derived from the model diagram and by omitting the redundant verification steps.

They propose a verification technique that checks the consistency among six kinds of UML models namely class diagram, statechart diagram, sequence diagram, collaboration diagram, activity diagram, and use case diagram. These six models are mainly used in the analysis phase of software development. They check the correspondence among the models by comparing the names of their defined elements. With such correspondences, the consistency of the models is verified using a knowledge-base (see Figure 34).

	Class diagram(C)	State chart(S)	Sequence chart(Q)	Collaboration diagram(L)	Use case diagram(U)
State chart (S)	1) Class of (S) and classes in (C) 2) confirmation of classes without state charts. 3) Attributes defining states in (S) and attributes in (C) 4) Range of attribute values in (C) and (S) 5) actions, activities in (S) and methods in (C)				
Sequence chart(Q)	1) Objects in (Q) and classes in (C) 2) Messages between objects in (Q) and associations between coressponding classes in (C) 3) Messages in (Q) and methods in (C)	1) Object in (Q) and class of (S) 2) Messages in (Q) and actions, activities in (S)			
Collaboration diagram(L)	1) Objects in (L) and classes in (C) 2) Messages between objects in (L) and associations between corresponding classes in (C) 3) Messages in (L) and method in (C)	1) Objects in (L) and class of (S) 2) Messages in (L) and actions, activities in (S)	1) Objects in (L) and in (Q) 2) Messages in (L) and in (Q) for directions, sequence, source, and destination		
Use case diagram(U)	1) Actors in (U) and classes in (C) 2) Use cases in (U) and methods in (C)	1) Actors in (U) and class of (S) 2) Use cases in (U) and actions, activities in (S)	1) Actors in (U) and objects in (Q) 2)use cases in (U) and messages in (Q)	1) Actors in (U) and objects in (L) 2) Use cases in (U) and messages in (L)	
Activity diagram(A)	1) Classes in (A) and in (C) 2) Actions in (A) and methods in (C) 3) Control flows between classes in (A) and associations in (C)	1) Classes in (A) and class of (S) 2) Actions in (A) and actions, activities in (S)	1) Classes in (A) and objects in (Q) 2) Actions in (A) and messages in (Q) 3) Control flows between classes in (A) and messages in (Q)	1) Classes in (A) and objects in (Q) 2) Actions in (A) and messages in (L) 3) Control flows between classes in (A) and message in (L)	1) Classes in (A) and actores in (U) 2) Actions in (A) and use cases in (U)

Figure 34: Knowledge-based Verification Items Among UML Models

In their research work, they developed an efficient consistency checking method for the identified verification items among UML models. And if no errors are found by their methodology, they guarantee the consistency among the six UML models in the object-oriented analysis.

However, A well-designed model should not only have consistent features, but also satisfy system requirements (expressed by system properties). Unfortunately, their proposed approach focuses only on consistency checking and doesn't deal with property verifications. They provided checking criteria for element correspondence among different UML models; it helps to reduce ambiguities and confusions in understanding the system design, but is not efficient to uncover design deficiencies or flaws in software specifications.

Our research work can be complementary to their consistency checking methodology, we focus on uncovering design deficiencies or errors by verifying system properties against UML models.

7.2 Mapping UML To Abstract State Machine

The Abstract State Machine (ASM) Project (formerly known as the Evolving Algebras Project) was started by Yuri Gurevich as an attempt to bridge the gap between formal models of computation and practical specification methods [25].

The ASM assumption is that any algorithm can be modeled at its natural abstraction level by an appropriate ASM. Based upon this consideration, members of the ASM community have sought to develop a methodology based upon mathematics which would allow algorithms to be modeled naturally, that is, described at their natural abstraction levels. The result is a simple methodology for describing simple abstract machines which correspond to algorithms. The ASM methodology has the following desirable characteristics: presision, faithfulness, understandability, scalability, generality.

A group led by Kevin Compton and Yuri Gurevich is working on using Abstract State Machines to define a semantic model for UML and then use ASM Model Checker to

design a verification tool for UML [4]. They proposed several subtools to implement the verification tool. First, a translation tool reads a UML model and translates it into an ASM model. Then the verification tool, which reads an ASM model and system properties provided by the designer, checks whether the ASM model satisfies the property specification. The last subtool is an analysis tool which is used to analyze the result returned by the model checker tool.

As first step, they defined ASM semantics for UML statechart and collaboration diagram. However, since it is actually an ASM model that is fed to the model checker for verification, system properties, which are drawn from the requirements, have to be expressed by ASM fomulas corresponding to the translated ASM models. And the detected errors returned by ASM model checker have to be matched to the original UML models. In their methodology, it is not mentioned how to achieve automation of the complete verification process. On the other hand, state space explosion, a general problem of model checkers, is not addressed by their methods.

Our methodology contains patterns for both expressing property formulas and proving process, therefore, the whole verification process is automatic. Moreover, we also address the state space explorsion problem for PVS model checker.

7.3 Formalizing UML Using PROMELA

PROMELA (Process Meta Language) [27] is a non-deterministic language, loosely based on Dijkstra's guarded command language notation and borrowing the notation for i/o operations from Hoare's CSP language. PROMELA is the input language of SPIN, which is a model-checker that supports the formal verification of distributed systems

[28]. The SPIN model-checker was developed at Bell Labs in the formal methods and verification group in the 1980s. Spin can be used as a full *LTL (Linear Temporal Logic) model checking* system, which produces a counter-example when an error is found during the verification process.

Researchers in TUCS group (Turku, Finland) have presented a formalization scheme for UML state machines using PROMELA. Their formalization consists of two parts as follows:

1. A formalization of the structure of UML statechart diagram that is both simple and declarative.

2. A formalization of the operational semantics of UML state machines that covers all the features of UML state machines.

The semantics presented in their work has been used to develop the vUML tool [26]. vUML is a tool for model checking UML models. The UML model is first translated into PROMELA, then SPIN is used to verify the generated PROMELA specification. However, vUML still uses the PROMELA language for describing guards and actions since there is no standard syntax for these elements in UML. In order to check the model for violations of the constraints, they have to define LTL formulas for system properties, but how to express a LTL formula in a UML diagram is an open problem. On the other hand, they don't present efficient approaches to reduce the state space for SPIN model checker. Moreover, their representation of counter-examples are not efficient for users to understand the errors and trace back the original UML models.

Our methodology contains automatic proof patterns for both PVS theorem prover and model checker. Proof formulas of system properies are generated systematically. No special formal background is required in our verification process.

7.4 Debugging UML Sequence Diagram and Statechart Diagram Using Model Checking

Maria, Pedro, and Ernesto proposed to apply model checking to compare the behavior described by a UML sequence diagram (or a collaboration diagram) with the behavior expressed by a set of UML state diagrams [44]. The state diagrams are the actual statecharts that describe the dynamic behaviors of all the entities in the sequence diagram. Their analysis is said to ensure the right correspondence between a sequence diagram giving the interaction among a group of objects, and the statecharts exhibiting their individual behavior.

Their methodology consists of three phases, first producing the configurations (global states) of the system by the conjunction of all the separate statecharts; then employing a sequence diagram to describe a potential desirable behavior (system property) of the system and checking the property against the designed statecharts; the third phase is a similar process with the second phase, which verifies the statecharts against non-desirable sequence diagrams. Their model checking tool can translate UML sequence diagram into an automaton to inspect the evolution of the global states previously inspected in the first phase.

Their contributions are that they applied model checking technique to UML design verification, meanwhile, they don't require software developers to have additional knowledge except the standard UML notations.

However, since they focus on the correspondence between a sequence diagram and statechart diagrams, their methodology is a partial verification method for the whole UML specifications. Besides, it is difficult for their method to detect inner design errors of a single object, which makes the methodology less useful for programmers. Moreover, as they mentioned, state space explosion is a serious problem for their methodology.

Our methodology can be used to analyze the behaviour of a system, a sub system, or even a single object. The analysis results are not only helpful to uncover design errors, but also useful for program testers to generate test cases.

7.5 Other Related Work

Other approaches related to UML verification are described respectively in [45], [46], [20]. All these approaches consider model checking in the verification and validation of UML models, and the main differences among them are in the translation scheme and the way of considering the properties to be verified.

Latella et al.'s [45] proposal for using temporal logic to define desirable (undesirable) scenarios also corresponds to partial verification, because a practical temporal formula only represents very specific fragments of the real executions in the system. The work by Mikk et al. [46] also considers temporal logic to represent the properties. These two works employ different semantic for statecharts. The first one describes a new semantic; while the second one uses the statechart semantics implemented by the commercial tools of I-logix. Useing temporal logic outside the UML framework, both of these two methodologies have to let users deal with the verification formulas of system properties.

HUGO [20] also employs Promela and SPIN as the core verification technology, but its users work with UML descriptions. HUGO verifies whether the desirable behavior described by a collaboration diagram is feasible for a set of UML state machines. The verification methodology of HUGO is similar with the approach introduced in section 7.4, thus they have the same limitations.

7.6 Summary

Based on the investigation and evaluation of the existing methodologies on UML verification, it is clear that UML models, due to their informal nature, require a formal transformation before being suitable for rigorous verification and validation. A form of basic consistency checking among UML models may be achieved directly on the diagram using the informal semantics and well-formedness rules. However, to check whether a UML model satisfies original customer expectation is impossible without defining additional constraints or assertions and a formal semantic ground.

Chapter 8

Conclusion And Further Work

8.1 Conclusion

The formalization of object-oriented analysis and design modeling languages provides a means to allow rigorous analysis, software comprehension and guarantee consistency in all software development phases. The rigor imposed by formalization also supports early detection of errors and bugs in the development process avoiding the overhead and cost of fixing them during the implementation or operational stages.

Even though the UML has been adopted as a standard object oriented modeling language for analysis and design, it is not yet formalized, thus it is difficult to rigorously verify UML models. On the other hand, the esoteric nature of formal methods and the lack of industrial-strength tool support impose significant barriers to their widespread penetration. The work presented in this thesis is part of work to develop a pragmatic framework where rigorous verification of UML models is made possible by using the power of formal methods.

More specifically, we focus here on the automation of UML model verification process using formal techniques. An automated formal analysis framework based on some identified proof patterns has been proposed in this thesis, and a corresponding implementation in the PrUDE toolkit has been developed.

The main contribution of this thesis is to present an abstract pattern for the UML statechart and propose general, yet efficient methodologies to verify UML statechart

using PVS theorem prover and model checker. Our verification process is completely automatic; no human interactions are needed during the proof checking process. Once the verification is triggered, users either receive successful notice or get counterexamples. Our verification methodologies can be applied to both infinite and finite state systems. The state space explosion problem caused by state variables are well addressed for model checking in our methodology.

The automated formal analysis framework proposed plays an important role in software engineering and reengineering environments in the sense that it can help in uncovering design errors at the specification level.

8.2 Further Work

In this thesis, we have presented a verification methodology for the UML statechart diagram and implemented it in PrUDE. In the future, we need to extend our framework to include more V&V strategies or approaches for other UML diagrams. Firstly, we need to provide complete formal semantics for the various UML models, and then develop the corresponding proof methods. Since we expect to hide formal aspects at the back-end in our methodology, achieving automation in the verification process is the main objective. In this respect, some foundation work has been proposed, for instance, the PVS formal semantics for the UML sequence and class diagram have been defined in [7] and [9] respectively.

One of the main limits of our current framework is that system complementary semantics and properties are still expressed using esoteric notation (e.g. OCL or PVS). We are investigating possible graphical notations that can be used to express system assertions

graphically in a user friendly way. Some candidate notations are being considered, for instance, the tabular notation in [37].

Another direction for future research is to explore ways to present the system-level analysis and simulation results to the user. Since the methodology calls for a UML user to provide the input specifications, it is only reasonable for the output results to be in a form that is meaningful to that user. Thus, we will investigate ways to map from the PVS analysis results back to the UML specifications.

Bibliography

[1] I. Sommerville, *Software Engineering*, Pearson Education, Harlow, England, 2001.

[2] B. Boehm, *Industrial Software Metrics Top 10 List*, IEEE Software, 4(5): 84 - 85, September 1987.

[3] G. Booch, J. Rumbaugh, I. Jacobson, *The Unified Modeling Language Reference Manual*, Rational Software Corporation, 1999, Addison Wesley Longman, Inc.

[4] K. Compton, Y. Gurevich, J. Huggings, W. Shen, *An Automatic Verification Tool For UML*, Technical Report, Feb. 25th, 2000.

[5] R. Kneuper, *Limits of Formal Methods*, Formal Aspects of Computing, 9:379-394, 1997

[6] J. B. warmer, A. G. Kleppe, *The Object Constraint Language: Precise Modeling With UML*, Addsion Wesley Longman Inc., Reading Massachusetts 01867, 1999.

[7] D. B. Aredo, I. Traore, K. Stolen, *An Outline of PVS Semantics for UML Class Diagram*, NWPT'99, Nordic Workshop on Programming Theory, Oct. 6-8, 1999, Upsala, Swedden.

[8] I. Traore, *An Outline of PVS Semantics for UML Statecharts*, Journal of Universal Computer Science (JUCS), Springer Pub. Co., November 2000.

[9] D. B. Aredo, *Semantics of UML Sequence Diagram in PVS*, Proceedings of the Workshop on Dynamic Behavior in UML Models, at UML2000, October 2-6, 2000, York, UK.

[10] I. Traore, *An Integrated V&V Environment for Critical System Development*, Proceeding Of 5th IEEE International Symposium on Requirements Engineering, August 2001, Toronto, Canada.

[11] Booch, *Booch Diagram*, Tutorial, http://www.smartdraw.com/software/booch.htm.

[12] Rumbaugh, *Rumbaugh Diagram*, Tutorial http://www.smartdraw.com/software/omt.htm.

[13] Jacobson, *Jacobson's OOSE Diagram*, Tutorial http://www.smartdraw.com/software/oose.htm.

[14] J. Crow, S. Owre, J. Rushby, N. Shankar, M. Srivas, *A Tutorial Introduction to PVS*, WIFT's95, Boca Raton, Florida, April 1995.

[15] S. Owre, N. Shankar, J.M.Rushby, *PVS Language Reference*, tutorial, http://pvs.csl.sri.com, September 1999.

[16] D. MacKenzie, *The Automation of Proof: A Historical and Sociological Exploration"*, Annals of the History of Computing, 1995.

[17] S. Owre, N. Shankar, J. M. Rushby, *PVS Prover Guide*, tutorial, http://pvs.csl.sri.com/, September 1999.

[18] E. M. Clarke, O. Grumberg, D. A. Peled, *Model Checking*, ISBN 0-262-03270- 8, 1999.

[19] N. Shankar, *Machine-Assisted Verification Using Theorem Proving and Model Checking*, Mathematical programming Methodology, M. Broy, Springer-Verlag, 1997.

[20] Schiafer T., Knapp A., Merz S., *Model Checking UML State Machines and Collaborations*, Electronic Notes in Theoretical Computer Science 55(3): 19-24, 2001.

[21] F. Dederichs, R. Weber, *Safety and Liveness from a Methodological Point of View*, Information Processing Letters, 36(1): 25–30, October 1990.

[22] R. V. Binder, *Testing Object-oriented System: Models, Patterns and Tools*, Reading, MA: Addition-Wesley Longman, 1999.

[23] J. Flynt, J. Maudlin, *An Example Specification for Implementing A Temperature Regulator Software Component*, in Component-based Software Engineering-Putting the Pieces Together, (eds.) G.T. Heineman and W.T. Council.

[24] A. Ohnishi, *Management And Verification Of The Consistency Among UML Models*, Technical Report, Department of Computer Science, Ritsumeikan University, Shiga 525-8577, Japan

[25] Y. Gurevich, *Specification and Validation Methods*, Evolving Algebras 1993: Lipari Guide, Oxford University Press, 1995.

[26] J. Lilius, I. P. Paltor, *VUML: a Tool For Verifying UML Models*, TUCS Technical Report NO. 272, May 1999.

[27] G. J. Holzmann, *Design and Validation of Computer Protocols*, Prentice Hall, 1991.

[28] G. J. Holzmann, *The Model Checker SPIN*, IEEE Transactions on Software Engineering, VOL. 23, NO. 5, May1997.

[29] Zs. Pap, I. Majzik1, A. Pataricza, A. Szegi, *Completeness and Consistency Analysis of UML Statechart Specifications*, Dept. of Measurement and Information Systems, Budapest University of Technology and Economics.

[30] R. France, A. Evans, K. Lano, B. Rumpe, *The UML As A Formal Modeling Notation*, OOPSLA'97 Workshop on Object-oriented Behavioral Semantics, p. 75-81. Atlanta, Georgia, USA, October 1997.

[31] IBM, *XML Parser for JAVA*, Tutorial, http://www.alphaworks.ibm.com/tech/xml4j.

[32] N. Leveson, *Center for Software Engineering Research: A White Paper,* http://sunnyday.mit.edu/research.html.

[33] J. L. Sourrouille, G. Caplat, *Constraint Checking in UML Modeling,* INSA – Dept IF. Bat. Blaise Pascal - 69621 Villeurbanne Cedex, France.

[34] I. Traore, D. B. Aredo, *Integrating Formal V&V and Structured Design Reviews*, Workshop on Inspections in Software Engineering (WISE'01), CAV'01 Conference, 23rd July 2001, Paris, France, pp 30-39.

[35] S. Graf, H. Saidi, *Construction of Abstract State Graphs with PVS*, Conference on Computer Aided Verification CAV'97, LNCS 1254, Springer Verlag, 1997.

[36] I. Traore, *A Framework for Rigorous Testing of Object-oriented Programs*, IEEE ECBS Conference, Workshop on Formal Specification of Computer-Based Systems (FSBCS), April 20th 2001, Washington D.C., USA, pp. 33-39.

[37] D.L. Parnas, *Tabular Representation of Relations*, CRL Report 260, McMaster University, Canada, February 1992.

[38] A. Moreira, R. Clark, *Combining Object-Oriented Analysis and Formal Description Techniques*, Proc. of ECCOP'94, LNCS, Bologna, Italy, 1994, vol.821, Springer, Verlag.

[39] R. E. Bryant, *Graph-based Algorithms for Boolean Function Manipulation*, IEEE Transactions on Computers C-35 (8): 667-691.

[40] P. Godefroid, D. Pirottin, *Refining Dependencies Improves Partial-order Verification Methods*, Proceedings of the 5th Conference on Computer-Aided Verification, LNCS, 697, pp.443-449, Springer, 1993.

[41] S. Bensalem, A. Bouajjani, C. Loiseaux, Sifakis, *Property Preserving Simulations*, in Von Bochmann and Probst [243], pp. 260-273.

[42] E. M. Clark, T. Filkorn, S. Jha, *Exploring Symmetry in Temporal Logic Model Checking*, in Courcoubetis [83], pp. 450-462.

[43] M. C. Browne, E. M. Clarke, O. Grumberg, *Reasoning About Networks with Many Identical Finite-state Processes*, Information and Computation 81(1): 13-31.

[44] M. Gallardo, P. Merino, E. Pimentel, *Debugging UML Designs with Model Checking*, in Journal of Object Technology, vol. 1, no. 2, July-August 2002, pp. 101-117.

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

[46] Mikk E., Lakhnech Y., Siegel M., Holzmann G., *Implementing Statecharts in Promela/SPIN*, Proceedings of Workshop on Industrial strength Formal Specification Techniques (WIFT'98), 1998.

Appendix A

The PVS Semantic of The Temperature Regulator Statechart

```
TempRegulatorStateMachine: THEORY

BEGIN
 %Import theory AbstractSyntax from the library
 prudelib: LIBRARY = "~/prude/semantic/lib/prudelib"
 IMPORTING prudelib@AbstractSyntax

 %Instance variables
 V: TYPE = [# desiredTemp : int,
              alarmMax : int,
              alarmMin : int,
              tempLimit : int,
              deviceTemp : int,
              deviceControl : bool,
              shutdownControl : bool,
           #]
  v, v1, v2: VAR V
  x: VAR Vertex
  defaultState(x): Vertex

 %StubState definition
 SmStub: finite_set[Vertex] = emptyset
 %IntialState definition
 initial2: Vertex
 initial3: Vertex
 SmInitial: finite_set[Vertex] = add(initial2, add(initial3,emptyset))
 %FinalState definition
 SmFinal: finite_set[Vertex] = emptyset
 %Join definition
 SmJoin: finite_set[Vertex] = emptyset
 %Fork definition
 SmFork: finite_set[Vertex] = emptyset
 %Branch definition
 branch3: Vertex
 SmBranch: finite_set[Vertex] = add(branch3, emptyset)
 %shallowHistory definition
 SmShallowHistory: finite_set[Vertex] = emptyset
 %deepHistory definition
 SmDeepHistory: finite_set[Vertex] = emptyset
```

```
%Junction definition
SmJunction: finite_set[Vertex] = emptyset
%SynchState definition
SmSynch: finite_set[Vertex] = emptyset
%State definitions
Working: Vertex

ax_Working_properties: AXIOM dsubvertex(Working) = emptyset AND
                                simpleState?(Working) AND
                                defaultState(Working)=Working
Waiting: Vertex
ax_Waiting_properties: AXIOM dsubvertex(Waiting) = emptyset AND
                                simpleState?(Waiting) AND
                                defaultState(Waiting)=Waiting
Disabled: Vertex
ax_Disabled_properties: AXIOM dsubvertex(Disabled) = emptyset AND
                                simpleState?(Disabled) AND
                                defaultState(Disabled)=Disabled

%Composite State definitions
Enabled: Vertex
 ax_Enabled_properties: AXIOM dsubvertex(Enabled) = {s: Vertex | s=initial3 OR
                                s=Waiting OR s=Working OR s=branch3 } AND
                                (NOT simpleState?(Enabled)) AND
                                isSequential(Enabled)  AND
                                defaultState(Enabled)=Waiting
 TempRegulatorState: Vertex
 ax_TempRegulatorState_properties: AXIOM
                                dsubvertex(TempRegulatorState) = {s:  Vertex
                                s=initial2 OR s=Disabled OR s=Enabled} AND
                                (NOT simpleState?(TempRegulatorState)) AND
                                isSequential(TempRegulatorState) AND
                                defaultState( TempRegulatorState)=Disabled

SmState: finite_set[Vertex] = add(TempRegulatorState, add(Enabled, add(Working,
                                add(Waiting, add(Disabled, emptyset)))))

%Root state definition
SmRoot: Vertex = TempRegulatorState

%Submachine State definition
SmSubmachine: finite_set[Vertex] = emptyset

%SignalEvent definitions
stopRegulation: Event
startRegulation: Event
```

```
setDeviceTemp: Event
setOperationParam: Event
reset: Event
SmSignalEvent: finite_set[Event] = add(stopRegulation, add(startRegulation,
                add(setDeviceTemp, add(setOperationParam, add(reset,emptyset)))))

%CallEvent definitions
SmCallEvent: finite_set[Event] = emptyset

%ChangeEvent definitions
SmChangeEvent: finite_set[Event] = emptyset

%TimeEvent definitions
SmTimeEvent: finite_set[Event]= emptyset

%Action definitions
setShutdownControl: Action
setDeviceControl: Action
clearWorkingParams: Action
updateOperationParam: Action
restoreDefaultOperationParams: Action
SmAction: finite_set[Action] = add(setShutdownControl, add(setDeviceControl,
                               add(clearWorkingParams, add(updateOperationParam,
                               add(restoreDefaultOperationParams, emptyset)))))

%Guard definitions
checkSafetyTrue: Condition
checkSafetyFalse: Condition
checkLimit: Condition
        SmGuard: finite_set[Condition] = add(checkSafetyTrue, add(checkSafetyFalse,
            add(checkLimit, emptyset)))

%Transition definitions
t6: Transition=(#source:=Working, trigger:=setDeviceTemp, guard:=checkSafetyTrue,
                                    effect:=setShutdownControl, target:=Waiting#)

t5: Transition=(#source:=Working, trigger:=setDeviceTemp, guard:=checkSafetyFalse,
                                    effect:=setDeviceControl, target:=Working#)

t3: Transition=(#source:=Working, trigger:=stopRegulation, guard:=EmptyC,
                                    effect:=clearWorkingParams, target:=Waiting#)

t2: Transition=(#source:=Waiting, trigger:=startRegulation, guard:=EmptyC,
                                    effect:=EmptyA, target:=Working#)
```

```
t1: Transition=(#source:=Disabled, trigger:=setOperationParam, guard:=checkLimit,
                                        effect:=updateOperationParam, target:=Enabled#)

t8: Transition=(#source:=initial2, trigger:=EmptyE, guard:=EmptyC, effect:=EmptyA,
                                        target:=Disabled#)

t7: Transition=(#source:=Enabled, trigger:=reset, guard:=EmptyC,
                                effect:=restoreDefaultOperationParams,target:=Disabled#)

SmTransition: finite_set[Transition] = add(t6, add(t5, add(t3, add(t2,add(t1, add(t8,
                                        add(t7, emptyset)))))))

%Context definitions
 SmContext: Context = behavioral

%State machine definition
sm: StateMachine =
   (#State     := SmState,
    StubState  := SmStub,
    SynchState := SmSynch,
    Initial    := SmInitial,
    Choice     := SmBranch,
    DeepH      := SmDeepHistory,
    ShallowH   := SmShallowHistory,
    Join       := SmJoin,
    Fork       := SmFork,
    Junction   := SmJunction,
    FinalState := SmFinal,
    SubmachineState := SmSubmachine,
            PseudoState:= union(SmDeepHistory, union(SmShallowHistory,
                    union(SmBranch, union(SmJoin, union(SmFork, union(SmJunction,
                    SmInitial))))))),
    Vertex     := union(PseudoState, union(SmStub, union(SmSynch, SmState))),
    CallEvent  := SmCallEvent,
    TimeEvent  := SmTimeEvent,
    ChangeEvent:= SmChangeEvent,
    SignalEvent:= SmSignalEvent,
    Event      := union(SmCallEvent, union(SmTimeEvent, union(SmChangeEvent,
                        SmSignalEvent))),
    Action     := SmAction,
    Condition  := SmGuard,
    Transition := SmTransition,
    Root       := SmRoot,
    Context    := SmContext
   #)
```

```
%import theory WellFormedness and FormalSemantics from the library

IMPORTING prudelib@WellFormedness[sm]
IMPORTING prudelib@FormalSemantics[sm,V]
IMPORTING prudelib@state[V]

%predicate definitions
vc: VAR VC

%simple state

% In the Working state, all the operational parameters must hold and be valid.

ax_pred_Working: AXIOM pred(Working) =
        (LAMBDA v:  alarmMax(v) >alarmMin(v) AND
                    alarmMax(v) <tempLimit(v) AND
                    desiredTemp(v)<alarmMax(v) AND
                    desiredTemp(v) > alarmMin(v) AND
                    tempLimit(v)=500)  AND
                    deviceTemp(v)>alarmMax(v) AND
                    deviceTemp(v)<alarmMin(v))

% In the Waiting state, all the operational parameters must have been initialized and
validated, therefore those parameter values must be valid. Meanwhile, all the detecting
values (deviceTemp) and output signals (deviceControl, deviceShutdown) must be in
default values.

ax_pred_Waiting: AXIOM pred(Waiting) =
        (LAMBDA v: tempLimit(v)=550 AND
                    alarmMax(v) >alarmMin(v) AND
                    alarmMax(v) <tempLimit(v) AND
                    desiredTemp(v)<alarmMax(v) AND
                    desiredTemp(v) > alarmMin(v) AND
                    deviceTemp(v)=0 AND
                    shutdownControl(v)=FALSE
                    deviceControl(v)=FALSE )

% In the Disabled state, all the state variables must be in the default values that designer
defined. In this example, we define the default values as above.

ax_pred_Disabled: AXIOM pred(Disabled) =
        (LAMBDA v: desiredTemp(v) =0 AND
                    alarmMax(v) = 0 AND
                    alarmMin(v) = 0 AND
                    tempLimit (v)= 500 AND
                    deviceTemp(v) =0 AND
```

```
                    deviceControl (v)= FALSE AND
                    shutdownControl(v) = FALSE )
```

%Guard conditions

% CheckSafetyTrue specifies whether the detected device temperature or probe temperature is out of the safe range that user defined.

```
ax_pred_checkSafetyTrue: AXIOM pred(checkSafetyTrue) =
        (LAMBDA v: deviceTemp(v) > alarmMax(v) OR
                    deviceTemp(v) < alarmMin(v)   )
```

% CheckSafetyFalse specifies whether the detected device temperature or probe temperature is in the safe range that user defined.

```
ax_pred_checkSafetyFalse: AXIOM pred(checkSafetyFalse) =
        (LAMBDA v: deviceTemp(v) < alarmMax(v) AND
                    deviceTemp(v) > alarmMin(v) )
```

% *CheckLimit* is to make sure that all the user input values are within the range of system temperature limitation.

```
ax_pred_checkLimit: AXIOM pred(checkLimit) =
        (LAMBDA v: desiredTemp(v) < alarmMax(v) AND
                    desiredTemp(v) > alarmMin(v) AND
                    alarmMax(v) <tempLimit(v) )
```

%Actions

% SetShutdowControl action just sets output signal – deviceShutdown to true, so that the monitored device is safely terminated.

```
ax_pred_setShutdownControl: AXIOM pred(setShutdownControl) =
        (LAMBDA vc: shutdownControl(next(vc)) = TRUE  )
```

% SetDeviceControl action sets the output signal-deviceControl by comparing detected device temperature to desired temperature. If deviceTemp<desiredTemp, the deviceControl is set to true and the device heats up. Otherwise, deviceControl is set to false and the device cools down.

```
ax_pred_setDeviceControl: AXIOM pred(setDeviceControl) =
        (LAMBDA vc: IF(deviceTemp(current(vc)) >= desiredTemp(current(vc)))
                            DeviceControl(next(vc)) = false
                    THEN
                            DeviceControl(next(vc)) = true
```

```
                    END IF)
```

% ClearWorkingParam action restores the default working variables(deviceTemp, deviceControl, deviceShutdown).

```
ax_pred_clearWorkingParams: AXIOM pred(clearWorkingParams) =
        (LAMBDA vc: deviceTemp(next(vc)) = 0 AND
                    deviceControl(next(vc)) = FALSE AND
                    shutdownControl(next(vc)) = FALSE   )
```

% RestoreDefaultOperationParam action restores the operational parameters to their default values.

```
ax_pred_restoreDefaultOperationParams: AXIOM
        pred(restoreDefaultOperationParams)=
        (LAMBDA vc: desiredTemp(next(vc)) = 0 AND
                    alarmMax(next(vc)) = 0 AND
                    alarmMin(next(vc)) = 0 AND
                    deviceTemp(next(vc)) =0 AND
                    deviceControl(next(vc)) = FALSE AND
                     shutdownControl(next(vc)) = FALSE   )
```

% Computation path definition

```
ConfigurationPair(v1:V, v2:V): bool =
        EXISTS (tr:{tr: Transition | SmTransition(tr) AND IsMeaningful(tr)}) :
                    (pred(source(tr))(v1) AND
                    pred(container(source(tr)))(v1) AND
                    pred(guard(tr))(v1) AND
                    pred(container(target(tr)))(v2) AND
                    DefaultConfigurationPredicate(target(tr))(v2) AND
                    (pred(effect(tr))(vc) WHERE vc=(#current:=v1, next:=v2#)))
```

%Properties to be checked

% If the detected device temperature is outside the range determined by safety parameters, then the device should be shutdown.

```
System_Safety_Property(v): bool =
        (deviceTemp(v) < alarmMin(v) OR deviceTemp(v) > alarmMax(v))
                => shutdownControl(v)
```

% When the temperature regulator is in working situation, the detected device temperature will vary around the desired temperature. Accordingly, two situations should eventually happen; one is that the device heats up when the detected temperature is less

than the desired temperature, the other is that the device cools down when the detected temperature is greater than the desired temperature.

```
System_Liveness_Property(v): bool =
        deviceTemp(v) < desiredTemp(v)  <=>  deviceControl(v)
```

% After the temperature regulator detects a dangerous situation and shutdowns the device, the regulator should terminate the regulation process and wait for the operator's instruction.

```
System_Safety_Property2(v): bool =
        ShutdownControl(v) => ( desiredTemp(v)<alarmMax(v) AND
                                desiredTemp(v)>alarmMin(v) AND
                                deviceTemp(v)=0 AND
                                deviceControl(v)=FALSE AND
                                tempLimit(v)=550)

END TempRegulatorStateMachine
```

VITA

Surname: Liu Given Name: Yanguo

Place of Birth: Liaoning, China Date of Birth: November 04, 1976

Educational Institutions Attended:

University of Victoria	2001 to 2002
Malaspina University and College	1999 to 2001
Harbin Institute of Technology	1995 to 1999

Degree Awarded:

B. Eng.	Harbin Institute of Technology	1999

Honors and Awards:

Excellent International Student Scholarship	2000
Excellent Academic Achievement Scholarship	2000
HIT Excellent Student Scholarship	1998

Publications:

M. Y. Liu, I. Traore, *PVS Proof-Patterns for UML-Based Verification*, IEEE ECBS Conference, Workshop on Formal Specification of Computer-Based Systems (FSBCS), April 8-11 2002, Lund, Swedden, pp 9-19.

I. Traore, M. Y. Liu, *UML-Based Verification and Testing of a Temperature Regulator Software Component*, submitted to Journal of Requirements Engineering (Springer), Nov.2001.

M.Y. Liu, H. Ye, I. Traore, *Using Formal Methods in Security Engineering: Case Study of a Patient Document Service*, Technical Report no. ECE01-3, Department of Electrical and Computer Engineering, University of Victoria, May 2001.

A. Hoole(1), I. Traore(2), M. Y. Liu(2), *Formal Analysis of an Agent-Based Medical Diagnosis Confirmation System*, The Second Goddard IEEE Workshop on Formal Approaches to Agent-Based System (FAABS II), October 28-30, 2002, Greenbelt, Maryland, USA.

PARTIAL CORYRIGHT LICENSE

I hereby grant the right to lend my thesis to users of the University of Victoria Library, and to make single copies only for such users or in response to a request from the library of any other university, or similar institution, on its behalf or for one of its users. I further agree that permission for extensive copying of this thesis for scholarly purpose may be granted by me or a member of the University designated by me. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Title of Thesis:

PVS Proof Patterns For UML-Based Verification

Author 

Yanguo Liu

August 9th, 2002