XFM: An Incremental Methodology for Developing Formal Models

Syed M. Suhaib, Deepak A. Mathaikutty and Sandeep K. Shukla
FERMAT LAB., Virginia Tech, Blacksburg, Virginia
and
David Berner
Institut de Recherche en Informatique et Systèmes Aléatoires (IRISA/INRIA)
35042 Rennes, France.

We present an agile formal methodology named eXtreme Formal Modeling (XFM), based on Extreme Programming (XP) concepts to construct abstract models from natural language specifications of complex systems. 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, coverage monitor generation, etc. This methodology for incrementally building PFMs works 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 consisting of a traffic light controller and a DLX pipeline. To make the regressive model checking steps feasible with current model checking tools, we need to control the model size increments at each subsequent step in the process. We therefore analyze the effects of ordering the LTL properties in XFM on the statespace growth rate of the model. We compare three different property-ordering methodologies: ad hoc 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 the predicate based ordering is the best among these orderings. Finally, we present a GUI based toolbox we implemented to build PFMs using XFM.

Categories and Subject Descriptors: B.6.3 [**Logic Design**]: Design aids—*Verification*; B.7.1 [**Logic Design**]: Types and design styles—*VLSI*; B.7.2 [**Integrated Circuits**]: Design aids—*Verification*; D.2.1 [**Software Engineering**]: Requirements/Specification—*Methodologies*; *Tools*; D.2.2 [**Software Engineering**]: Tools and Techniques—*Computer-aided software engineering*; D.2.4 [**Software Engineering**]: Program Verification—*Validation*; D.2.1 [**Software Engineering**]: Requirements/Specification—*Methodologies*; *Tools* ; F.3.1 [**Logics and Meaning of Programs**]: Specifying and Verifying and Reasoning about Programs—*Assertions*; *Specification techniques*

Additional Key Words and Phrases: Extreme formal modeling, extreme programming, formal specification, formal verification, prescriptive formal models, property ordering, property refactoring, SPIN, SMV

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

[0]This work has been supported by the NSF project CRCD/EI-0417340 and CCR-0237947

techniques. Formal verification and formal methods 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 easily accessible to design engineers, one has to build methodologies and toolsets that enable the engineers to easily utilize the effectiveness of formal methods without being thwarted by the complexity of the method itself.

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 leads 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 easy to build such golden reference models 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 in an ad hoc manner.

In this article, we present a formal model building methodology based on the principles of *Extreme Programming* (XP) [Wells 2001; Wood and Kleb 2003] methodology in software engineering. 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 the incremental building of a PFM, the model is regressively verified. This model helps in creating testbenches, 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 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 article, our focus is on a specific methodology that enhances the efficacy of formal verification.

In the industry, building formal models for verification purposes is used in two different usage modes: Descriptive formal models (DFM) [Bentley 2001] 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. Most of the verification these days is done on DFMs. Since, DFMs are built from the implementation, there is a high possibility that some of the vital implementation bugs are not modeled unless automatically abstracted from the implementation. Another drawback with these types of models is that they may include unwanted complexity and irrelevant behaviors from the implementation.

On the other hand, Prescriptive Formal Models (PFM) [Shimizu et al. 2000; Clarke et al. 2000; Berner et al. 2004] 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.

Currently, in our experience from the industry, whenever PFMs are built and verified, it is done in an ad hoc manner. Figure 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 includes less behaviors than the specification allows. 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.

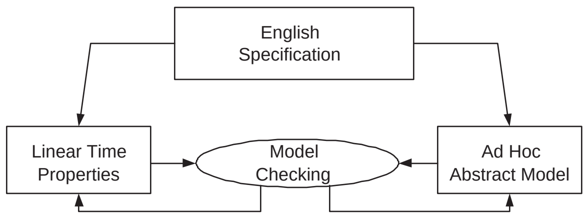


Fig. 1. Current practice in capturing a formal model from natural language specification

1.2 Motivation

In our experience in developing formal models for systems such as real-time meta-scheduler and EDF scheduling algorithm [Li et al. 2004], we feel 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 the "*model formally first and design later*" (MOFDEL) approach.

Our methodological motivation is derived from *XP* [Wells 2001] where a "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 aspects 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.

With verification being a major part of the design cycle, our motivation is to facilitate the verification process by creating PFMs as golden reference models for verification. Usually the golden reference model is not built or even if it is built, it is usually captured in a high level programming language such as C or C++, which does not have a well-defined semantics. Bits and pieces of PFMs are often built in the verification IPs, such as verification IPs constructed in Vera [Synopsis 2004], Specman [Verisity 2004], etc., but these are usually done after the design is developed, and are often constructed in an ad hoc fashion. We believe building PFMs a priori to design implementation in a verifiable language would benefit the industrial design flow.

Building correct PFMs is one of the challenging problems, and we address this by using XFM. 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 implementation specific 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. As we incrementally build a PFM, our approach attempts to control state space growth in small increments, rather than sudden large growths in state space resulting in a failure to carry out regressive model checking.

1.3 Main Contributions of this Work

In this section, we summarize the main contributions of this work. At the risk of repeating ourselves, we enlist the main facets of this article for readers' convenience. We propose a methodology for an agile formal method: eXtreme Formal Modeling (XFM), based on 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 illustrate our methodology with examples of a traffic light controller and a DLX pipeline [Suhaib et al. 2004a;

Suhaib 2004].

Since XFM 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: *ad hoc ordering, property based ordering* and *predicate based ordering* of properties. We use our property ordering approaches with XFM methodology on the models of ISA bus [Shanley and Anderson 1995] and the arbitration phase of Pentium Pro bus [Shanley 1998]. We found out that the predicate based ordering approach is the most effective way of XFM and with formal reasoning we conclude the same [Suhaib et al. 2004; Suhaib et al. 2004b].

We also build a platform independent GUI tool that provides a GUI interface to facilitate the usage of our XFM approach with property ordering. For model checking purposes, the tool is interfaced with the SMV model checker [McMillan 1993].

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

This article is organized as follows: Section 2 describes related work in the field. In Section 3, we discuss in detail the methodology of XFM along with examples of a traffic light controller and a DLX pipeline. In Section 4, we discuss the theory behind ordering the properties for XFM and the schemes we use for our methodology. We illustrate the ordering with examples of the monitor of the ISA bus and the arbitration phase of the Pentium Pro bus. In Section 5, we discuss the XFM toolkit followed by conclusion in Section 6.

2. RELATED WORK

Efforts are being made to use the XP methodology to build large systems fast in high quality, but this approach has not been connected so far to formal modeling and verification. Some related work in the field of using formal methods with XP has been done by Herranz and Moreno-Navarro at the TU Madrid in [Herranz and Moreno-Navarro 2003a; 2003b]. They describe the integration of some XP practices to formal methods in the SLAM software tool [Herranz 2003]. While our work involves the use of XP to model complex concurrent systems, their approach is directed towards sequential software programs. General information about XP and agile techniques can be found in [Wells 2001; Beck 2000; Williams 2003; Wood and Kleb 2003]. Our approach in using XP ideas to formal modeling is distinct from a recent work in [Henzinger et al. 2003], where they show how to reuse previous model checking results to incrementally model check a modification of a model. Their technique can be plugged into XFM in the essential regression steps of XFM.

3. EXTREME FORMAL MODELING (XFM)

As for any system development, it is important to have a concise and clearly written specification for the system. Some time must be spent on the specification 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 writing correct LTL properties.

Many of the 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**" - short narratives that point out specific implementation details and requirements. These user stories act as a detailed guideline for the programmer. To refactor problems and to update tests after a bug is found are also principles that are as beneficial to the capturing of formal methods as they are for common programming projects.

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 one or more LTL properties.

Algorithm 1 XFM Approach

```
/* For a given system, we have the behaviors 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 $us_i$ user story
Let $\Pi = \cup_i \Pi(us_i) = \{\pi_1, \pi_2, ..., \pi_m\}$ be the complete ordered set of properties.
/*We show 3 different ordering approaches for property modeling later in this article */
Let $\Pi_i = \{\pi_1, \pi_2, ..., \pi_i\} \subseteq \Pi$, so $\Pi_i$ represents the first $i$ properties in the specific order chosen.
Let $X(\Pi_i)$ be the model that satisfies all the properties in $\Pi_i$
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 $\Pi_i$
simultaneously.
    $X := X(\Pi_i)$
    $ModelCheck(X, \Pi_i)$
    if $ModelCheck$ fails for a $\pi_k$ /* This is the regression step */
        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
```

We can now check if the LTL property correctly expresses the behavior of the user story. LTL properties can be visualized as finite state machines (FSM) and LTL 2 BA [Oddoux 2001] eases this step by displaying the corresponding FSM. It is important that during implementation, only the behavior of this property is taken into account. After listing out all LTL properties, we select one property from the list, 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 behaviors 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, the error can be located 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 is straightforward to find the bug as it usually is related to the most recent additions. The complete effort of modeling and bug fixing grows incrementally along with the size of the model. Algorithm 1 gives a formal representation of the basic XFM methodology.

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 helps reveal missing functionality. However, in an ad hoc approach, 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. With two illustrative examples (of a control intensive traffic light controller, and the DLX pipeline) we present this methodology and show the benefits.

3.1 Examples and Results

A simple traffic light illustrates the main steps, tools, and techniques involved. The design of a DLX microprocessor pipeline [Hennessy et al. 2002] shows how this methodology works for a bigger model, and how the model evolves with the incremental approach.

3.1.1 *Traffic Light Model.* This example of a traffic light controller shows a pedestrian crossing at a traffic light. **When a pedestrian pushes the switch, the car signal turns red, and the pedestrian signal turns green. After thirty seconds, the pedestrian signal turns red again and car signal turns green such that cars can go. At any point of time, the cars and the pedestrians cannot go simultaneously.** This description is the English specification. Now, we construct LTL properties describing this system. Let us assume that p, c, and sw are predicates in the system where p is high when the pedestrian signal is red and c is high when the car signal is red. We denote sw high when the switch pressed. We start with the most important property that states that both pedestrian and car, can never get the green signal at the same time: []!(!p && !c).

The next property states that whenever the car signal turns red, they eventually become green. Table I lists all LTL properties for this example. LTL does not allow expressing exact timing, only relative occurrences of events. However, in the model, we add a timer that counts to thirty before the pedestrian's signal turns red and the car's signal turns green. 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).

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 I. LTL properties for traffic light (c = cars stop, p = ped stop, sw = button is pressed)

*Prop*3 and *Prop*4 state that when no switch is pressed, the cars keep driving and the pedestrians keep stopping. As we check these properties against the formal model, we realize that they can be verified without making many modifications to the system.

The 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 are, and without LTL 2 BA it is not easy to figure out if it is correct. After implementing the functionality of these properties into the model, simulation shows that it works as specified, so we have found all properties.

3.1.2 *Model of a DLX Pipeline Control.* The pipeline control of the DLX RISC processor model [Hennessy et al. 2002] is a well-known and reasonably large example to show the use of XFM. The DLX has a 5-stage pipeline, which means up to five instructions can run 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 II shows the cycle usage for the 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	

Table II. Cycles for Different Instruction Types

Starting from this system description, we identify the first user story. One of the most basic behavior states 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 second user story expresses the fact 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

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

Table III. LTL properties for pipeline (examples)

one instruction. 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 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 such as **cat1** in Table III.

In the next iteration, we introduce the possibility to control the instructions from outside. This is done by "enable signals", one for each instruction. The LTL expression says that an instruction does not advance unless the enable signal is given. Again we obtain 25 properties in the style of **cat2** in Table III. The changes in the model for these properties are small, so all of them can be verified without problems.

The following iteration is adding some synchronization. The user story states that the control enables each instruction in each cycle. Once the instruction advances, it is setting its enable signal to zero, thus signaling the control that it is ready for the next cycle.

Another important behavior of a pipeline is to prohibit the multiple usages of resources. If at no time the fetch, decode, execute, address bus, and data bus units are used by more than one instruction there are no resource conflicts. **Cat4** in Table III expresses this in LTL for the fetch cycle of the first instruction. Again the category consists of 25 properties, one for each cycle. In order to satisfy this property in the model, we are introducing a control process that in an initialization phase starts each instruction successively, and later makes sure that every instruction advances in each cycle. The verification of all properties and simulation finishes this iteration step. With only four categories of properties the basic functionality of the pipeline is now verified and working.

To make the model of the pipeline more realistic, we select the user story that defines the different instruction types and their different cycle sequences from Table II. It turns out that this does not result in a new category of properties, but rather implies 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 II. This makes sure that an arithmetic instruction for example goes 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 III. Changes in the abstract model to reflect this are limited to update the FSM description for each instruction that means introducing the notion of an instruction type, and adding the transitions to and from the wait stage. These changes are transparent to the control logic since after the changes still each instruction takes 5 cycles to finish, therefore preventing the occurrence of structural hazards. Of course, there are still more details that could be added to the pipeline, such as data dependencies and forwarding, but the steps for building the model are always the same.

4. PROPERTY ORDERING

In this section, we analyze the effect of ordering the linear time properties while using the XFM methodology in building 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. We compare three different model building methodologies, (i) Ad hoc selection of user stories (ad hoc ordering), (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 [Shanley and Anderson 1995] and a model of the arbitration phase of the Pentium Pro processor bus [Shanley 1998] using Cadence SMV [McMillan 1993].

4.1 Preliminary Definitions

Let $X = \{x_1, x_2, \ldots, x_l\}$ be 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 $x_i = v$ or $x_i \neq v$ 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 a finite variable set X, and below we indicate this number by n. Now let $P = \{p_1, p_2, \ldots, p_n\}$ be the set of all such predicates and $\Pi = \{\pi_1, \pi_2, \ldots, \pi_m\}$ be the set of properties to be modeled.

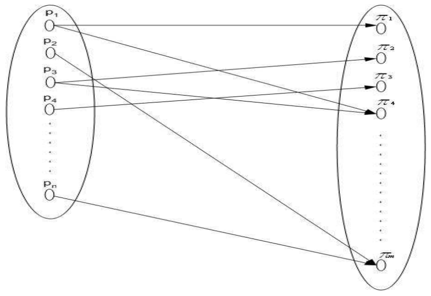


Fig. 2. Predicate-Property Representation

Definition 4.1. **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)**.

The relationship between the properties and predicates can be represented by a bi-partite graph, where one set of nodes is marked with the predicates, and another set of nodes is 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 $p_i \in \pi_j$. A predicate describes the behavior of the system with respect to a specific functionality. A predicate-property relationship graph is shown in Figure 2.

Definition 4.2. **Weights of Predicates and Properties:** $\delta(p_i)$ = out degree of a node marked p_i in G is called the weight of the predicate p_i. It indicates how many times p_i appears in different properties. We denote this by $F(p_i)$ 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 the *entanglement* of that property with other properties.

4.2 Importance of Ordering

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. These behaviors 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 concern is to achieve a model that has smaller state space searched for the property in order to verify it. With smaller state space, verification would be faster and efficient. Both the 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. We experiment with different property ordering approaches using XFM to construct the model.

The first approach, ad hoc ordering, is based on the designers selection of properties for modeling. This approach may cause the model to blow up initially and remain same throughout or may cause the model to decrease in size in between. Second, we have the property based ordering scheme, where the order is based on the weight of each property. We first compute the total weight of the predicates in the system. We start modeling the property π_f with the least $W(\pi_f)$. Also, note that if multiple properties have the same weight, we model them in any order. Finally, we have the predicate based ordering scheme, in which we do an elimination of the properties based on the frequency of the predicates. In this scheme, we model the properties containing the least frequent predicate. Once the above properties are modeled, that specific functionality of the system has been completely modeled. The remaining properties are independent of the modeled functionalities. Therefore, by modeling using a predicate based ordering, we model in a more incremental fashion. We find this to be the most effective of the property ordering schemes for XFM. The scheme for predicate based ordering is shown in Algorithm 2.

Recall that, our goal for ordering properties is twofold: (i) To keep the number of states searched for verification as low as possible in the incremental model building

Algorithm 2 Predicate based ordering

```
Let $\Pi$ = $\{\pi_1, \pi_2, ..., \pi_m\}$ be the set of properties of size $m$
Let $P$ = $\{p_1, p_2, ..., p_n\}$ be the set of predicates of size $n$
/* $W(\pi_f) = A_f \cdot \hat{\delta}$ where $A_f$ is the column vector corresponding to property $\pi_f$ in PP matrix */
Let $L = \emptyset$ /* $L$ is an ordered list of properties initially empty */
while $\Pi$ is not empty
  for all $p_i \in P$, such that $\digamma(p_i)$ 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 $\digamma(p_i)$ in $P$ and one or more predicates
    from $min_p$ appears in all $\pi^i_{min_p}$ in $D$
  end for
  Select $\pi_j$ from $D$ such that $W(\pi_j)$ is minimum
  Remove $\pi_j$ from $\Pi$ such that $\Pi = \Pi - \{\pi_j\}$ /* For multiple $\pi_j$, select one randomly*/
  Add $\pi_j$ to the end of list $L$
  Update all $\digamma(p)$ for all $p$'s in $\pi_j$
  /* Effect of removing $\pi_j$ 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 for XFM modeling */
```

process, as long as we can, (ii) To keep the changes needed at later stages of the model building minimal. 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 realize 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 are 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 property, we have to introduce more states.

In the following series of lemmas and theorem, we show that in predicate based modeling, usually when a sequence of properties is modeled they share predicates, and therefore, the statespace does not increase drastically 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 have any shared predicates, and hence the state space can suddenly go up drastically, as seen in the experimental evidence presented later.

P_{min} denotes 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 1. *At the end of every iteration, for any predicate p, $\delta(p)$ either reduces by 1 or remains the same.*

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

Proof sketch: If the modeled property contains p, then $\delta^{'}(p) = \delta(p)$ - 1, and since $\delta(p)$ is the smallest, and no other property's δ value can reduce by more than 1, p still remains among predicates whose δ value is minimum. On the other hand, if $\delta(p) > 1$ 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 $\delta(q)$ becomes smaller than $\delta(p)$ and p leaves P_{min} in the next iteration.

OBSERVATION 2. *If $\delta(p) = 1$ and if one of the properties selected to be modeled contains p, then p does not have to be modeled again.*

LEMMA 2. *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 is removed from P_{min}, but p remains. If p also is in the modeled property, then again the condition of the lemma is met.

LEMMA 3. *Once a P_{min} is chosen, P_{min} may lose some predicates from one iteration to the next, but it does 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 decreases, so that p stays in P_{min} through the next iteration. Now if $p' \in P_{min}$ is not in the modeled property, its δ value remains unchanged, and so it is removed from P_{min}. However, since for all $q \notin P_{min}$, the reduction in δ value is at most 1, they cannot enter P_{min}.

OBSERVATION 3. *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 4. *If P_{min} is singleton, then no new P_{min} can be constructed until all properties containing that predicate are modeled.*

Proof sketch: Follows from the previous lemma.

OBSERVATION 4. *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 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 4.2.1. *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.*

Proof sketch: Follows from previous observations and lemmas. 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. Remember Π_i denotes the set of properties until π_i.

Experimentally we find 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 Pro bus for the three distinct approaches to XFM.

4.3 Case Studies Based on Property Ordering

4.3.1 *ISA Bus Architecture.* The XFM approach is applied to formalize the monitor model of the ISA bus architecture, which is compact. Following the XFM guidelines, we first write the user stories from the natural specifications dictating the behavior of the ISA bus protocol. The user stories are converted into a series of LTL properties. These properties are then sorted in accordance with the schemes mentioned earlier and an incremental model is constructed using the XFM approach. 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 indicated in [Shanley and Anderson 1995]. Based on the functions, the signals are grouped into address signals, data signals, and system management signals. The bus clock drives the ISA bus and introduces the notion of timing. When the unit is initially powered up, the reset signal on the ISA bus remains asserted until the power supply voltages stabilize.

Furthermore, 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 is 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 a 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 in Table IV. The results of the state space search for the properties for all three approaches are illustrated in Section 4.4.

4.3.2 *Bus Arbitration of the Pentium Pro Processor.* One of the most important concerns of the Pentium Pro bus is how it handles arbitration between its symmetric and priority agents [Shanley 1998]. 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 is acquired, the request agent initiates the request phase of the transaction. Some of the behavior handled includes arbitration among the symmetric agents, arbitration by the priority agents and its effect on the symmetry agents. The user stories incorporate the behavior when the symmetric agent locks the bus and a priority agent is giving a request. The ability of bus parking

Table IV. LTL properties for the ISA bus

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	Whenever the data transfer takes place, the address is valid	G(dtstart → addressvalid)
6	When the device is powered and data transfer is complete, the current bus cycle ends in the next bus clock.	G((dtcomp && b_clock && power) → X endbuscycle)
7	When the 8 bit device is powered and if the data transfer can start in the same bus cycle, the lower path of the data bus is used to transfer the data.	G((isacardselect && dtstart && power && !endbuscycle) → X lowerpath)
8	When the 8 bit device is powered and if the data transfer can start in the same bus cycle, the upperpath of the data bus is 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, the lower path of the data bus is 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, the upper path of the data bus is 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, 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	Whenever 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, the upper path of the data bus is used for transfer.	G((highen && isacardselect && address && dtstart && power && !endbuscycle) → X upperpath)

by an agent is also modeled.

The Pentium architecture for 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

Table V. Sample user stories and LTL properties for Pentium Pro Arbitration

1	If only agent 2 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)*
2	If agent 2 and agent 3 are requesting 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*$\bar{0}$ *&&* Arbit *&& X !reset* $\rightarrow$ *X Rid*$\bar{2}$ *&& X busstate)*
3	If agent0, agent2 and agent3 are requesting the bus and if agent0 had the ownership of the bus previously then agent 2 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)*
4	If agent0 is the owner of the request signal group and one or more of the other agents are requesting the bus then agent0 deasserts its request and allows the other agents to arbitrate the ownership	*G(busstate && Rid*$\bar{0}$ *&& (breq1 || breq2 || breq3) && Arbit &&* $\rightarrow$ *!breq0)*
5	If agent0 and agent1 are requesting 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* $\rightarrow$ *X Rid*$\bar{4}$ *&& X busstate)*
6	If agent1 and the priority agent are requesting 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)*

the request signal group in order to communicate with an external device. The bus arbitration decides which of the processors gets the ownership of the bus, based on a built in rotational priority assignment scheme. In order 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. An arbitration event, defined as passing of ownership from one agent to another, occurs under the following circumstances: (i) *None of the agents are requesting during one clock and then one or more are seen requesting in the next clock.* and (ii) *The current symmetric owner of the request signal group relinquishes ownership of the bus and one or more of the other agents have been requesting the bus.* In either case, the symmetric agents must collectively decide which of them assumes ownership of the request signal group in the next clock. In the Arbitration event, when only one of the symmetric agents is requesting the bus, it gets the ownership of the request signal group. Whereas, if two or more of the symmetric agents are requesting 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 0, 1, 2, 3, 0, ... (Agent IDs). Each agent has a *bus state* and a *Rid*, which is used to keep track of the bus status and the ID of the current owner of the bus respectively. The processor may retain ownership after completing a transaction in case it needs 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. If an agent is requesting ownership within 4 arbitration events, it is 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 as one or more of the symmetric agents, the priority agent wins. The only case where the priority agent is unsuccessful in winning the ownership of the bus is the case where 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. Some of the user stories and the corresponding properties of the arbitration event are mentioned in Table V [Suhaib 2004].

4.4 Experimental Results

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

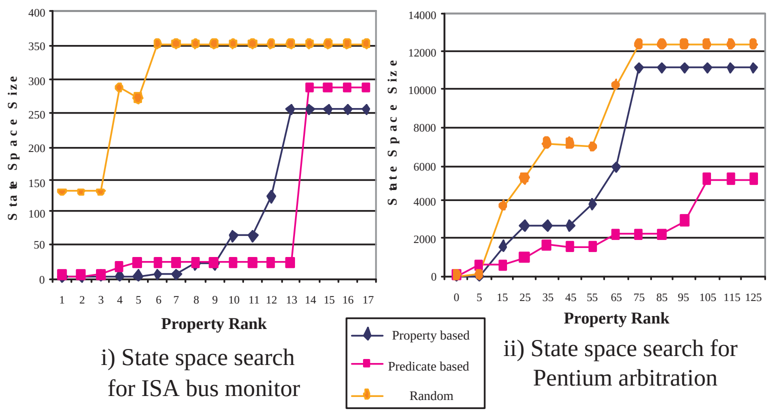


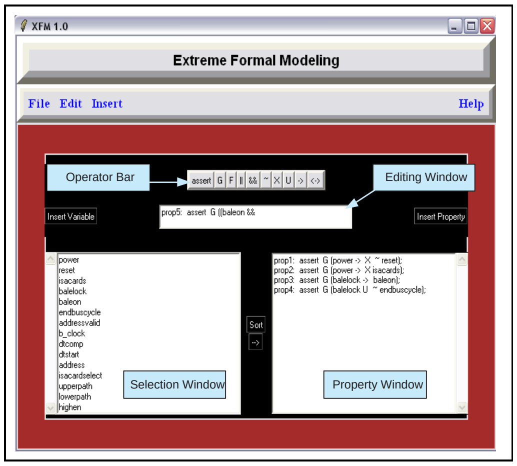
Fig. 3. State space search graphs

In the model of the ISA Bus architecture, 16 properties are modeled; they are derived from 14 distinct user stories and include 15 predicates. We find that during the ad hoc approach the model state space grows significantly large after modeling the first property. The main reason is that the property selected required the user to build the complete model in one shot in order to verify the property. This approach causes inclusion of irrelevant behavior in the model, which is not seen in the other two approaches due to an incremental modeling methodology. Therefore, the property based and the predicate based ordering result in a smaller number of states searched than the ad hoc modeling approach of properties. The number of states searched for verification by predicate based ordering results in better performance in terms of more properties being verified with a lower number of states searched than the property based ordering.

On the other hand, the arbitration model of the Pentium bus is more complex than the ISA bus model. The arbitration model consists of 126 properties from as many user stories and includes 24 predicates. The state space search graph for the verification of properties has a huge discrepancy between the model built by ad hoc ordering as to the other two approaches. The model built using the ad hoc approach becomes complex in the early stage of modeling requiring a large number of states to be searched to verify the properties. The property based sorting has a smaller state space search for the initial set of properties but the number of states for verification grows as more properties are added. After about 75 properties, the state space search becomes stable since the model is complete at this point. This ordering approach is better than the ad hoc approach, but the predicate based property ordering gives better results with the graph of the state space search being more incremental throughout. The results of the state space search of the ISA bus as well as the arbitration model are shown in Figure 3.

5. XFM GUI TOOLKIT

We develop a user-friendly Graphical User Interface (GUI) for users to build effective models by using our agile methodology and ordering schemes. The GUI is written using the *tcl/tk* toolkit [Ousterhout 2002] in *wish* version 8.4. The GUI interface of XFM is shown in Figure 4. 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 [McMillan 1993].



```
prop5:  assert  G ((baleon &&
```

```
power
reset
isacards
balelock
baleon
endbuscycle
addressvalid
b_clock
dtcomp
dtstart
address
isacardselect
upperpath
lowerpath
highen
```

```
prop1:  assert  G (power ->  X  ~ reset);
prop2:  assert  G (power ->  X isacards);
prop3:  assert  G (balelock ->  baleon);
prop4:  assert  G (balelock U  ~ endbuscycle);
```

Fig. 4. XFM GUI toolkit

The user can load a list of variables or properties from a file and can write or insert the properties in the property window. After the user has a complete list of properties in the property window, they can be sorted based on the ordering schemes selected. Three property ordering schemes are implemented: *Ad hoc, Property based and Predicate based* ordering. 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 tool accepts the properties in the format used by SMV. If the format of the properties does not adhere to the specified format, SMV gives an error. Once the properties are sorted, a new window shows the sorted properties. The user can develop the model in the window directly or use an editor of his/her choice. To follow the XFM approach, the user has to manually comment (/*...*/) the properties other than the one that is being currently modeled. The user has to remove the comments 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. Once verified, the user can return to the tool and follow the XFM methodology until all the properties are modeled. The XFM toolkit can be downloaded from [Suhaib et al. 2004].

6. CONCLUSION

We have developed an incremental methodology for building PFMs by adding user stories expressed as LTL formulae gleaned from the natural language specifications, one by one, into the model. We have applied XFM on various case studies such as a traffic light controller, and the DLX pipeline using the SPIN [Holzmann 2003] and SMV [McMillan 1993] model checkers. We also have experimented with various property ordering schemes in XFM to make it a truly incremental development methodology. We analyzed the effect of these orderings while using XFM in building PFMs. We show the difference between the ad hoc ordering, property based ordering and predicate based ordering schemes. From the experimental results, we conclude that the predicate based ordering is better than property based and ad hoc schemes. We also formally justify such a conclusion. To facilitate our methodology for other users, we have also developed a toolkit for XFM that sorts the properties specified by the user, enables modeling and model checking using Cadence SMV. The tool can also be interfaced with other model checkers. We believe that this tool can successfully assist engineers in making effective formal models for verification purposes.

REFERENCES

Beck, K. 2000. *Extreme Programming explained: Embrace change.* Addison Wesley.

Bentley, B. 2001. Validating the Intel Pentium 4 Microprocessor. In *Proc. of Design Automation Conference*. 244–248.

Berner, D., Suhaib, S., Shukla, S., and Talpin, J. 2004. *Capturing Formal Specification into Abstract Models.* Kluwer Academic Publishers, 325–346.

Clarke, E. M., German, S. M., Lu, Y., Veith, H., and Wang, D. 2000. Executable protocol specification in ESL. In *Proc. of Formal Methods in Computer-Aided Design*. 197–216.

Hennessy, J. L., Patterson, D. A., and Goldberg, D. 2002. *Computer Architecture: A Quantitative Approach*, 3rd ed. Morgan Kaufmann, San Mateo, CA.

Henzinger, T. A., Jhala, R., Majumdar, R., and Sanvido, M. A. 2003. Extreme model checking. In *Proc. of International Symposium on Verification: Theory and Practice.* Lecture Notes in Computer Science, Springer-Verlag.

Herranz, A. 2003. The SLAM website.
http://lml.ls.fi.upm.es/slam.

Herranz, A. and Moreno-Navarro, J. 2003a. Formal extreme (and extremely formal) programming. In *Proc. of 4th International Conference on Extreme Programming and Agile Processes in Software Engineering, XP 2003*, M. Marchesi and G. Succi, Eds. Number 2675 in LNCS. Genova, Italy, 88–96.

Herranz, A. and Moreno-Navarro, J. 2003b. Rapid prototyping and incremental evolution using SLAM. In *Proc. of 14th IEEE International Workshop on Rapid System Prototyping, RSP 2003)*. San Diego, California, USA.

Holzmann, G. J. 2003. *The SPIN Model Checker: Primer and Reference Manual*. Addison Wesley, Boston, MA.

Li, P., Ravindran, B., Suhaib, S., and Feizabadi, S. 2004. A Formally Verified Application-Level Framework for Utility Accrual Real -Time Scheduling On POSIX Real-Time Operating Systems. *IEEE Transactions on Software Engineering 30,* 9 (September), 613–629.

McMillan, K. 1993. *Symbolic Model Checking*. Kluwer Academic Publishers.

Oddoux, D. 2001. LTL 2 BA : fast algorithm from LTL to buchi automata. http://www.liafa.jussieu.fr/~oddoux/ltl2ba/.

Ousterhout, J. K. 2002. *Tcl and the Tk Toolkit*, Third ed. Addison-Wesley Professional Computing Series. Pearson Education Inc.

Shanley, T. 1998. *Pentium Pro and Pentium II bus System Architecture*, Second Edition ed. Addison-Wesley Inc.

Shanley, T. and Anderson, D. 1995. *ISA System Architecture*, 3rd ed. Addison-Wesley Publishing Company.

Shimizu, K., Dill, D. L., and Hu, A. J. 2000. Monitor-based formal specification of PCI. In *Proc. of Formal Methods in Computer-Aided Design*. 335–353.

Suhaib, S. 2004. Incremental methodology for developing formal models. M.S. thesis, Virginia Polytechnic and State University.

Suhaib, S., Jhala, A., and Shukla, S. 2004. XFM toolkit. http://fermat.ece.vt.edu/XFM.html.

Suhaib, S., Mathaikutty, D., Berner, D., and Shukla, S. 2004a. Extreme formal modeling for hardware models. In *Proc. of 5th International Workshop on Microprocessor Test and Verification (MTV'04)*.

Suhaib, S., Mathaikutty, D., Berner, D., and Shukla, S. 2004b. Property ordering effects in an incremental formal modeling methodology. In *Proc. of Thirteenth International Workshop on Logic and Synthesis (IWLS'04)*.

Suhaib, S., Mathaikutty, D., and Shukla, S. 2004. Effects of property ordering in an incremental formal modeling methodology. In *Proc. of IEEE International High Level Design Validation and Test Workshop (HLDVT'04)*.

Synopsis. 2004. Vera: Testbench automation. http://www.synopsys.com/products/vera/vera_ds.html.

Verisity. 2004. Specman elite: Testbench automation. http://www.verisity.com/products/specman.html.

Wells, D. 2001. Extreme Programming: A gentle introduction. http://www.extremeprogramming.org/.

Williams, L. 2003. The XP programmer - the few minutes programmer. *IEEE Software 20,* 3 (May/June), 16–20.

Wood, W. A. and Kleb, W. L. 2003. Exploring XP for scientific research. *IEEE Software 20,* 3 (May/June), 30–36.