PIERRE DE FERMAT 1601-1665
RF
LA POSTE 2001
4,50 F
0,69 €
$x^n + y^n = z^n$
n'a pas de solution pour des entiers n > 2

F E R M A T

Formal Engineering Research using Methods, Abstractions and Transformations

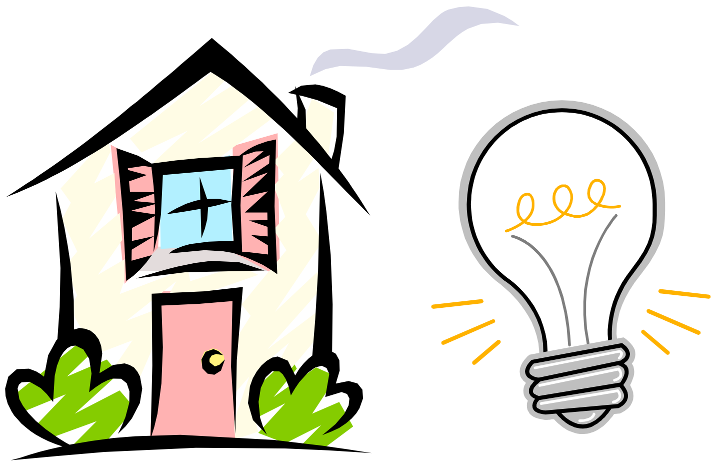
Technical Report No: 2004-01

Extreme Formal Modeling to Capture Specification for a Smart Building Control System into a Constructively Correct Model.

Syed M. Suhaib, David Berner
Sandeep K. Shukla

{ssuhaib, shukla}@vt.edu,
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 has been used in the industry for two different usage modes: (i) Descriptive Formal Models (DFM) are used to capture an implementation in an abstract model to submit to analysis by model checking tools, (ii) Prescriptive Formal Models (PFM) are used to capture natural language specifications in 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. To illustrate our methodology we chose PROMELA as modeling language, and linear time properties as specification language. However, our approach could be used for other modeling and specification languages. We illustrate the methodology with an extensive example of a Smart Building control. It features intelligent control and interaction of illumination, heating, cooling, safety, security, and appliances. We find this methodology very useful as an well defined way to go about constructing PFMs from documented specifications and guarantee correct by construction PFMs. From our experience in industrial formal verification, we have found verification engineers grappling with how to go about building PFMs, and we position XFM as a prescription for verification engineers. We recognize that expert formal Methodists can build correct PFMs without following through such an incremental methodology, but for wide acceptance of formal methods and verification by designers and validation engineers XFM will be quite useful. It must be noted that our approach in using extreme programming ideas to formal modeling is distinct from a recent work by Henzinger et.al., where they show how to reuse a 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..



VirginiaTech
VIRGINIA POLYTECHNIC INSTITUTE AND STATE UNIVERSITY

Extreme Formal Modeling to Capture Specifications for a Smart Building Control System into a Constructively Correct Model

Syed Suhaib
Virginia Tech
Blacksburg, VA USA

David Berner
INRIA/IRISA
Rennes, France

Sandeep Shukla
Virginia Tech
Blacksburg, VA USA

Contents

List of Figures

List of Tables

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 used in the industry for two different usage modes: (i) Descriptive Formal Models (DFM) are used to capture an implementation into an abstract model to submit to analysis by model checking tools, (ii) Prescriptive Formal Models (PFM) 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. To illustrate our methodology we chose PROMELA as modeling language, and linear time properties as specification language. However, our approach could be used for other modeling and specification languages. We illustrate the methodology with an extensive example of a Smart Building control system. It must be noted that our approach in using extreme programming ideas to formal modeling is distinct from a recent work by Henzinger et.al., 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.

1 Introduction

Due to increasing recognition of the need for formal verification and formal methods in hardware and software industry, there has been a surge in activities in the industry to build formal models and apply formal model checking, or use the models for test generation (e.g. [17]). From our experience, building formal models for verification purposes is being used in the industry for two different usage modes: (i) Descriptive Formal Models (DFM) [2] are used to capture an implementation into an abstract model to submit to analysis by model checking tools, (ii) Prescriptive Formal Models (PFM) [4, 18, 3] 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 Extreme Formal Models (XFM) as a methodology to incrementally build a correct PFM from a natural language specification. 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 [18, 17] 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 [4] 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 [14] 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 the paper and regressively model check it to ensure constructively correct models.

To illustrate our methodology we chose PROMELA [12] as modeling language, and linear time properties as specification language. However, our approach could be used for other modeling and specification languages as well. We illustrate the methodology with an extensive example of a Smart Building control. It features intelligent control and interaction of illumination, heating, cooling, safety, security, and appliances. We find this methodology very useful as a well defined way to go about constructing PFMs from documented specifications and guarantee correct by construction PFMs. From our experience in industrial formal verification, we have found verification engineers grappling with how to go about building PFMs, and we position XFM as a prescription for verification engineers. We recognize that expert formal methodists can build correct PFMs without following through such an incremental methodology, but for the wide acceptance of formal methods and verification by designers and validation engineers XFM is quite useful. It must be noted that our approach in using extreme programming (XP) ideas to formal modeling is distinct from a recent work by Henzinger et.al. [9], where they show how to reuse previous model checking results to incrementally model check the modification of a model. Their technique can be plugged into XFM in the essential regression steps of XFM, though.

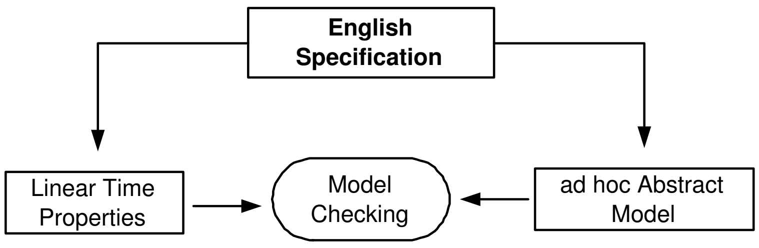


Figure 1. State of the art to capture a formal model

In a typical industry PFM modeling flow (Figure 1), a formal model checker is used to verify properties of the model. There are several drawbacks in this approach. One is that the 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. Another problem is that there is no way of guaranteeing the inclusion of all properties, thus reducing the significance of the model. Finally, there is a tendency that the model includes more functional details than its specification describes. Implementation details can get into the abstract model, that make it have unwanted properties.

XFM attempts to overcome these problems and restrictions. It exploits XP techniques in order to capture formal specification into abstract models. Figure 2 depicts XFM's incremental approach to formal modeling. From a spoken language specification, we first derive a simple formal property and then build an abstract model for this property. After that, we check whether this property holds for the model. Once this property is verified, 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 functionality describes in the natural language specification. Whenever a property fails to validate, it usually is straightforward to find the bug as it must be related to the latest additions. The overall effort of modeling and bug fixing grows linearly along with the size of the model.

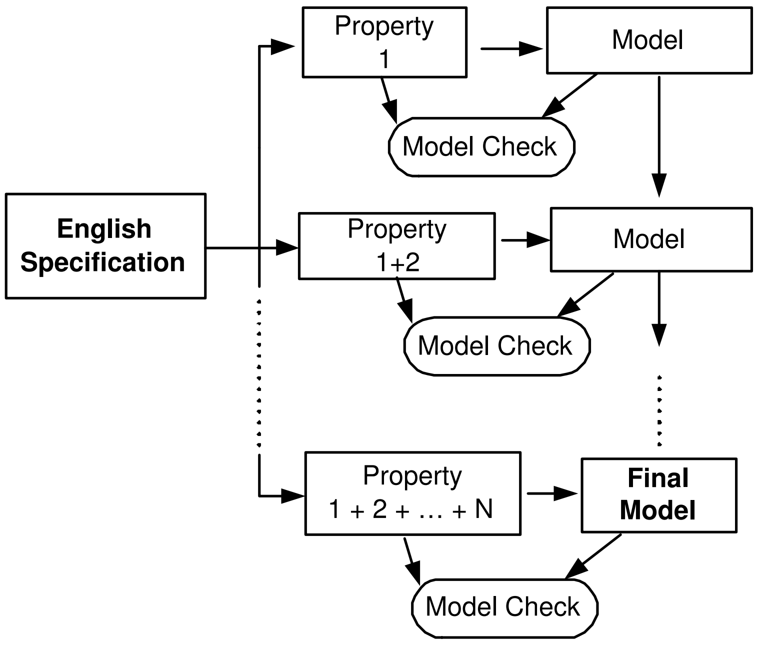


Figure 2. Capturing a formal model with XFM

1.1 Motivation for the Smart Home example

As technology advances, the cost and size of electronic components are becoming cheaper, smaller, and consume less power. As a result, these components are becoming an increasingly important part of our environment. Ubiquitous computing is no more science fiction, but instead an emerging technological area. There are electronic devices such as sensors, cameras, personal assistants, and microprocessors that provide us with information and transparently perform tasks without our knowledge of its working. This is what makes our environment "smart". A Smart Home uses these techniques to make living conditions more convenient and adapts to its residents' needs. A truly "smart" home does not directly burden its residents with technology, but reduces the presence of technology by automating tasks of everyday life. However, for "smart"-ness, the technology must be constructed such that the resident is unaware of the complex technology behind the automation. So, one important objective of a Smart Home is to take charge of obvious and repetitive tasks. Although the need for explicit control is taken away, the resident has supervisory control over the system.

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

2 Related Work

General information about XP and agile techniques can be found in [23, 7, 1, 22]. There are some projects that also use agile methods in the domain of modeling and verification. Herranz and Moreno-Navarro from TU Madrid describe in [11, 10] the integration of some XP practices to formal methods using the SLAM software tool [8]. Their environment generates sequential programs from formal expressions using an assertion based JAVA development framework. While our work involves the use of XP to model complex concurrent hardware systems their approach is directed towards sequential software programs. Henzinger et.al. [9] show how to reuse previous model checking results to incrementally model check the modification of a model. While this a completely different approach, these technique can be used in XFM in order to speed up the single model checking steps. Besides the typical method to capture formal models, there is other research going on to speed up this process, such as the automatic synthesis of models from temporal constraints [20], but high quality models are still created manually by qualified engineers.

In this paper, we formally model a Smart Home system and verify its properties. A lot of research [5, 16] as well as implementation models of Smart Homes exist. The OXYGEN project of MIT [13] focuses on pervasive human centered computing and an intelligent room. The Internet Home built by Cisco [21] involves security, remote monitoring, and control of appliances. The Adaptive House built by the University of Colorado [15] also contains appliances, entertainment centers as well as temperature and lighting control units. However, it has the approach of a non-invasive technology, where usage patterns are learned over time and then successively automated. This is in contrast to the Cisco Internet Home where the control is fixed and the focus is on the ability to remotely control the home environment.

3 Modeling Approach

3.1 Extreme Programming

Rules of XFM are based on rules of XP. Many XP rules do directly apply to XFM. There is not one XP methodology, but rather a collection of techniques. One basic XP technique is to define “user stories” - individual cards that point out specific implementation details and requirements. These user stories act as a detailed guideline for the programmer. Then there is a rule to write tests before the actual code. These tests strictly adhere to what is defined in the user stories. Then the actual implementation is programmed in a way, that just these tests are satisfied. Aspects like generality and extensibility are ignored. User stories are successively added in incremental steps, always building upon the previous model. Whenever the model seems inappropriate for the current user story or when its implementation seems circumstantial, it is considered to refactor the model. Whenever a bug is found, the associated test is updated, and after each iteration step, all tests are run to check for broken code. Other XP techniques that proved to be beneficial to most programming projects are collective code ownership, working in pairs, and a short team meeting every morning.

3.2 Modeling Steps

Figure 2 illustrates the flow of the presented modeling approach and Table 1 shows the single steps that have to be performed in order to follow this methodology. In a first step, a user story has to be defined. As in XP, it describes a simple but precise aspect of the system. In a second step this aspect is expressed in linear time. As a third step a model has to be built that expresses only this first property. If there is already an existing model, it has to be extended with just the functionality noted in the current user story. After the model satisfies the current property, all previous properties have to be checked for in a forth step and in case a property in not valid any more, the model has to be changed accordingly. The fifth step asks to check if there are more user stories to cover. If yes, we continue at step one, otherwise the model-building process is finished.

Table 1. XFM modeling steps

1	Write down user story
2	Express it as a linear time property
3	Construct a model that expresses only this first property or extend existing model to just satisfy this property
4	Make sure the model still satisfies all previous properties
5	If there are more user stories left, go back to step 1 stop if all specified functionality is incorporated

3.3 Difficulties and Details

The XFM steps are in detail not always as simple as they might seem. But as most problems encountered are known from the traditional capturing of formal models, they can be dealt with in the same way. The first challenge is to get the natural language specification complete and correct. Then, the order in which the user stories are specified can make a difference in the complexity of the model. There is no ultimate answer for this, but experience helps a lot and even if an initial choice was not so good, the model can be simplified in a later refactoring step. The building of formal properties is not always straightforward. However, it can be assisted with methods and tools such as specification patterns [6] and property elucidation [19], and their sanity can be checked for with tools such as LTL 2 BA [14], which transforms an LTL property into a finite state machine. Step three of the process requires - just as in XP - the discipline of the engineer to only model the functionality specified in the property. This is also the stage where refactoring has to be done when needed in order to keep the model concise. In the last step we have to make sure, that the model contains all specified functionality. Again there is no general solution for this, but simulating the model is one way to get more confidence about this. Contradicting properties are discovered when during an iteration not all properties can be satisfied any more.

Putting this new methodology to work does not mean ignoring all previous results in overcoming known obstacles, but contrarily it encourages the use of all these techniques for the single steps to obtain an even better result. As always, results significantly improve and can be obtained faster once the engineer gets more experienced and familiar with the process.

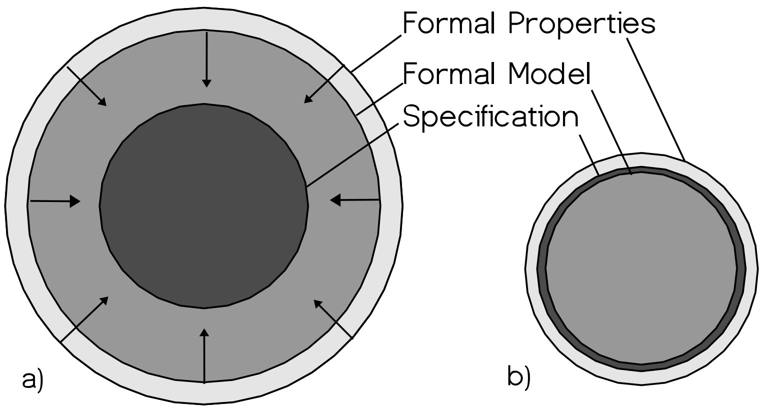


Figure 3. Behavior during the modeling process (a) and for the modeling result (b)

3.4 Advantages

Following an XFM based methodology, brings about a number of advantages. One of them is speed. The small debugging steps between the iterations can be done faster than debugging the complete model in the end. Besides speed, the quality of the resulting formal model is higher. Figure 3a illustrates how the amount of behavior for the properties and the abstract model develop during the capturing process. At any point, the behavior of the formal properties is more general than that of the abstract model. Both are more general than the behavior of the specification. In each iteration step their behavior is confined by adding additional properties and details to the model. Since the specification is not altered, the behavior of the specification does not vary during this process. At the end of the capture of the model ideally all three behaviors are identical, but in practice there are always small gaps (Figure 3b). The gap between the formal properties and the specification indicates how much system functionality is not expressed in the formal properties, and the gap between the formal model and the specification marks the amount of functionality the model contains that is unaccounted for in the specification. The latter is small because we only add functionality from simple user stories. The former gap is small as the process makes sure that the gap between properties and model never gets big in the first place.

4 The Smart Home Case Study

4.1 Natural Language Specification

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

4.1.1 Lighting.

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

4.1.2 Temperature Control.

Temperature control involves temperature sensors in every room, and the control of heating and air conditioning. For each room, three temperature levels can be defined: comfort level, power-save level, and the vacation level. The comfort level is the temperature the resident likes to have when the room is being used. The power-save level defines the temperature for a room that is not being used. It saves power by lowering AC and heating, but still keeping the temperature at a level so that the room can reach comfort level in a reasonable time. The vacation level defines the temperature for a room that is not used for a longer period of time. In the vacation level, only heating is operated to prevent the room from freezing.

4.1.3 Security.

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

4.1.4 Safety.

Safety is dealing with malfunctioning devices which may cause unforeseen hazards such as fire, flooding etc. One of the main components of safety are the smoke detectors which are installed in every room to monitor smoke levels. Also it is monitored if the temperature is in a safe range. Once a certain safety constraint is violated, a safety alert is raised and displayed on the screens available in every room. In addition a message is sent to the residents phone and email box. If a hard safety constraint is violated, fire extinguishing sprinklers are activated and the fire department is notified.

4.2 Capturing of the Formal Model

In order to capture the formal model we identify a basic functionality of the system, specify a formal linear time property for this functionality and then build a model that satisfies this property. Our first property states that if it is sufficiently bright outside, the light is never switched on in the room. Table 2 shows the corresponding LTL property. To make sure that this property expresses what we intend to express, we enter the LTL formula into the LTL 2 BA tool in order to get the corresponding automaton. Figure 4(a) shows the FSM corresponding to property 1 and Table 3 lists the definitions necessary to verify it with SPIN. Since this property is quite simple it is not difficult to see, that it expresses the correct behavior, but for more complex properties it is important to make sure they are correct before checking the model for it. Now we start writing a model in PROMELA that satisfies exactly this one property. Here it is important not to introduce more functionality than specified. However, the model always contains parts that have no correspondence in a formal property. This is because certain things such as initialization of variables and changes of state variables can not be expressed in linear time properties. Figure 5 shows the PROMELA model for the first property. It features the initialization process *init* and

Table 2. LTL Properties for the Smart Home model

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

the process *KLuminosity*, where the variable *kactive* is set according to the current luminosity. In order to keep the state space of the model small, we only consider three values of brightness. All other values do not result in any change of *kactive*.

Only after the first property is verified, we come up with a second property. This property states that if the room has not been used for more than 10 minutes, the light is always off. The corresponding LTL property in Table 2 is similar to property 1. A process *KPsense* has to be added to the model that detects if a person is in the kitchen and it counts the minutes since the kitchen was used last. If nobody has been there for more than 9 minutes, the light is switched off.

Property 3 describes that if there is light in the kitchen, it must be sufficiently dark outside and someone has used the kitchen within the last 10 minutes. For the formal model, we only have to add small

Table 3. Definitions for the LTL Properties

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

changes to reflect this property. Basically we have to make sure that the light is dependent on *kactive*. Once these changes are performed, the property verifies and we determine the next property. Number 4 defines when the light is off in the kitchen. If the light is off either nobody has been using the room recently or it is sufficiently bright without light. This property is a good example for the interactive model building process. While implementing the changes in the model we realize that when it is just getting bright in the morning, the light is switched on just before *klactive* is switched off. This causes the property to fail. Now if we inverse the assignments, property 3 fails. This is why we extend the property including instants where in the next state *klactive* is true. To make sure, that this actually happens in the next state we have to enclose the two assignments in an *atomic* statement. Figure 4(b) shows the corresponding FSM generated by LTL 2 BA. It has very few transitions but close examination confirms that

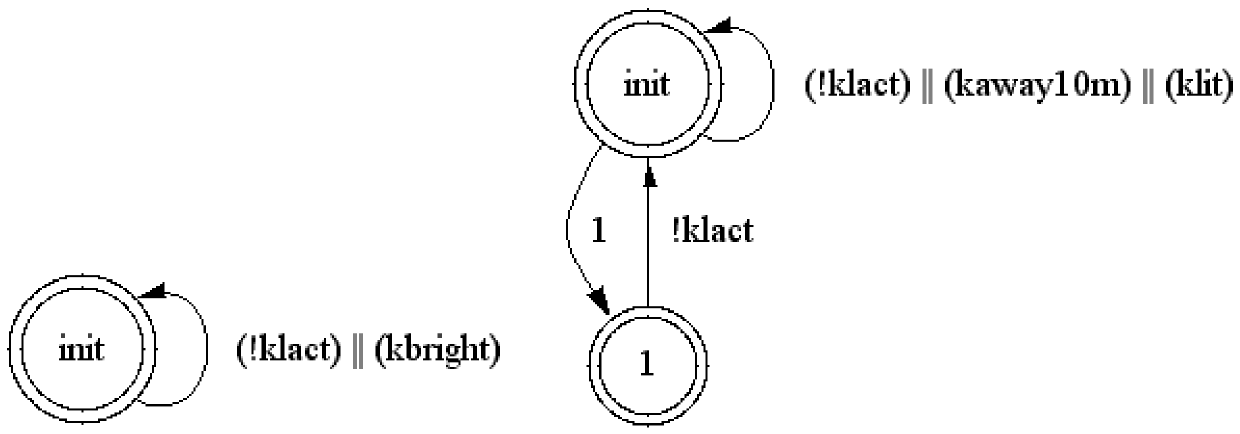


Figure 4. FSMs of properties 1 (a) and 4 (b)

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

Figure 5. Promela code for first property

the automaton follows our intention. When building the properties independent from the model, such details would easily get omitted. This causes later the property to fail and it may be time consuming to locate the error.

As the model now contains most of the lighting functionality, we now add security properties. Property 5 exploits the motion sensors. When no person is identified in the room with a valid RFID and there is motion detected an alarm is activated. The alarm signal might ring a bell and send a text message to the residents cell phone and email. As the alarm signal can only occur after the condition is detected, it is most appropriate to use the until operator U for this property. Table 2 shows all the LTL properties. The implementation in the formal model does add some lines to the *KPsense* process, but it does not incur any fundamental changes.

The next user-story describes the temperature control mechanism. It says that the AC/Heating system has three modes of operation, each having a temperature level assigned. The first level is the comfort level, it defines the temperature when the room is used. Then there is a power save temperature that is active whenever nobody has been using the room for more than two hours. And finally there is a vacation mode that is active when a room has not been used for more than two consecutive days. From this user story we obtain the three properties 6, 7, and 8. In the PROMELA model there is already a counter that counts the time since the room has been used last so we have again few changes in the model. After modifying the model, small corrections have to be performed when checking all previous properties.

Until here, our model only comprises one room, the kitchen. As we have sensors for every room, most of the control is based on the room-level. All properties so far can be replicated for any other room. The same is valid for the model. This makes our model modular in approach. However, in some rooms there are additional functions that should be added. For example we want to control the intensity of the light in the living room and in the bedrooms. Intensity in the bedroom is set to zero, when going to bed and in the living room it may be set to any desired value for example 25% for watching movies. To add a switch that controls the light intensity results only in trivial LTL properties, and therefore is not worth checking for. But additionally it says in the specification that when it gets dark outside and bright again (the next day), intensity should always be at 100. This requirement is expressed in LTL property 9 (see Table 2). To reflect this in the model, we first of all have to create the living room. This is done by copying all kitchen processes and rename them and their variables. In addition to *KPsense* we get *LPsense* and so on. Then we replicate all LTL properties and the appropriate definitions and check all these new properties as well as the ones for the kitchen. Once this is done a new process *LIntensity* is added that switches between different levels of intensity if it is sufficiently dark in the living room. To verify property 9 we change the intensity back to 100 as soon it gets bright enough outside, i.e. as soon as *lactive* turns to zero.

As we have several rooms now, it makes sense to introduce functions that spawn several rooms. Safety properties such as the detection of smoke or hazardous temperatures are properties that concern the whole house. In order to ensure the safety in the house, we add two properties that depend on the smoke detectors and on the temperature sensors. Property 10 issues a safety hazard when a certain amount of smoke is detected or if the temperature in any room is high or low. A safety hazard notifies the resident about the incident but does not take any immediate action yet. Once the intensity of the smoke reaches another critical level, the sprinkler system is launched and the fire fighters are notified. In the model, these changes induce the creation of two small processes that monitor the temperature and smoke sensors in all rooms.

5 Conclusion

In this paper we demonstrate the usage of the XFM methodology developed by us in [3] for the formal modeling of a large control application for a smart building. It 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. The incremental approach used in the paper helps that the constructed abstract model satisfies all properties, and it ensures that the model contains only little unwanted behavior that might cause a later conformance check against an implementation to fail. 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.

The modeling example of the Smart Home demonstrates the power of the approach, and even if there exist more sophisticated smart spaces, we succeeded in building a reasonably complex smart environment with little effort. Yet most importantly is not the large amount of functionality and the different levels of interaction, but the confidence that this model complies with the specification without containing much superfluous behavior. It shows that this methodology lowers the hurdle for engineers to move to formal verification or improving their results and confidence while still being open for particular solutions on the single steps.

References

[1] Kent Beck. *Extreme Programming explained: Embrace change*. Addison Wesley, 2000.

[2] Bob Bentley. Validating the intel pentium 4 microprocessor. In *Design Automation Conference*, pages 244–248, 2001.

[3] David Berner, Syed Suhaib, Sandeep Shukla, and Harry Foster. XFM: Extreme Formal Method for Capturing Formal Specification into Abstract Models. Technical Report no. 2003-08, Virginia Tech, FERMAT Lab, Blacksburg, VA, 2003.

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

[5] M. Coen, B. Phillips, N. Warshawsky, L. Weisman, S. Peters, and P. Finin. Meeting the computational needs of intelligent environments: The metaglue system. In *1st Int. Workshop on Managing Interactions in Smart Environments (MANSE'99)*, pages 201–212, Dublin, Ireland, December 1999. Springer-Verlag.

[6] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Property specification patterns for finite-state verification. In *Proc. 2nd Workshop on Formal Methods in Software Practice (FMSP-98)*, pages 7–15, New York, 1998. ACM Press.

[7] Michigan eXtreme Programming Enthusiasts (MXPE). Extreme programming: A gentle introduction. http://extremeprogramming.org.

[8] Babel group at TU Madrid. The slam website. http://lml.ls.fi.upm.es/slam.

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

[10] A. Herranz and J.J. Moreno-Navarro. Formal extreme (and extremely formal) programming. In Michele Marchesi and Giancarlo 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.

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

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

[13] MIT. Project OXYGEN - pervasive human centered computing. http://oxygen.lcs.mit.edu, 2003.

[14] Dennis Oddoux. LTL 2 BA: Buchi automata from LTL. http://liafa.jussieu.fr/˜oddoux/ltl2ba.

[15] University of Colorado. Adaptive Home. http://cs.colorado.edu/˜mozer/nnh.

[16] Stephen Peters and Howie Shrobe. Using semantic networks for knowledge representation in an intelligent environment. In *PerCom ’03: 1st Annual IEEE International Conference on Pervasive Computing and Communications*, Ft. Worth, TX, USA, March 2003. IEEE.

[17] K. Shimizu and D. Dill. Deriving a simulation input generator and a coverage metric from a formal speci cation, 2002.

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

[19] Rachel L. Smith, George S. Avrunin, Lori A. Clarke, and Leon J. Osterweil. Propel: an approach supporting property elucidation. In *Proceedings of the 24th international conference on Software engineering*, pages 11–21. ACM Press, 2002.

[20] B. Steffen, T. Margaria, and M. von der Beeck. Automatic synthesis of linear process models from temporal constraints: An incremental approach, 1997.

[21] Cisco Systems. Internet home. http://cisco.com/warp/public/3/uk/ihome, 2000.

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

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