XFM: An Incremental Methodology for Developing Formal Models

Syed Mohammed Suhaib

Thesis submitted to the Faculty of
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Engineering

Sandeep K. Shukla, Chair
Dong Ha, Member
Binoy Ravindran, Member

May 6, 2004
Blacksburg, Virginia

Keywords: Extreme Programming, Extreme Modeling, Property Ordering, Formal Verification, Property Refactoring, Prescriptive Formal Models, SPIN, SMV, Embedded Systems

XFM: An Incremental Methodology for Developing Formal Models

Syed Mohammed Suhaib

Abstract

We present a methodology of an agile formal method named eXtreme Formal Modeling (XFM) recently developed by us, based on Extreme Programming concepts to construct abstract models from a natural language specification of a complex system. In particular, we focus on Prescriptive Formal Models (PFMs) that capture the specification of the system under design in a mathematically precise manner. Such models can be used as golden reference models for formal verification, test generation, etc. This methodology for incrementally building PFMs work by adding user stories (expressed as LTL formulae) gleaned from the natural language specifications, one by one, into the model. XFM builds the models, retaining correctness with respect to incrementally added properties by regressively model checking all the LTL properties captured theretofore in the model. We illustrate XFM with a graded set of examples including a traffic light controller, a DLX pipeline and a Smart Building control system. To make the regressive model checking steps feasible with current model checking tools, we need to keep the model size increments under control. We therefore analyze the effects of ordering LTL properties in XFM. We compare three different property-ordering methodologies: 'arbitrary ordering', 'property based ordering' and 'predicate based ordering'. We experiment on the models of the ISA bus monitor and the arbitration phase of the Pentium Pro bus. We experimentally show and mathematically reason that predicate based ordering is the best among these orderings. Finally, we present a GUI based toolbox for users to build PFMs using XFM.

Acknowledgements

I am extremely grateful to my advisor, Dr. Sandeep K. Shukla for advising, supporting and motivating me throughout this work and for keeping me focussed in my research. We acknowledge the support of NSF CAREER grant CCR-0237947 which provided the funding for the work reported in this Thesis. I would also like to thank Dr. Binoy Ravindran for supporting me for a part of my Master's degree.

I would like to acknowledge my roommate and colleague Debayan for motivation and support, and also for introducing me to the "murky" side. I would like to thank Hiren for motivating me to write better. I also like to acknowledge David, Deepak and Abhijeet for their help and support in the XFM development.

I would like to thank my friends Habeeb, Animesh, Madhup, Shekhar and Gaurav for their support. Finally, I would also like to thank Ahmed uncle and family, and Ambareen and family for motivating me to work hard for my Master's degree.

Dedication

I dedicate this thesis to ...

Amma (Mrs. Rana Ashraf)

Abba (Mr. Syed Mohammed Ashraf)

Ayya (Mrs. Tarana Mariam)

and

Ahad Bhai (Mr. Abdul Ahad)

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Computational systems, consumer electronics, avionics and other mission critical systems are dependent on complex hardware and software components. Most often, these systems entail a complexity beyond the scope of ordinary validation techniques. Formal verification and formal methods for test generation, etc. have been emerging as viable techniques for mitigating this increasing system complexity and the resulting validation challenges.

Formal Methods include describing the behaviors of systems mathematically and reasoning about them to prove correctness and analyze their behavior and performance. The key aspects of formal methods include specification, verification and testing techniques for enhancing the quality of the software and hardware development. It is known from industrial trends that the validation cycle is often the limiting factor for the decrease in time-to-market. Some industry experts estimate that about 70% of the design cycle is spent on verification.

However, formal methods often themselves are complex, difficult to use, and require

mathematical sophistication. To make formal methods available to regular engineers, one has to build methodologies and tool sets that enable the engineers to easily utilize the effectiveness of formal methods without being thwarted by the complexity of the method itself.

In this thesis, we present a formal model building methodology based on the principles and philosophy of *Extreme Programming* [6, 65] methodology in software engineering. We believe that one of the *major gaps between formal systems engineering and the requirement specification is the translation from natural language specification to formal models.* Most design teams start with a requirements document written in a natural language and capture the main functionalities, interpret them, which often lead to subtle functional bugs in the resulting product. Therefore, there is a need for a golden reference model in a formal framework that is not only correct with respect to the intent of the required product, but also unambiguous through the rigorous semantics of the formal language which it is coded in. It is not very easy to build such golden reference model because there is no specific methodology either in academia or in the industry that prescribes the steps for building such models. In most cases, such models are never built or are built using ad hoc methods.

Our proposal to use a modeling methodology, which we call *Extreme Formal Modeling* (XFM) is intended to change this practice and to lead designers to adopt this methodology to build these golden models. We call such models *Prescriptive Formal Models* (PFM). During its incremental building, model is regressively verified. This model helps in creating test-benches, validating implementation, create coverage monitors etc.

However, there are intricacies in this methodology, especially related to the order in which the model is incrementally enhanced with newer features. We address this problem by building specific heuristics and their theoretical justifications.

1.1 Prescriptive vs Descriptive formal models

In the industry, building formal models for verification purposes has been used in two different usage modes: Descriptive formal models (DFM) [22] and Prescriptive formal models (PFM). Descriptive formal models are used to capture an implementation in an abstract model to submit to analysis by model checking tools. Since, DFMs are built from the implementation, there is a high possibility that some of the vital implementation bugs are left out unless automatically abstracted from the implementation. Another drawback with these kinds of models is that they may include unwanted complexity and irrelevant behaviors.

On the other hand, Prescriptive Formal Models (PFM) [56, 28, 29] are used to capture natural language specifications in a formal model to analyze consistency of the specification. In this approach of model building, the *intended behavior* of the system is described and not what a specific implementation does. These specifications describe how the system should work. They are also used as a reference model to compare a DFM against it. Since these types of models are built from specification, unwanted complexity and unnecessary details that are present in the implementation but not in the specification are omitted.

1.2 Industrial Trends

The design productivity has not kept up with the increase in complexity of computational systems as well as the increase in the size and complexity of the circuit designs. Although one of the key steps in the design cycle is *verification*, progress has been slow in improving formal verification methodology in the industry. Most of the progress

reported in the literature deal with enhancement of the verification engines, but not so much in verification methodologies. In this thesis, our focus is on a specific methodology.

For efficient verification, the model of the system has to be correctly specified. In the industry, most of the verification is done on DFMs, which are abstract model of the implementation of the system. We believe that a more top-down approach by first building PFMs is more likely to succeed.

Whenever PFMs are built and verified, they are done in an ad hoc manner. Figure 1.1 shows that an ad hoc abstract model is usually built from an English specification and checked against formal properties with a model-checker. At times, the ad hoc abstraction is built from an implementation, which is then checked against the implementation for conformance. There are several drawbacks in this approach. First, the ad hoc building of both the model and the properties is error prone and the effort of model building and debugging grows along with the size of the model. Next, as there is no way to control the inclusion of all properties, some may be overlooked, thus reducing the significance of the model. Then, if a property fails, it is tedious to debug the model. Few indications exist where the bug is located. Finally, there is a tendency that the model will include more behavior than the specification will allow. This is because often implementation bias gets into the abstract model. In addition, implementation details in the abstract model may introduce unwanted complexity and may later cause problems in a conformance check. In [28] the specification language ESL is described in which all properties are specified together, and then an automaton is synthesized from the complete ESL specification. This wholesome approach often has the problem that (i) the inconsistency in the properties or mistakes in capturing the intended property are found late, (ii) the synthesis of automata may explode in size when everything is considered together.

Usually the golden reference model is not built or even if they are built, they are usually

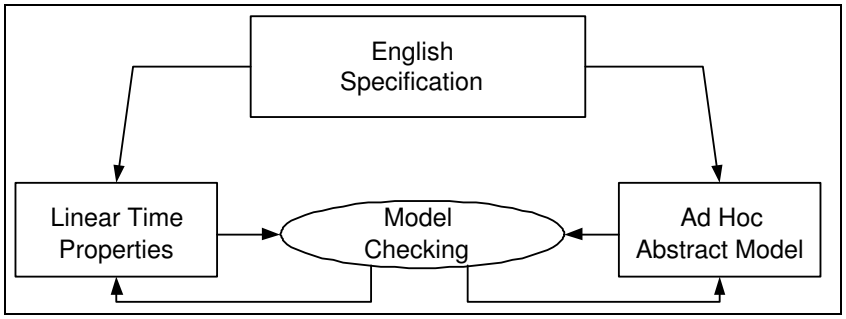

Figure 1.1: Current practice in capturing a formal model from natural language specification

in a high level programming language such as C/C++, which of course does not have a well-defined semantics. We believe building PFMs in a verifiable language would change the industrial practices.

1.3 Motivation

In our experience in developing formal models for systems such as realtime meta-scheduler, EDF scheduling algorithm, etc [44], we felt that the design preceded the modeling, which then makes it difficult to abstract the design into a formal model. Also going directly into an implementation, which is heavy with lots of implementation artifacts, the properties intended for the design often get hidden. So we advocate "*formally model first and design later*" approach.

Our methodological motivation is derived from *Extreme Programming* [6] where "test-first" approach is recommended in building designs. In the "test-first" approach, the customer gives a set of *user stories* which are converted to tests. The user stories are short descriptions that convey the exact detail of the required functionality of the design, which

enables the programmers to be certain about the features that the customers request. By creating tests first, an urge arises to test everything that is valuable to the customer. We first create one test to define some small aspect of the problem. Then we create the simplest code that can make that test pass. We then create a second test. Now we add to the code that we just created to make this new test pass, and no more. We continue with the procedure until there is nothing left to test. The code created is simple and concise, implementing only the required features. This regression technique of testing the code at every phase makes the system gain a better test coverage.

Figure 1.2 shows the design cycle paradigm. With verification being the major part of the

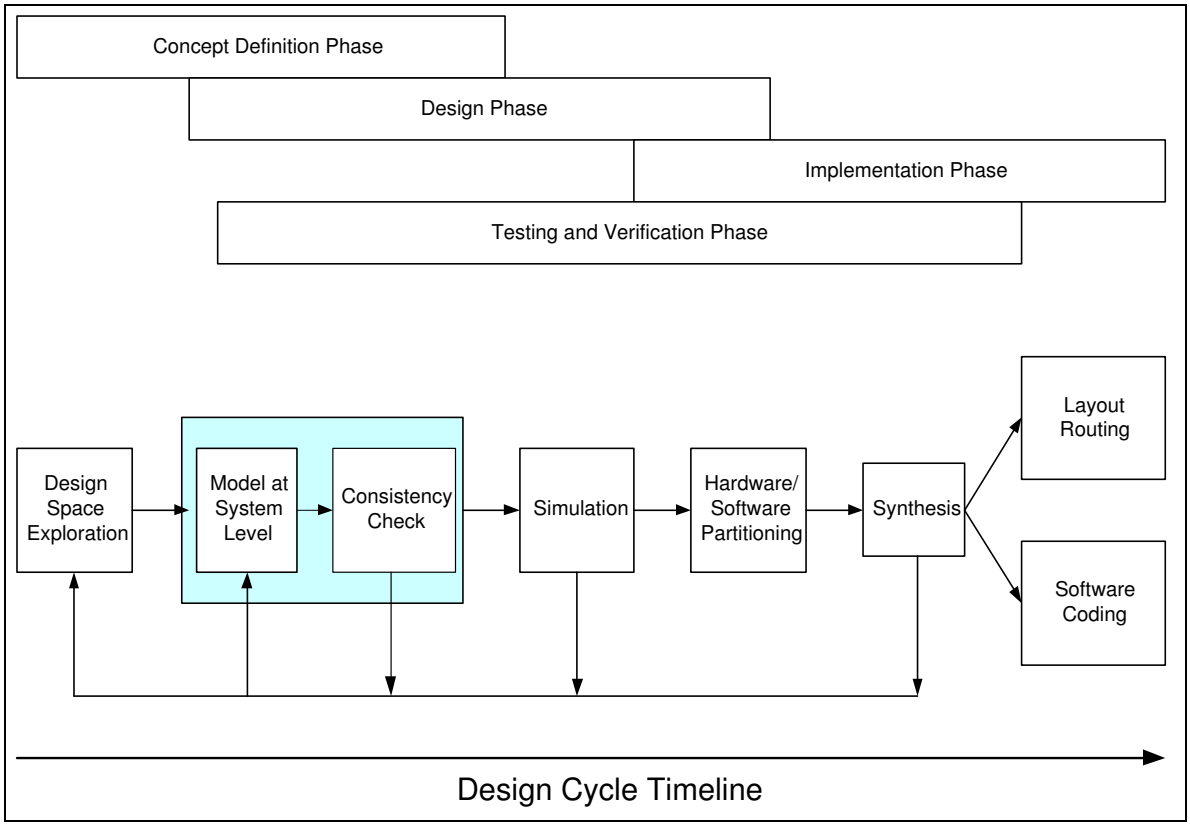

Figure 1.2: Design Cycle

design cycle, our motivation is to decrease the verification aspect of it. Building correct PFMs is one of the challenging problems. PFMs, as described earlier, capture natural language specifications in a formal model to analyze consistency of the specification. It is vital that these PFMs contain all the relevant system properties that need to be analyzed. PFMs with unwanted complexity and irrelevant behavior may result in a longer verification time implying a larger *state space* search for verification. One of the key ideas of efficient verification is to reduce the number of reachable states that must be searched to verify the properties. Lower number of states required to verify a property will result in an efficient verification.

1.4 Main Contributions of this work

We propose a methodology for an agile formal method: eXtreme Formal Modeling (XFM), based on Extreme Programming (XP) concepts to construct abstract models from a natural language specification of a complex system. We show how to incrementally build PFMs by adding user stories one by one into the model. XFM uses a "property driven" approach to build formal models. Models are built based on the properties instead of being built depending on the implementation. We regressively model check the model against all the properties created so far. If at any step the model does not satisfy a specific property, we locate and fix the error in the model. Due to our regressive approach, the debugging is easily done since we only have to backtrack one step whereas debugging a complete model for locating an error is difficult. Another advantage is that the model constructed using our methodology does not contain much irrelevant behaviors since the model is constructed to tightly fit the properties considered so far, whereas in the "standard" modeling approach, constructing the complete model at once may cause the model to contain irrelevant behaviors. We illustrate our methodology with

various examples of traffic light controller, DLX pipeline and a Smart Building control system. Since our methodology is based on modeling the properties, it is important to decide what order of properties should be used to build the abstract model. The properties contain predicates that describe certain behaviors. The complete set of these behaviors modeled in a specific manner constitute the model. Our ordering schemes are based on the frequency of the predicates present in the system. We analyze the effect of ordering properties for building PFMs with XFM with three different property ordering approaches: *arbitrary ordering*, *property based ordering* and *predicate based ordering* of properties. We used our property ordering approaches with XFM methodology on the models of ISA bus [54] and the arbitration phase of Pentium Pro bus [53]. We found out that the predicate based ordering approach was the most effective way of XFM and we formally reason the same as well.

We have also built a platform independent GUI tool that provides a GUI interface to facilitate the usage of our extreme formal modeling approach with property ordering. The user can load a list of predicates and construct a list of properties that can be sorted based on the ordering scheme selected for modeling. With the tool, the user can also invoke a text editor and build the model. For model checking purposes, the tool is interfaced with the SMV [47] model checker.

To the best of our knowledge, this is the first methodology for building formal models in an agile development style. The GUI is also useful in enabling users to easily build models incrementally and correctly through regressive steps. The property set developed as user stories are expressed formally, and hence can be used for coverage monitor and test generation as well.

1.5 Organization

This thesis is organized as follows: In Chapter 2 we provide relevant definitions used through out the thesis. We discuss formal methods and its application with a simple example of a traffic light controller. Using this example, we discuss the basis of temporal logic. We also talk about model checking and the tools that we use in our methodology.

In Chapter 3, we present our incremental framework of modeling systems with extreme formal modeling. We describe the steps required to go about using XFM to build PFMs. We illustrate our methodology with examples of a traffic light controller and a 5-stage DLX pipeline.

In Chapter 4, we present a more detailed example of a smart home system. The model of the smart home system is based on the ideas of various other projects like Oxygen project of MIT [19], the Internet home built by Cisco [4] and the adaptive house build by the University of Colorado [2]. The smart home system features intelligent control and interaction of illumination, heating, cooling, safety, security, and appliances. Our model of the smart home system exemplifies the usage of formal and agile methods for such high reliability applications.

In Chapter 5, we present a heuristics for ordering the properties using XFM. We realize that the selection of the order of modeling properties would make a difference in the number of reachable states needed for their verification. We illustrate the case study of the ISA bus and the arbitration phase of the Pentium Pro bus using three approaches: *random, property based and predicate based ordering* of properties.

In Chapter 6, we describe a platform independent graphical user interface based tool set that facilitates using XFM methodology with special GUI window for property editing, sorting, and model building.

Chapter 7 consists of some discussion on related and future work.

Chapter 2

Preliminary Definitions and Terminologies

In this chapter, we discuss the relevant definitions pertaining to our work on Extreme Formal Modeling. We discuss formal methods along with some examples. We then elaborate on model checking since we use this particular formal method in this thesis for the implementation of extreme modeling. We further describe some frameworks and tools used for model checking. We also discuss extreme programming and the necessary components required for it.

2.1 Formal Methods

Formal Methods include ways of describing the behavior of systems mathematically and reasoning about them to prove correctness and analyze their behavior and performance. Formal methods are needed in many fields of software and hardware systems, and more

significantly used for the safety critical systems to ensure that they behave in a way they are supposed to behave. Errors in such systems may result in damages in various aspects. Various techniques of formal analysis have been developed, each having its own advantages and disadvantages.

One such method of formal analysis is model checking [25, 51]. Model checking by using temporal logic was developed by Clarke and Emerson [25, 26] as well as by Queille and Sifakis [51] in the early 1980s. Later on, many model checking tools such as SPIN [41], SMV [47], UPPAAL [43], etc were developed. Model checking has been one of the popular formal methods used in the industry and we have used some of these model checking tools to illustrate our methodology. We discuss more about the tools and methods used in our work in the later part of this chapter. There are other techniques for formal analysis such as theorem proving [61, 16, 8, 18], equivalence checking [55, 45], etc. Theorem proving systems are computer programs capable of proving theorems and proofs by formulations of the problem as axioms, hypotheses and a conjectures whereas equivalence checking involves checking for equivalence between two systems which may be described using HDLs or as gate-level circuits, or in any other suitable formalism.

Formal methods employ various modeling frameworks [32] for specifying and describing various systems. For a system to be correctly specified for its intended behavior, the correct framework has to be chosen. Various modeling frameworks such as finite state automata [41], petrinets [62], stochastic modeling framework [45], etc. exists in the industry today for analyzing formal methods.

To illustrate the applications of formal methods, we will consider an example of a traffic light controller shown in figure 2.1, which shows an intersection with two signals: A and B. The finite state machine representation of the traffic light controller is shown in

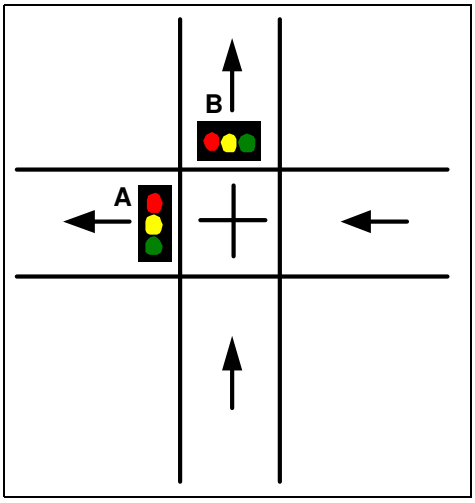

Figure 2.1: Traffic light controller

figure 2.2 which consists of four states and a set of transitions. Each state has specific values for signals A and B. The signals can have three possible values: red, yellow and green. The set of possible transitions that A and B can take is described in the automaton in Figure 2.2. The formal definition of this system is as follows:

Let $X = \{A, B\}$ be the set of state variables.

Let $W = \{red, yellow, green\}$ be the domain of A, B

then, $P = \{(x = w) \mid x \in X, w \in W\}$ is the set of primitive predicates.

Let $S = \{s_1, s_2, s_3, s_4\}$ be the set of states. Note that $S \subseteq X \times W$. $X \times W$ has some combinations where (x, w) are not possible.

then, $P_{s_i} \in P$ is the set of predicates true in s_i for example $P_{s_0} = \{A = green, B = red\}$ in state s_0.

Let $L = \{(x := w) \mid x \in X, w \in W\}$ be the set of action labels.

Let L_{t_k} be the set of actions taken on t_k transition. For example, L_{t_1} represents the action taken on t_1 which is $A := yellow$.

Let $T = \{(s_0, L_{t_0}, s_0), (s_0, L_{t_1}, s_1), (s_1, L_{t_2}, s_2), (s_2, L_{t_5}, s_2), (s_2, L_{t_4}, s_3), (s_3, L_{t_3}, s_0)\}$ be the set

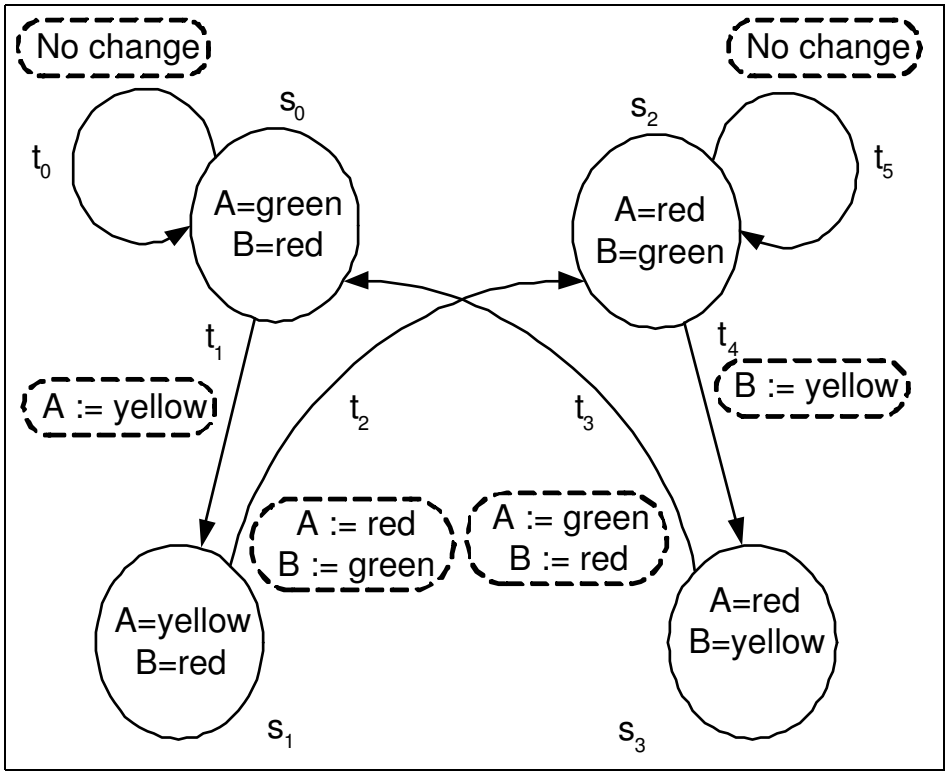

Figure 2.2: Finite State Machine of a traffic light controller

of transitions where $T \subseteq (S \times L \times S)$

Next, we define what model checking is, how it is done and also discuss what tools we use to experiment our methodology.

2.2 Model Checking

Model Checking [27] is an automated technique for formally verifying formal models of systems. Applying model checking to a design consists of several tasks that include modeling, specification and verification. The modeling aspect involves converting a

design into a formal language representation that can be accepted by the model checking tool. This procedure may require abstracting away irrelevant details of the system that are not relevant for the actual intended behavior. Abstraction helps to reduce the model size of the system, which prevents the problem of *state space explosion* which we discuss later in this section. Specification of the properties require stating the properties in some logical formalism which can be expressed in many different ways such as temporal logic properties [50], finite state machines [41], first order logics [24], etc. The model should be designed such that it satisfies the required properties. Once we have the specified property and the formal model of the system, we then verify the properties against the design. The model checking system may either terminate with a successful verification of the system or will give a counter example in the form of a trace indicating that the system failed to satisfy the given property. The user can then locate and fix the error, which either may be in the modeled design or may have occurred due to incorrect specification of the system that is being modeled. The problem of model checking can be described as follows:

Let M be the model and let $S = \{s_0, s_1, ..., s_i\}$ be the set of states in the model.

Let φ be the formulae to be checked.

then,

model checking problem is whether $M, s_k \models \varphi$, which means whether the formulae φ is satisfied in s_k.

Referring to the example of the traffic light controller discussed in section 2.1, let us consider a safety property that states whenever signal A is green, then signal B is red. We have the property φ being $[](A = green \rightarrow B = red)$. This property will hold true for the model at all states since whenever, signal A is green which is *true* at state s_0, then signal B is red. Apart from state s_0, signal A is never green. Now, let us consider another property that states whenever signal A is red, signal B is green. We have the

corresponding φ as $[](A = red \rightarrow B = green)$. This property will not be satisfied since in state s_3, signal A is red but signal B is yellow and not green. Therefore, the model checker will generate a trace showing that at state s_3, the property does not hold.

There are some pros and cons for using model checking as the formal method of approach. One major advantage it has over theorem proving is that it requires lesser human interaction. Whereas, theorem proving systems require an expert in the domain of application to solve the problem in a reasonable amount of time since the interactions may occur at a much higher levels where the user determines the intermediate lemmas to be proved. On the other hand, model checking requires some user interaction to fix the model when an error trace has been generated for an unsatisfied property. Another advantage that model checking has is that every reachable state in the system is checked to ensure that the property is satisfied.

The main challenge in model checking is the problem of state space explosion [47]. This problem occurs in large and complex systems with many components that interact with each other or in systems with data structures that can assume many different values. Some of the techniques can be applied to alleviate the problem of state space explosion that include abstraction [30], symbolic representation [33], partial order reduction [42], symmetry [27], etc. These techniques help reduce the model size of the system thereby easing the process of verification.

2.2.1 Temporal Logic

Temporal logic [27] is a formalism that allows us to describe and reason about causal as well as temporal relations of properties. The analysis of systems by using temporal logic

was initially used by Amir Pnueli in 1977 [50]. Temporal logic is used for specifying order of events in time without introducing the time explicitly. Operators such as *Eventually*, *Always* or *Never* are used to describe events and their temporal aspects in a system.

We talk about some of the common temporal operators in the next section. There are many advantages of using temporal logic to specify systems. One of the main advantages is that it has a well-defined semantics. It is independent of the modeling language as well as has enough expressiveness. Temporal logics can be classified into *linear time temporal logics* and *branching time temporal logics* with the distinction being how they handle the branching in the underlying computational tree where a computational tree shows all possible execution paths starting from the initial state. In linear time temporal logic, events are illustrated by operators in a single computational path where as in the branching time temporal logic, the temporal operators quantify over different paths reached from a given state [47]. Referring to the traffic light example in section 2.1, let us consider a property that states if signal B is yellow, it will eventually become red. This property in LTL is specified as $[](B = yellow \rightarrow <> (B = red))$. On the other hand, in CTL, the property can be specified as $AG(B = yellow \rightarrow AF\ (B = red))$.

In our thesis we mainly focus on linear time temporal logic (LTL) which we define in the following section.

2.2.2 Linear Time Temporal Logic

In Linear time temporal logic(LTL), it is possible to define properties for all reachable states in a single computational path. We show some of the commonly used syntax and semantics of LTL with the example of the traffic light controller discussed in section 2.1. For a complete formal treatment of LTL, one should refer to [27]. Revisiting the example,

we have two signals A and B. Both signals can have three different values: red, yellow and green. We show some of the properties based on the example to illustrate the temporal operators below:

Example 2.2.1 *Always - It is denoted by [] symbol.*
Property: Always both signals A and B cannot be green at the same time.
LTL property: $[](\neg(A = green \;\&\&\; B = green))$. *The property captures the notion that at any time, signals A and B will never be green simultaneously. The property is considered to be a* safety *property.*

Example 2.2.2 *Eventually - It is denoted by <> symbol.*
Property: When signal A is yellow, it will eventually change to red.
LTL property: $[](A = yellow \rightarrow \; <> \; (A = red))$. *The property captures the notion that whenever A is equal to yellow, it will eventually become red.*

Example 2.2.3 *Next - It is denoted by X symbol.*
Property: If the signal A is yellow, it will change to red in the next state.
LTL property: $[](A = yellow \rightarrow X\ (A = red))$. *The property captures the notion that whenever signal A is yellow, in the next state, it will change to red.*

Example 2.2.4 *Until - It is denoted by U symbol.*
Property: The signal B will not change to green until A changes to red.
LTL property: $[]((B = \neg green)\ U\ (A = red))$. *The property captures the notion that always B will not change to green until A is red.*

The properties of the example were not difficult to express but at times, temporal properties are not always easy to write [31]. Therefore, possibilities of errors exist if a user

wants to express the behaviors in LTL. A simple way to check if the intended behavior that the user is trying to specify is correctly expressed in LTL is to convert the LTL property into a finite state machine [41]. Finite state machine is a framework for formal modeling and provides an ease of visualizing the expressed behavior. There are tools that automatically convert LTL properties to finite state machines and one such tool that we use in our thesis is LTL2BA [10] which is explained in the next section.

2.2.3 LTL2BA

LTL 2 BA [10] (LTL to Büchi automata) generates a Büchi automaton [41] representing a LTL expression. This visualization is instrumental in verifying that the intended behaviors are correctly expressed in the specifications. LTL 2 BA also generates PROMELA [41] code from an LTL expression. PROMELA is the specification language used by the SPIN [41] model checker which we explain later in this thesis. LTL2BA - in theory - can be used to obtain the abstract models directly and automatically from the LTL properties. However, in practice this does not work too well since (1) it is not possible to describe some implementation details such as initialization of variables and changing state variables, and, (2) it is not possible to concatenate several properties since the resulting automaton may be huge. This is because LTL 2 BA is building a monolithic automaton, where the concurrency of independent parts is not captured. Therefore, LTL2BA cannot be used to build models for the properties, but can be used to visualize that the property being expressed is correct. We use this tool often for *sanity check*.

Small changes in a LTL property, like the misplacement of a parenthesis, can change the meaning completely. For example [](a→<> (b U c)) represents a simple 2-state automaton, but if we move one parenthesis before the <>, it becomes a 7 state automaton. Such mistakes are hard to detect, therefore it is important to check whether the properties

specified are correct before passing them to the model checker.

2.3 SPIN

SPIN [41] is a model checker used extensively for verification of software systems. It was developed at BELL Labs [3] by Gerard Holzmann [41]. It was recognized by ACM (the Association for Computing Machinery) with Software System Award [13]. The specification language used by SPIN is a high level language used to specify systems descriptions and is called PROcess MEta LAnguage (PROMELA) [41]. SPIN is used to trace logical design errors of various system designs and checks the logical consistency of the specification. SPIN avoids constructing a global state graph as a prerequisite for verification; instead, it works on-the-fly, which means that the states are created during the verification process. Its basic building blocks include asynchronous processes, message channels, synchronizing statements, and structured data [41]. Once the model is built, the user can simulate it with the built-in simulator that supports random, interactive as well as guided simulation schemes. Also various optimization techniques such as partial order reduction [23], statement merging [52], hashing [64], etc, make SPIN faster on certain classes of verification problems.

SPIN reports on deadlocks, unspecified receptions, flags incompleteness, race conditions, and unwarranted assumptions about the relative speeds of processes [14]. It verifies LTL properties by considering LTL properties as never claims in the form of Büchi automata. It then verifies the model against the Büchi automata. If the property is not satisfied, it generates a counter example in the form of a trace file. This trace file can be loaded into the simulator provided by the tool and checked for the errors in the model. For some properties that are not expressible by LTL formulae, it also considers verifying an

automata in the form of a never claim against the model. For more information on SPIN, refer to [14].

2.3.1 PROMELA

PROMELA [41] is an acronym for *PROcess MEta-LAnguage* and is intended to make easy abstractions of system designs. It is not an implementation language, but a description language. Specification in PROMELA consists of processes, message channels and variables. The processes are global objects running in parallel unless specified. Message channels can be declared either globally or locally within a process and are used to pass data from one channel to another. Message channels are *randez-vous* [40] in PROMELA meaning that a handshake occurs for send and receive operations in different processes. Variables can also be declared either globally or locally within the processes. Depending on the state of the system, any statement in PROMELA is either executable or blocked. Given below is an example code of blocked:

```
/* The code illustrates a blocking instance. The execution will block at while until the
value of a is equal to value of b where a and b are variables. Once they become equal,
then the value of c will be added to a*/
while (a!=b) {
    skip;
}
a = a + c;
```

For more reference on the PROMELA syntax, refer to [41]. SPIN is mostly used for verification of concurrent software systems rather than descriptions of hardware circuits. On the other hand, SMV [47] which we discuss next is used widely for hardware verification systems.

2.4 SMV

SMV is another model checking tool used for verification of hardware systems. It was developed by Ken McMillan [47]. It allows several forms of specification, including the temporal logics CTL and LTL, finite automata, embedded assertions, and refinement specifications. It also includes an easy-to-use graphical user interface and source level debugging capabilities. SMV automatically verifies a design for all possible input sequences for properties of combinational logic and interacting finite state machines. When a property fails to verify, then a counterexample trace is produced which helps locating the bugs and fixing the model. Since SMV model checks all possible reachable states in the system, it may lead to a problem of state space explosion. SMV provides a number of tools to help the user reduce the verification of large, complex systems to small finite state problems. These techniques include refinement verification, symmetry reduction, temporal case splitting, data type reduction, and induction [47]. We use Cadence SMV [7] as one of the tools for model checking in our thesis. Reference of SMV syntax and semantics of Cadence SMV can be found at [12].

2.5 Vacuity check

Vacuity can indicate a serious flaw either in the model or in the property. Antecedent failure, where a formulae containing an implication proves true because the antecedent or the precondition of the implication is never satisfied, is one of the reasons that a property can be vacuously true. Revisiting our example from the section 2.1, consider the following scenario:

Property: When signal A is green and signal B is yellow then in the next state B will

change to green.
LTL property: $[](($A = green$ \&\& $B = yellow) \rightarrow X\ B = green)$.
The scenario mentioned above is wrong and should never happen. However, since the antecedent of the property is always false, the property will verify to be true which is incorrect. Therefore, it is important to check that at some point of time, the antecedent becomes true. One of the ways to verify that can be to check if the negation of the antecedent is never true [21]. For example, the LTL property: $[]\neg(A = green$ \&\& $B = yellow)$ will be satisfied implying that the antecedent condition never occurs. Instead, the model checker should generate a trace for the property if the antecedent was true at any point, thus ensuring that the property is not vacuously true. Since, we use Cadence SMV as one of the model checking tools, we can also specify the property in branching time logic. For example, the vacuity check for above mentioned property in CTL can be specified as: $EF\ (A = green$ \&\& $B = yellow)$ which states that there will exist a path in the system which will have signal A as green and signal B as yellow. This check is also referred to as *trigger check*.

2.6 Extreme Programming

Extreme programming [6] (XP) has been popularized in the object oriented software community in the recent years. It introduced novel guidelines and concepts of an agile methodology that seem to increase programming productivity significantly while producing higher quality error free code [65, 63]. Extreme programming is a "*Test-driven*" approach. The customer gives a set of user stories that are converted to a test. These tests are run every time whenever a newer version of code is released. This regression technique of testing the code at every phase makes the system gain a better test coverage. It involves building of code by starting from a simple design and

performing a thorough test on each stage while improving upon the design. The XP team keeps the design thorough and completely relevant to the requested functionality of the system. Its core functionalities involve getting user stories from the user. The user stories are short descriptions that convey the exact detail of the required functionality of the design, which enables the programmers to be certain about the features that are being requested by the customer. Extreme programming is an incremental and iterative process, therefore having a good design from the start is essential. One of the focuses of extreme programming is the validation of the software at all times. Programmers develop the software by writing the test cases first from the user stories provided by the customers. Refactoring and continuous regression are also one of the main aspects of extreme programming which enables the programmer to improve the design of the system throughout the entire developmental process. The improvement of the design is done by cleaning up the code and removing duplication of the design aspects. More extreme programming practices such as working in pairs, collective programming, etc exist which we do not focus on for our methodology. A complete reference of extreme programming can be found at [6, 65, 63].

We show in the next chapter how we use some of the concepts we describe here for formal methods.

Chapter 3

XFM Methodology

3.1 Introduction

We introduce an agile formal method (named XFM) based on extreme programming concepts to construct abstract models from a natural language specification of a complex system. In our experience, both in carrying out formal verification for major microprocessor chips, as well as, working on tools and methodologies, we have found that the major challenges faced by industrial formal verification engineers are two fold: (i) Making sure that the natural language specification of the system is translated into a sufficiently complete set of formal properties to be used in model checking of an implementation, (ii) In conformance based formal verification using abstraction techniques, creating an abstract model which satisfies all formal properties intended in the natural language specification. Most of the times, it is hard to validate the sufficiency/completeness of the property suite developed from the natural language, or to make sure that the abstract model is constructed correctly. By 'correctly' we mean that the set of behaviors

of the abstract model is not only a super set of the set of behaviors of an implementation, but also a subset (in the best case, equal) to the set of behaviors intended/allowed by the natural language specification.

3.2 Extreme Formal Modeling (XFM)

As for any system development, it is important to have a concise and clearly written specification of the system. Some time must be spent on the specs to get an overview of the whole system and maybe visualize its main structure. Both, a clear system specification and a deep understanding of the system are crucial for good LTL properties.

Many of the Extreme Programming (XP) rules can be applied directly and successfully in XFM. For instance, one of the main XP rules is to write tests before the actual code (test-driven approach). In XFM, this rule maps to specifying the linear time property before writing the abstract model (property-driven approach). Another important XP technique is to add functionality as late as possible, incrementally increasing the complexity of the model. Iterations are small steps in the development process. At the start of each iteration the goals are identified and written down in the form of "**user stories**" - individual cards that point out specific implementation details and requirements. These user stories act as a detailed guideline for the programmer. To refactor problems, to update tests after a bug is found, and to work in pairs are also principles that are as beneficial to the capturing of formal methods as they are for common programming projects.

Figure 3.1 presents XFM's incremental approach to formal modeling. The initial part of our XFM procedure involves breaking down the English specification to user stories. We select a user story that describes basic functionality of the system, and transform it into an LTL property.

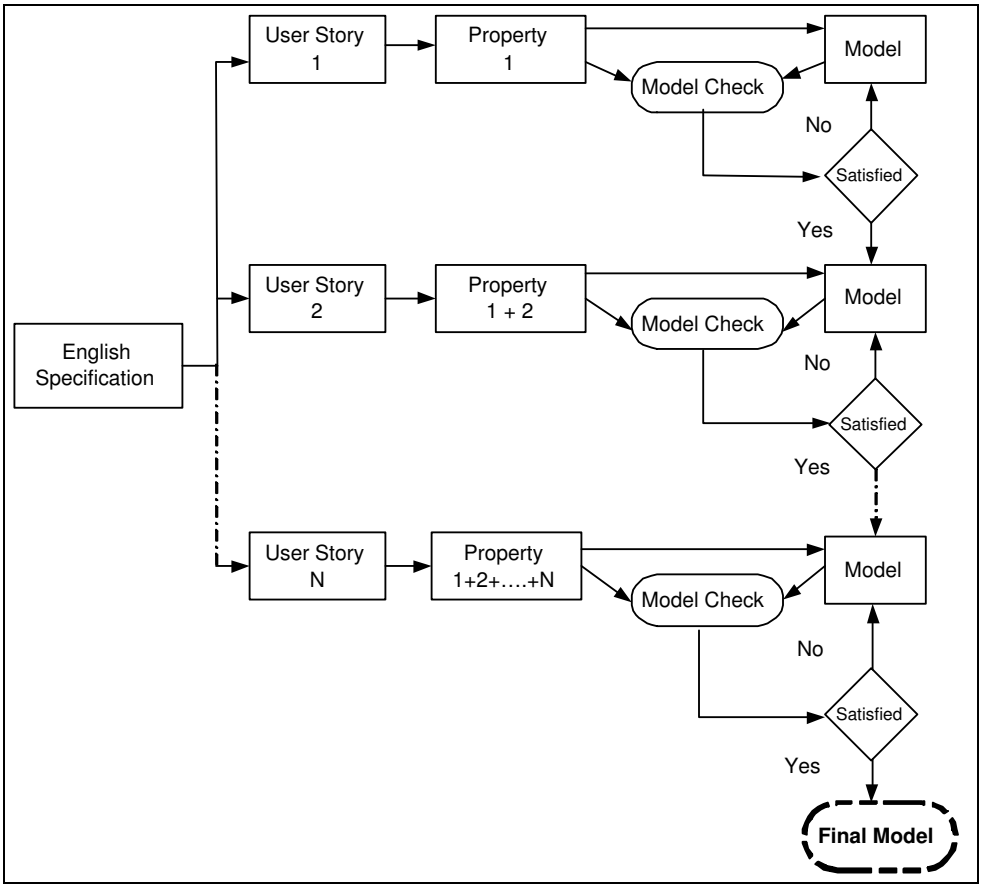

Figure 3.1: Capturing a formal model with XFM

At this stage, we can check if the LTL property correctly expresses the behavior of the user story. LTL properties can be visualized as finite state machines and LTL 2 BA eases this step by displaying the corresponding FSM. It is important that while implementing the model, only the behavior of this property is taken into account. After listing out the complete list of linear time properties, we select one property from the list and build an abstract model for this property and model check if it holds for the model. Once the property is satisfied, we take a second property, extend the model according to this property, and model check for both properties. This procedure is repeated until the abstract model contains all behavior from the English specification and all the properties

in the list are satisfied. The controlled and incremental model building results in a compact and structured abstract model. If the model checker fails to validate the property, we can locate the error with the help of a trace file generated by the model checker, fix the bug and rerun verification. Whenever a property fails to validate, it usually is straightforward to find the bug as it must be related to the latest additions. The complete effort of modeling and bug fixing grows incrementally along with the size of the model. Algorithm 3.2 gives a formal representation of XFM.

Algorithm 3.2: XFM Approach

{ For a given system, we have the behavior in terms of natural language specification which are converted to user stories}

Let $US = \{us_1, us_2, ..., us_n\}$ be the set of all user stories for the system

Let $\Pi(us_i) = \{\pi_j, \pi_{j+1}, ..., \pi_k\}$ be the set of properties for a user story

Let $\Pi = \cup_i \Pi(us_i)$ $= \{\pi_1, \pi_1, ..., \pi_m\}$ be the complete set of properties after a specific ordering has been chosen.

/*We show 3 different ordering approaches for property modeling later in this thesis*/

Let $\Pi_i = \{\pi_1, \pi_1, ..., \pi_i\} \subseteq \Pi$, so Π_i represents the first i properties in the specific order chosen.

Let $X(\Pi_i)$ be the model which satisfies all the properties in Π_i

`Initial` $X = \emptyset$, $i = 0$,

`Step1:` $i := i + 1$

Step2: Build Abstract Model $X(\Pi_i)$, the model is built to satisfy all the properties in Π_i simultaneously.

$X := X(\Pi_i)$

$ModelCheck(X, \Pi_i)$

/* This is the regression step */

if *ModelCheck* `fails for a` π_k

`go to Step2`

`/* to change the model suitably so the failing property can pass*/`

else if $i = m$

X is the required model

else

go to Step1

3.3 XFM Advantages

Our preliminary model building exercises show that our methodology is superior to the traditional way of capturing formal specification. One of the key advantages is that XFM involves an iterative technique. The evolving model facilitates debugging whenever a property is found to be unsatisfied. After each stage, we make sure that the model concurs to the specification through model checking of all previously specified properties. If a property fails, the error can be easily located in the parts of the model that have been changed in the last iteration.

Another contribution is that our model is built using the properties. Hence, it does not include much unintentional details. Figure 3.2 illustrates how the amount of behavior for the properties and the abstract model develop during the capturing process. At any point, the behavior of the formal properties is more general than that of the abstract model. At the beginning, however, they are both much more general than the behavior of the specification. In each iteration step, their behavior is confined by adding additional properties and details to the model. Obviously, the behavior of the specification does not

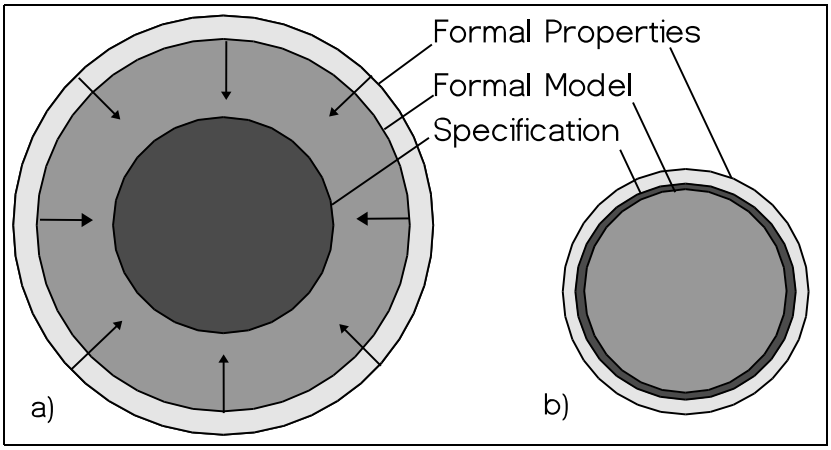

Figure 3.2: Modeling process (a) and modeling result (b)

vary during this process since the specification is not altered. Ideally, in the end, all three behaviors are identical with the specified behavior, but in practice, there will always be a small gap. However, this gap will be much smaller for the XFM methodology than for the traditional approach. The fact that the behavior of the model is closely linked to the properties entails a close to complete set of properties once the model is complete; simulation of the model will help reveal missing functionality. In the conventional approach, however, the model tends to contain much more functionality than specified, but less properties than needed as there is no mechanism that guarantees the exposure of all properties of the spec. The overall time to build and validate the model is substantially less, especially for large systems. This is mainly due to the iterative aspect. Since the model is checked for each property after each iteration, the time needed to debug is less. This is in contrast to debugging the entire model at once for satisfying all properties for the traditional approach.

Following an XFM based methodology, brings about a list of advantages. One of them is speed. The small debugging steps between the iterations can be done much faster than debugging the complete model in the end. Besides speed, more importantly the quality

of the resulting formal model will be better. Figure 3.2a illustrates how the amount of behavior for the properties and the abstract model develop during the capturing process. At any point, the behavior of the formal properties is more general than that of the abstract model. Both are more general than the behavior of the specification. In each iteration step, their behavior is confined by adding additional properties and details to the model. Since the specification is not altered, the behavior of the specification does not vary during this process. Once the model is completely build, all three behaviors are identical, but in practice there will always be small gaps (Figure 3.2b). The gap between the formal properties and the specification indicates how much system functionality is not expressed in the formal properties, and the gap between the formal model and the specification marks the amount of functionality the model contains that is unaccounted for in the specification. The latter is small because we only add functionality from simple user stories. The former gap is small as the process makes sure that the gap between properties and model never gets big in the first place. It is quite clear that in an ad hoc methodology of model building, debug time is long, the resulting formal model contains much more functional details than specified, and the properties are far from covering the whole specification.

Our XFM based methodology addresses these problems, and with two illustrative examples (of a control intensive traffic light controller, and the DLX pipeline) we present this methodology and show the benefits.

3.4 Examples and Results

In order to demonstrate the power of XFM, we present two examples from different domains and of different complexity. A simple traffic light will illustrate the main steps,

tools, and techniques involved. The design of a DLX [34] microprocessor pipeline will show how this methodology works for a bigger model, and how the model evolves with the incremental approach.

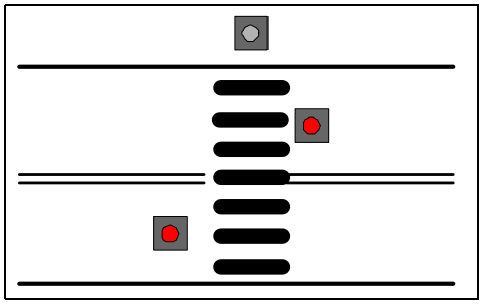

Figure 3.3: Sketch of the Pedestrian Traffic Light

3.4.1 Traffic Light Model

The example of traffic light controller is a simple example of a pedestrian crossing with a traffic light (Figure 3.3). When a pedestrian pushes a button, the traffic signal turns red, and the pedestrian signal turns green. After thirty seconds, the pedestrian signal becomes red again and the red light goes off for the cars. Therefore, this description is the English specification. Now, we construct LTL properties describing this system. Let us assume that p is high when pedestrian signal is red and c is high when car signal is red. We will denote sw high as switch pressed. We start with the most important property that states that both pedestrian and car, can never get the GO signal at the same time: []!(!p && !c). We verify that the property concurs with the specification with LTL 2 BA. Figure 3.4(a) shows the automaton corresponding to this property. The corresponding model is as simple and contains only one state.

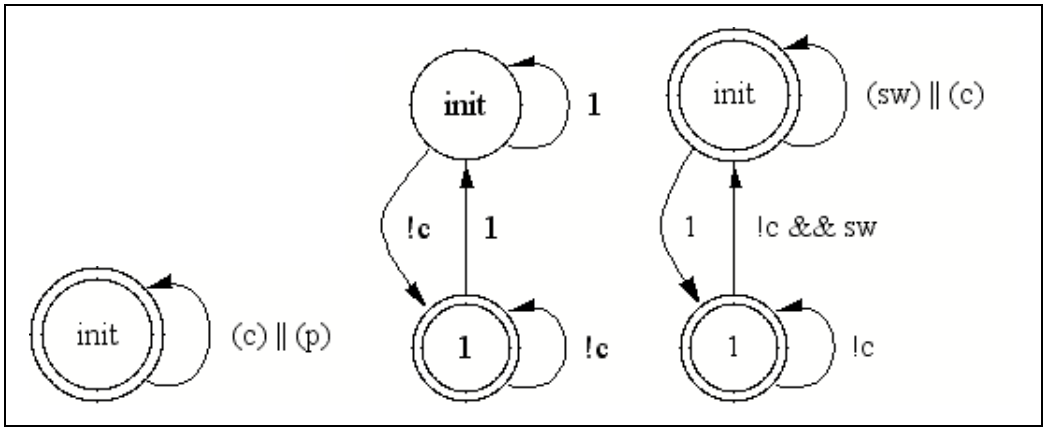

Figure 3.4: FSMs of traffic properties 1 (a), 2 (b), 3 (b)

The next property states that whenever the car signal turns red, they will eventually become green (Figure 3.4(b)). Table 3.1 lists all LTL properties for this example. LTL does not allow to express exact timing, only relative occurrences of events. However, in the model we add a timer that counts to 30 before the pedestrians stop and the cars can run again. The model now includes two states, one where cars go (!c) and pedestrians stop (p) and the other where pedestrians go (!p) and cars stop (c).

*Prop*3 and *Prop*4 state that when no switch is pressed, the cars keep driving and the pedestrians keep stopping (Figure 3.4(c)). As we check these properties against the formal model, we realize that they can be verified without making any modifications to the system, and a closer look at the properties shows that their behavior is already satisfied by property 1 (Figure 3.4(a)).

One functionality that is still missing is the inclusion of the switch. When cars go and the switch is pressed, eventually pedestrians should be allowed to walk before the switch turns off. This property is a bit longer than the others, and without LTL 2 BA it is not

Prop1	Never both signals can be green at same time	$[]!(!c\&\&!p)$
Prop2	If cars cannot go, they will go eventually	$[](c \rightarrow <>!c)$
Prop3	No button pressed, cars keep going	$[]((!c\&\&!sw) \rightarrow X!c)$
Prop4	No button pressed means that the pedestrian signal cannot turn green	$[]((p\&\&!sw) \rightarrow Xp)$
Prop5	When the switch is pressed while the cars go, pedestrians will go before the switch is turned off. $[](sw\&\&p \rightarrow swU(!p\&\&sw)\&\&(!p\&\&sw \rightarrow !pU(!p\&\&!sw)))$	

Table 3.1: LTL properties for traffic light (c = cars stop, p = ped stop, sw = button is pressed)

easy to figure out if it is correct (Figure 3.5). After implementing the functionality of these properties into the model, simulation shows that it works as specified, so we have found all properties. Listing 3.4.1 shows the PROMELA code for the complete Traffic light model.

Listing 3.4.1: Promela code for Pedestrian Crossing

```
bool sw, c, p;
int time;
active [0] proctype signal() {
cargo:
    p=1; c=0;
    if
        :: (1) → sw =1; time=30; goto pedgo
        :: (1) → goto cargo
    fi;
```

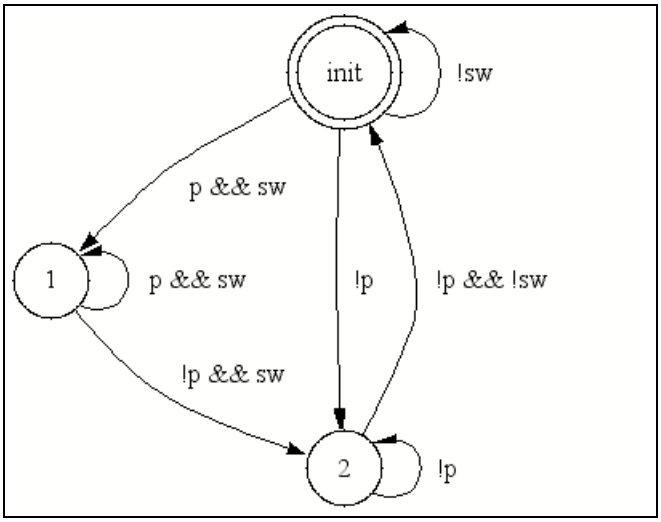

Figure 3.5: Graph for traffic property 5

```
pedgo:
    c=1; p=0; sw =0;
    time = time -1;
    if
        :: (time > 0) → goto pedgo
        :: (time == 0) → goto cargo
    fi
}
init {
    p = 1; c = 0; sw = 0;
    run signal();
}
```

3.4.2 Model of a DLX pipeline control

The capability of the XFM approach can be seen when working on large systems. The pipeline control of the DLX RISC processor model [34] is a well-known and reasonably

large example to show the use of XFM. The DLX contains a 5-stage pipeline, which means up to five instructions can execute concurrently. The cycles for the instructions are instruction fetch (IF), instruction decode (ID), execute (EX), memory access (MEM), and write back (WB). However, not all instruction types use the same cycles in the same order. Table 3.2 shows the cycle usage for the different instruction types.

Table 3.2: Cycles for Different Instruction Types

	IF	ID	EX	MEM	WB
Arithmetic	X	X	X		X
Load	X	X	X	X	X
Store	X	X	X	X	
Branch	X	X	X	X	

Starting from this system description, we identify the first user story. One of the most basic behavior is that each instruction executes in a certain order. So, generally speaking, instructions execute in the order $IF \rightarrow ID \rightarrow EX \rightarrow MEM \rightarrow WB$. In LTL this can be expressed as $[](if \rightarrow Xid)$, always ID after IF and then the same for ID and EX, EX and MEM, MEM and WB, and finally WB and IF. The automaton generated with LTL 2 BA (Figure 3.7(a)) shows that the LTL expression is sound. These five properties can be represented with a circular automaton that satisfies our first user story.

The second user story states that this order of execution still has to hold when we consider five concurrent instructions in the pipeline. In order to keep the model small we decide to use five concurrent processes each of which handles one instruction (Figure 3.9). Since the processes run independently, the first property does not hold any more. It is not guaranteed that directly after the first instruction is in the fetch stage it advances to the decode stage, since in the meantime other processes may get execution time. What we can guarantee however, is that we will never go directly into any of the other stages.

Now this has to be expressed for each cycle in each instruction, which means we get 25 LTL properties similar to cat1 in Table 3.3.

```
proctype instruction1() {
    inst_if:
      if
      :: st1=fet; goto inst_id fi;
    inst_id:
      if
      :: st1=dec; goto inst_ex fi;
    inst_ex:
      if
      :: st1=ex;  goto inst_mem fi;
    inst_mem:
      if
      :: st1=mem; goto inst_wb  fi;
    inst_wb:
      if
      :: st1=wb;  goto inst_if  fi;  }
```

Figure 3.6: PROMELA code for one single instruction through DLX

In the next iteration, we introduce the possibility to control the instructions from a controller. This is done by "enable signals", one for each instruction. The LTL expression will say that an instruction will not advance unless the enable signal is given (Figure 3.7(b)). Again we obtain 25 properties in the style of cat2 in Table 3.3. The changes in the model for these properties are small, so all of them can be verified without problems.

The next user story considers the control of the synchronization of the system. It states that the control enables each instruction in each cycle. Once the instruction cycle advances, the enable signal is set to zero, thus signaling the controller that it is ready for

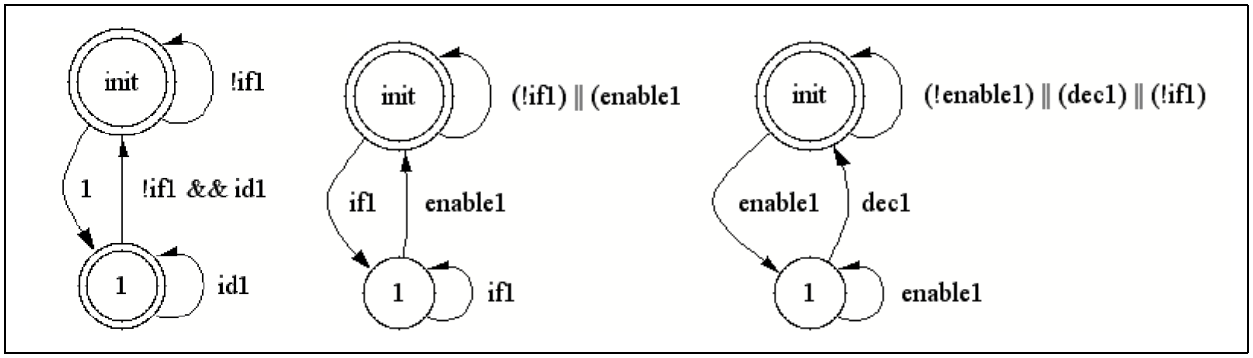

Figure 3.7: Graphs of pipeline properties 1 (a), 3 (b), and 4 (c)

the next cycle. This category of properties is somewhat more complex, but with the help of the LTL 2 BA tool we finally get cat3 in Table 3.3. It states that whenever an enable signal of a pipeline is true, it will change to the next stage before the enable signal goes down, unless the enable signal is already low (Figure 3.7(c)). Again, we have to check all the stages to satisfy the corresponding property. Once the model is constructed for all the corresponding properties and model checked, we again regressively model check the previously modeled properties.

Another important behavior of a pipeline is to prohibit multiple usages of resources. If one unit is using a stage of a pipeline, then no other pipeline can execute in the same stage of the system. For example, if an instruction is executing in IF stage, no other instruction can be executed in the same stage until the previous instruction completes from the IF stage. This ensures that there are no resource conflicts. Cat4 in Table 3.3 expresses this in LTL for the fetch cycle of the first instruction. Again the category will consist of 25 properties, one for each cycle. In order to satisfy this property in the model we introduce a control process that during initialization phase starts each instruction successively, and later makes sure that every instruction advances in each cycle. Again the verification of all properties and simulation finishes up this iteration step. With only 4 categories of properties the basic functionality of the pipeline is now verified and

Table 3.3: LTL properties for pipeline (examples)

label	LTL property
cat1	$[](if1 \rightarrow !(Xex1 \| \| Xmem1 \| \| Xwb1))$
cat1b	$[]((ex1 \&\& (load1 \| \| store1 \| \| branch1)) \rightarrow !(Xif1 \| \| Xwb1 \| \| Xdec1 \| \| Xwait1))$
cat2	$[]((if1 \&\& !enable1) \rightarrow (if1 U enable1))$
cat2b	$[]((wait1 \&\& !enable1) \rightarrow (wait1 U enable1))$
cat3	$[](if1 \rightarrow ((enable1 U dec1) \| \| !enable1))$
cat3b	$[]((ex1 \&\& (load1 \| \| store1 \| \| branch1)) \rightarrow ((enable1 U mem1) \| \| !enable1))$
cat4	$[]((if1 \&\& enable1) \rightarrow ((!(if2 \&\& enable2) \| \| (!(if3 \&\& enable3)) \| \| (!(if4 \&\& enable4)) \| \| (!(if5 \&\& enable5)))U !enable1))$

working.

To make the model of the pipeline a bit more realistic, we select the user story that defines different instruction types and their different cycle sequences from Table 3.2. It turns out that this does not result in a new category of properties, but rather induces changes to existing properties. This step illustrates that in the iterative process, not only does the model evolve, but also the properties can evolve and get more complex later in the modeling process. To satisfy the requirement, we extend our basic instruction automaton with a wait stage and transitions according to Table 3.2 (Figure 3.8). This will make sure that an arithmetic instruction for example, will now go from EX to WAIT and then to WB. We have to change some properties in category 1 and 3, and add properties in all four categories. Resulting LTL examples are shown in cat 1b and 3b in Table 3.3. Changes in the abstract model to reflect this are limited to update the FSM description for each instruction to the automaton of Figure 3.8 that means introducing the notion of an instruction type, and adding the transitions to and from the wait stage. For the controller, these changes are transparent since after the changes still each instruction takes 5 cycles

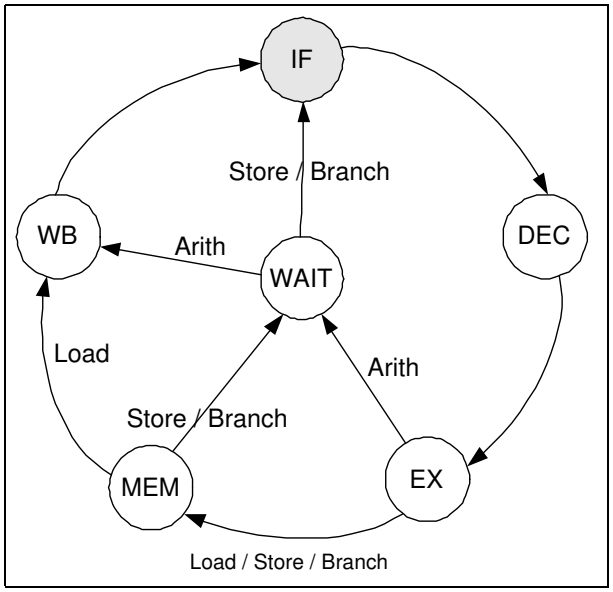

Figure 3.8: Automaton for one instruction

to finish, therefore preventing the occurrence of structural hazards. Figure 3.9 shows the complete structure of the pipeline model. Of course there would still be many more details that could be added to the pipeline, such as data dependencies and forwarding, but the steps will always be the same.

3.4.3 Summary

We present a novel approach to use mechanisms from extreme programming to capture formal models. Instead of building an ad hoc formal model and come up with properties to check it against, we show that when building the model along with the properties, the model grows incrementally and gets a natural structure. The major benefits are the speedup of the model-building process and the high quality of the model compared with the traditional approach. Since we handle small steps, each step will add limited

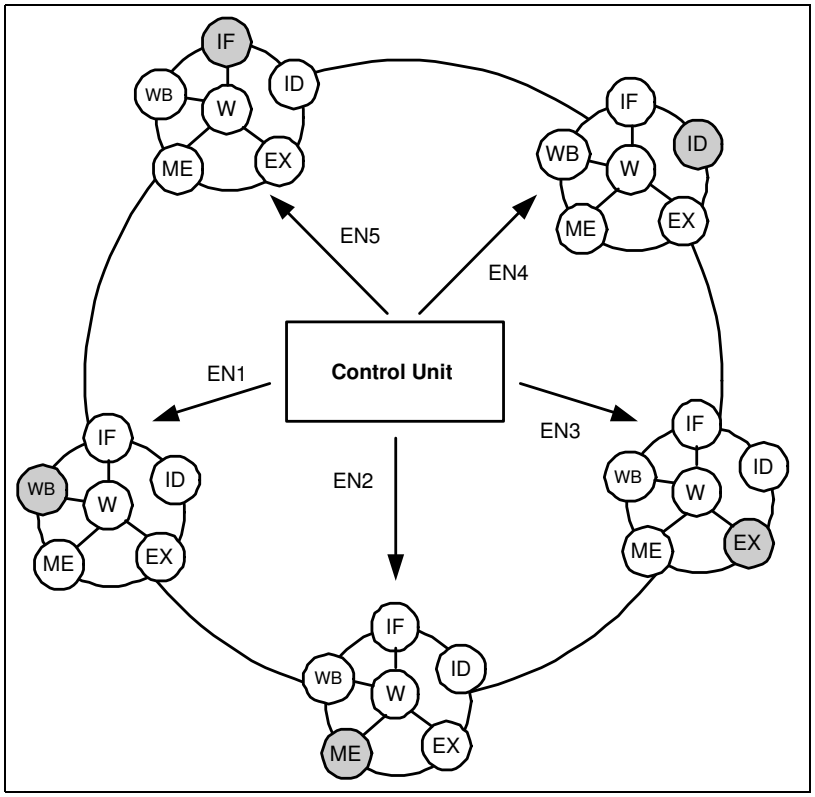

Figure 3.9: Overall Pipeline Design

functionality to the model, so the debugging process is much more directed. Another major benefit resides in the scalability. For example if ad hoc models are built, they are usually monolithic, but with XFM, complex models get broken down to small problems and can be built as concurrent state machines more easily. The time required to build models using this methodology grows incrementally along with the size of the model, whereas the design effort in a conventional methodology grows exponentially with the size.

Chapter 4

Another extensive case study: Smart Home

As technology advances, the cost and size of electronic components are becoming cheaper, smaller, and consume less power. As a result, these components are becoming an increasingly important part of our environment. Ubiquitous computing is no more science fiction, but instead an emerging technological area. There are electronic devices such as sensors, cameras, personal assistants, and microprocessors that provide us with information and transparently perform tasks without our knowledge of its working. This is what makes our environment "smart". A Smart Home uses these techniques to make living conditions more convenient and adapts to its resident's needs. However, for "smart"-ness, the technology must be constructed such that the resident is unaware of the complex technology behind the automation. Therefore, one important objective of a Smart Home is to take charge of obvious and repetitive tasks. Although the need for explicit control is taken away, the resident has supervisory control over the system.

It is desirable for a person that the environment is aware of his/her presence, acts according to his/her preferences, and is able to communicate with ease in many ways. However, while people are familiar with and accept that programs on their PC sometimes crash or behave inadequately, it is unacceptable that their home environment shows any unwanted behavior. It is unthinkable that a house refuses access to its resident, or those vital functions such as heating, lighting, or the security system does not behave the way that they should. On account of this, the development of the control for a Smart Home not only has to be done with diligence, but it has to include a methodology that completely rules out the possibility of such failures. Extreme Modeling is a methodology that not only provides for a correct model of the system, but also makes the process of model building and capturing of formal specifications faster and more intuitive.

4.1 The Smart Home Details

There are many approaches to what a Smart Home may look like and to what functionalities it should be able to perform without user assistance. The most important functionalities for a Smart Home include the control of lighting, temperature, security, and safety. However, some also include other features such as entertainment and specific smart appliances such as a smart refrigerator or a smart microwave. Systems also differ greatly in their level of awareness of the resident, where a higher level of awareness enables the system to adapt to personal preferences of different family members depending on their current location in the house and the time of day. Finally, a big design decision is the level and ways of human interaction with the system. This may be as simple as a central control console or it can be done with multiple ways such as portable devices, voice recognition, and remotely over the Internet, or by phone. With exuberant control on a small console, there exists a risk of a faulty behavior by the system. In this section,

we present with some of the necessary control specifications of the Smart Home control system.

4.1.1 Natural Language Specification

The example of a Smart Home we are describing here includes all basic functionalities for lighting, temperature control, security, and safety as well as some extended features for advanced control. We describe the functionalities separately for the different categories although it is not possible to do a clear separation, as some functions require interaction of several categories.

Lighting:

Lighting control illuminates rooms depending on the brightness outside the house. There are light sensors at every window. If during the day it is brighter than a pre-defined value, light in the room will switch off. Once it gets dark, the light will switch on. Obviously, rooms without windows such as the bathrooms do not have light-sensors. Every person carries an RFID (Radio Frequency identification) in their jewelry or shoes to detect the presence of a person, and the lights in the house are controlled depending on the presence of persons in the rooms. If a room has not been used for more than 10 minutes, light will switch off. In addition, for some rooms such as the living rooms and the bedrooms, the resident can choose a lighting intensity from 0 to 100 percent. Whenever the room is used again, light switches to the previous selected intensity. However, the next evening, as the light switches on when it is getting dark outside, it always starts with 100 percent intensity.

Temperature Control:

Temperature control involves temperature sensors in every room, and the control of heating and air conditioning. For each room, three temperature levels can be defined: comfort level, power-save level, and the vacation level. The comfort level is the temperature the resident likes to have when the room is being used. The second level is the power-save level that defines the temperature for a room that is not being used. It saves power by lowering AC and heating, but still keeping the temperature at a level so that the room can reach comfort level in a reasonable time. The vacation level defines the temperature for a room that is not used for a longer period. In the vacation level, only heating is operated to prevent the room from freezing. Whenever a person is detected in a room, the system goes to comfort level. If a room has not been used for a certain time for example 2 hours, temperature will be lowered to power save level. If the room has not been used for say more than two days, it will fall back to vacation mode.

Security:

Security uses motion sensors in every room to detect if someone is present in a room or not. If there is motion detected but no person of the household identified in the past 5 minutes, an alarm signal is issued. When the alarm signal is issued, the safety alarm starts and a message is sent to the resident's cellular phone along with an email. If the alarm is not turned off within next 5 minutes, a signal is sent to the security for help.

Safety:

Safety is dealing with malfunctioning devices that may cause unforeseen hazards to the house such as fire, flooding etc. It may also involve mistakes made by the residents that may cause such an event. One of the main components of safety is the smoke detectors, which are installed in every room to monitor smoke levels. In addition, it checks if the temperature is in a safe range. Once a certain safety constraint is violated, a safety alert is raised and displayed on the screens available in every room. In addition, a message is sent to the resident's phone and email box. If a hard safety constraint is violated, fire-extinguishing sprinklers are activated and the fire department is notified.

Table 4.1: LTL Properties for the Smart Home model

1	If light is over 500 lumen light will not switch on	[]($klact \rightarrow kbright$)
2	If the room is not used for more than 10 minutes, the light is always off	[]($kaway10m \rightarrow !klit$)
3	If there is light in the room, either it is sufficiently dark outside and someone is using the room or someone has been in the room in the last 10 minutes	[]($klit \rightarrow (khere10m \&\& klact)$)
4	If the light is off, either it is sufficiently bright already, or it is just getting sufficiently bright or nobody has been in the room recently.	[]($!klit \rightarrow (!klact \| \| X\ !klact \| \| kaway10m)$)
5	If no identified person is in the room and there is motion the alarm is triggered	[]($(((kempty \&\& kmotion)\ U\ alarm) \| \| !kmotion \| \| (kmotion \&\& !kempty))$)
6	Temperature is at comfort level if the room has been used within the last 15 minutes	[]($ktcomfort \rightarrow khere15m$)
7	Temperature is at power save level if the room has not been used for more than 2 hours.	[]($ktpowersave \rightarrow kaway2h$)
8	Temperature is at vacation level if the room has not been used for more than 2 days.	[]($ktvacation \rightarrow kaway2d$)
9	At dawn (when the light gets active) intensity is always at 100 percent.	[]($((!llact\ U\ lintmax) \| \| llact)$)
10	If there is some smoke or a high temperature in any room next a firehazard is issued	[]($(lowsmoke \| \| temphigh) \rightarrow X\ safetyhazard$)
11	If there is heavy smoke or a very high temperature in any room next the sprinkler system is started	[]($(heavysmoke \| \| tempveryhigh) \rightarrow X\ spkl$)

Table 4.2: Definitions for the LTL Properties

1	#define kbright klum < 500	Brightness level in the kitchen is under 500 lumen
2	#define klact klactive	True if kitchen light can be switched on, false if bright enough
3	#define klit klight	Output for the kitchen light.
4	#define ktcomfort ktl==comfort	Temperature in the kitchen set to comfort
5	#define ktpowersave ktl==powersave	Temperature in the kitchen set to power save
6	#define ktvacation ktl==vacation	Temperature in the kitchen set to vacation
7	#define kempty kpsens==Empty	No person is in the kitchen
8	#define alarm alert	
9	#define khere15m kcount<16	
10	#define khere10m kcount<11	Kitchen has been used within the last 10 minutes
11	#define kaway10 kcount > 10	Kitchen is unused for more than 10 minutes
12	#define kaway2h kcount > 15	Kitchen is unused for more than 2 hours
13	#define kaway2d kcount > 20	Kitchen is unused for more than 2 days
14	#define kpsense !(kpsens == Empty)	No person detected via RFID
15	#define kmotion kmot	The motion detector
16	#define lowsmoke ((ksmoke==LOW)||(lsmoke==LOW))	There is some smoke in a room
17	#define heavysmoke ((ksmoke==HIGH)||(lsmoke==HIGH))	There is heavy smoke in a room
18	#define temphigh ((ktemp > 80)||(ltemp > 80))	The temperature in a room is unusually high
19	#define tempveryhigh ((ktemp > 100)||(ltemp > 100))	The temperature in a room is very high
20	#define spkl sprinkler	The sprinkler system
21	#define lintmax lint == 100	Full light intensity in the living room

4.1.2 Capturing of the Formal Model

In order to capture the formal model we identify basic functionalities of the system, specify formal linear time properties for the functionalities and then build a model that satisfies those properties. Our first property states that if it is sufficiently bright outside, then the light will never switch on in the room. Table 4.1 shows the corresponding LTL property. To make sure that this property expresses what we intend to express, we enter the LTL formula into the LTL 2 BA tool to get the corresponding automaton. Figure 4.1(a) shows the FSM corresponding to property 1 and Table 4.2 lists the definitions necessary to verify it with SPIN. Since this property is quite simple it is not difficult to see, that

it expresses the correct behavior, but for more complex properties it is important to make sure they are correct before checking the model for it. Now, we start writing a model in PROMELA that satisfies exactly this one property. It is important to try not to introduce functionality early; however, the model always contains parts that have no correspondence in a formal property. This is because certain things such as initialization of variables and changes of state variables cannot be expressed in linear time properties. Listing 4.1.2 shows the PROMELA model for the first property. It features the initialization process *init* and the process *KLuminosity*, where the variable *kactive* is set according to the current luminosity. In order to keep the state space of the model small, we only consider three levels of brightness. All other values will not result in any change of *kactive*.

Listing 4.1.2: Promela code for the first property

```
bool kactive;
int klum;
proctype KLuminosity() {
    do
        :: klum < 400 → klum = 200; kactive =1
        :: (100 < klum) && (klum < 700) → klum = 400; kactive =0
        :: (klum > 700) → klum = 600; kactive =0
    od;
}
init {
    klum = 200;
    kactive =1;
    run KLuminosity();
}
```

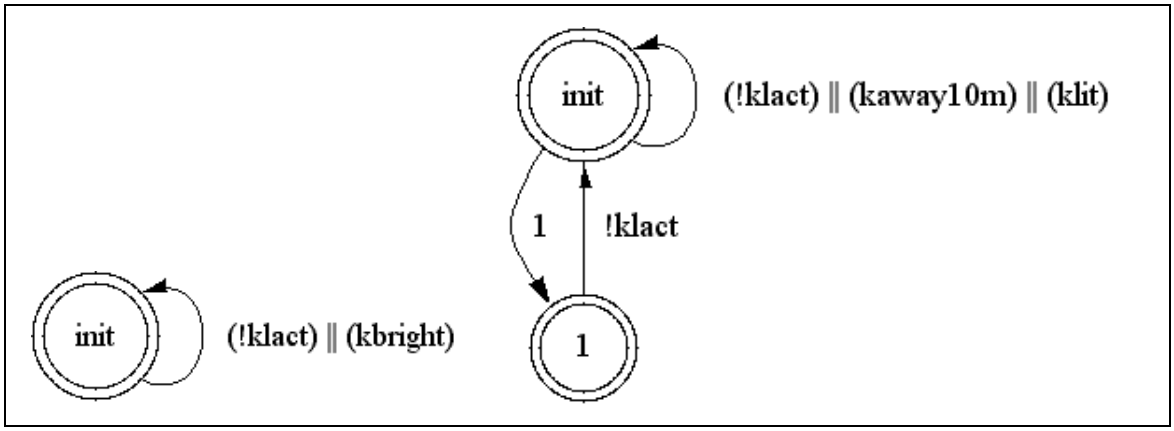

Figure 4.1: FSMs of properties 1 (a) and 4 (b)

Only after the first property is verified, we come up with a second property. This property states that if the room has not been used for more than 10 minutes, the light is always off. The LTL property in Table 4.1 is similar to property 1. A process *KPsense* has to be added to the model that detects if a person is in the kitchen and it counts the minutes since the kitchen has been used last. If nobody has been there for more than 9 minutes, the light is switched off.

Property 3 describes that if there is light in the kitchen, it must be sufficiently dark outside and someone used the kitchen within the last 10 minutes. The LTL property is quite easy to understand. For the formal model, we only have to add small changes to reflect this property. We have to make sure that the light is dependent on *kactive*. Once these changes are updated, the property verifies and we determine the next property. Property 4 defines when the light is off in the kitchen. If the light is off, that means, that either nobody has been using the room recently or it is sufficiently bright without light. This property is a good example for the interactive model building process. While implementing the changes in the model we realize that when it is just getting bright in the morning, the light is switched on just before *klactive* is switched off. This will cause the property to fail. Now if we inverse the assignments, property 3 will fail.

This is why we extend the property including instants where in the next state *klactive* is true. To make sure, that this actually happens in the next state we have to enclose the two assignments in an *atomic* statement. Figure 4.1(b) shows the corresponding FSM generated by LTL 2 BA. It has very few transitions but confirms that the automaton follows our intention. When building the properties independent from the model, such details get omitted. This will cause the property to fail and it may be time consuming to locate the error.

As the model now contains most of the lighting functionality, we add a security property next. Property 5 exploits the motion sensors. When no person is identified in the room with a valid RFID and there is motion detected an alarm is activated. The alarm signal might ring a bell and send a text message to the resident's cell phone and email. As the alarm signal can only occur after the condition is detected, it is most appropriate to use the until operator U for this property. Table 4.1 shows all the LTL properties. The implementation in the formal model does add some lines to the *KPsense* process, but it does not incur any fundamental changes.

Next, we consider the temperature control mechanism. As there are three basic properties that are similar and closely related, we decide to add all three properties in one iteration step. The properties say that the AC/Heating system has three modes of operation, each having a temperature level assigned. The first level is the comfort level; it defines the temperature when the room is used. Then there is a power save temperature that is active whenever nobody has been using the room for more than two hours. Finally, a vacation mode is active when a room has not been used for more than two consecutive days. The resulting LTL properties are simple (Property 6, 7, 8). In the PROMELA model, there is already a counter that counts the time since the room has been used last so we have again few changes in the model. After modifying the model, small alterations have to be performed when checking all previous properties.

At this point, our model only comprises one room, the kitchen. As we have sensors for every room, most of the control is also based on the room-level. All properties mentioned can be replicated for any other room as well. The same is valid for the model. This makes our model modular in approach. However, in some rooms there are additional functions that should be added. For example, we want to control the intensity of the light in the living room and in the bedrooms. Intensity in the bedroom will be set to zero, when going to bed and in the living room it may be set to a desirable level. The fact alone to have a switch that controls the intensity will result only in trivial LTL properties, and therefore is not worth checking for. But in the specification, it says, that if it gets dark the next day, intensity should always be at 100. This requirement is expressed in LTL property 9 (see Table 4.1). To reflect this in the model, we create the model of the living room by copying all kitchen processes and rename them and the variables. In addition to *KPsense*, we get *LPsense* and so on. Then we replicate all LTL properties and the definitions and check all these properties as well as the ones for the kitchen. Once this is done a new process *LIntensity* is added, that switches between different levels of intensity if it is sufficiently dark in the living room. In order that property 9 verifies, we change the intensity back to 100 as soon it gets bright enough outside, as *lactive* is turned to zero.

Safety properties such as the detection of smoke or hazardous temperatures are properties that concern the whole house. In order to ensure the safety in the house, we add two properties that depend on the smoke detectors and on the temperature sensors. Property 10 issues a safety hazard when a certain amount of smoke is detected or if the temperature in any room is high or low. A safety hazard will notify the resident about the incident but not take any immediate action yet. Once the intensity of the smoke reaches another critical level, the sprinkler system is launched and the fire fighters are notified. This is specified by property 11. In the model, these changes induce the creation of two

small processes that monitor the temperature and smoke sensors in all rooms.

4.1.3 Summary

We demonstrate the usage of the XFM methodology for the formal modeling of a large control application for a smart building. The modeling example of the Smart Home demonstrates the power of the approach, and even if there exist more sophisticated smart spaces, we succeeded in building a reasonably complex smart environment with little effort. Yet most important is not the large amount of functionality and the different levels of interaction, but the confidence that this model fully complies with the specification without containing much superfluous behavior.

Chapter 5

Property Ordering Effects on XFM

In this chapter, we analyze the effect of ordering the linear time properties while using Extreme Formal Modeling (XFM) methodology in building "Prescriptive Formal Models" (PFMs). During incremental model building, the PFMs often blow up in size in terms of the state space, and the main tenet of XFM being regressive model checking, blown up models often make it impossible to carry out the XFM methodology. In this section, we compare three different model building methodologies, (i) No specific ordering of user stories (arbitrary), (ii) Sorting of the user stories based on a weighting scheme (property based sorting), and (iii) Predicate based sorting of user stories based on an eliminative scheme (predicate based sorting). We show that the predicate based sorting scheme is the most effective way to carry out XFM model building. We illustrate the schemes and the comparison by modeling a monitor for the ISA bus [54] and model of the arbitration phase of Pentium Pro processor's bus [53] using SMV [47]. We also provide formal representation of the proposed schemes for ordering the properties for model building.

5.1 Preliminary Definitions

Let $X = \{x_1, x_2, ..., x_l\}$ the set of variables in the system being modeled. Let $D(x_i)$ denote the domain of variable x_i. In most cases $D(x_i) = \{0, 1\}$, when x_i is a Boolean variable. Let us call $x_i = v$ and $x_i \neq v$ where $v \in D(x_i)$ as predicates of our system. In other words, in every state of the system, for each variable x_i, one can evaluate these predicates easily. It is easy to see, that the number of such predicates in the system is $\Sigma_{x_i \in X}(2 \cdot \mid D(x_i) \mid)$, which is finite for finite variable set X, and below we will indicate this number by n.

Now let $P = \{p_1, p_2, ..., p_n\}$ be the set of all such predicates and $\Pi = \{\pi_1, \pi_2, ..., \pi_m\}$ are the set of properties to be modeled.

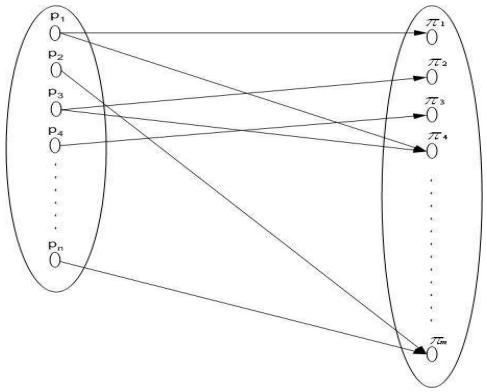

Figure 5.1: Predicate-Property Representation

The relationship between the properties and predicates can now be represented by a bi-partite graph, where one set of nodes are marked with the predicates, and another set of nodes are marked with the properties. Edges go from a node marked with predicate p_i to a node marked with a property π_j if the predicate p_i is used in the property π_j. We write this condition in short hand as $p_i \in \pi_j$ Such a bipartite graph to represent the predicate-property relationship is shown in Figure 5.1.

Table 5.1: The Predicate-Property Representation Graph of the different Behaviors of a System

	π_1	π_2	π_3	π_4	.	.	π_m	$\hat{\delta}(p)$
p_1	1	1	0	0	.	.	.	4
p_2	0	0	0	1	.	.	.	1
p_3	1	1	1	1	.	.	.	6
p_4	0	1	1	0	.	.	.	4
.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.
p_n	0	1	1	1	.	.	.	4

Definition 5.1.1 Predicate-Property Relationship Graph: *A bipartite graph* $G = (P \cup \Pi, E)$ *where* $P \cap \Pi = \emptyset$, $E \subseteq P \times \Pi$ *s.t* $(p_i, \pi_j) \in E$ *iff* $p_i \in \pi_j$, *is named a* **Predicate-Property Relationship Graph (PPRG)**.

Definition 5.1.2 Weights of Predicates and Properties: $\delta(p_i)$ = *out degree of node marked* p_i *in G is called the weight of the predicate* p_i. *It indicates how many times in different properties* p_i *appears. We denote this by* $\digamma(p_i)$ *or by* $\delta(p_i)$ *interchangeably. The weight of a property* π_j *is now defined as* $W(\pi_j) = \sum \delta(p_i)$ *where* $(p_i, \pi_j) \in E$, *and the weight of a property is a measure of* entanglement *of a property with other properties.*

Definition 5.1.3 PP Matrix: *The Predicate-Property Representation Graph (PPRG) can also be represented in a matrix form as shown Table 5.1. Such a matrix denoted by PP is a* $n \times m$ *matrix, with* A_j *being the* j^{th} *column vector.*

The following observations can be made easily about *PP* matrix.

Observation 5.1.4 *The i^{th} row element of column vector A_j of PP is* 1 *if and only if predicate p_i appears in property π_j.*

Observation 5.1.5 *The j^{th} element of i^{th} row of the matrix PP is* 1 *if and only if predicate p_i appears in property π_j.*

If we denote A_i^T as the i^{th} row of *PP* in column format then we can observe that

Observation 5.1.6 $\delta(p_i) = A_i^T \cdot \vec{1}$ *where $\vec{1}$ a column vector of size $m \times 1$ with all 1's.*

Let $\hat{\delta}$ denote the column vector of size $n \times 1$ containing the δ values for each predicate.

Observation 5.1.7 *The weight of a property π_j is obtained as*

$$W(\pi_j) = A_j \cdot \hat{\delta}$$

Now if we define $A_j \leq A_k$ as equivalent to $A_j[i] \leq A_k[i]$ for all i where $A_j[i]$ denotes the i^{th} row element of A_j, then we can observe the following:

Observation 5.1.8 $A_j \leq A_k$ *implies that* $W(\pi_j) \leq W(\pi_k)$.

5.2 Importance of Ordering

XFM is an incremental modeling approach with the model expanding in the state space after addition of properties at each increment. It is important to focus on the order of modeling the properties because if the state space blows disproportionately, then using

XFM will become difficult. At times, arbitrary ordering of the properties blow up the models in the middle or in early stages of XFM making it hard to carry on XFM. Our goal is to choose the properties in a way such that the state space reached during verification of the model is incrementally growing and within bound. Therefore, a scheme is required to order the properties which would be modeled to get an incremental representation as we keep verifying remaining properties.

One of the important factors in building an abstract model for formal verification is to make sure that it does not contain any unnecessary behavior. This behavior may get included in the updated model when it is not seen as an incremental approach but rather as unstable in terms of state space searched. Another important factor for a correct incremental approach is to achieve a model that results in a smaller state space for the verification of a property. With a smaller state space, verification would be faster and more efficient. Both these concerns are resolved by modeling using the XFM approach with a consistent ordering of properties. Our goal of XFM does focus on the fact that we want to satisfy the property with the least changes in the model from its previous increment.

Listing 5.2: Property based ordering

Let $\Pi = \{\pi_1, \pi_2, .., \pi_m\}$ be the set of properties and the set has size m

Let $P = \{p_1, p_2, ..., p_n\}$ be the set of predicates and this set has size n

Let $F(p)$ be the frequency of the predicate p in the system. Note that $F(p) = \delta(p)$ in PPRG

{The weight of the property will be calculated by the following segment}

for all $\pi_i \in \Pi$

for all $p_j \in P$
if $(p_j \in \pi_i)$
$W(\pi_i) = W(\pi_i) + F(p_j)$
end if
end for
end for

{In property based ordering, we sort the properties according to their weights. We start modeling the property with the least $W(\pi_f)$. Also, note that if multiple properties have the same weight, we order them in any order} {Note that from PP matrix, one can compute $W(\pi_f)$ as $A_f \cdot \hat{\delta}$ where $\hat{\delta}$ is the column vector with the predicate frequencies.}

We have experimented with different property ordering schemes while doing XFM to construct the model. The first scheme is the "standard approach" which we call Arbitrary ordering, where properties are modeled in a random order. This approach may cause the model to blow up initially and remain the same throughout or may cause the model to decrease in size in between. The arbitrary sorting scheme is not a good approach to select properties for modeling with XFM due to the lack of the notion of property dependencies and the weight of its predicates. While the predicate dictates a specific behavior of the system, the property conveys a relation or dependence between the predicates. Therefore, in order to achieve an incremental and stable model, the properties need to be ordered in some scheme such that the properties that are dependent on the same set of predicates are modeled in close proximity of the ordering sequence.

Secondly, we have the property based ordering sequence, which is based on the weight

of each property. We use a predicate-property matrix to represent the system and the summation of all the predicates in a property gives the total weight of the property. Once we have the weight of all the properties, we sort them in ascending order such that we have the property with the lowest weight on top of the ordering list. Properties can have same or different weights depending on the frequency of the predicates in all the properties. In the property based ordering, we start to construct the model based on the least weighted property, which is less "dependent" in terms of frequency of that predicate in the other properties. We continue formulation till all the properties in the property list are modeled. Listing 5.2 shows the formal representation of the property based ordering approach.

Finally, we have the predicate based Sorting scheme, in which we do an elimination of the properties based on the frequency count of the predicates and dynamically update the predicate-property matrix to reflect the absence of the property. We set the column corresponding to the eliminated property to zero's and reduce the weight of the predicates in the other properties which were dependent on this property (since they shared the same predicate). We continue formulation till we have null matrix. The order of elimination will generate the sorted sequence.

Listing 5.2: Predicate based ordering

Let $\Pi = \{\pi_1, \pi_2, ..., \pi_m\}$ be the set of properties, and this set has size m
Let $P = \{p_1, p_2, ..., p_n\}$ be the set of predicates, and this set has size n
/* Note that $W(\pi_f) = A_f \cdot \hat{\delta}$ where A_f is the column vector corresponding to property π_f in PP matrix */

Let $L = \emptyset$

/* L is an ordered list of properties initially empty */

while Π is not empty

for all $p_i \in P$, such that $F(p_i)$ is minimum

Obtain $D = \{\pi^1_{min_p}, \pi^2_{min_p}, ..., \pi^t_{min_p}\}$, where min_p is the set of predicates with least $F(p_i)$ in P and one or more predicates from min_p appears in all $\pi^i_{min_p}$ in D

end for

Select π_j from D such that $W(\pi_j)$ is minimum

/* If there are multiple such π_j, select one randomly*/

Remove π_j from Π such that $\Pi = \Pi - \{\pi_j\}$

Add π_j to the end of list L

Update all $F(p)$ for all p's in π_j

/* Effect of removing π_j is equivalent to deleting the column A_j from PP matrix and updating the $\hat{\delta}$ such that $\hat{\delta} := \hat{\delta} \ominus A_j$ where $\ominus$ indicates element by element subtraction between two vectors */

for all remaining $\pi_i \in \Pi$

Update $W(\pi_i)$ with new $\hat{\delta} \cdot A_i$

end for

end while

/* The ordered list L now contains the predicate based order to be used for XFM Modeling */

In this scheme, we model the property containing the least frequent predicate which is

a behavior in the system described by that predicate, which is least dependent on the other behaviors. Once the above behavior is modeled, that portion of the system may not to be modified for other properties. Therefore, by modeling using a predicate based ordering, we model in a more incremental way. It was found to be the most effective of the property ordering schemes for XFM. Listing 5.2 shows the formal representation of predicate based ordering approach.

Consider an example, where we have a system whose behaviors are expressed as Linear Time properties:

$\pi_1 : \mathbf{G}(A \;\&\&\; B)$
$\pi_2 : \mathbf{G}(A || C \;\rightarrow\; \mathbf{F}(D))$
$\pi_3 : \mathbf{G}(A \;\rightarrow\; C \;\mathbf{U}\; B)$

The **Predicate-Property Representation Graph** and the resultant **Sorting schemes** for the example are shown in Table 5.2 and Table 5.3 respectively. According to the PPRG, we see that D is the least occurring predicate where as the weight of π_1 is the least. Therefore according to the property-ordering scheme, we model π_1 first. Then, we consider the other two properties. Since, both π_2 and π_3 have same weights, we randomly choose to model one of them. On the other hand, for the predicate based ordering, we consider π_2 to be modeled first since it contains the least weighted predicate. After π_2 is selected, we then update the frequencies of other properties with respect to the predicates modeled by π_2. We get C as the least frequent predicate after D is removed. Therefore, we model π_3 that contains predicate C and finally model π_1.

We later show our results from modeling the ISA bus architecture as well as the arbitration of the Pentium bus for the three distinct approaches to XFM.

Table 5.2: The PPRG for the example

	π_1	π_2	π_3	$\delta(p_i)$
A	1	1	1	3
B	1	0	1	2
C	0	1	1	2
D	0	1	0	1

Table 5.3: The Ordering for the example

property based ordering	predicate based ordering
π_1	π_2
π_2	π_3
π_3	π_1

Before we show our experimental evidence to illustrate the relative merits of all the different sorting orders, let us look back into the PPRG and understand the differences between the different sorting orders. Our goal for ordering properties is two fold: (i) To keep the state space search as low as possible in the incremental model building process, as long as we can, (ii) To keep the changes needed at later stages of the model building on the parts which were built earlier.

We realize that to achieve these two goals, we need to make sure that the properties that have the least entanglement with other properties must be modeled first, so that we can minimize changes on parts modeling these early properties. We also realized that if we model a property, and the next property we model has more predicates in common with this property, the changes in the state space would be minimal since each predicate may introduce new states. In other words, if we model a property with a collection of predicates, and then we model another property, which has very little overlap in

predicates with the last modeled one, we have to introduce lot more new states in most cases.

In the following series of observations, lemmas and theorems we establish a crucial characteristic of the predicate based ordering. We show that in predicate based modeling, usually when a sequence of properties is modeled they share predicates, and therefore, the state space does not increase drastically that often from one step to the other. In the property based scheme, since the modeling is done based on property weights, two subsequently modeled properties may not share any predicates, and hence state space may go up drastically, which we will see in the experimental evidence that we present later.

In the following, we assume P_{min} denote the set of predicates with the least occurrences among the properties not modeled yet. In other words $P_{min} = \{p \mid \delta(p) \leq \delta(q) \forall q \in P\}$. At every iteration of the predicate based sorting, this set P_{min} is updated, since δ values for predicates change based on what property is modeled at that iteration.

Observation 5.2.1 *At the end of every iteration, for any predicate p, $\delta(p)$ either reduces by 1 or remains the same.*

Proof sketch: Recall that if π_j is the property modeled in an iteration, then at the end of the iteration $\hat{\delta}$ becomes $\hat{\delta} \ominus A_j$, and all entries in A_j are either 0 or 1.

Lemma 5.2.2 *If $\delta(p) > 1$ and $p \in P_{min}$, then p remains in P_{min} in the next iteration if and only if p was in the modeled property in the current iteration.*

Proof sketch: If the modeled property contains p, then $\delta'(p) = \delta(p) - 1$, and since $\delta(p)$ was the smallest, and no other property's δ value cannot reduce by more than 1, p still

remains among predicates whose δ value is minimum. In the other direction, if $\delta(p) > 1$ and it is one of the minimum, and if the modeled property does not contain p, then by the algorithm, the modeled property has to contain $q \in P_{min}$, and hence $\delta(q)$ will become smaller than $\delta(p)$ and p will leave P_{min} in the next iteration.

Observation 5.2.3 *If $\delta(p) = 1$ and if one of the properties selected to be modeled contains p, then p is never again needed to be modeled.*

Lemma 5.2.4 *if $\delta(p) = 1$, then always $p \in P_{min}$ until a property containing p is modeled.*

Proof sketch: Let us assume that there is $q \in P_{min}$, then also $\delta(q) = 1$ since $p \in P_{min}$ and $\delta(p) = 1$. If q is selected to be modeled, then $\delta^{'}(q) = 0$ and it will removed from P_{min}, p will remain since it has the least $\delta(p)$. If p also was in the modeled property, then again the condition of the lemma is met.

Lemma 5.2.5 *Once a P_{min} is chosen, P_{min} may lose some predicates from one iteration to the next, but will not gain any new predicates, until the current P_{min} becomes completely empty.*

Proof sketch: By definition of P_{min}, for all $q \notin P_{min}$, $\delta(q) > \delta(p)$ for all $p \in P_{min}$. Now, when a property is modeled in the current iteration, it must contain at least one $p \in P_{min}$, and hence its δ value will decrease. So that p will certainly stay in P_{min} through the next iteration. Now if $p' \in P_{min}$ is not in the modeled property, it's δ value will remain unchanged, and so it will be removed from P_{min}. However, since for all $q \notin P_{min}$, the reduction in δ value is maximum 1, they cannot enter P_{min}.

Observation 5.2.6 *If P_{min} has more than 1 entry, and some p is chosen, and $\delta(p) > 1$, and π_p is the property being modeled, then in the next iteration all $q \notin \pi_p$ goes out of P_{min}.*

Lemma 5.2.7 *If P_{min} is singleton, then until all properties containing that predicate are modeled, no new P_{min} can be constructed.*

Proof sketch: Follows from the previous lemma.

Observation 5.2.8 *The size of P_{min} either decreases from one iteration to the next or remains the same until a property is chosen to be modeled such that it does not contain all the predicates currently in P_{min}. In either case, until $P_{min} = \emptyset$, no new predicate enters P_{min}*

The time we choose a fresh new P_{min} till it becomes *empty*, let us call that set of iterations a *phase* of the modeling. Therefore, the entire predicate based sorting ordered XFM goes through these phases. Based on the previous lemmas and observations, we can now claim the following:

Theorem 5.2.9 *In a single phase of the modeling, all properties that are modeled overlap in some predicates. Also, in each phase, for at least one predicate, all properties containing that predicate are modeled in a single phase.*

This theorem, points out that the state space increase in this methodology is more controlled, because if we model properties that have predicates common with the previously modeled properties, we are likely to introduce less states in the next increment of the model, as some predicates of the new property are already modeled.

Given this understanding of predicate property relationships, we anticipate that a small change in the property based modeling can help control its state space growth pattern as well. The idea is to choose properties based on how entangled they are with the previously modeled property. In the approach, we recalculate the property weights

after each iteration. The following Theorem indicates an improvement possibility in this approach as entanglements play a role in property sequencing.

Remember Π_i denotes the set of properties until π_i.

Theorem 5.2.10 *If modeling property π_i is immediately followed by modeling of property π_j in the original property based ordering, then in the improved property ordering π_l where $l \neq j$ will be modeled immediately after π_i, if and only if Π_i and Π_l has more entanglement (common predicates) at that point of the XFM process than Π_i and π_j*

Proof sketch: Let $\Pi_i \cap \pi_j$ denote the entanglement between Π_i, and π_j indicating set of overlapping predicates between the two. $\mid \Pi_i \cap \pi_j \mid$ denote the size of the overlap.

Now suppose π_i is followed by π_j in the original property based ordering. That means $W(\pi_j) \leq W(\pi_l)$ (by the property based ordering algorithm). Now in predicate based ordering, this must have changed to $W'(\pi_j) > W'(\pi_l)$, where W' denotes the adjusted weight of the two at this point of the XFM process. In predicate based ordering, every time a property is modeled, weights of predicates in that property are reduced from the weights of all the properties that contained them. So at this point the $W'(\pi_j) = W(\pi_j) - \mid \Pi_i \cap \pi_j \mid$ and $W'(\pi_l) = W(\pi_l) - \mid \Pi_i \cap \pi_l \mid$. Hence $W(\pi_j) - \mid \Pi_i \cap \pi_j \mid > W(\pi_l) - \mid \Pi_i \cap \pi_l \mid$, which implies that $\mid \Pi_i \cap \pi_l \mid - \mid \Pi_i \cap \pi_j \mid > W(\pi_l) - W(\pi_j)$. Now from the original property based ordering we know $W(\pi_l) - W(\pi_j) \geq 0$, and hence $\mid \Pi_i \cap \pi_l \mid \geq \mid \Pi_i \cap \pi_j \mid$. This means that π_l has atleast as much entanglement with the previously modeled properties than π_j has. In fact, if the equality is broken by number of predicates in property based ordering, in most cases it will be more entanglement.

Experimentally it was found that predicate based ordering is the most effective of the property ordering schemes for XFM. In the next section, we show our results from modeling the ISA bus architecture as well as the arbitration of the Pentium bus for the

three distinct approaches to XFM.

5.3 Case Study for the model

5.3.1 ISA Bus Architecture

The XFM rules are applied on the monitor model of the ISA bus architecture to get a compact model. Following the XFM guidelines, we first wrote the user stories of the ISA pipeline from the natural specifications dictating the behavior of the ISA bus protocol. The user stories are converted into a sequence of linear time properties, which are then ordered using one of the previously explained schemes to model the system in an incremental approach, thereby formally verifying the bus protocol.

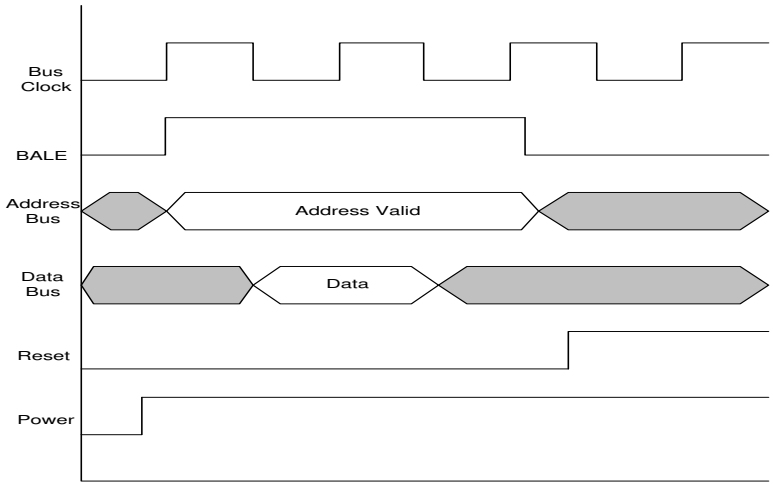

Figure 5.2: ISA bus timing diagram

Model

The model of the ISA bus architecture is one of the basic models of the first IBM PC. The main component of the ISA bus architecture is the expansion bus. The expansion bus interfaces the memory with the I/O cards. The model of the ISA bus protocol is based on the signal specifications specified in [54]. Based on the functions, the signals are grouped into address signals, data signals and the control signals. The bus clock drives the ISA bus and brings in the notion of timing. The timing diagram of the ISA bus is shown in Figure 5.2.

When the unit is initially powered up, the reset signal on the ISA bus remains asserted until the power supply voltages have stabilized. Also the ISA cards are prevented from functioning until the power has stabilized. When a bus cycle is initiated by the CPU, the target address is placed on the address bus once the Buffered Address Latched Enable(BALE) signal appears on the bus. The BALE signal is used to indicate that the address has been successfully decoded. Once the address becomes valid, the data transfer proceeds either on the upper or lower paths of the data bus based on whether it is an 8 bit or 16 bit expansion card that initiated the bus cycle with or without the System Bus High enable asserted. The specification of the ISA bus along with the corresponding properties is illustrated Table 5.4.

The properties of the ISA bus were sorted in accordance with the schemes mentioned earlier and an incremental model was constructed using the XFM approach. The result of the state space search for the properties are mentioned in section 5.4.

Table 5.4: LTL Properties for the ISA Bus Model

1	When the system powers up, all bus signals are reset.	G(power → X !reset)
2	The cards are prevented from doing anything until power stabilizes.	G(power → X isacards)
3	The BALE signal occurs once in a bus cycle.	G(balelock → baleon) G(balelock U !endbuscycle) G((baleon && power && !endbuscycle) → X balelock)
4	At the end of the bus cycle, the address is not valid, the bale signal is not high and data transfer is not complete.	G(endbuscycle → X(!addressvalid && !baleon && !dtstart && !balelock && !dtcomp))
5	When ever the data transfer takes place, the address is valid	G(dtstart → addressvalid)
6	When the device is powered and data transfer is complete, then current bus cycle ends in the next bus clock.	G((dtcomp && b_clock && power) → X endbuscycle)
7	When the a 8 bit device is powered and if the data transfer can start in the same bus cycle, then the lower path of the data bus will be used to transfer the data.	G((isacardselect && dtstart && power && !endbuscycle) → X lowerpath)
8	When the a 8 bit device is powered and if the data transfer can start in the same bus cycle, then the upperpath of the data bus will never used to transfer the data.	G((isacardselect && dtstart && power && endbuscycle) → X !upperpath)
9	When the 16 bit device is powered up and the data transfer can start in the same bus cycle to an even addressed location along with the high enable signal set low, then the lower path of the data bus will be used for transfer.	G((address && isacardselect && dtstart && power && !endbuscycle && !highen) → X lowerpath)
10	When the 16 bit device is powered up and the data transfer can start in the same bus cycle to an even addressed location along with the high enable signal set high, then the upper path of the data bus will be used for transfer.	G((!address && isacardselect && dtstart && power && !endbuscycle) → X upperpath)
11	Data transfer completes after Data transfer starts.	G(dtcomp → dtstart)
12	When the power off signal is received then balelock, baleon, data transfer, bus clock, buscycle is reset and address is not valid.	G(!power → X(!baleon && !dtstart && !balelock && !isacards && !b_clock && !dtcomp && !endbuscycle && !addressvalid))
13	When ever the address is valid, then the baleon signal has already arrived.	G(addressvalid → baleon)
14	When the 16 bit device is powered up and the data transfer can start in the same bus cycle to an even addressed location along with the high enable signal set high, then the upper path of the data bus will be used for transfer.	G((highen && isacardselect && address && dtstart && power && !endbuscycle) → X upperpath)

5.3.2 Pentium Pro Processor's Bus Arbitration

One of the most important concern of the Pentium Pro bus is how it handles arbitration between its symmetric and priority agents [53]. Before a request agent can issue a new transaction to the bus, it must arbitrate for and win ownership of the request signal group. Once ownership has been acquired, the request agent initiates the request phase of the transaction. The arbitration diagram is shown in Figure 5.3.

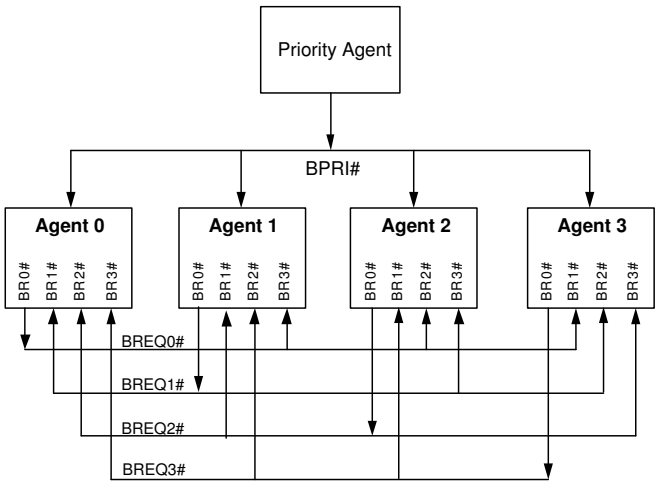

Figure 5.3: Pentium Bus Arbitration

The Pentium bus arbitration model captures the behavior of the bus arbitration. Some of the behavior handled includes the arbitration among the symmetric agents, arbitration by the priority agents and its affect on the symmetry agents. Also, the user stories incorporate the behavior when the symmetric agent locks the bus and a priority agent is giving a request. The Pentium architecture of arbitration contains four symmetry agents and a priority agent, where a symmetric agent is a processor capable of handling any task. At a given instance in time, one or more of the processors may request ownership of the request signal group in order to communicate with an external device. The bus arbitration decides which of the processors gets the ownership next based on a built

in rotational priority assignment scheme for which each of the processors always keeps track of whether any of them currently owns the signal group, and which of them owned the group last or still owns it and which of them gets to use it next. In order for them to track this information, each agent knows its own agent ID as well as the agent ID of the processor that last gained ownership of the request signal group.

The arbitration event, which is the passing of the ownership from one symmetric to another, occurs under the following circumstances:

- *When none of the agents are requesting during one clock and then one or more are seen in the next clock.*
- *When the current symmetric owner of the request signal group relinquishes ownership of the bus and one or more of the other agents having been requesting for the bus.*

In either case, the symmetric agents must collectively decide which of them will assume ownership of the request signal group in the next clock. Each symmetric agent has a *Rid* and a bus state which are their internal states used for keeping track of the bus status and the ID of the current owner of the bus.

In the Arbitration event, when only one of the symmetric agents is requesting for the bus, it wins the arbitration and gets the ownership of the request signal group.

User Story : If only agent2 is requesting the ownership of the bus then it is given the ownership of the bus and all the other agents will update their rotating ID to '2', irrespective of who was the previous owner.

$G(!breq0 \;\&\&\; !breq1 \;\&\&\; breq2 \;\&\&\; !breq3 \;\&\&\; Arbit \;\&\&\; X\; !reset \rightarrow X\; Rid\bar{2} \;\&\&\; X\; busstate)$

Whereas, if two or more of the symmetric agents are requesting for the bus, one of them wins the arbitration and gets the ownership of the request signal group based on who had the ownership previously. The sequence in which the processor gains ownership is based on rotating IDs: 0, 1, 2, 3, 0, 1,... (Agent ID's).

User Story : If agent2 and agent3 are requesting for the bus and if agent 0 had the ownership of the bus previously then agent 2 wins the arbitration and gains ownership of the bus.

$G(!breq0 \;\&\&\; !breq1 \;\&\&\; breq2 \;\&\&\; breq3 \;\&\&\; Rid\bar{0} \;\&\&\; \text{Arbit} \;\&\&\; X\; !reset \rightarrow X\; Rid\bar{2} \;\&\&\; X\; busstate)$

User Story : If agent0, agent2 and agent3 are requesting for the bus and if agent0 had the ownership of the bus previously then agent2 wins the arbitration and gains ownership of the bus.

$G(breq0 \;\&\&\; breq2 \;\&\&\; breq3 \;\&\&\; !breq1 \;\&\&\; Rid\bar{1} \;\&\&\; Arbit \;\&\&\; X\; !reset \rightarrow X\; Rid\bar{2} \;\&\&\; X\; busstate)$

The processor may retain ownership after completing a transaction in case it may need the request signal group again in the future. This is referred to as bus parking. When a processor parks the ownership of the bus, it may retain the ownership until another processor requests ownership. In other words, be fair to the other processors.

User Story : If agent0 is the owner of the request signal group and one or more of the other agents are requesting for the bus then agent0 deasserts its request and allows the

other agents to arbitrate for the ownership.

G(busstate && Rid$\bar{0}$ && (breq1 || breq2 || breq3) && Arbit && → !breq0)

If an agent is requesting for the ownership, within 4 arbitration events it will be given the ownership of the request signal group.

While the symmetric agents are very polite to each other, the system may include another agent that plays by different rules, referred to as a priority agent. When a priority agent is requesting ownership at the same time that one or more of the symmetric agents are also requesting ownership, the priority agent wins. The only case where the priority agent will be unsuccessful in winning ownership of the bus is the case when a symmetric agent has already acquired ownership and has asserted a LOCK signal. This prevents the priority agent from acquiring ownership until the symmetric agent deasserts the LOCK signal.

User Story : if agent0 and agent1 are requesting for the ownership along with the priority agent and the LOCK signal has not been asserted, then the priority is given the ownership.

G(breq0 && breq1 && !breq2 && !breq3 && Rid$\bar{3}$ && BPRI && !LOCK && Arbit && X !reset → X Rid$\bar{4}$ && X busstate)

User Story : If agent1 and the priority agent are requesting for the ownership at the same time when the LOCK is asserted then the ownership is not given to the priority agent.

G(breq0 && breq1 && !breq2 && !breq3 && Rid$\bar{3}$ && BPRI && LOCK && Arbit && X !reset

$\rightarrow$ *X Rid$\bar{0}$* && *X busstate)*

5.4 Experimental Results

The ISA Bus and the arbitration event of the Pentium bus were modeled and number of reachable states searched for the verification of the properties was noted for each of the three different approaches of ordering and modeling properties. The two models differed in the number of properties for modeling as well as the size of the model. Vacuity checks were performed on all properties in order to ensure that they were not vacuously true.

In the model of the ISA Bus architecture, 16 properties were modeled which were derived from 14 distinct user stories and included 15 predicates. The data for the number of reachable state space searched for verification of each property were noted after each successful verification. This data was analyzed for all three distinct property ordering approaches. It was noted during the random ordering approach that the model state space grew significantly large after modeling the first property. The main reason being that the order of properties selected required the user to build the complete model in one step in order to verify the property in the early stages. This approach caused inclusion of non-relevant behavior in the model, which was not seen in the other two approaches due to an incremental modeling methodology. Therefore, the property based and the predicate based ordering resulted in less number of states searched than the randomly ordered modeling approach of properties. The number of states searched for verification by predicate based ordering resulted in better performance in terms of more properties being verified with a lower number of states searched than the property based ordering. The results of the state space search of the ISA bus is shown in Figure 5.4

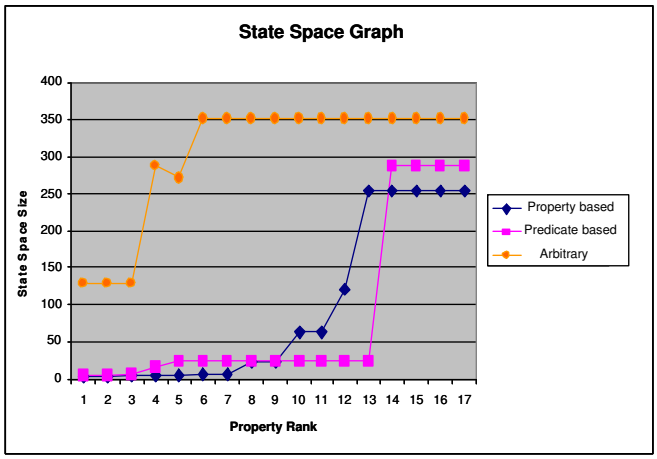

Figure 5.4: State Space Search Graph of ISA Bus Model

On the other hand, the arbitration model of the Pentium bus was more complex than the ISA bus model. The arbitration model consisted of 126 properties from as many user stories and included 24 predicates. The state space search graph for verification of properties had a huge discrepancy between the random model and the other two approaches. The model built using the random approach became complex in the early stage of modeling requiring a large number of states to be searched to verify the properties. The property based sorting had a smaller state space search for the initial set of properties but the states for verification grew as more properties were added. After about 75 properties, the state space search became stable since the complete model had be constructed at that point. This ordering approach was better than the random approach, but the predicate based property ordering gave better results with the graph of the state space search being more incremental and uniform throughout. The results of the state space search of the arbitration phase of the Pentium Pro bus is shown in Figure 5.5.

We develop and experiment with various incremental development methodologies of XFM based on property ordering schemes. We use the methodology to build correct PFMs from specification in natural languages. Our experimental results suggests that

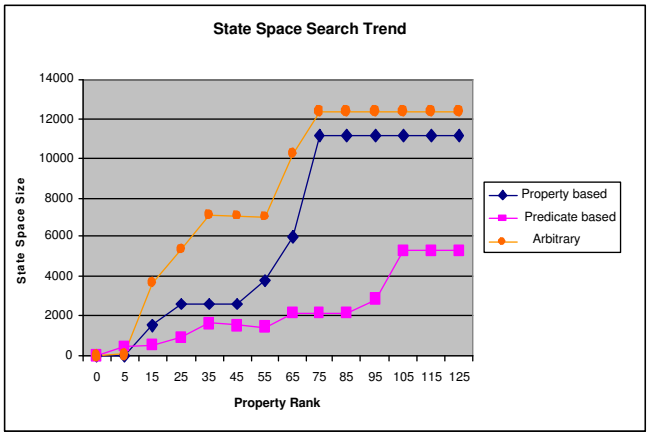

Figure 5.5: State Space Search Graph of Arbitration Model of Pentium Bus

property based and predicate based modeling approaches yield a better methodology for building incremental PFMs with respect to the random ordering which is one of the standard approaches. We also provide an enhancement to the property based ordering approach and in the future, we plan to experiment with this approach. We have also developed a graphical user interface for XFM such that user can easily sort the properties using our proposed methodologies. Using this tool, the user can build and model check the model with SMV which is explained in the next chapter.

Chapter 6

XFM GUI toolkit

In Chapter 5, we developed a methodology which allows us to analyze the effects of ordering LTL properties for building PFMs with XFM using three different model building approaches: *arbitrary ordering*, *property based ordering and predicate based ordering* of properties. We have developed an 'easy to use' graphical user interface for users to build efficient models by using our agile methodology and ordering schemes, which we discuss in this chapter . The Graphical User Interface (GUI) has been developed in *tcl/tk* toolkit [49]. Details of the functionalities of the GUI and the setup requirements are discussed in the following sections:

6.1 Tool Setup

Since the XFM tool is written in *tcl/tk* [49], the user needs to install *wish* version 8.4, which can be obtained for PCs from *cygwin* [5] and for Unix systems from Tcl Developer Xchange [15]. Nevertheless, the user can also download and install *Activetcl* from the

Active State [1] which contains *wish* version 8.4 as well.

Once the user has setup *wish*, he/she then needs to install Cadence SMV, which can be obtained from [46]. After the SMV has been successfully installed, the *vw* file needs to be copied from */smv/lib* directory to the directory of the XFM tool. *vw* is used for invoking the GUI interface of SMV. The XFM tool has been tested on a Windows platform as will as on Redhat version 9.0. The tool can be downloaded from [17].

6.2 Graphical User Interface

The GUI interface of XFM is shown in Figure 6.1. The GUI consists of many widget classes and is interfaced with a C/C++ based ordering program. The GUI also has the capabilities to model check the model with SMV [47].

6.2.1 Menu

The menu consists of *File, Edit, Insert and Help* menu options. The File menu includes three options. The first option, 'Open', is to load the contents of the file that is selected by the user in the *selection window*. The contents of the file can be either variables or properties used for construction of the model. The 'Save' option in the *File* menu saves the contents of the property window. The third option, 'Exit', closes the GUI interface.

The *Edit* menu consists of 'Cut', 'Copy' and 'Paste' options which work similar to standard options and are applicable to the *property window*. The *Insert* menu consists of a list of operators that can be used to write the properties. When selected, these operators are inserted into the editing window for writing properties. The help menu gives

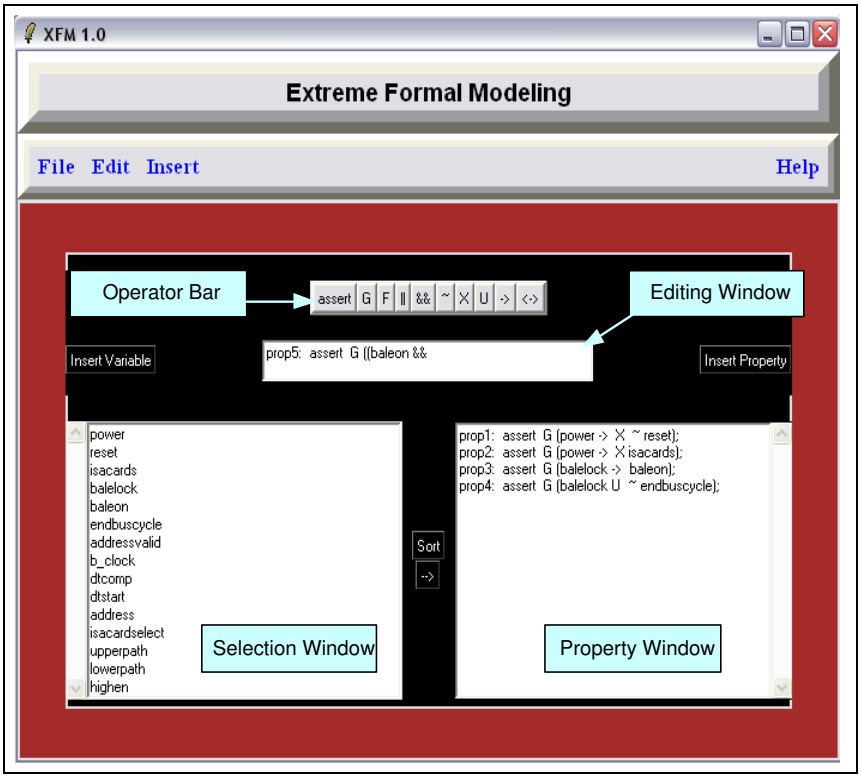

Figure 6.1: XFM GUI toolkit

information about the tool.

6.2.2 GUI setup

The main GUI interface consists of a listbox and two text boxes. The list box is referred to as the *selection window*. When a file is loaded, the data from the loaded file is displayed in the selection window. One text box, referred as *editing window*, is used for editing properties and the other text box, referred as *property window*, is for displaying the list of properties for sorting. A button bar referred as *operational bar* is used for inserting operators into the editing window. There are four separate buttons on the GUI:

- **Insert Variable** button when pressed copies variables selected in the selection window to the editing window.
- **Insert Property** button when pressed copies the property written in the editing window to the property window.
- '→' button when pressed copies the selected items from the selection window to the property window directly. This option can be used when the user loads a list of properties from the file.
- **Sort** button can then be pressed for sorting the contents of the property window. When the button is pressed, a small frame pops up that allows the user to select the three ordering options: *Arbitrary, Property based and Predicate based* ordering. The snapshot of the selection is shown in Figure 6.2. When a sorting option is selected, the sort program executes in the back-end and the sorted properties are displayed in a new window. The sort program has been developed in C++.

6.2.3 Acceptable Property Format

The tool accepts the properties in the format used by the SMV. If the format of the properties do not adhere to the specified format, SMV will give an error. An example of a property format is shown below:

property1: assert G (sugar → sweet);

In the property mentioned above, *property*1 is a name given to identify the property. A property name should be given to all properties which should be unique such that it is easier to refer to the property. Also, note the colon (:) preceding the property name, which distinguishes it as the property name. The sorting algorithms may assume the

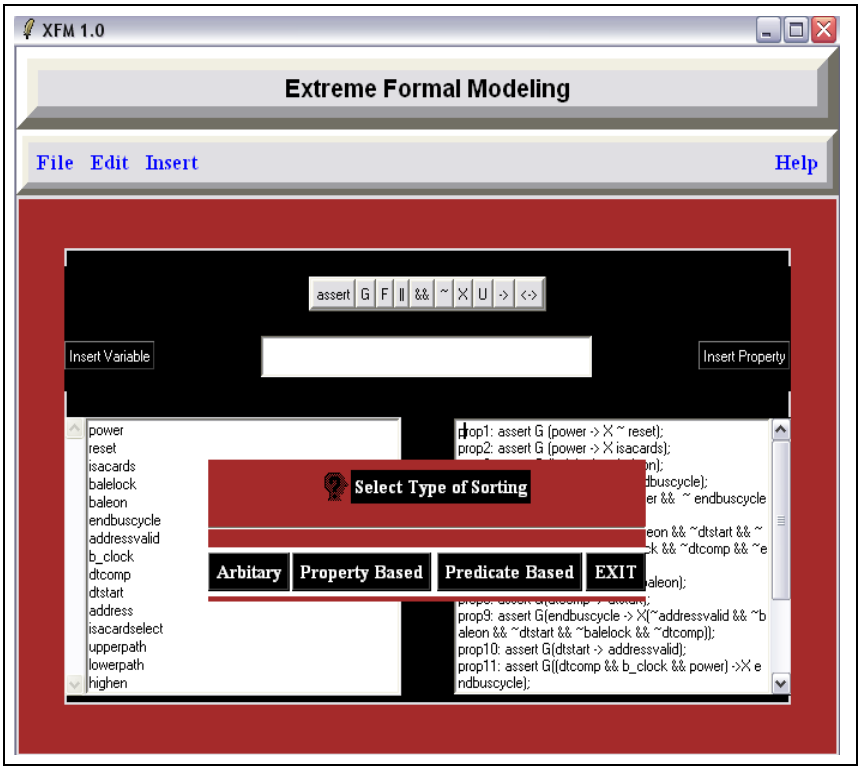

Figure 6.2: Sorting Option Frame

property name to be a predicate if the colon is missing. In addition, SMV will give an error message if the name and the colon is not present. Following the property name, the key word *assert* is used for denoting a property to SMV. Another important part of the property is its terminating semi-colon (;). The sorting algorithm identifies a property with its terminating semi-colon and if missing, the sorting algorithm will not consider it a property. SMV follows the standard format of the properties. If the property is incorrectly specified, SMV will return an error.

The tool can be used to sort properties in formats acceptable by other tools as along as the properties terminate with a semi-colon(;) and no specific keywords other than the ones mentioned in the operator list exist.

6.3 GUI Objective

The tool serves two objectives:

Objective A: Focus on property writing and sorting aspect. It allows the user to write or load a list of properties in the property window and sort them according to the selected sorting scheme.

Objective B: Focus on the model building and model checking aspect and interface with SMV. The properties must adhere to the format of properties accepted by SMV as discussed earlier.

Objective A

As mentioned earlier, the first objective of the GUI is to have a set of properties in the property window and sort them. There are two ways for doing this: The user can load a list of variables from the file, or, write properties in the editing window and then copy them to the property window. To write the properties in the editing window, the user can either use the insert menu, or use the operator bar, or can simply type the property. The user can also load a list of properties from a file and copy them directly to the property window with the '$\rightarrow$' button. One or more properties can be selected simultaneously by pressing *shift/Ctrl* key and selecting the properties.

Once the user has all the necessary properties in the property window, the user can sort the properties by clicking on the 'Sort' button. The properties in the property window are saved in a temporary file by this operation. The user can select one of the three options on the frame for ordering properties or can press exit to return back to the main

GUI as shown in Figure 6.2. When one of the sorting options is selected, the algorithm executes on the back-end and the sorted properties are displayed in the new window as shown in Figure 6.3.

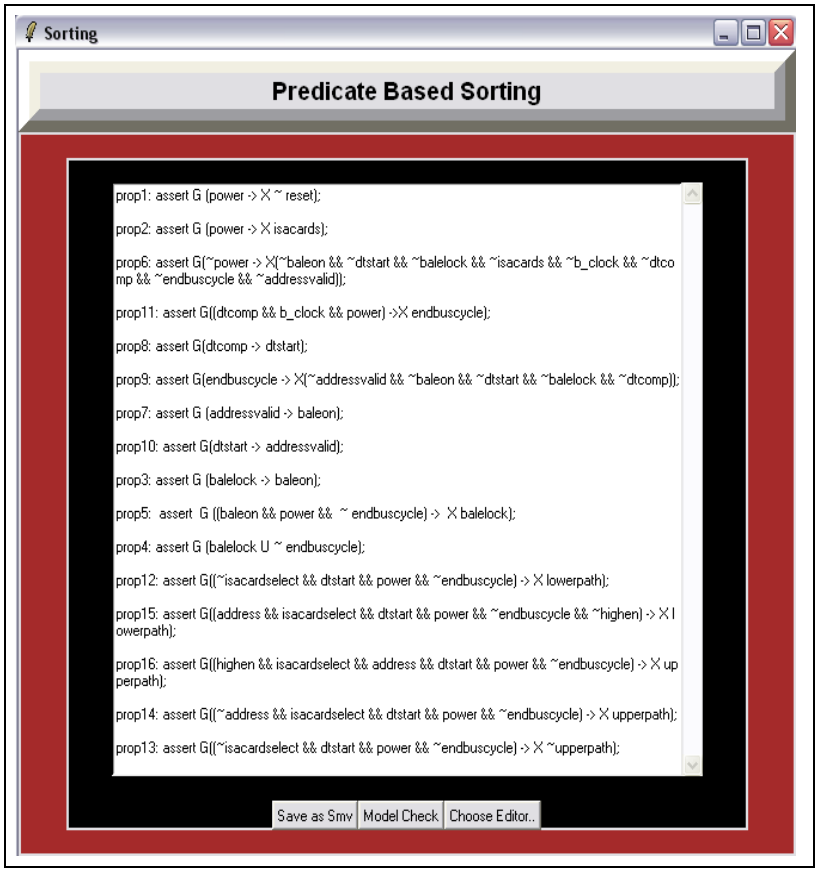

Figure 6.3: Model building window

Objective B

The second objective of the GUI focuses on building the model and model checking it with SMV. The user can build the model in the text box provided or can select an editor of his/her choice by clicking the *Choose Editor* button. To follow the XFM approach, the

user will have to manually comment (/*...*/) the properties other than the one which is being currently modeled. The user will have to remove the comment from the properties one by one as per the XFM methodology for model building. On each iteration, the user can save and model check the model by clicking on *Model Check* button. Once verified, the user can return to the tool and follow the XFM methodology.

This tool helps the user to list, sort, model and model check the model in compliance with the XFM methodology presented in the thesis. We believe that this tool will assist engineers in making efficient models for verification purposes.

Chapter 7

Conclusion

Computer systems control many aspects of our lives. These computer systems are composed of various hardware and software components. Electronic commerce, highway and air traffic control systems, medical instruments, telephone systems are some of the examples of highly computerized systems. Failures in such systems have caused serious consequences including deadly accidents, shutting down of vital systems and monetary loss. Therefore, verification of electronic designs has become increasingly important in the industry. The use of formal methods has become popular in the industry for such purposes. The key aspects of formal methods include specification, verification and testing techniques for enhancing the quality of the software and hardware development. It is known from industrial trends that the validation cycle is often the limiting factor for the decrease in time-to-market. Formal methods are complex, difficult to use, and require mathematical sophistication and to make formal methods available to regular engineers, one has to build methodologies and tool sets that enable the engineers to easily utilize the effectiveness of formal methods without being thwarted by the complexity of the method itself. There exists a gap between natural language specifications

and formal models. In most cases in the industry, the ad hoc abstract model is built from an implementation, which is then checked against the implementation for conformance. Several drawbacks are present with this approach. One major drawback is that the ad hoc building of both the model and the properties is error prone and the effort of model building and debugging grows along with the size of the model. There is no way to control the inclusion of all properties; some may be overlooked, thus reducing the significance of the model. Then, if a property fails, it is tedious to debug the model. Few indications exist where the bug is located. Finally, there is a tendency that the model will include more behavior than the specification will allow. This is because often implementation bias gets into the abstract model. In addition, implementation detail in the abstract model may introduce unwanted complexity and may later cause problems in a conformance check.

7.0.1 Main Contributions of this thesis

- XFM methodology and its importance
- Property Ordering Heuristics
- GUI toolkit to support XFM

In this thesis, we have developed a methodology of an agile formal method (XFM) and a GUI based toolbox to enable formal methods practitioners to use our methodology through a graphical user interface. Our methodology is based on Extreme Programming (XP) concepts instead of programs. We apply XP to construct abstract models from a natural language specification of a complex system.

One of the usage modes for building formal models is Prescriptive Formal Models (PFM), which states the specification in a formal manner. We show how to incrementally build

PFMs by adding user stories one by one into the model. "User Stories" are description of behaviors in a system and in our toolbox; they are captured as Linear Time Temporal Logic (LTL). Other formalisms for expressing properties can be easily integrated. The methodology involves taking an LTL property and building the model to satisfy that property. Once the model is verified for the selected property, we select a second property and build up on the same model for satisfaction. This incremental model building procedure is repeated until all the properties are modeled. We regressively model check the model against all the modeled properties. If at any step the model does not satisfy a specific property, we locate and fix the error in the model. Due to our regressive approach, the debugging is easily done since most of the times we only have to backtrack one-step whereas debugging a complete model for locating an error is difficult. Another advantage is that the model constructed using our methodology does not contain any non-relevant behavior since the model is constructed based on the properties, whereas in the "state of the art" modeling approach, constructing the complete model at once may cause the model to contain non-relevant behavior. We illustrate our methodology with various real world examples of traffic light controller, DLX pipeline and a Smart Building control system.

Since our methodology is based on modeling the properties, it is important to decide what order of modeling should be used to build the abstract model. The LTL properties contain predicates that describe a certain behaviors. The complete set of these behaviors modeled in a specific manner constitute the model. Our ordering schemes are based on the frequency of the predicates present in the property set. We analyze the effect of ordering LTL properties for building PFMs with XFM with three different model-building methodologies: arbitrary ordering, property based ordering and predicate based ordering of properties. We used our property ordering approaches with XFM methodology on the models of ISA bus and the arbitration event of Pentium Pro bus.

We found out that the predicate-based ordering approach was the most effective way of XFM and we mathematically reason the same as well. We have also developed a GUI based toolbox to enable formal methods practitioners to use our methodology through a graphic user interface. Some of key the features of this gui toolbox include property building, property sorting, model building and model checking. The user can select one of the property sorting schemes presented in this thesis. The syntax of the properties accepted by the tool adheres to the format of SMV model checker. This XFM toolkit is interfaced with the SMV model checker for model checking purposes.

Some projects also use agile methods in the domain of modeling and verification. Herranz and Moreno-Navarro from TU Madrid describe in the integration of some XP practices to formal methods using the SLAM software tool [11]. Their environment generates sequential programs from formal expressions using an assertion based JAVA development framework. While our work involves the use of XP to model complex concurrent hardware systems, their approach is directed towards sequential software programs. Henzinger et.al. [36] show how to reuse previous model checking results to incrementally model check the modification of a model. While this is a completely different approach, this technique can be used in XFM in order to speed up the single model checking steps. Besides the typical method to capture formal models, there is other research going on to speed up this process, such as the automatic synthesis of models from temporal constraints [58], but high quality models are still created manually by qualified engineers.

To the best of our knowledge, this is the first methodology for building formal models in an agile development style, and first theoretical framework development. The GUI is also useful in enabling users to easily build models incrementally and correctly through regressive steps. The property set developed as user stories are expressed formally, and hence can be used for coverage monitor and test generation as well.

Bibliography

[1] *Active State*, http://www.activestate.com.

[2] *Adaptive Home- Colorado, Boulder*, http://cs.colorado.edu/~mozer/nnh/index.hmtl.

[3] *Bell Labs*, http://www.bell-labs.com/.

[4] *Cisco Internet Home*, http://www.cisco.com/warp/public/3/uk/ihome.

[5] *Cygwin*, http://www.cygwin.com.

[6] *Extreme Programming: A gentle introduction*, http://www.extremeprogramming.org/.

[7] *Getting Started with SMV*, http://docs.cs.uct.ac.za/smv/tutorial/node1.html.

[8] *HOL: The High Order Logic Theorem Prover*, http://cs.anu.edu.au/student/comp8033/hol.html.

[9] *Langley Formal Methods*, http://shemesh.larc.nasa.gov/fm/fm-what.html.

[10] *LTL 2 BA : fast algorithm from LTL to buchi automata*, http://www.liafa.jussieu.fr/~oddoux/ltl2ba/.

[11] *The SLAM website*, http://lml.ls.fi.upm.es/slam.

[12] *SMV language reference*, http://www.cs.wpi.edu/ kfisler/Courses/525V/S02/Readings/smv-cadence.pdf.

[13] *Software System Award*, http://www.acm.org/awards/ssaward.html.

[14] *SPIN root*, http://spinroot.com/spin/whatispin.html.

[15] *Tcl Developer Xchange*, http://www.tcl.tk/.

[16] *The PVS Specification and Verification System*, http://pvs.csl.sri.com/.

[17] *XFM tool download*, http://fermat.ece.vt.edu/XFM.html.

[18] *Z/EVES*, http://www.ora.on.ca/z-eves/.

[19] *MIT Project Oxygen - pervasive human centered computing*, http://oxygen.lcs.mit.edu, 2003.

[20] Kent Beck, *Extreme programming explained: Embrace change*, Addison Wesley, 2000.

[21] I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh, *Efficient detection of vacuity in temporal model checking*, Formal Methods of System Design, March 2001, pp. 141–163.

[22] Bob Bentley, *Validating the Intel Pentium 4 Microprocessor*, Design Automation Conference, 2001, pp. 244–248.

[23] C. T. Chou and D. Peled, *Formal verification of a partial order reduction technique for model checking*, Automated Reasoning, 3, vol. 23, November 1999, pp. 265–298.

[24] Koen Claessen, Reiner Hahnle, and Johan Martensson, *Verification of hardware with First-Order Logic*, PaPS, 2002.

[25] E. M. Clarke and E. A. Emerson, *Design and synthesis of synchronization skeletons using branching time temporal logic*, In Proc. of the Workshop on Logic of Programs (Yorktown Heights, New York), vol. 131, Springer-Verlag, May 1981, pp. 52–71.

[26] E. M. Clarke, E. A. Emerson, and A. P. Sistla, *Automatic verification of finite-state concurrent systems using temporal logic specifications.*, Issue 2, vol. Vol. 8, pp. 244–263, ACM Transactions On Programming Languages and Systems, April 1986.

[27] Edmund Clarke, Orna Grumberg, and Doron Peled, *Model checking*, The MIT Press, 1999.

[28] Edmund M. Clarke, Steven M. German, Yuan Lu, Helmut Veith, and Dong Wang, *Executable protocol specification in ESL*, Formal Methods in Computer-Aided Design, 2000, pp. 197–216.

[29] D.Berner, S. Suhaib, S. Shukla, and H. Foster, *XFM: Extreme Formal Method for Capturing Formal Specification into Abstract Models*, Tech. Report 2003-08, Virginia Tech, 2003.

[30] Devadas, Ma, and Newton, *On the verfication of sequential machines at different levels of abstraction*, 24th DAC, June 1987, pp. 271–276.

[31] Matthew Dwyer, George Avrunin, and Corbett, *Patterns in property specifications for finite-state verification*, In Proc. of the 21st International Conference on Software Engineering (1999).

[32] Matthew B. Dwyer, *Automated analysis of software frameworks*, Workshop on Foundation of Component-Based Systems, September 1997.

[33] Mike Gordon, *Specification and verification of hardware*, University of Cambridge, October 1992.

[34] John L. Hennessy, David A. Patterson, and David Goldberg, *Computer architecture: A quantitative approach*, 3rd ed., Morgan Kaufmann, San Mateo, CA, may 2002.

[35] Jesper G. Henriksen, *Logics and automata for verification - expressiveness and decidability issues*, Ph.D. thesis, Basic Research in Computer Science (BRICS), University of Aarhus, Denmark, 2000.

[36] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Marco A.A. Sanvido, *Extreme model checking*, Proceedings of the International Symposium on Verification: Theory and Practice, Lecture Notes in Computer Science, Springer-Verlag, 2003.

[37] A. Herranz and J.J. Moreno-Navarro, *Formal agility. how much of each?*, Taller de Metodologias Agiles en el Desarrollo del Software. JISBD 2003 (Alicante, Espana), Grupo ISSI, 11 2003, pp. 47–51.

[38] ———, *Formal extreme (and extremely formal) programming*, 4th International Conference on Extreme Programming and Agile Processes in Software Engineering, XP 2003 (Genova, Italy) (Michele Marchesi and Giancarlo Succi, eds.), LNCS, no. 2675, May 2003, pp. 88–96.

[39] ———, *Rapid prototyping and incremental evolution using SLAM*, 14th IEEE International Workshop on Rapid System Prototyping, RSP 2003) (San Diego, California, USA), June 2003.

[40] C. A. R. Hoare, *Communicating sequential processes*, Comm. of the ACM, 8, vol. 21, Aug 1978, pp. 666–677.

[41] Gerhard J. Holzmann, *The SPIN model checker: Primer and reference manual*, Addison Wesley, Boston, MA, September 2003.

[42] Thomas Kropf, *Introduction to formal hardware verification*, Springer, 1999.

[43] K. Larsen, P. Petterson, and W. Li, *Model-checking for real time systems*, In Proc. of the 10th International Conference Funcamentals of Computation Theory. **LNCS 965** (1995), 62–88.

[44] P. Li, B. Ravindran, S. Suhaib, and S. Feizabadi, *A Formally Verified Application-Level Framework for Utility Accrual Real -Time Scheduling On POSIX Real-Time Operating Systems*, Tech. Report 2003-05, Virginia Tech, 2003.

[45] M. Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Francheschinis, *Modelling with generalized stochasic petri nets*, John Wiley and Sons, 1995.

[46] Ken McMillan, *Cadence SMV*, http://www-cad.eecs.berkeley.edu/ kenmcmil/smv/.

[47] _____, *Symbolic model checking*, Kluwer Academic Publishers, 1993.

[48] Andreas S. Meyer, *Principles of functional verification*, Newnes, 2004.

[49] John K. Ousterhout, *Tcl and the tk toolkit*, third ed., Addison-Wesley Professional Computing Series, Pearson Education Inc., 2002.

[50] Amir Pnueli, *The temporal logic of programs*, In 18th Annual Symposium on Foundations of Computer Science (1977), 46–57.

[51] J. P. Queille and J. Sifakis, *Specification and verification of concurrent programs in CESAR*, Proc. of the 5th Intl. Symp. on Programming **LNCS 137** (1981), 195–220.

[52] H. Schoot and H. Ural, *An improvement on partial order model checking with ample sets*, University of Ottawa, Canada, computer science technical report tr-96-11 ed., Sept 1996.

[53] Tom Shanley, *Pentium pro and pentium ii bus system architecture*, second edition ed., Addison-Wesley Inc., 1998.

[54] Tom Shanley and Don Anderson, *Isa system architecture*, 3rd ed., Addison-Wesley Publishing Company, 1995.

[55] Shi-Yu-Huang and Kwang-Ting Cheng, *Formal equivalence checking and design debugging*, vol. 12, Kluwer Academic Publishers, 1998.

[56] Kanna Shimizu, David L. Dill, and Alan J. Hu, *Monitor-based formal specification of PCI*, Formal Methods in Computer-Aided Design, 2000, pp. 335–353.

[57] Ronak Singhal, E. Venkatraman, Evan Cohn, John Holm, David Koufaty, Meng-Jang Lin, Mahesh Madhav, Markus Mattwandel, Nidhi Nidhi, Jonathan Pearce, and Madhusudanan Seshadri, *Performances analysis and validation of the intel pentium 4 processor on 90nm technology*, Intel Technology Journal **08** (2004).

[58] B. Steffen, T. Margaria, and M. von der Beeck, *Automatic synthesis of linear process models from temporal constraints: An incremental approach*, International Workshop on Automated Analysis of Software (New York), ACM Press, January 1997, pp. 127–141.

[59] S. Suhaib, D. Berner, and S. Shukla, *Extreme Modeling in PROMELA: Formal Modelling and Verification of a Smart Building Control System*, Tech. Report 2003-11, Virginia Tech, 2003.

[60] _____, *Extreme Formal Modeling to Capture Specifications for a Smart Building Control System into a Constructively Correct Model*, Tech. Report 2004-01, Virginia Tech, January 2004.

[61] Geoff Sutcliffe, *Automatic theorem proving*, http://www.cs.miami.edu/ tptp/OverviewOfATP.html.

[62] Filip Thoen and Francky Catthoor, *Modeling, verification and exploration of task-level concurrency in real-time embedded systems*, Kluwer Academic Publishers, 2000.

[63] Laurie Williams, *The XP programmer - the few minutes programmer*, IEEE Software **20** (2003), no. 3, 16–20.

[64] Pierre Wolper and Denis Leroy, *Reliable hashing without collision detection*, Proc. 5th Int. Conference of Computer Aided Verification (Elounda, Greece), Springer Verlag, pp. 59–70.

[65] William A. Wood and William L. Kleb, *Exploring XP for scientific research*, IEEE Software **20** (2003), no. 3, 30–36.

Vita

Syed Mohammed Suhaib

Date of Birth	:	October 29, 1980
Marital Status	:	Single
Visa Status	:	F-1

Office Address:

FERMAT Research Lab.
340 Whittemore Hall
Blacksburg, VA 24060
Email: ssuhaib@vt.edu
URL: http://www.fermat.ece.vt.edu/
Tel: (540) 257-4444

Research Interests

(a) Formal Methods and Modelling

(b) Formal Verification and Model Checking of Software, Hardware and Real-Time Systems

(c) Behavioral Type Systems

(d) Embedded Systems Design

(e) Software Design

Education

(a) Ph.D., Computer Engineering, (Expected May 2007), Virginia Polytechnic Institute and State University.

(b) M.S., Computer Engineering, (May 2004), Virginia Polytechnic Institute and State University.

M.S. Thesis Title: "XFM: An Incremental Methodology for Developing Formal Models"

(c) B.S., Computer Engineering, (May 2002), University of Arkansas.

Current Work

(a) Extreme Formal Modelling and Verification

Developed an agile formal method named **XFM** based on extreme programming concepts to construct abstract model from a natural language specification of real world complex systems using SPIN and SMV model checkers.

(b) Model Extraction of Multi-Threaded Graph Models for Formal Verification of System Level Models

Implemented a model extraction scheme from MTG models to specification language of UPPAAL model checker for formal verification.

(c) Formal specification and Verification of Real-Time Systems

Modelled and Verified utility accrual Real-Time systems using the UPPAAL model checker.

Skills

– Formal Verification Tools: SPIN, Cadence SMV, Timestool, UPPAAL.

– Languages: C/C++, Tcl/TK, SystemC, Java, Winsock API, Intel/Motorola Assembly, Visual Basic, SQL, HTML.

– Environments: Windows 95/ 98/NT/2000/XP, DOS, Linux, Solaris, QNX.

– Application Packages: Ptolemy II, MS Office, MS Visio, Corel Word Perfect, Mentor Graphics, Ocelot, Visual Studio, Borland, Dreamweaver, Adobe Photoshop, Flash.

Experience

(a) January 2003 to present: Graduate Research Assistant, Virginia Tech., Blacksburg, VA.

- Extreme Modelling based approach to build a model based on the properties and incrementally verifying them simultaneously.
- Developed an XFM tool for formal modeling and model checking
- Using Model Extraction schemes for formalizing and verifying systems.
- Using various Model Checking tools such as SPIN, SMV, TimesTool and UPPAAL for verification of complex systems such as RISC CPU Pipeline, ISA Bus protocol, Pentium Pro bus, Smart Home Control System as well as for Utility based protocols for Real-Time scheduling.

(b) August 2001 to May 2002: Lab Operator Computer Lab, University of Arkansas, AR.

- Maintain Workstations for student.
- Assist students with various technical difficulties.

(c) August 2002 to December 2002: Technical Support for Law School Computing Services, University of Arkansas, AR.

- Provided Support for Network Administrator including configuring of systems on the network, maintaining and upgrading systems and maintaining School of Law web-page.

Publications

[1] S. Suhaib, D. Mathaikutty, and S. Shukla. Effects of Property Ordering in an Incremental Formal Modeling Methodology. (To appear in) *Proceedings of International Workshop on Logic and Synthesis*, 2004.

[2] D.Berner, S. Suhaib, S. Shukla, and J-P Talpin. Capturing Formal Specification into Abstract Models. (To appear in) Formal Methods and Models for System Design, 2004.

[3] S. Suhaib, D. Berner, and S. Shukla. Extreme formal modeling to capture specification for a smart building control system into a constructively correct model. FERMAT Lab Technical Report 2004-01, Virginia Tech, January 2004.

[4] S. Suhaib, D. Berner, and S. Shukla. Extreme modeling in PROMELA: Formal modeling and verification of a smart building control system. FERMAT Lab Technical Report 2003-11, Virginia Tech, 2003.

[5] D.Berner, S. Suhaib, S. Shukla, and H. Foster. XFM: Extreme formal method for capturing formal specification into abstract models. FERMAT Lab Technical Report 2003-08, Virginia Tech, 2003.

[6] S. Suhaib and S. Shukla. Automated extraction of multi-threaded graph models for formal verification of system level models. Technical Report 2003-03, Virginia Tech. FERMAT Lab, 2003.

[7] P. Li, S. Suhaib, B. Ravindran, and S. Feizabadi. A formally verified application-level framework for utility accrual real-time scheduling on posix real-time operating systems. (To appear in) 'IEEE transaction on Software Engineering', 2004.

[8] D. Bhaduri, M. Chandra, H. D. Patel, S. Sharad, and S. Suhaib. Systematic abstraction of microprocessor RTL models to enhance simulation efficiency. In *the proceedings of Microprocessor Test and Verification*, 2003.

Selected Projects

(a) Car Tracking Model: Developed an extension to multi-domain model of a car tracking system from Ptolemy II manual.

(b) Air Traffic Controller: Designed and implemented Air Traffic controller system on POSIX compliant RTOS.

(c) Source Code Analyzer: Designed and Developed a Source Code Analyzer of any C program displaying details of the code.

(d) Pipeline Architecture: Designed a 5 stage Pipeline Architecture in Mentor Graphics. Task also involved simulation of the system.

(e)Web-Browser: Developed a Java based Web-Browser with capabilities of a regular browser.

(f)Uc-Linux Kernel: Implementation of Real-Time Scheduling Algorithms such as RMA, EDF and DASA on a micro-controller in the ucLinux kernel.