A Method for Model Checking Feature Interactions

Thomas Pedersen, Thibaut Le Guilly, Anders P. Ravn, and Arne Skou
Department of Computer Science, Aalborg University, Selma Lagerlöfs Vej 300, 9220 Aalborg East, Denmark
{*tp,thibaut,apr,ask*}*@cs.aau.dk*

Keywords: Feature Interaction, Control Systems, Model-driven Development, Home Automation, Model Checking, Timed Automata.

Abstract: This paper presents a method to check for feature interactions in a system assembled from independently developed concurrent processes as found in many reactive systems. The method combines and refines existing definitions and adds a set of activities. The activities describe how to populate the definitions with models to ensure that all interactions are captured. The method is illustrated on a home automation example with model checking as analysis tool. In particular, the modelling formalism is timed automata and the analysis uses UPPAAL to find interactions.

1 INTRODUCTION

Feature interactions appear when independently well functioning processes operate concurrently. An example from home automation is an automatic window that sets off an alarm, because of the movement it causes while opening and closing. To unify processes like these in a harmonious interplay, engineers can conduct a behavioural analysis of the system and if necessary make appropriate modifications. In the example, the window could be closed by the alarm system. However, this analysis is non-trivial, as building control systems not only have many features [Metzger and Webel, 2003], but also have complex interactions with their environments [Kolberg et al., 2003]. Therefore we investigate tool support for analyzing feature interactions in systems.

Existing approaches use different models and tools, for example: dependency graphs [Metzger and Webel, 2003], Symbolic Model Verifier (SMV) [Leelaprute et al., 2005], SPIN [Matsuo et al., 2006], SAT solver [Classen et al., 2008] and Java Modelling Language (JML) [du Bousquet et al., 2009]. Others handle feature interactions at run-time [Wilson et al., 2008]. All of these approaches provide an architecture for structuring the models of the system. Some allow detailing the features [Inada et al., 2012,Leelaprute et al., 2005], or consider the relationship between devices and environment [Wilson et al., 2008,Nakamura et al., 2013]. Existing work also provides an overall methodology for analysis of existing implementations [du Bousquet et al., 2009], or for eliciting safety properties [Yan et al., 2007]. See Section 6 for more details on related work.

In this paper, we integrate previous definitions of home automation systems into one suitable for verification of systems, when their features are known in advance. In addition, we propose a set of activities to identify all relevant elements in the definitions to have a complete verifiable system. The definitions thus become boxes populated with models using the activities. The method is tool independent, but we show a proof-of-concept using timed automata and the UPPAAL model checker [Behrmann et al., 2004]. This allows features to be described at a detailed level, for example with branches, loops, and time. Feature interactions can be detected using verification – a full state-space exploration of the system. In particular, our proof-of-concept enhances previous work by using graphical modelling and including timing aspects. More concretely we contribute with:

- A definition of control systems and interactions, that combines previous definitions from software product line engineering [Classen et al., 2008] with those from an online feature manager [Kolberg et al., 2003]. In addition we add disturbances to complete the system models (Section 2).
- A scenario (Section 3) showing a set of activities to identify necessary models (Section 4).
- A proof-of-concept, where the informal descriptions of the home automation system are transcribed to timed automata and analysed to find feature interactions using the UPPAAL tool. This leads to clarification and disambiguation of the

concepts introduced during modelling (Section 5).

Section 7 concludes with further work and discussion of the benefits and limitations of the method.

2 CONTROL SYSTEMS AND INTERACTIONS

Our definition of a home automation system, roughly correspond to the layers used in the online feature manager [Kolberg et al., 2003], but here we generalise to conventional control concepts. The relationships between the components is shown in Figure 1. The home automation system is a control system, that consists of a set of features, F. Each feature, $f \in F$ is a concrete and well defined unit within the control system, that controls an aspect of the plant (e.g. home). An example is a heating feature, f_{Heat}, with the purpose of keeping the home comfortable at a comfortable temperature, see Figure 2.

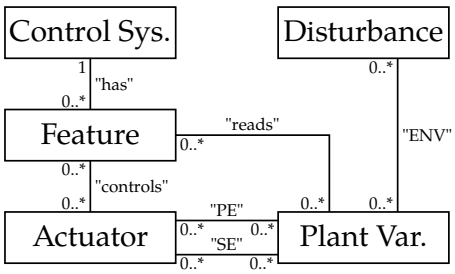


Figure 1: Relationship between components in a control system. The effects of actuators and disturbances are classified in primary (PE), side (SE), and environmental (ENV).

2.1 The Plant

A plant has a state, a set of devices and a set of disturbances. The state is a set of measurable plant variables, V. An example of a variable is a room temperature, v_{temp}. Sensors measure variables and allow a feature, f, to read their value. For the scope of this paper we shall exclude delays and noise due to sensors, and say that sensors are implicitly defined in a variable, v. This means that f will read v directly. The set of devices D is the combined set of sensors and actuators. As we just excluded sensors it will only consist of actuators. An actuator is a device, $d \in D$, that is controllable by features in F and influences values of variables in V. An example of an actuator is a heater, $d_{heater} \in D$, that increases v_{temp}. The disturbances, P, also change values of variables in V, but in contrast to actuators these are uncontrollable by features in F. An example of a disturbance is the thermodynamics of the walls, $p_{wall} \in P$, that decreases v_{temp} (under the assumption that it is colder outside than inside).

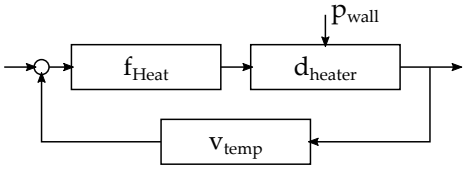


Figure 2: An example of a heating feature, f_{Heat}, controlling a heater, D_{heater} based on the value of a temperature variable, v_{temp}.

2.2 Features and Interactions

To find unwanted feature interactions in the system, we need to introduce the notion of feature consistency from requirement engineering. The ith feature, $f_i \in F$, is a tuple, (R_i, W_i, S_i), where R_i is the requirements, W_i is the assumptions about the environment, and S_i is its specification [Classen et al., 2008]. An example for f_{Heat}:

R_{Heat}**:** The temperature, v_{temp}, must be kept between 19°C and 22°C.

W_{Heat}**:** d_{heater} warms up the room, but p_{wall} cools it down.

S_{Heat}**:** When $v_{temp} \leq 19$°C then turn on d_{heater}.

The feature f_i is consistent if we take R_i, S_i, and W_i to be predicates, and $S_i, W_i \vdash R_i$ or in plain English: if the specification, under the given assumptions about the environment, entails the requirements. To find feature interactions we combine features and check whether the combinations are consistent as well. Let the tuple (R, W, S) be the combination of all n features in a system, then the consistency of the full system is formally given as: $S = \bigwedge_{i=1}^{n} S_i$, $W = \bigwedge_{i=1}^{n} W_i$, $R = \bigwedge_{i=1}^{n} R_i$, and $S, W \vdash R$.

2.3 Refinement of Features

To use the consistency proof $S, W \vdash R$, we need to refine the tuple (R, W, S), using the previous definitions of a home automation system. Consider a feature, f_i, with its specification, S_i. A difference between requirement engineers and feature managers is that the former works with specifications and the latter handles actual implementations. We shall let it be up to the modeller to decide whether to model the specification or the implementation.

Let each $d \in D$ be a specification of how the device works and each $p \in P$ be a specification of how the disturbance effects the variables in V. For example: a device d_{heater} increases v_{temp}, and a disturbance

p_{wall} decreases v_{temp}. This gives a refinement of the assumptions into a conjunction of device and disturbance specifications. Assuming that the variables in V are implicitly defined through the sets D and P:

$$W_i = \bigwedge_{s=1}^{i_k} d_s \wedge \bigwedge_{s=1}^{i_m} p_s$$

2.4 Effects

We can have multiple devices in D and multiple disturbances in P manipulating the same variable in V, used by a number of features in F. Potentially, the result is many interactions between elements in D, P and F. It can therefore be advantageous to reduce the set of interactions, by removing some of the acceptable ones. A notion for this is primary and side effects [Wilson et al., 2008]. For example, the primary effect of a refrigerator is to cool the food, but a side effect is that it warms up the room. The latter is an interaction with the air conditioner, but one we can accept (since we prefer both us and our food to stay cold). We cannot exclude the heat production from our models, since we still have to ensure that the air conditioner produces enough cold air to counter the refrigerator. Figure 1 shows primary (PE) and side effect (SE) describing the relationships between D and V, these notations are used in the requirements R. Furthermore, we add environmental effect (ENV) as a third type to describe the effects the disturbances in P have on the variables in V.

2.5 Communication

Home automation systems are deployed using either a centralised gateway or a distributed architecture. The centralised gateway is a platform that implements a common interface to a heterogeneous collection of devices which communicate using low level protocols, e.g. Homeport [Le Guilly et al., 2013]. The exact choice of interface to devices is not a concern for this paper. We assume that communication is without delays and loss of messages. This is reasonable, because the network load is typically low for home automation, and because there are network protocols that guard against message loss.

3 A HOME AUTOMATION SCENARIO

Imagine that a home owner hires a company to install a home automation system with a number of features. The company prepares the system in an installation package by combining on-the-shelf products and connecting them. Before doing the actual installation, the company asks the very natural question: *Is this package going to behave as we expect it to?* As the home automation domain consists of many vendors with different products [Wilson et al., 2008], we are going to assume it is infeasible to enumerate all variations of on-the-shelf products. We will also assume that the company needs to adapt components for each installation to tailor it to the customer. Thus, it becomes infeasible to check all possible configurations beforehand. Instead, we focus on conducting an offline analysis of the *concrete* configuration to be installed. This gives us three advantages:

1. We do not have to consider an exponentially increase of possible configurations,
2. The company can reason about how to solve each interaction, as they know them before run-time,
3. The company can adapt components to avoid these interactions and re-verify the system. The method thus becomes iterative.

3.1 User Requirements

We consider a user requirement to be an informal description, by the owner of the home, that defines how the system is supposed to work. Throughout this paper, we let the owner ask for a home automation system for a single room, that includes Heating, Ventilation, and Air-conditioning (HVAC), a Humidity Control Service (HCS), and a Home Security System (HSS). The set U of user requirements[1] are:

u_{HVAC}**:** The temperature shall be kept between 19°C and 22°C, by using both a heating and a cooling device.

u_{HCS}**:** An automatic ventilation system, that will improve humidity.

u_{HSS}**:** An alarm that will sound on intrusion.

4 IDENTIFICATION OF COMPONENTS

In the previous sections, we have seen all specifications as predicates. However in a design phase it is more intuitive to use the corresponding models, thus we shall continue by treating the devices, D, disturbances, P, and features, F, to be sets of models and

[1]We use U for the set of informal *user* requirements, and reserve the letter R for the more refined and concrete requirements as identified by the company in section 4.

not predicates[2]. Recall that D and P operate on the variables in V. We keep each requirement $r \in R$ as a predicate, because they specify properties of the system. During this section we will identify the necessary elements of F, V, D, and P to have a complete verifiable system. We will assume that each element $f \in F$, $d \in D$, $v \in V$ and $p \in P$ will have a corresponding model in the modelling tool. The identification is accomplished by answering a series of questions in a well defined sequence:

1. Which features are required in the system?
2. For each feature $f \in F$: which input is needed to make decisions?
3. For each feature $f \in F$: which actuators does the feature need to perform its task, and which actions on them?
4. For each device $d \in D$: which effects do each device have on each variable?
5. For each variable $v \in V$: which disturbances are there on it?

For the requirements, we start with the set U and refine each of them into a number of more specific requirements. The extent of a requirement should match the input of the modelling tool, such that it can answer each of them. We will not translate them to UPPAAL syntax before Section 5, thereby leaving the choice of tool open. When writing the requirements, R, it is also important to remember the definition of consistency from Section 2.2. Each feature should have their own set of requirements, and the requirements for the heating feature f_{Heat} should not reference devices used in the HCS feature f_{HCS}. This also means that requirements will be reusable for configurations where we have no HCS features installed.

4.1 Required Features

By exploring the user requirements, U, we can identify and plan for four features to be present in the system.

f_{Heat}: A HVAC heating feature, which increases the temperature, by turning on a heater, when it becomes too cold.

f_{Cool}: A HVAC cooling feature, which decreases the temperature, by turning on an air conditioner, when it becomes too hot.

[2]If we are very formal, we could say that we substitute a model semantics $\mathcal{M}(C)$ for a predicate C; but this seems an overkill since the predicates are not formalised in the examples.

f_{HCS}: A HCS feature to add fresh air to the room, by opening a window. We do not consider what causes this, just allowing it to do so arbitrarily.

f_{HSS}: A HSS feature, sounding an alarm on intrusion, by use of a movement sensor.

We list each feature $f \in F$ in the first column in Table 1.

4.2 Needed Input

The inputs needed by the features to make decisions are listed in Table 1. The types and purpose of variables in V are also relevant when modelling the features. For instance, v_{temp} is a continuous variable and v_{move} is a boolean. For our set-points, user requirement u_{HVAC} defines the acceptable range. In an attempt (which we will later verify) to allow the two features to co-exist we set slightly overlapping set-points. f_{heat} will start at $v_{temp} \leq 19°\text{C}$ and heat until $v_{temp} \geq 21°\text{C}$. f_{cool} will start at $v_{temp} \geq 22°\text{C}$ and stop at $v_{temp} \leq 20°\text{C}$.

Table 1: Variables and devices needed by each feature.

Feature $f \in F$	Variables $V_f \subseteq V$	Devices $D_f \subseteq D$	Actions
f_{Heat}	$\{v_{temp}\}$	$\{d_{heater}\}$	on/off
f_{Cool}	$\{v_{temp}\}$	$\{d_{aircon}\}$	on/off
f_{HCS}	$\{\}$	$\{d_{window}\}$	open/close
f_{HSS}	$\{v_{move}\}$	$\{d_{alarm}\}$	on

4.3 Needed Actuators

Also the devices and actions are listed in Table 1. The HVAC devices can both be turned on and off, the window can be opened and closed, the alarm can only be turned on. For this example, it is not necessary to consider what turns off the alarm again.

4.4 Device Effects

The effects are shown in Table 2, were *PE* is primary effect and *SE* is side effect. It is important to consider this connection after identifying the set of variables. Consider for example d_{window}. Here, we could easily have considered that it improves humidity in the room. However, it is not relevant as nothing acts on humidity. Instead it is important to ask whether changing the state of the window causes change to v_{move} and v_{temp} – which it does.

4.5 Disturbances

The included disturbances for our example:

Table 2: Mapping between devices and variables.

	v_{temp}	v_{move}
d_{heater}	*PE*	
d_{aircon}	*PE*	
d_{window}	*SE*	*SE*
d_{alarm}		

v_{temp} slowly approaches the outside temperature, caused by a wall, p_{wall}.

v_{move} is caused by a burglar, $p_{burglar}$, while breaking into the house.

4.6 HVAC Requirements

We recall that, the user requirement u_{HVAC} on v_{temp} was: *The temperature shall be kept between 19°C and 22°C, by using both a heating and a cooling device.* For v_{temp}, we can only guarantee that nothing is working against these set points:

r_1: When $v_{temp} < 19°$C, no device, $d \in D$, can decrease it as its primary effect.

r_2: When $v_{temp} > 22°$C, no device, $d \in D$, can increase it as its primary effect.

For d_{heater} and d_{aircon}, we know that they are less efficient if something works against them:

r_3: When d_{heater} is on, no device, $d \in D$, can decrease v_{temp} as its primary effect.

r_4: When d_{aircon} is on, no device, $d \in D$, can increase v_{temp} as its primary effect.

For f_{heat} and f_{cool}, we have to guarantee that they get v_{temp} back into the comfortable range, if it moves out. This also ensures that nothing is preventing the devices from operating forever. The features also make the assumption that nothing will turn off their device while they are operating it:

r_5: When $v_{temp} < 19°$C then eventually $v_{temp} \geq 19°$C.

r_6: When $v_{temp} > 22°$C then eventually $v_{temp} \leq 22°$C.

r_7: d_{heater} remains on for the duration that f_{Heat} is active.

r_8: d_{aircon} remains on for the duration that f_{Cool} is active.

4.7 HCS Requirements

For the HCS service, the user requirement u_{HCS} was *An automatic ventilation system, that will improve humidity.* For d_{window}, we know that devices like heaters and air conditioners are less efficient if it is open:

r_9: When d_{window} is open, no device, $d \in D$, can decrease the environment temperature as its primary effect.

r_{10}: When d_{window} is open, no device, $d \in D$, can increase the environment temperature as its primary effect.

For the HCS, we check if the window can eventually open and that it always closes again some time later:

r_{11}: d_{window} can eventually open.

r_{12}: When d_{window} is open it can always close again.

4.8 HSS Requirements

The user requirement, u_{HSS}, was *An alarm that will sound on intrusion.* We need to ensure that the alarm is on if and only if the burglar is active. This translates to the following two queries:

r_{13}: d_{alarm} is never on when $p_{burglar}$ is not present.

r_{14}: d_{alarm} always turns on when $p_{burglar}$ is present.

Now that we have investigated and recorded the sets of specifications S, assumptions W, and requirements R, it remains to prove that $S, W \vdash R$. For this we use timed automata to model S and W and a temporal logic for R.

5 TIMED AUTOMATA AS MODELLING TOOL

The purpose of this section is to translate the informal descriptions from Section 4 into the formalism of a modelling tool. This will show that the identified models and requirements are sufficient to find feature interactions. The chosen tool is UPPAAL [Behrmann et al., 2004]. It is a well-known tool [Behrmann et al., 2011] for modelling, simulation, and verification of networks of timed automata. It has become popular for verification of real-time systems, including protocols, controllers, and schedulers.

5.1 Automata and Queries in UPPAAL

We present a subset of the UPPAAL syntax, sufficient for our needs. Refer to [Behrmann et al., 2004, Behrmann et al., 2006] for the full syntax and semantics for UPPAAL.

In UPPAAL, a project is a tuple $(A, Var, Chan, Q)$, where $A = A_1|A_2|\ldots|A_n$ is a composition of n parallel timed automata, Var is a set of variable names, $Chan$ is a set of channel names, and Q is a set of UPPAAL queries. An automaton A_i is given by the tuple (L, l_0, C, E, I), where L is a set of locations, l_0 is

the initial location, C is a set of clocks[3], E is a set of edges, and I assigns invariants to locations. As usual we shall depict automata graphically, see Figure 3-6. An edge $(l,g,a,u,l') \in E$ between two locations, l and l' is annotated with a guard, g, over the sets Var and C, an action a over the set $Chan$, and an update, u, over Var and C.

A UPPAAL query, $q \in Q$ is given in a subset of timed computation tree logic (TCTL). The query types used in this paper are: $A\Box\ \varphi$ which means 'φ is true for all reachable states', $E\Diamond\ \varphi$ which means 'φ is true for some reachable state', and $\varphi \rightsquigarrow \psi$ which means 'when φ is satisfied ψ follows this or some state later[4]'.

5.2 Variables

We associate each variable v in V to a variable x in Var in the UPPAAL project. The granularity is important to reduce the number of states UPPAAL has to explore later. In our example, we use integer values in degrees Celsius for v_{temp}. Note that the other models will restrict the range to $[18,24]$, thus resulting in only seven states. An alternative solution can be to segment it into ranges, e.g. 'low', 'comfortable', and 'high'.

To specify what actions are allowed on each variable, v_i, and to solve a technical problem with detecting why it changed (primary, side or environmental effect), we add a guardian timed automaton, $A_x \in A$ around x, shown in Figure 3. The available actions, $act_x[0],...,act_x[5] \in Chan$, represent either direction of each effect type; primary `PE`, side `SE`, and environment `ENV`. In UPPAAL, an action is modelled through a channel, where one automaton is the sender and another is the receiver. The semantics is that both automata take their transition simultaneously. Our variable automaton will be the receiver; the device and environment automata will be the senders.

The timed automaton, shown in Figure 3 represent a generic template where x represent an environment variable, each act_x an action that can be performed to update its value and $[Min,Max]$ its domain. It is a generic template instantiated with x, act_x, and the constants min and max, for each variable. The constants are used to initialise the variable x with a non-deterministic value between min and max. This ensures that we also cover cases where the room temperature is too low or too high from the beginning. To construct queries using effect types, we pass each action through their own location, e.g. `INC_PE`. Thereby we can query for `(temp > 21 &&`

[3]In our models, only the identifier `t` will refer to clocks.

[4]$\varphi \rightsquigarrow \psi$ is equivalent to $A\Box\ (\varphi \Longrightarrow A\Diamond\ \psi)$

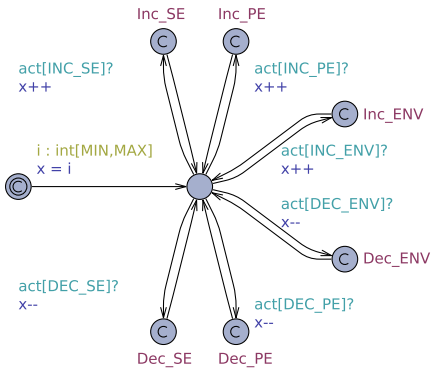


Figure 3: Generic variable template.

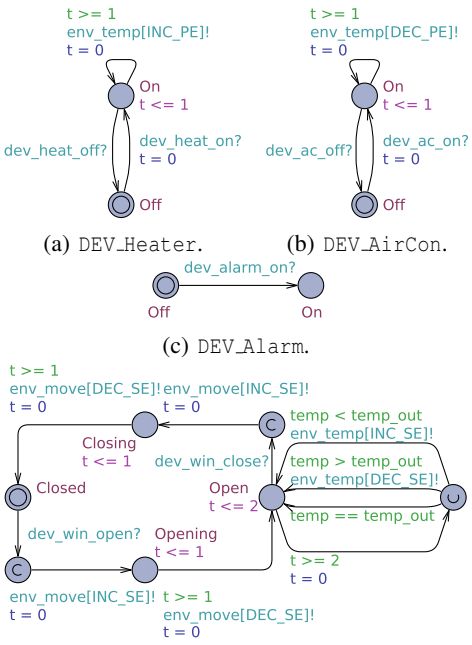
`VAR_Temp.Inc_PE)` to check whether a device has increased the temperature as its primary effect, to a value above 21C. A *committed* location, shown by the letter `C` inside the circle, means that time is not allowed to pass while the automaton is here.

In our example, we have two kinds of variables; a continuous V_{temp} and a boolean V_{move}. For both we use the same template, but interpret x differently. For the continuous v_{temp}, x is simply the temperature of the room in degree Celsius. For the boolean v_{move}, $x > 0$ is true and $x == 0$ is false – e.g. the variable keeps count of how many objects are moving. Notice that queries on primary/side/environment effects apply only to the change of the variable in the boolean case, which suffices for the example. Alternatively, one should use three values to account for each effect type.

5.3 Devices

The general scheme for adding devices is to add the set of actions, from Table 1, as $Act \in Chan$ with the interface to the device. For example, d_{heater} can be turned on and off and d_{window} can be opened and closed. The automaton $A_i \in A$ constructed for each device $d \in D$ will be the receiver on these channels, and will be the sender on the variable channels created above.

The HVAC devices, d_{heater} and d_{aircon} are depicted in Figure 4(a) and 4(b) respectively. They have two states, on and off. The heater increase v_{temp} by one each time unit while it is on. The air conditioner decreases it in a similar manner. The clock t keeps track of the time, and an invariant on the `On` location forces it to update. The window device, d_{window}, in Figure 4(d) has more states to capture when opening and closing. When the window is opened, v_{temp} moves towards the outside temperature, one degree Celsius every second time unit. This means that the temperature will converge faster (see Figure 6(a) as well) towards



(a) DEV_Heater. (b) DEV_AirCon.

(c) DEV_Alarm.

(d) DEV_Window.

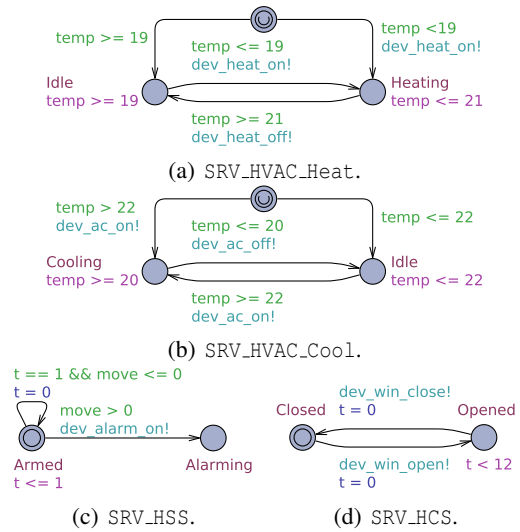
Figure 4: Devices.

the outside one, when the window is open. Again we use the time aspect, with the clock t, to capture how quickly v_{temp} changes and how long the window takes to open and close. The alarm is simple. It cannot be turned off in this example.

5.4 Features

A feature, $f \in F$ is a timed automaton, $A_i \in A$, which reads the values of global variables in V and performs actions by synchronising on device channels. The features in our example do not add further channels or global variables. However, if the system later turns out to have feature interactions, new channels can be used to allow the features to communicate. Communication and coordination between features can be used to resolve feature interactions.

The features in our HVAC services are f_{Heat} in Figure 5(a) and f_{Cool} in Figure 5(b). Both are similar and start in an initial state, from which they switch to `Idle` or `Heating`/`Cooling`, depending on the current value of v_{temp}. From here they attempt to keep v_{temp} between their respective set-points, by switching on and off their respective devices. The feature f_{HSS}, Figure 5(c), checks v_{move} every time unit and sounds d_{alarm} if it is true. Finally, the feature f_{HCS}, in Figure 5(d), opens and closes d_{window}. No condition is set, thus it can do so freely, except that the window cannot be open for more than 12 time units at a time. Allowing d_{window} to be open forever will



(a) SRV_HVAC_Heat.

(b) SRV_HVAC_Cool.

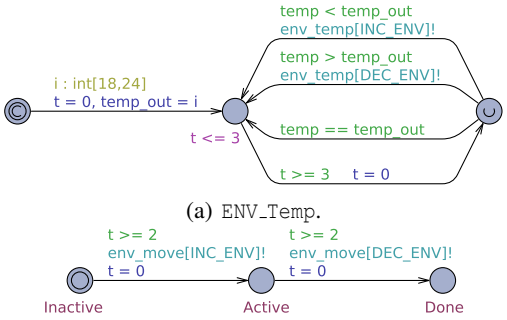
(c) SRV_HSS. (d) SRV_HCS.

Figure 5: Feature models.

break requirements, because it will prevent d_{heater} and d_{aircon} from running forever. This is actually a feature interaction which was prevented in the design.

5.5 Disturbances

The disturbances, P, are modelled similar to features, but instead of using devices, they control the variables in V directly using the environment effect type. Thus each disturbance $p \in P$ is a timed automaton, $A_i \in A$.



(a) ENV_Temp.

(b) ENV_Burglar.

Figure 6: Environment.

The wall, p_{Wall} in Figure 6(a), starts by choosing a non-deterministic outside temperature, to ensure we cover all the cases: lower than the set-points, between the set-points, and higher than the set-point. Each three time units the environment moves the v_{temp} closer to this outside temperature. The burglar, $p_{Burglar}$ in Figure 6(b), waits at least two time units before breaking in (to ensure that all models are settled), and stays inside the house for at least two time units when active, before leaving again with the loot.

Table 3: Requirements queries in UPPAAL and their results.

	UPPAAL Query	Result	Belongs to
q_0	`A[] not deadlock`	Passed	
q_1	`A[] !(temp < 19 && VAR_Temp.Dec_PE)`	Passed	`VAR_Temp`
q_2	`A[] !(temp > 22 && VAR_Temp.Inc_PE)`	Passed	`VAR_Temp`
q_3	`A[] !(DEV_Heater.On && VAR_Temp.Dec_PE)`	Passed	`DEV_Heater`
q_4	`A[] !(DEV_AirCon.On && VAR_Temp.Inc_PE)`	Passed	`DEV_AirCon`
q_5	`temp < 19 --> temp >= 19`	Passed	`SRV_HVAC_Heat`
q_6	`temp > 22 --> temp <= 22`	Passed	`SRV_HVAC_Cool`
q_7	`A[] !(SRV_HVAC_Heat.Heating && !(DEV_Heater.On))`	Passed	`SRV_HVAC_Heat`
q_8	`A[] !(SRV_HVAC_Cool.Cooling && !(DEV_AirCon.On))`	Passed	`SRV_HVAC_Cool`
q_9	`A[] !(DEV_Window.Open && VAR_Temp.Inc_PE)`	Failed	`DEV_Window`
q_{10}	`A[] !(DEV_Window.Open && VAR_Temp.Dec_PE)`	Failed	`DEV_Window`
q_{11}	`E<> DEV_Window.Open`	Passed	`SRV_HCS`
q_{12}	`DEV_Window.Open --> DEV_Window.Closed`	Passed	`SRV_HCS`
q_{13}	`A[] !(DEV_Alarm.On && ENV_Burglar.Inactive)`	Failed	`SRV_HSS`
q_{14}	`ENV_Burglar.Active --> DEV_Alarm.On`	Passed	`SRV_HSS`

5.6 Checking the Requirements

To perform consistency checks, the models are combined in one network. Both the assumptions, W, and specifications, S, are timed automata in UPPAAL. We can therefore use their parallel composition to combine them; $A = W_1|\ldots|W_n|S_1|\ldots|S_n$. Recall that a W_i is composed of devices from D and disturbances from P. In UPPAAL, each requirement is represented by a UPPAAL TCTL query q that evaluates to true if the model satisfies it, and to false otherwise. We can combine queries by logical conjunction; $\bigwedge_{i=1}^{n} q_i$. The consistency check for feature f_i thus becomes $W_i \parallel S_i \models R_i$. In practice, for the system to be consistent, we require that all results must list as *Passed*. All queries are shown in Table 3 and the complete exhaustive check for all queries in the table finishes in less than 30 seconds on a modern laptop.

In the queries, we checked safety properties ($q_{0-4,7-10,13}$), liveness properties ($q_{5,6,12,14}$), and reachability (q_{11}). Both states (e.g. `VAR_Temp.Dec_PE`) and variables (e.g. `temp`) are used in the queries, but not time. Timing requirements can be incorporated with the UPPAAL tool, either explicitly using constraints on clocks in the queries or indirectly by specifying timing constraints in the models and checking reachability. Exploring fairness properties is not feasible, but here one could consider using probabilistic transitions. Other work has already shown examples on how to model non-functional requirements see [Repasi et al., 2012] and analyse reliability [Le Guilly et al., 2015] in a probabilistic version of UPPAAL.

If queries have syntactical inconsistencies (wrong variable names, etc.), the UPPAAL tool will protest already when parsing the query. For queries that evaluate to false, we can use counter-example generated by Uppaal to understand the cause. Notice that queries q_9 and q_{10} fail, because nothing prevents d_{heater} and d_{aircon} from manipulating v_{temp} while d_{window} is open. q_{13} fails because d_{window} moves, thus starting d_{alarm}. All three are therefore examples of feature interactions to be resolved.

6 RELATED WORK

The definitions of features, consistency, and interactions from requirement engineering [Classen et al., 2008], outlined in Section 2.2, are reused in this paper. We have refined those and presented a novel approach to perform consistency checks. Their tool makes use of the event calculus and a SAT solver.

The feature manager [Kolberg et al., 2003, Wilson et al., 2008] works online. However, we were still able to utilise the layers and notions of primary/side effects in our method, see Section 2.3. We note that, queries $q_{1-4,9,10}$ are equivalent to asking if a lock on the v_{temp} has been violated. Similar for queries $q_{7,8}$ on d_{heater} and d_{aircon}.

A third approach to feature interaction analysis is based on object-oriented descriptions [Nakamura et al., 2005]. Generally they share the same layered view with features, devices and variables. Compared to our approach, their early work considered simple service descriptions without branches, loops, or time. Branches and loops are later added into the language, where they apply the SMV model checker [Leelaprute et al., 2005]. This language, and their queries, are roughly equivalent with ours. The exceptions being handling of time and that their approach is purely textual. They have tried using JML [du Bousquet et al., 2009], concluding that verification was long, not easy, and only partially successful. They do, however, provide some interesting perspectives on the

need to abstract from implementation specific details and on requirements elicitation by reading safety instructions. They consider appliance/service pairs in most of their work, with chains of services first considered on Event-Condition-Action (ECA) rules [Inada et al., 2012]. Another important consideration is the granularity of environmental impacts [Nakamura et al., 2013], where they ask *how much* a device changes a variable.

There is also work on analysing requirements from an existing development platform, by construction of dependency graphs [Metzger and Webel, 2003]. Compared to our work they consider larger systems on a more abstract level, where they hint at possible interactions at the requirement level. They do not model the internal workings of the features.

Another methodology for verification of smart environments have also been proposed [Corno and Sanaullah, 2014]. Both methods contain roughly the same steps, but in a different order. They start from an existing system, thus is able to enumerate all devices in the system. In our case, we started from the user requirements, thus had to identify the devices by identifying the features first. They provide more details on the individual steps, which can be used while constructing models in our method as well. In contrast we put more emphasis on the models themselves.

Other work with UPPAAL provide a good example of a model for a larger home automation system [Augusto and McCullagh, 2007]. Their focus is not on feature interactions and they do not provide much on requirements. Another example uses stochastic and hybrid models [David et al., 2015] in UPPAAL to more precisely model energy aware buildings with differential equations [David et al., 2013]. There is also an example of how to find non-functional feature interactions in UPPAAL in the automotive domain [Repasi et al., 2012].

There is also work on software product lines and safe configurations [Classen et al., 2010]. Here they concentrate on modelling feature compositions and developing efficient algorithms for model checking the exponential number of combinations in software product lines. Another example of similar work is integration of a component concept in UPPAAL [David et al., 2010]. Both work on a more foundational level on models, including transition systems and algorithms, and do not directly consider the home automation domain and features that interacts with their environments.

7 CONCLUSION AND FUTURE WORK

In this paper we consider systems, where the configuration is known at the time of the analysis. In software product lines, this method can be used to verify the individual combinations of features. We refined the definitions of existing work and suggested a set of activities to identify elements of the system. We argue that the activities help to identify which models are needed to capture all aspects of the problem domain, and to identify the relationships between them. The use of UPPAAL has lead to clarification and disambiguation of the concepts introduced in the related work, which is reflected in our definitions.

For the relationship between devices and variables we used the three effect types; primary, side, and environmental. As briefly outlined in Section 6, others have started to consider a relationship where the effects are measured in degree of impact. It would therefore be possible to include this in the models. While our activities work well for constructing the models, how to obtain the requirements is still an open question. An inspiration could be elicitation of safety properties [Yan et al., 2007]. When interactions are found, there is no automatic way of resolving them. Either requirements can be relaxed or features can be further constrained. In both cases it requires insight into the system.

From several years of experience with model checking, we have learnt that verification easily suffers state space explosions, as the number of models and states increase. In our example we had a single room with three services – in total 12 models. We showed that on these limited number of components it is feasible to do the exploration. With more components, we would consider a static pruning of the model, removing features that do not share outputs and thus cannot interfere. Another option would be switching to the statistical version of UPPAAL, UPPAAL SMC, that performs simulations on the model instead of full state-space exploration. This could help to improve scalability, but at the cost of a less accurate answer with a probability for satisfying a property (or not). We are currently planning on continuing this line of work, by studying these options for handling larger number of components.

ACKNOWLEDGEMENTS

The underlying research for this paper is partially supported by the European FP7-ICT project INTrEPID, and the Danish DI ITEK ITOS and TotalFlex

projects[5].

REFERENCES

Augusto, J. C. and McCullagh, P. (2007). Ambient intelligence: Concepts and applications. *Computer Science and Information Systems*, 4(1):1–27.

Behrmann, G., David, A., and Larsen, K. (2004). A tutorial on UPPAAL. *Formal methods for the design of real-time systems*, pages 33–35.

Behrmann, G., David, A., Larsen, K. G., Pettersson, P., and Yi, W. (2011). Developing uppaal over 15 years. *Software: Practice and Experience*, 41(2):133–142.

Behrmann, G., David, R., and Larsen, K. G. (2006). A tutorial on Uppaal 4.0.

Classen, A., Heymans, P., and Schobbens, P.-Y. (2008). What's in a feature: A requirements engineering perspective. In *Fundamental Approaches to Software Engineering*, pages 16–30. Springer.

Classen, A., Heymans, P., Schobbens, P.-Y., Legay, A., and Raskin, J.-F. (2010). Model checking lots of systems: efficient verification of temporal properties in software product lines. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 335–344. ACM.

Corno, F. and Sanaullah, M. (2014). Modeling and formal verification of smart environments. *Security and Communication Networks*, 7(10):1582–1598.

David, A., Du, D., Guldstrand Larsen, K., Legay, A., and Mikuionis, M. (2013). Optimizing control strategy using statistical model checking. In Brat, G., Rungta, N., and Venet, A., editors, *NASA Formal Methods*, volume 7871 of *Lecture Notes in Computer Science*, pages 352–367. Springer Berlin Heidelberg.

David, A., Larsen, K. G., Legay, A., Mikučionis, M., and Poulsen, D. B. (2015). Uppaal SMC tutorial. *International Journal on Software Tools for Technology Transfer*, pages 1–19.

David, A., Larsen, K. G., Legay, A., Nyman, U., and Wasowski, A. (2010). Timed I/O automata: a complete specification theory for real-time systems. In *Proceedings of the 13th ACM international conference on Hybrid systems: computation and control*, pages 91–100. ACM.

du Bousquet, L., Nakamura, M., Yan, B., and Igaki, H. (2009). Using formal methods to increase confidence in a home network system implementation: a case study. *Innovations in Systems and Software Engineering*, 5(3):181–196.

Inada, T., Igaki, H., Ikegami, K., Matsumoto, S., Nakamura, M., and Kusumoto, S. (2012). Detecting service chains and feature interactions in sensor-driven home network services. *Sensors*, 12(7):8447–8464.

[5]www.fp7-intrepid.eu, itek.di.dk, www.totalflex.dk

Kolberg, M., Magill, E. H., and Wilson, M. (2003). Compatibility issues between services supporting networked appliances. *Communications Magazine, IEEE*, 41(11):136–147.

Le Guilly, T., Olsen, P., Ravn, A., Rosenkilde, J., and Skou, A. (2013). Homeport: Middleware for heterogeneous home automation networks. In *Pervasive Computing and Communications Workshops (PERCOM Workshops), 2013 IEEE International Conference on*, pages 627–633.

Le Guilly, T., Olsen, P., Ravn, A. P., and Skou, A. (2015). Modelling and analysis of component faults and reliability. In Petre, L. and Sekerinski, E., editors, *From Action System to Distributed Systems: The Refinement Approach*. Accepted for publication.

Leelaprute, P., Nakamura, M., Tsuchiya, T., Matsumoto, K.-i., and Kikuno, T. (2005). Describing and verifying integrated services of home network systems. In *Software Engineering Conference, 2005. APSEC '05. 12th Asia-Pacific*, pages 10 pp.–.

Matsuo, T., Leelaprute, P., Tsuchiya, T., Kikuno, T., Nakamura, M., Igaki, H., and Matsumoto, K. (2006). Automatically verifying integrated services in home network systems. In *Proc. International Technical Conference on Circuits/Systems, Computers and Communications (ITC-CSCC2006)*, volume 2, pages 173–176.

Metzger, A. and Webel, C. (2003). Feature interaction detection in building control systems by means of a formal product model. In *FIW*, pages 105–122.

Nakamura, M., Igaki, H., and Matsumoto, K.-i. (2005). Feature interactions in integrated services of networked home appliances. In *Proc. of Intl. Conf. on Feature Interactions in Telecommunication Networks and Distributed Systems (ICFI05)*, pages 236–251.

Nakamura, M., Ikegami, K., and Matsumoto, S. (2013). Considering impacts and requirements for better understanding of environment interactions in home network services. *Computer Networks*, 57(12):2442–2453.

Repasi, T., Giessl, S., and Prehofer, C. (2012). Using model-checking for the detection of non-functional feature interactions. In *Intelligent Engineering Systems (INES), 2012 IEEE 16th International Conference on*, pages 167–172.

Wilson, M., Kolberg, M., and Magill, E. H. (2008). Considering side effects in service interactions in home automation-an online approach. *Feature Interactions in Software and Communication Systems IX*, page 172.

Yan, B., Nakamura, M., du Bousquet, L., and Matsumoto, K.-i. (2007). Characterizing safety of integrated services in home network system. In Okadome, T., Yamazaki, T., and Makhtari, M., editors, *Pervasive Computing for Quality of Life Enhancement*, volume 4541 of *Lecture Notes in Computer Science*, pages 130–140. Springer Berlin Heidelberg.