Extreme Formal Modeling (XFM) for Hardware Models

Syed Suhaib, Deepak Mathaikutty and Sandeep Shukla
FERMAT Lab
Virginia Polytechnic and State University,
Blacksburg, VA 24060.
{ssuhaib, damathai, shukla}@vt.edu

David Berner
Institut de Recherche en Informatique et Systèmes Aléatoires (IRISA/INRIA)
35042 Rennes, France.
david.berner@irisa.fr

Abstract

In this paper, we show the usefulness of an agile formal method (named XFM) based on extreme programming concepts to construct abstract models from a natural language specification of a complex system. Building formal models for verification purposes is being employed in the industry for two different usage modes: (i) Descriptive Formal Models (DFM) which, are used to capture an implementation into an abstract model to submit to analysis by model checking tools, (ii) Prescriptive Formal Models (PFM) which, are used to capture natural language specifications into a formal model to analyze consistency of the specification and also as a reference model to compare a DFM against it. We propose XFM as a methodology to incrementally build a correct PFM from a natural language specification. In this paper, using XFM, on various examples related to microprocessors, we build the models of DLX pipeline in SPIN, the ISA bus monitor and arbitration phase of the Pentium Pro bus in 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 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.

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.

We propose Extreme Formal Models (XFM) as a methodology to incrementally build a correct PFM from a natural language specification. PFMs can be used to generate checkers, coverage monitors and formal properties for verifying the DFM for the corresponding processors. Our approach is necessitated by the absence of a prescription on how to go about building these models from natural language documents in the literature. For example, in [6] SMV specifications of bus protocols are developed as PFM but the goal of these PFM is to check consistency or test generation and it does not prescribe any incremental methodology based on regression. In [1] 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. We have tried our hands on some of the ω-automata synthesis tools [3] and usually when the number of properties are sizable, such synthesis tools do not work well. It might be better and more feasible to construct the model by hand incrementally as is explained later in this paper and regressively model check it to ensure constructively correct models.

Our methodology is based on the ideas of Extreme Programming (XP). We show the flow and demonstrate the benefits of this XP based methodology with three hardware based examples: the DLX pipeline, monitor of the ISA bus and the arbitration phase of the Pentium Pro bus. Our experiments show that this methodology not only constructs abstract models in sufficiently shorter time than the time taken in constructing ad hoc abstract models from implementation or specification, but also provides models that are constructively correct and closer to the intended specification.

The paper is organized as follows. In Section 2, we discuss some of the rules of XP. In Section 3, we discuss the guidelines of XFM which is followed by case studies and its experimental results in Section 4 and 5. We conclude the paper with a brief summary in Section 6.

2 Extreme Programming

Extreme programming [9] (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 [11, 10]. 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. A complete reference of extreme programming can be found at [9, 11, 10].

3 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.

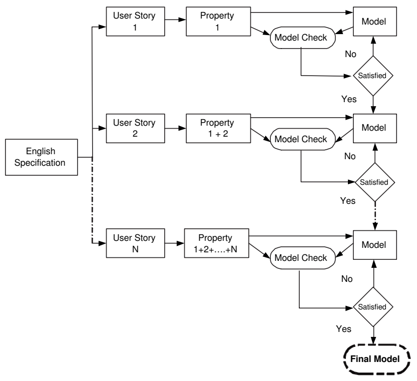

Figure 1. Capturing a formal model with XFM

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

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 building 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 1 gives a formal representation of XFM.

Algorithm 1 XFM Approach

```
{For a given system, we have the behavior in terms of a natural language specification that is converted into 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
Let $\Pi_i = \{\pi_1, \pi_1, ..., \pi_i\} \subseteq \Pi$, so $\Pi_i$ represents the first $i$ properties.
Let $X(\Pi_i)$ be the model that satisfies all 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.
    $X := X(\Pi_i)$
    $ModelCheck(X, \Pi_i)$ /* This is the regression step */
    if $ModelCheck$ fails for a $\pi_k$
      go to Step2 /* to change the model so the failing property can pass */
    else if $i = m$
      $X$ is the required model
    else
      go to Step1
```

4 Experiments

4.1 Model of a DLX pipeline control

The pipeline control of the DLX RISC processor model [2] 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 1 shows the cycle usage for the different instruction types.

Starting from this system description, we identify the first user story. One of the most basic behaviors 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

Table 1. 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	

$[](if \rightarrow Xid)$, 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 is 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 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 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 such as cat1 in Table 2.

Figure 2. PROMELA code for one single instruction

```
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;  }
```

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 will say that an instruction will not advance unless the enable signal is given. Again we obtain 25 properties in the style of cat2 in Table 2. The changes in the model for these properties are small, so all of them can be verified without prob-

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	$[]((if1 \&\& enable1) \rightarrow ((!(if2 \&\& enable2) \| \| (!(if3 \&\& enable3))$ $\| \| (!(if4 \&\& enable4)) \| \| (!(if5 \&\& enable5))) U !enable1))$

Table 2. LTL properties for pipeline (examples)

lems.

The following iteration is adding some synchronization. Our user story says 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.

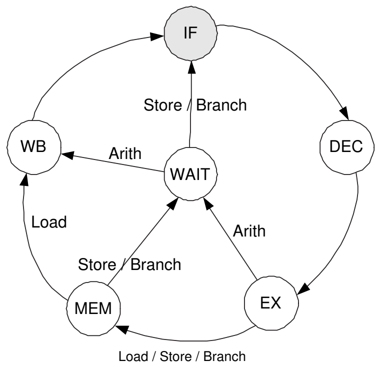

Figure 3. Automaton for one instruction

Another important behavior of a pipeline is to prohibit the multiple usage 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 2 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 are introducing a control process that in an initialization phase will start each instruction successively, and later makes sure that the 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 working.

To make the model of the pipeline a bit more realistic, we select the user story that defines the different instruction types and their different cycle sequences from Table 1. 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 1 (Figure 3). 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 2. Changes in the abstract model to reflect this are limited to update the FSM description for each instruction to the automaton of Figure 3 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 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, so we will not continue this for this paper.

4.2 ISA Bus Architecture

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 [5]. Based on the functions, the signals are grouped into address signals, data signals and the system management signals. The bus clock drives the ISA bus and brings in 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 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 with detail in [7].

4.3 Pentium Pro Processor's Bus Arbitration

One of the most important concerns of the Pentium Pro bus is how it handles arbitration between its symmetric and priority agents [4]. 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 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 ability of bus parking by an agent is also modeled. 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 is the passing of the ownership from one symmetric agent to another. It occurs under the following circumstances:

- *None of the agents is requesting during one clock cycle and then one or more are seen requesting in the next clock cycle.*
- *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 access to 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 cycle.

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. Though, complete set of user stories and their corresponding properties are illustrated in [7], we mention some of the user stories here.

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

G(*!breq0 && !breq1 && breq2 && !breq3 && Arbit && X !reset → X Rid2 && X busstate*)

Whereas, if two or more of the symmetric agents is 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 0, 1, 2, 3, 0, 1 (Agent ID's).

User Story : If agent 2 and agent 3 are requesting access to the bus and if agent 0 currently owns the bus then agent 2 wins the arbitration and gains ownership of the bus.

G(*!breq0 && !breq1 && breq2 && breq3 && Rid0 &&* Arbit *&& X !reset → X Rid2 && X busstate)*

Each agent has an 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.

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 access to the bus then agent0 deasserts its request and allows the other agents to arbitrate the ownership.

G(busstate && Rid0 && (breq1 || breq2 || breq3) && Arbit && → !breq0)

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

5 Experimental Results

The ISA Bus and the arbitration phase of the Pentium bus were modeled and results of their state space growth were recorded. Both these models differ in the number of properties to be modeled as well as the size of the model. Vacuity check was also performed on all the properties to ensure that they are not vacuously true. In the model of the

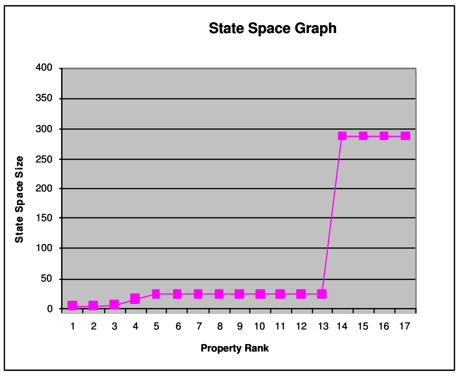

Figure 4. ISA bus state space graph

ISA Bus architecture, 16 properties were modeled. These properties were derived from 14 distinct user stories and 15 predicates. The data for the state space reached for each property were noted after they were verified. The results of the state space search of the ISA bus model is shown in Figure 4. The order in which is the properties were modeled was selected based on a predicate-based ordering scheme which we define in [8]. It can be seen from the graph that the state-space growth is incremental with the growth of the model. While modeling the 14th property, we had to add a large factor of non-determinism into the system and as a result the state space increases drastically which can be seen in the graph. The reason for the addition of the non-determinism is due to the lack of knowledge of the behavior of some predicate modeled for that property. On the other

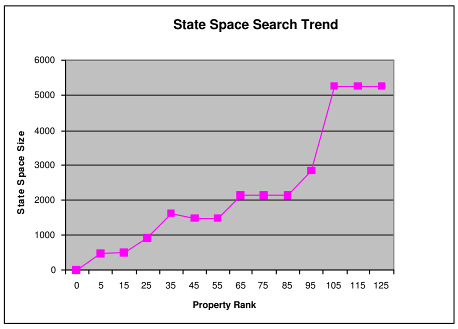

Figure 5. Pentium bus state space graph

hand, the model of the arbitration phase of the Pentium bus is more complex than the model of the ISA bus model. The arbitration model consisted of 126 properties with 24 predicates. The results of the state space search of the arbitration phase model of the Pentium Pro bus is shown in Figure 5. It can be seen that our approach of modeling scales well for large systems and the state space growth is incremental with the growth of the model.

6 Conclusion

We present and demonstrate the usage of agile methods for the formal modeling of hardware systems. The presented methodology focuses on the concept of incremental formal modeling based on properties from a natural language specification. Each property represents a specific behavior based on which the abstract model is constructed. Since XFM involves an iterative technique, the evolving abstract model facilitates debugging whenever a property is found unsatisfied. In each iteration step the behavior is confined by adding additional properties and details to the abstract model. The fact that the behavior of the abstract model is closely linked to the properties entails a close to complete set of properties once the abstract model is complete. In the conventional approach, however, the abstract model tends to contain much more functionality than specified, but less properties than needed as there is no mechanism that provides for the exposure of all properties contained in the specification.

References

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

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

[3] D. Oddoux. LTL 2 BA : fast algorithm from LTL to buchi automata. http://www.liafa.jussieu.fr/˜oddoux/ltl2ba/.

[4] T. Shanley. *Pentium Pro and Pentium II bus System Architecture*. Addison-Wesley Inc., second edition edition, 1998.

[5] T. Shanley and D. Anderson. *ISA System Architecture*. Addison-Wesley Publishing Company, 3rd edition, 1995.

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

[7] S. Suhaib. XFM: An incremental methodology for developing formal models. Master's thesis, Virginia Tech, May 2004.

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

[9] D. Wells. Extreme Programming: A gentle introduction. http://www.extremeprogramming.org/.

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

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