Workshop Proceedings of the 8th International Conference on Intelligent Environments
J.A. Botía et al. (Eds.)
IOS Press, 2012

doi:10.3233/978-1-61499-080-2-401

Consistency in Context-Aware Behavior: a Model Checking Approach

Davy PREUVENEERS [a,1], Yolande BERBERS [a]
[a] *IBBT-DistriNet, KU Leuven, 3001 Leuven, Belgium*

Abstract. Context-aware systems have been in the research epicenter for more than a decade. By sensing, reasoning and acting on various kinds of relevant situational information, context-aware systems can make well-informed decisions and manifest autonomous behavior through pre-defined and user customizable rules that define what to do in particular circumstances. However, there is still a significant gap between deploying such context-aware systems in lab environments under ideal circumstances and deploying them in the real world with non-specialist end-users. These end-users may experience a lack of understanding or frustration when a system does not behave as expected, especially in the presence of unforeseen human interventions, exceptional circumstances and unexpected events. This paper investigates challenges and opportunities of using model checking as an approach for improving the reliability and consistency of smart environment applications whose behavior is driven by context-aware adaptation rules, and reports on initial results with the SPIN model checker.

Keywords. context, event-condition-action rules, model checking

Introduction

In the ubiquitous computing paradigm, devices and applications are able to interact with one another and often have an awareness of the situation of their users to create a smart environment that is proactive and supportive. Context-aware systems [1] are able to adapt their behavior to the current context without explicit user intervention by taking environmental context into account. As Dey [2] stated, context is any information that can be used to characterize the situation of an entity. An entity is a person, place or object that is considered relevant to the interaction between a user and an application, including the user and the application themselves.

However, the vision of ubiquitous computing as described in Mark Weiser's seminal article has still not become reality. The complexity of integrating ubiquitous computing in our daily lives and making our environment intelligent, proactive and, moreover intuitive to use, has proven to be very high. As these ubicomp features interact with each other in sophisticated ways [3], so does the complexity that will adversely impact the reliability [4] of the smart environment. Recent work [5] confirms that little is reported in the literature on methodologies to increase the reliability of software for the devel-

[1]Corresponding Author: Davy Preuveneers, Dept. Computer Science, Celestijnenlaan 200A, B-3001 Heverlee, Belgium; E-mail: davy.preuveneers@cs.kuleuven.be.

opment of intelligent environments. Assistive technologies are often too naively or too optimistically developed assuming that users always understand what these systems are doing or that systems know what users want.

In this paper, we will address the challenge of robustness and mitigating the increased potential for failure in a smart environment that evolves over time (i.e. in a dynamic context). We will particularly focus on smart context-aware systems whose dynamic behavior is driven by context-aware decision and adaptation rules (following the typical *event-condition-action* paradigm). The complexity of ensuring robustness is caused by the fact that ensuring consistency among these rules is not straightforward and that many features may interact with one another [6]. Robust intelligent systems [7] ensure acceptable performance, not only under ordinary conditions but also under unusual conditions that stress the developers' assumptions of the system. However, it is difficult for a developer *at design time* to discover and eliminate all errors when the application will be active outside the intended limited context. As a result, fragile context-aware applications are subject to subtle or more severe errors that only make their presence known in unusual circumstances. This problem is aggravated when users can customize *at runtime* the contextual triggers (event-condition-action rules) that modify the behavior of a ubicomp system, causing inconsistencies among these rules (e.g. turning on the heating and the air conditioning at the same time).

In section 1, we present a use case scenario in the smart home environment from which we distill some context-aware adaptation rules. These rules will be used in section 2 to discuss more in detail the kind of inconsistencies that may emerge from these rules. In section 3 we will elaborate on our model checking approach and present some qualitative results with the SPIN model checker in section 4. We conclude and propose further work in section 5.

1. Motivating Scenario: a Smart Home Use Case

Jack and Jill have installed a smart home automation system. It connects all the devices and appliances in their home so they can communicate with each other and also with their owners. Their ZigBee-based smart home solution provides applications related to lighting, home security, home theater and entertainment, and thermostat regulation. It is equipped with sensors to measure the temperature at different places in the house, motion sensors to detect someone's presence, ambient light sensors, as well as other sensors for outdoor use. The home automation system also comes with switches and dimmers to control the curtains and window blinds, the lighting, the heating and the air conditioning in each room. A control panel provides easy remote access to each device.

Jack and Jill set up the system to dynamically adapt to the current context rather than merely relying on pre-programmed timers. Sensor values will drive the behavior of the home automation system. Initially, they would like to configure the basic lightning, heating and air conditioning. They want the home automation system to behave according to the following preferences:

- **Heating:** The heating should only be turned on if it is less than 19 °C inside the house. Sensors in the house will monitor the environment and the thermostat will aim for a temperature of 21 °C.

$$\textbf{if } (temperature_{in} < 19°C) \textbf{ then } heating = on \tag{1}$$

$$\textbf{if } (temperature_{in} > 21°C) \textbf{ then } heating = off \tag{2}$$

- **Lighting:** When it is getting dark inside, the lights should be turned on only in those rooms where people are present. The lights should be turned off automatically if no presence is detected for more than 5 minutes.

$$\textbf{if } (brightness_{in} = low) \textbf{ and } (motion = true) \textbf{ then } lights = on \tag{3}$$

$$\textbf{if } (motion = false)_{\Delta t=5min} \textbf{ then } lights = off \tag{4}$$

- **Air conditioning:** If it is getting more than 27 °C in the house, then turn on the air conditioning until the temperature drops to 24 °C. Motion sensors will scan the room for occupancy and when no movement is detected it will send a signal to the air conditioning unit to switch it off.

$$\textbf{if } (temperature_{in} > 27°C) \textbf{ then } airconditioning = on \tag{5}$$

$$\textbf{if } (temperature_{in} < 24°C) \textbf{ then } airconditioning = off \tag{6}$$

$$\textbf{if } (motion = false) \textbf{ then } airconditioning = off \tag{7}$$

Later on, Jack and Jill also add outdoor sensors to their smart home automation system and they want to update the previous behavior and objectives with the following automation preferences:

- **Lightning:** If it is daylight outside, open the curtains or window blinds and also turn off the lights.

$$\textbf{if } (brightness_{out} = high) \textbf{ then } curtains = open \textbf{ and } lights = off \tag{8}$$

- **Air condition:** If it is more than 30 °C outside, close the curtains and window blinds to reduce the temperature increase inside the house.

$$\textbf{if } (temperature_{out} > 30°C) \textbf{ then } curtains = close \tag{9}$$

The above rules are a typical illustration of how a layman end-user (i.e. not a domain expert) would specify the preferred behavior of the smart system in terms of the storyline described above. Although these rules seem obvious at first sight, the careful reader will have already noticed that dependencies between these high-level automation objectives can be a recipe for trouble (e.g. possible contradictions in rules (5) and (7)) with the intended smart behavior escalating quickly into chaos.

2. Classification of Inconsistencies

Errors and inconsistencies cannot be avoided for systems with humans-in-the-loop, because people are not always consistent, dependable, responsible, trustworthy, etc. They are not designed to perform, and they definitely not always perform as designed (as envisioned by the developer of the system). The following sources for inconsistencies can be explained in terms of three causes:

1. **System factors:** devices behaving erratically
2. **Human factors:** people behaving erratically
3. **Unknown contexts:** unstated operational/environmental conditions

In this section, we will provide an overview of inconsistencies that can emerge due to dependencies or conflicts among the above rules.

2.1. Non-Deterministic System Behavior

Deterministic systems are characterized by the fact that all accepted behavior is adequately expressed for all possible circumstances. In non-deterministic systems, especially those with humans-in-the-loop, the current state of a system is not always predictable. This causes uncertainty about the behavior of a system, and a lack of understanding of why a system does not behave as expected. Without rule (2), the state of the heating would not be complete because the rules do not specify the contextual constraints for each state of the system. However, even with rule (2) the state of the system remains undetermined for a temperature of 20 °C. This leads to the following requirement:

> *The contextual constraints for each system state should be specified non-ambiguously so that the system's behavior can be explained with respect to the current context.*

2.2. Erratic System Behavior

In order to mitigate the increased potential for failure in a smart environment, systems should be prepared for failure rather than trying to prevent every failure (i.e. design for failure). Smart systems that can tolerate partial failures, deal with contextual inconsistencies (e.g. an erroneous sensor reading) or improper human handling, and minimize the impact of a failure through graceful degradation will improve the robustness and the quality of experience for the end-users. In rules (1) and (2) we assume that if the temperature is below 19 °C, the heating will be turned on. If there is a failure in the actuator, the temperature will not increase. Such failure can be detected with a rule that will monitor the execution or progress of other rules.

$$\textbf{if } (temperature_{in} < 19°C) \textbf{ then } (temperature_{in,t=0..10min} > 19°C) \tag{10}$$

As we turn on the heating, the effect of this actuation is not immediate. With this rule, we express that the actuation will lead to a new sensed contextual reality somewhere in the next 10 minutes. If this does not happen, the system can detect its own failure. For the sake of legibility, the above simplified notation means that somewhere in the next 10 minutes the temperature will be above $19°C$ (i.e. existential quantification), and not that this statement is true for the whole period (i.e. universal quantification). Similar to pre-conditions and post-conditions in object-oriented programming, assertions can help monitor and verify the expected behavior of the system. This leads to the following requirement:

> *To detect erratic system behavior early, every rule that modifies the system or the environment should be accompanied with an assertion that verifies the expected state of the system or the environment after the actuation.*

This assertion can be implemented as a rule. In the above example, a temperature sensor is used to verify the state of the environment during the next 10 minutes. However, note that the temperature sensor used to implement the assertion can also produce erroneous

values, i.e. the verification can fail. This means that we may be able to detect erratic system behavior, but we may not always accurately identify the cause of the error.

2.3. Conflicting Context Conditions and Actions

When multiple rules affect the same system component, inconsistencies in the contextual triggers may cause a non-deterministic component state. These inconsistencies are often due to unbounded and/or overlapping contextual circumstances. For example, with rules (8) and (9) the state of the curtains is uncertain during a hot summer day because both rules would be triggered. Rules (5) also contradicts with (7) when $motion = false$ and $temperature_{in} > 27°C$. This leads to the following requirement:

> *To avoid conflicting operations or effects, each context should give rise to only one state for each component in the system.*

We can relax this requirement if we are able to prioritize the execution of the rules, i.e. know the order in which the rules should fire. This example investigated contextual conflicts for an individual system component. Next, we will show how inconsistencies can emerge due to separate system components affecting the environment at the same time in opposite ways.

2.4. Hidden Dependencies and Invalid Global States

Consistent behavior of the individual components does not guarantee reliable functioning of the home automation system as a whole. For example, the temperature in a room can be affected by the heating, the air conditioning and the state of the curtains. Obviously, turning on the heating and the air conditioning at the same time is not a good idea. However, these implicit dependencies are nowhere modeled in the above rules. These hidden dependencies should be made as explicit as possible. At the least, invalid global states of the system or the environment need to be explicitly modeled − independent of the context-driven adaptation rules − to guarantee that certain inconsistent behavior does not occur or is detected early.

> *Invalid global states should be explicitly modeled.*

These kind of assertions are similar to invariants in object-oriented programming that guarantee that certain predicates must always be true before and after a sequence of operations.

2.5. Non-Terminating Loops

A difficult problem in rule based systems is the presence of loops. The consequence statements of one rule can cause another rule to fire, possibly causing non-terminating loops. Intelligent environments are typically characterized by non-terminating processes,

but some of these execution loops can cause unwanted behavior. For example, we want the temperature to gradually increase, so that there is no initial overshoot causing the air conditioning to become active, and vice versa. The problem with detecting these inconsistencies is the temporal aspect of the temperature variations in a loop. If such a cycle would take hours to complete (e.g. heating turned on in the morning and air conditioning turned on in the afternoon), there is not a problem. However, if this cycle would be much shorter, there is something wrong. Therefore, we should make sure that:

Context-aware adaptation rules should have safeguards in place to avoid non-terminating loops of unwanted behavior.

As loops cannot always be detected through static analysis of the rules, safeguards can for example express how often a rule can be executed in a given time period.

In the above subsections, we identified several requirements for rule-based dynamic and adaptive smart environment applications that would help to make these systems more reliable. In the following section, we will investigate and discuss which kind of inconsistencies can be detected at design time of the smart system through simulation and verification with the SPIN model checker.

3. A Model Checking Approach with SPIN

Model checking in software engineering [8] has been around for more than a decade to formally verify models of distributed software systems. The motivation for using model checking to verify context-driven applications is straightforward. There are often too many possible configurations and contextual situations that an application can be in, that it is impossible to verify all these combinations with a set of test cases. In this section, we will discuss the feasibility of using the SPIN model checker, and highlight the major benefits and weaknesses.

3.1. Promela

The SPIN (Simple Promela Interpreter) model checker relies on a model of the system. This model is specified in the Promela language (PROcess MEta-LAnguage). The objective is to model rules as concurrent processes and to model the behavior of the system as a finite state system.

The following example is a simple Promela model that represents a subset of the functionality of the home automation system. We use the Linear Temporal Logic (LTL) formula `[]( !airco || !heating )` to simulate and verify that the airco and the heating are never turned on at the same time.

```
mtype = { airco_on, airco_off, heating_on, heating_off }
mtype = { activated }

chan system_actions = [0] of { mtype };
chan sensor_reaction = [0] of { mtype, mtype };
```

```
bool airco = false;
bool heating = false;
int temperature = 18;

ltl p1 { []( !airco || !heating ) }

active [1] proctype Airco()
{
end:    do
        :: (temperature < 24) -> system_actions!airco_off; airco=false;
        :: (temperature > 27) -> system_actions!airco_on; airco=true;
        od
}

active [1] proctype Heating()
{
end:    do
        :: (temperature < 19) -> system_actions!heating_on; heating=true;
        :: (temperature > 21) -> system_actions!heating_off; heating=false;
        od
}

active [1] proctype Environment()
{
end:    do
        ::system_actions?heating_on -> sensor_reaction!activated,heating_on;
        ::system_actions?heating_off -> sensor_reaction!activated,heating_off;
        ::system_actions?airco_on -> sensor_reaction!activated,airco_on;
        ::system_actions?airco_off -> sensor_reaction!activated,airco_off;
        od
}

active [1] proctype Sensor()
{
end:    do
        ::sensor_reaction?activated,heating_on ->
            if
            :: (temperature < 5) -> temperature = temperature + 7;
            :: (temperature < 10) -> temperature = temperature + 5;
            :: (temperature < 15) -> temperature = temperature + 3;
            :: else temperature = temperature + 1;
            fi
        ::sensor_reaction?activated,airco_on ->
            if
            :: (temperature > 50) -> temperature = temperature - 12;
            :: (temperature > 35) -> temperature = temperature - 10;
            :: (temperature > 30) -> temperature = temperature - 8;
            :: else temperature = temperature - 1;
            fi
        ::sensor_reaction?_,_ -> temperature = temperature + 2;
        ::sensor_reaction?_,_ -> temperature = temperature - 2;
        od
}
```

Note that it is not our objective to create Promela models by hand, but rather to au-

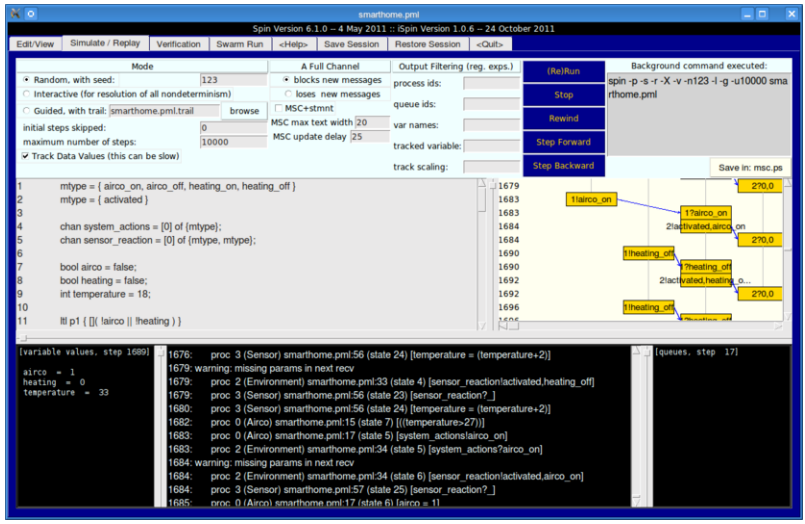

Figure 1. Simulation with the SPIN model checker

tomatically generate them from a simple rule set description that specifies the desired autonomous behavior and adaptation in the smart environment. The reason for this approach is twofold:

- The rule set is easier to specify and maintain, especially for users unfamiliar with the peculiarities and syntax of Promela.
- The same rule set can be used for transformation to other notations as input for other consistency checkers.

For example, we are exploring similar transformation techniques to OWL2RL [9] to model inconsistency rules and analyze them based on its underlying propositional logic. How the transformation from the rule set description to Promela is carried out is beyond the scope of this paper.

3.2. Simulation and Verification

In Figure 1, we show a simulation of this model with SPIN. Verifying this model results in the following success message:

```
spin -a  smarthome.pml
ltl p1: [] ((! (airco)) || (! (heating)))
gcc -DMEMLIM=1024 -O2 -DXUSAFE -DNOCLAIM -w -o pan pan.c
./pan -m10000  -a
Pid: 25147
error: max search depth too small

(Spin Version 6.1.0 -- 4 May 2011)
+ Partial Order Reduction

...
```

```
pan: elapsed time 0.62 seconds
No errors found -- did you verify all claims?
```

The aforementioned Promela does not represent the full behavior as reflected by the rules in the previous section. Some of the missing features (such as the lights and the motion) can easily be included in a similar fashion. However, it seemed not straightforward to model all features of the smart system behavior.

As a detailed discussion of the full Promela model is outside the scope of this paper, we will highlight in the following subsections which concerns were hard to model and which aspects we were not able to represent.

4. Qualitative Evaluation

The above Promela model is only a brief example of what can be modeled with Promela and analyzed with SPIN. After a preliminary assessment, it is clear that it definitely does not address all the requirements that we outlined before. We can make the following observations:

1. The first requirement argues that it should be possible to explain the state of a system based on the current context, as well as the rules that gave rise to the current state. This concern is often referred to as the intelligibility [10] of a system, a user centered property leading to a better understanding and stronger feelings of trust.
2. The second requirement identifies inconsistencies with respect to expected behavior. Obviously, these errors cannot be addressed purely with design time verification tools. Runtime support is necessary to ascertain whether the intended outcome of a rule has been achieved or not.
3. The third requirement states that all components should work together to achieve the same global new context. Leelaprute et al. [3] have how SPIN can be used to detect such feature interactions. We could have specified in our example:

   ```
   ltl p0 { !<> []( airco || heating ) }
   ```

4. The forth requirement stating which global states should never happen, can be implemented easily with Linear Temporal Logic formulas using negation and the 'always' operator represented as [] P, stating that P is always true in all system executions. In our example, we specified:

   ```
   ltl p1 { []( !airco || !heating ) }
   ```

   Note that this LTL formula is more strict than the previous one. With the above `p0` formula, we are sure there can never be a situation where both the airco and heating are on indefinitely, whereas with the `p1` formula, we are sure that the airco and heating are never on at the same time.
5. Non-terminating loops cannot (to the best of our knowledge) be expressed easily in Promela. Many works that use SPIN to detect feature interaction assume concurrency while representing their correctness claims as LTL formulas. To address this requirement, we need expressiveness (and most likely runtime support) to specify undesired interleaved processes (e.g. the airco and heating taking turns every 5 minutes).

In the following subsections, we will highlight the major benefits and weaknesses of SPIN and Promela with respect to the requirements.

4.1. Benefits

Counter example: One of the main advantages of SPIN is the ability to produce a counter example with a trace of events that lead to a situation where the claims to be verified are not confirmed. However, we found cases where the produced counter example was not realistic (e.g. the sun shining at night) and we had to finetune the model to exclude these false positives.

Non-deterministic system behavior: Following a particular modeling approach (using channels and the Promela `timeout` concept, it is possible to use a simulation approach where at some point the simulation blocks due to a particular context for which no further autonomous behavior is specified (e.g. a temperature of 20 °C with rules (1) and (2)). However, it is not guaranteed that through simulation you will find all these incomplete context specifications.

Conflicting conditions/actions: It is possible to identify different contexts that give rise to conflicting actions (cfr. rules (8) and (9) for an illustration). The technique to follow is based on the `atomic` construct and the implementation of a concept similar to mutexes. If a rule modifies the state of a component, it increases and decreases the mutex for this component in an `atomic` construct. If each context triggers one rule only, the mutex for each component will never be higher than 1.

4.2. Weaknesses

Explicit notion of time: While the Linear Temporal Logic allows for expressing qualitative properties with respect to time, it is not possible to quantitatively express time. We may guarantee that eventually it will get warmer if the heating is turned on. However, it would be much better if we could also specify that this should happen in the next 10 minutes. See rule (10) for an example of this concern. The inability to express time creates a similar concern regarding the non-terminating loops. It is impossible to specify the minimal length of acceptable adaptation cycle (e.g. heating turned on in the morning and air conditioning in the afternoon).

Modeling external influences: It is not straightforward to model a smart environment accurately. The measured temperature in a smart environment is not only the result of state of the air conditioning, the heating, the curtains, etc. but also influenced by external factors such as the temperature outside. In the previous Promela model, we pursued a simplistic approach by having the temperature randomly go up or down with a degree (exploiting the non-deterministic selection in the Promela `do`-loop statement). However, this is not a realistic representation of the real world, which makes the outcome of the simulation subject to interpretation.

Hidden dependencies: It is very easy to overlook dependencies among context variables and the impact of one activity on multiple contextual variables. For example, closing

the curtains will not only keep the heat outside on a sunny day, but will also affect the brightness inside the house. If the behavior is not accurately described, it is unlikely we will be able to capture all inconsistencies in the context-aware rules through verification.

State space explosion: This is a common problem with model checkers. In the aforementioned example, the possible context combinations were rather small due to the limited amount of contextual state variables. As a result, we did not experience any scalability problems or memory constraints that would force us to simplify the abstractions in our model. However, for more complex situations with multiple context variables and context-aware rules and assertions, it is fairly likely that the exponential state space explosion will become a critical concern to verify the consistency of the context-aware rules.

No systematic extraction techniques: The current approach assumes the existence of a Promela that resembles the behavior of the context-aware systems. Classical approaches start off with an initial design of the system which is then manually handcrafted into an abstract Promela verification model. If the verification succeeds, the system is implemented. Unfortunately, this process does not guarantee that the implementation is consistent with the Promela model. Modern approaches start off with the implementation from which the verification model is automatically abstracted. Unfortunately, to the best of our knowledge, such systematic techniques have not been proposed for context-aware systems.

5. Conclusion

In this paper, we investigated an approach to expose context dependent faults in a smart environment system that is driven by rules following the event-condition-action paradigm. We started off with some general observations of how inconsistencies in context-aware rules can lead to erratic system behavior, either through faults in these rules or due to interferences between these rules. We listed some requirements that would help to mitigate the risk of failures, and we experimented with the SPIN model checker to investigate its feasibility as a model checker to debug these inconsistencies. We illustrated the process with a simple smart home automation scenario as a typical example of a smart environment. Experiments with students have shown that formal verification of a context-aware system by means of model checking is not easy, even after an initial training with SPIN on some less complex examples. Many consider it more complex than implementing the system itself, in some cases even too complex. This experience is well captured in one of the quotes of Brian Kernighan, well-known for his co-authorship of the first book on the C programming language:

> *"Everyone knows that debugging is twice as hard as writing a program in the first place. So if you are as clever as you can be when you write it, how will you ever debug it?"* - Brian Kernighan

In the preliminary experiments we carried out with SPIN, we found the lack of an explicit representation of time to be counter-intuitive. Being able to verify that a certain property will hold some time in the future does not allow us to really express what people expect from a system.

Another aspect we are further exploring is the automatic translation of a given rule set and a list of global assertions into a compatible Promela model, and how verification failures can be traced back to the original rule set specification. For this purpose, we need to explicitly model how state changes in a single component of the smart system can interfere with the state of another component (e.g. turning on both the heating and the air conditioning at the same). In a similar way, given that the effects of a state change (e.g. the heater being turned on) are not immediate, we are exploring models to realistically represent how an actuator change influences what a sensor will measure over time. The purpose of such models would be to capture the expectations of the system. Any significant deviations from such common or expected behavior could be identified as abnormal behavior more easily, which would hopefully lead to an increased reliability and robustness against unforeseen human interventions, exceptional circumstances and unexpected events in smart environments.

In some experiments, the simulator of SPIN provided us with useful feedback with contextual situations where erratic behavior would occur. However, we also identified that some of the traces in SPIN were not realistic. In one of the traces, SPIN produced a counter example that claimed an error would occur if the sun is shining at night. Obviously, this error occurred due to hidden dependencies in the context variables. We are currently investigating approaches to capture these facts into the Promela model.

Acknowledgments

This research is partially funded by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, and by the Research Fund KU Leuven.

References

[1] M. Baldauf, S. Dustdar, and F. Rosenberg. A survey on context-aware systems. *Int. J. Ad Hoc Ubiquitous Comput.* **2**:4 (2007), 263–277.

[2] G.D. Abowd, A.K. Dey, P.J. Brown, N. Davies, M. Smith, and P. Steggles. Towards a better understanding of context and context-awareness. In *HUC '99: Proceedings of the 1st international symposium on Handheld and Ubiquitous Computing*, 304–307, London, UK, Springer-Verlag, 1999.

[3] P. Leelaprute, T. Matsuo, T. Tsuchiya, and T. Kikuno. Detecting Feature Interactions in Home Appliance Networks. In *Proceedings of the 2008 Ninth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing*, 895–903, IEEE Computer Society,2008.

[4] R. Cook. How Complex Systems Fail. Technical report, Cognitive technologies Laboratory, University of Chicago, 2000.

[5] J.C. Augusto and P.J. McCullagh. Safety Considerations in the Development of Intelligent Environments. In P. Novais, D. Preuveneers, and J.M. Corchado, editors, *ISAmI*, *Advances in Intelligent and Soft Computing* **92** (2011), Springer, 197–204.

[6] H. Igaki and M. Nakamura. Modeling and Detecting Feature Interactions among Integrated Services of Home Network Systems. *IEICE Transactions* **93-D**:4 (2010), 822–833.

[7] A. Schuster. *Robust Intelligent Systems*. Springer Publishing Company, Incorporated, 1 edition, 2008.

[8] G.J. Holzmann. The Model Checker SPIN. *IEEE Trans. Softw. Eng.* **23**:5 (1997), 279–295.

[9] D. Reynolds. OWL 2 RL in RIF. http://www.w3.org/2005/rules/wiki/OWLRL, 2009.

[10] B.Y. Lim, A.K. Dey, and D. Avrahami. Why and why not explanations improve the intelligibility of context-aware intelligent systems. In *Proceedings of the 27th international conference on Human factors in computing systems*, CHI '09, 2119–2128, New York, NY, USA, ACM, 2009.