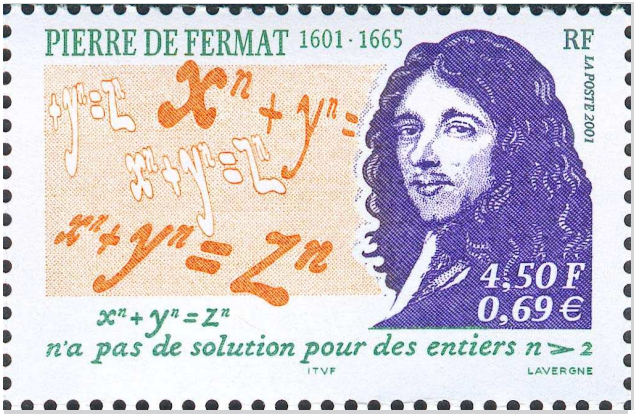


F E R M A T

Formal Engineering Research using Methods, Abstractions and Transformations

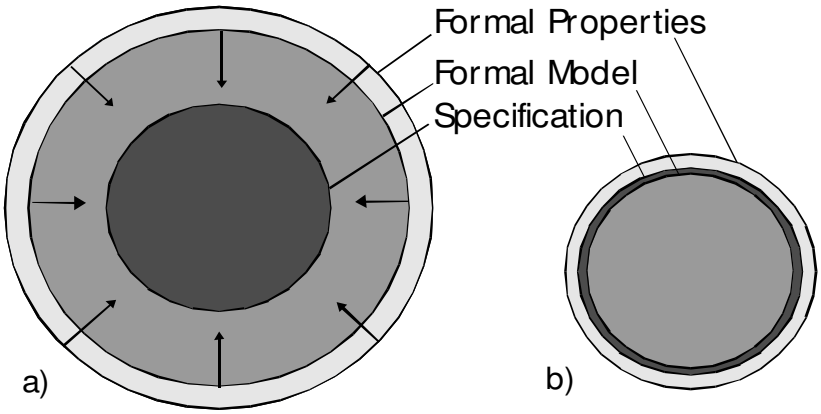
Technical Report No: 2003-08

XFM: Extreme Formal Method for Capturing Formal Specification into Abstract Models

David Berner, Syed M. Suhaib,
Sandeep K. Shukla and Harry Foster

david.berner@irisa.fr,
{ssuhaib, shukla}@vt.edu,
harry@jasper-da.com

Abstract -- In this paper 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, 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. 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. Our experiments show that this methodology not only constructs abstract models with sufficiently shorter time than the time taken in constructing ad hoc abstract models from implementation or specification, but also provides models which are constructively correct and closer to the intended specification.



VirginiaTech
VIRGINIA POLYTECHNIC INSTITUTE AND STATE UNIVERSITY

XFM: Extreme Formal Method for Capturing Formal Specification into Abstract Models

David Berner
INRIA/IRISA
Rennes, France

Syed Suhaib
Sandeep Shukla
Virginia Tech
Blacksburg, VA USA

Harry Foster
Jasper D.A.
Mountain View, CA USA

Contents

List of Figures

List of Tables

Abstract

In this paper 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, 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. 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. Our experiments show that this methodology not only constructs abstract models with sufficiently shorter time than the time taken in constructing ad hoc abstract models from implementation or specification, but also provides models which are constructively correct and closer to the intended specification.

1 Introduction

Extreme programming (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 signifi cantly while producing higher quality error free code [11] [10]. Some of the features of extreme programming are use of 'user stories', 'test-fi rst' development, 'refactoring', 'continuous regression' etc., which we have found very useful in creating more dependable software if applied properly. In the spirit of our successful use of extreme programming in software development, we decided to experiment with a parallel methodology in formal model construction. The initial results look very promising and are reported in this paper.

In our experience, both in carrying out formal verifi cation for major microprocessor chips, as well as, working on tools and methodologies, we have found that the major challenges faced by industrial formal verifi cation engineers are two fold: (i) Making sure that the natural language specifi cation of the system is translated into a suffi ciently complete set of formal properties to be used in model checking of an implementation, (ii) In conformance based formal verifi cation using abstraction techniques, creating an abstract model which satisfi es all formal properties intended in the natural language specifi cation. Most of the times, it is hard to validate the suffi ciency/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/allowedby the natural language specifi cation.

Our XFM based methodology addresses these problems, and with two illustrative examples (of a control intensive traffi c light controller, and the DLX pipeline) we present this methodology and show the benefi ts. Our experiments show that this methodology not only constructs abstract models with suffi ciently shorter time than the time taken in constructing ad hoc abstract models from implementation or specifi cation, but also provides models which are constructively correct and closer to the intended specifi cation.

Figure 1 and 2 show our comparison of the current state of the art in capturing formal specifi cations against our XFM approach. Figure 1 shows that an ad hoc abstract model is usually built from an English specifi cation and checked against formal properties with a model-checker. Some times, to make

matters worse, the ad hoc abstraction is built from an implementation itself, which is then checked against the implementation for conformance. In our view, that defeats the purpose of formal verification, except that the abstraction step might uncover bugs unknown to the implementors. There are several drawbacks in this approach. First of all, the ad hoc building of both the model and the properties is error prone and the effort of model building and debugging grows exponentially 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. Often implementation detail gets into the abstract model. These tendencies might make the model have undesirable properties and hence the implementation being checked against it may have those too. Also implementation detail in the abstract model may introduce unwanted complexity and may later cause problems in a conformance check.

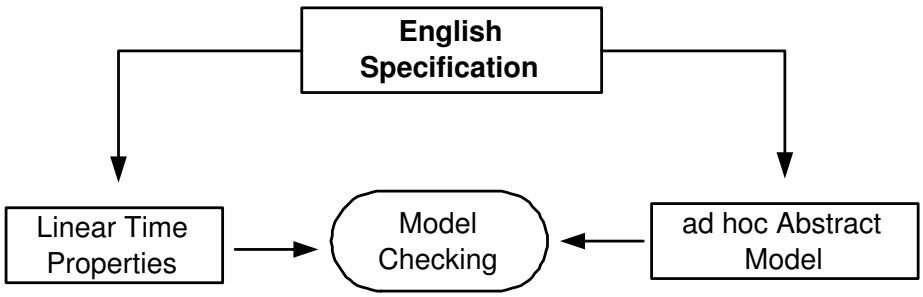


Figure 1. State of the art to capture a formal model

Figure 2 presents XFM's incremental approach to formal modeling. From the English specification, we first derive a simple formal property, then 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 spec (Figure 2). One way to make sure that it does is by simulating the model. The controlled and incremental model building results in a compact, structured abstract model. 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 linearly along with the size of the model.

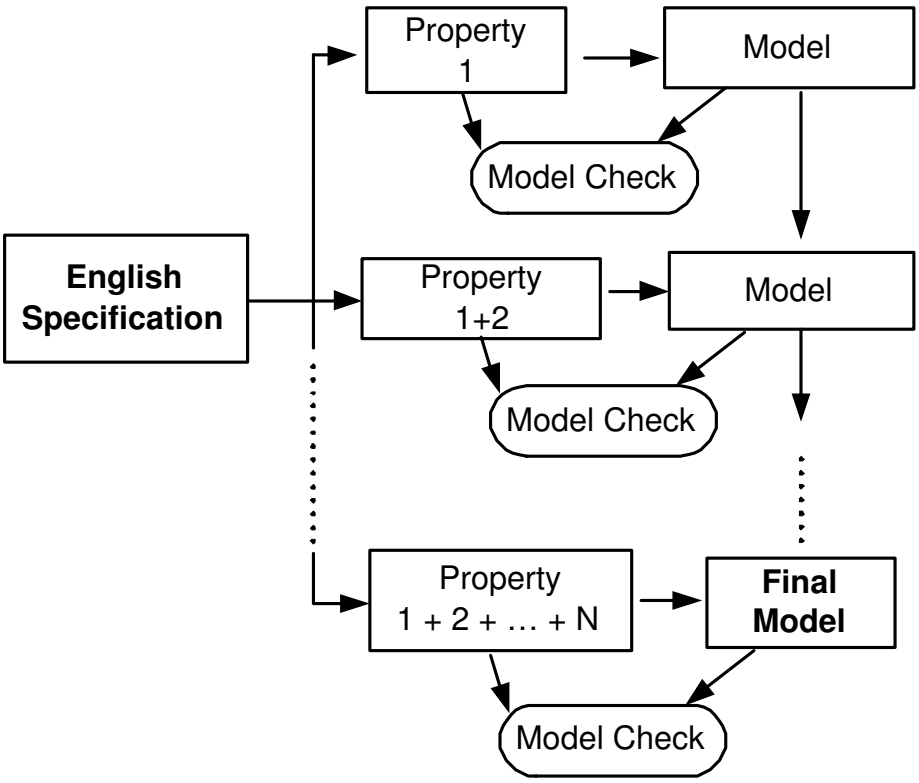


Figure 2. Capturing a formal model with XFM

For illustration purposes, we are considering linear-time properties, and using the SPIN [8] model checker. Most of the linear-time properties we use are in the form of LTL (linear-time temporal logic) [4]. Generating Finite State Machines (FSM) from LTL expressions allows for a visual representation of the expression. From this representation we can verify whether the original LTL expression was correctly constructed. These properties can be supplied to the SPIN model checker [8]. Properties that are not expressible in LTL can be modeled as an automaton description, which also can be supplied to SPIN as a property.

The remainder of this paper is organized as follows. Section 2 describes related work in the field. Section 3 points out the major contributions. In Section 4 we discuss in detail the methodology of XFM. In Section 5 we present two examples for XFM and Section 6 concludes this paper.

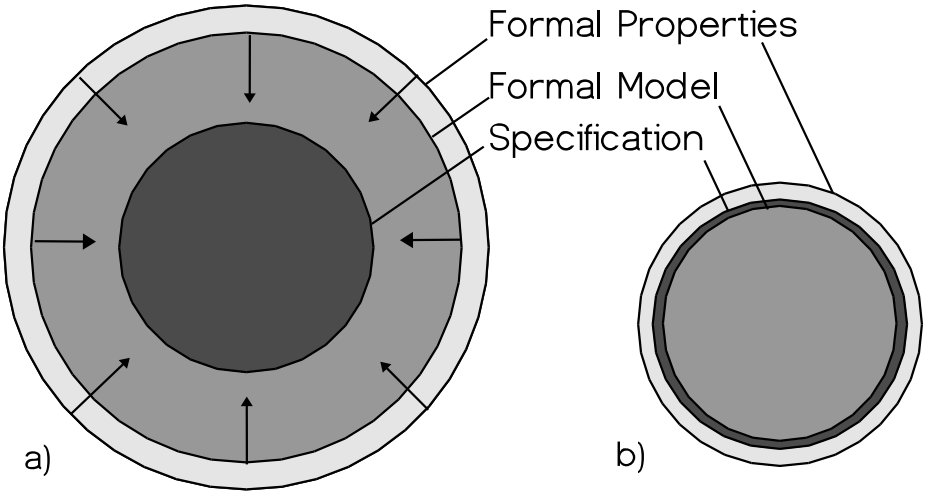


Figure 3. Modeling process (a) and modeling result (b)

2 Related Work

Although an emerging field, there exists a lot of literature about extreme programming [3] [10] [11], as well as many online resources e.g. [1]. However, hot many seem to have discussed these techniques for formal modeling. [6], and [7] connect formal methods and XP using formal declarative specifications in an asserting based JAVA development framework generating JAVA code. Their environment just generates sequential programs, so there is no notion of concurrency for hardware development.

3 Major Contributions

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 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 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 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.

4 Modeling approach

4.1 Tools

As our model checking environment we chose the SPIN model checker because it is one of the most popular model checkers in todays modeling of concurrent systems. However, we could use any model checker for the same compelling results. For SPIN, the models are specified in PROcess MEta LAnguage (PROMELA), a system description language. Its basic building blocks are asynchronous processes, message channels, synchronizing statements, and structured data. Once the model is built, the user can simulate it with the built-in simulator and verify formal properties. Verification properties can be entered in LTL or in the form of PROMELA never claims, for properties that are not expressible in LTL.

LTL is the leading technique for specification of temporal rules. It extends propositional logic with the four operators “always” (condition holds always in the future), “eventually” (condition holds sometime in the future), “next” (condition holds in the next cycle), and “until” (condition A holds until condition B, afterwards do not care). All LTL operators are listed in Table 1. As to the expressiveness of LTL, it is complete with respect to first order logic [5]. Temporal expressions that cannot be expressed in LTL, can be provided to SPIN in the form of a never-claim automaton.

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, if we move one parenthesis before the <>, it is a 7 state automaton. Even if these kinds of mistakes are hard to detect, it is especially important that properties are correctly defined before checking the model for it. There is a tool called LTL 2 BA [2] (LTL to Büchi automata) that generates a Büchi automaton representing any LTL expression. This visualization is instrumental in verifying that the expression matches the specification. LTL 2 BA also generates PROMELA code from an LTL expression. Therfore it could - in theory - be used to obtain the abstract model directly and automatically from the LTL properties. However, in practice this does not work since it is neither possible to describe concurrent processes in LTL nor to describe implementation details such as initial states or changing state variables.

Table 1. LTL Operators

!A	negation
A→B	implication
A ↔ B	equivalence
A && B	and
A || B	or
[]A	Always
A U B	until
<> A	eventually
X A	next

4.2 Extreme Programming Techniques

As stated earlier, XFM works on the lines of XP. 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. In XFM this rule maps to specifying the LTL property before writing the abstract model. Another important XP technique is to add functionality as late as possible, keeping the model simple for as long as possible. 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 as much as possible, 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. The benefit of other XP techniques such as a stand up meeting in the mornings, collective code ownership and moving people around depends on the type of the project, the size of the team, and on personal and corporate preferences.

4.3 Details of 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.

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 transforms it into an LTL property. The next step is to check if the LTL property correctly expresses the behavior of the user story. LTL 2 BA eases this step by displaying the corresponding FSM. If the property is sound, we start building the model corresponding to this property. It is important that while implementing the model only the behavior of this property is taken into account. If the model checker fails to validate the property, we can locate the error with the help of the trace file generated by the model checker, fix the bug and rerun verification. Once verification is successful we pick the next basic user story, transform it into an LTL property and extend the model obtained from the previous property to satisfy this one as well. This procedure is repeated until it contains everything that is specified in the English spec. The final model can be simulated to ensure that all specified functionalities are incorporated. If a certain functionality is found missing, we identify the corresponding LTL property and extend the model accordingly. After correct simulation, the model and the list of LTL properties should be complete.

never both have a green signal	$[]!(!c\&\&!p)$
If cars cannot go, they will go eventually	$[](c \rightarrow <>!c)$
No button pressed, cars keep going	$[]((!c\&\&!sw) \rightarrow X!c)$
No button pressed the pedestrians cannot walk	$[]((p\&\&!sw) \rightarrow Xp)$
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 2. LTL properties for traffi c light (c = cars stop, p = ped stop, sw = button is pressed)

5 Examples and Results

In order to demonstrate the power of XFM we present two examples from different domains and of different complexity. A simple traffi c light will illustrate the main steps, tools, and techniques involved. The design of a DLX [9] microprocessor pipeline will show how this works for a bigger model, and how the model evolves with the incremental approach.

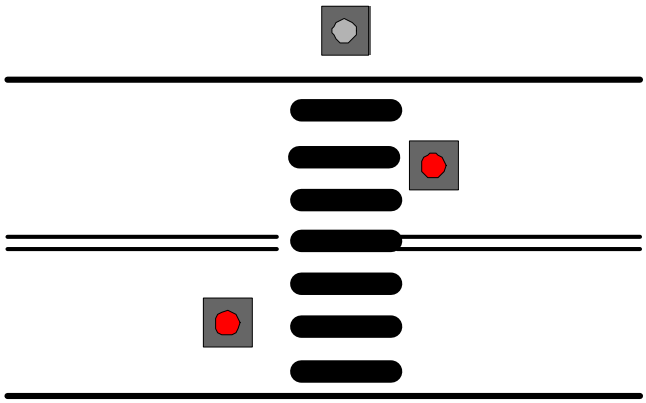


Figure 4. Sketch of the Pedestrian Traffi c Light

5.1 Traffic Light Model

This is a very simple example of a pedestrian crossing with a traffi c light (Figure 4). When a pedestrian pushes a button, the lights turn red, and the pedestrians can walk. After one minute the pedestrians

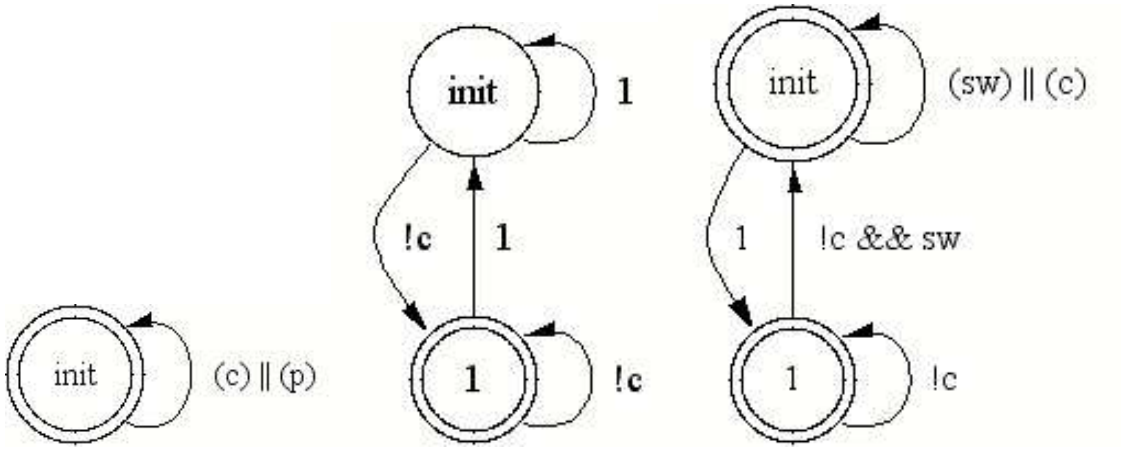


Figure 5. FSMs of Traffi c Properties 1 (a), 2 (b), 3 (b)

get a red light and the cars red light goes off. So this description is the English specifi cation. Now, we construct LTL properties describing this system. 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 specifi cation with LTL 2 BA. Figure 5(a) shows the automaton corresponding to this property. The corresponding model is just as simple, just one state.

The next property we come up with states that whenever the cars stop, they will eventually go (Figure 5(b)). Table 2 lists all LTL properties for this example. LTL does not allow to express exact timing, only relative occurrences of events. But in the model we add a timer that counts to 60 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 and cars stop.

The following two properties state that when no switch is pressed, the cars keep driving and the pedestrians keep stopping (Figure 5(c)). As we check these properties against the formal model, we realize that they can be verifi ed without making any modifi cations to the system, and a closer look at the properties shows that their behavior is already satisfi ed by property 1 (Figure 5(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 easy to fi gure out if it is correct (Figure 6). After implementing the functionality of these properties into the model, simulation shows that it works as specifi ed, so we have found all properties. Figure 7 shows the PROMELA code for the complete

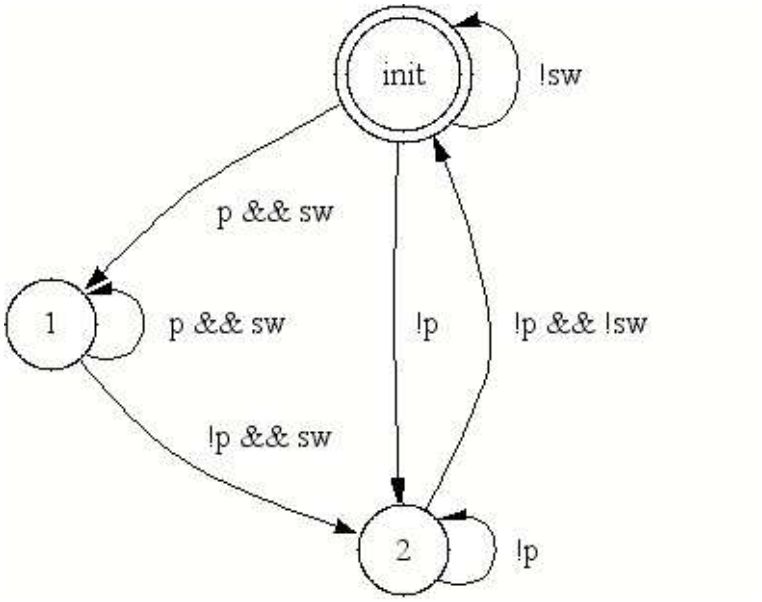


Figure 6. Graph for Traffi c Property 5

Traffi c light model.

Table 3. 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	

5.2 Model of a DLX pipeline control

The actual power of the XFM approach develops when working on large systems. The pipeline control of the DLX RISC processor model [9] 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 fi ve 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 3 shows the cycle usage for the different instruction types.

```
 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;
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();
}
```

Figure 7. Promela code for Pedestrian Crossing

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 $[](if \rightarrow Xid)$, 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 9(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 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 (Figure 11). 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 like cat1 in Table 4.

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

```
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 8. PROMELA code for one single instruction

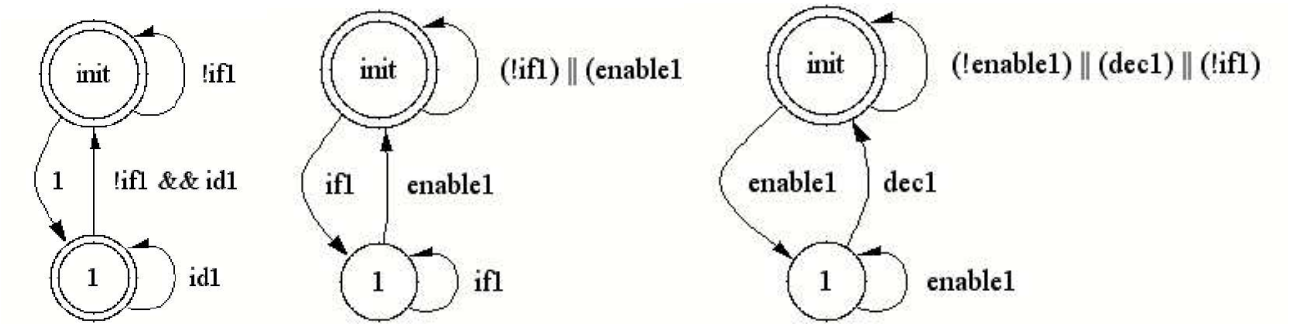


Figure 9. Graphs of pipeline properties 1 (a), 3 (b), and 4 (c)

advance unless the enable signal is given (Figure 9(b)). Again we obtain 25 properties in the style of cat2 in Table 4. 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. 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. This category of properties is somewhat more complex, but with the help of the LTL 2 BA tool we finally find cat3 in Table 4. It reads that whenever a stage is true, it will change to the next stage before the enable sign goes down, unless the enable sign is already low (Figure 9(c)). Again we get one of these properties for each stage that is 25. Once the correct

Table 4. LTL properties for pipeline (examples)

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

properties are found, the changes in the model are small, and after all properties verified (including the previous ones), we can check the correct behavior with the builtin SPIN simulator.

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 4 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 3. 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 3 (Figure 10). 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 4. Changes in the abstract model to reflect this are limited to update

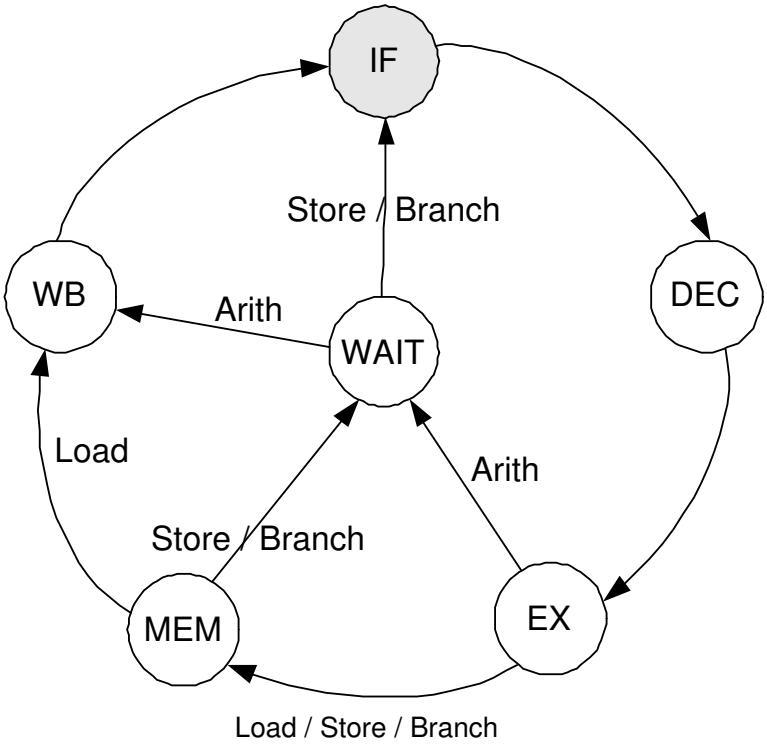


Figure 10. Automaton for one instruction

the FSM description for each instruction to the automaton of Figure 10 that means introducing the notion of an instruction type, and adding the transitions to and from the wait stage. For the control these changes are transparent since after the changes still each instruction takes 5 cycles to finish, therefore preventing the occurrence of structural hazards. Figure 11 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, so we will not continue this for this paper.

6 Conclusion

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 will grow linearly and get 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

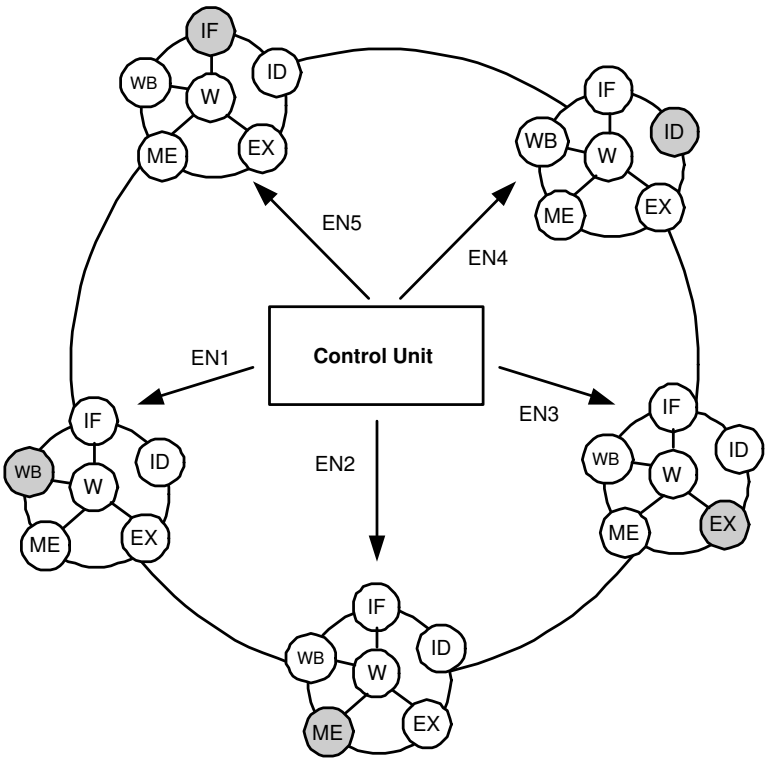


Figure 11. Overall Pipeline Design

functionality to the model, so the debugging process is much more directed. Another major benefi t resides in the scalability. For example if ad hoc models are built, then 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 built models using this methodology grows linearly in the size of the model, whereas the design effort in a conventional methodology grows exponentially with the size.

References

[1] Extreme programming: A gentle introduction. http://www.extremeprogramming.org/.

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

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

[4] E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B. Elsevier Science Publishers, Amsterdam, 1990.

[5] J. G. Henriksen. *Logics and Automata for Verification - Expressiveness and Decidability Issues*. PhD thesis, Basic Research in Computer Science (BRICS), University of Aarhus, Denmark, June 2000.

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

[7] A. Herranz and J. Moreno-Navarro. Formal extreme (and extremely formal) programming. In M. Marchesi and G. Succi, editors, *4th International Conference on Extreme Programming and Agile Processes in Software Engineering, XP 2003*, number 2675 in LNCS, pages 88–96, Genova, Italy, May 2003.

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

[9] D. G. John L. Hennessy, David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Mateo, CA, 3rd edition, may 2002.

[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 scientifi c research. *IEEE Software*, 20(3):30–36, May/June 2003.