J Reliable Intell Environ (2016) 2:75–91
DOI 10.1007/s40860-016-0020-z

ORIGINAL ARTICLE

User constraints for reliable user-defined smart home scenarios

Thibaut Le Guilly[1] · **Michael K. Nielsen**[1] · **Thomas Pedersen**[1] · **Arne Skou**[1] · **Jesper Kjeldskov**[1] · **Mikael Skov**[1]

Received: 21 January 2016 / Accepted: 12 April 2016 / Published online: 4 May 2016

Abstract Defining control scenarios in a smart home is a difficult task for end users. In particular, one concern is that user-defined scenarios could lead to unsafe or undesired state of the system. To help them explore scenario specifications, we propose in this paper a system that enables specification of constraints restricting the control commands that can be used inside user-defined scenarios. The system is based on timed automata model checking abstracted by event condition action rules. A prototype was implemented, including a user interface to interact with the user. The usability of the system and interface was evaluated in a user study which results are reported here.

Keywords Smart home · User-defined scenarios · Formal methods · Reliability · Safety · Graphical user interface

1 Introduction

In his seminal paper "The Computer for the 21st Century", Weiser concludes by stating that machines need to fit human environments to make them less frustrating to use [37]. This aim has been an inspiration to much of the research conducted in the area of ubiquitous, pervasive and intelligent environments, trying to make computers an integral part of the people's surroundings, using them in ways similar to other common objects, without much thought. However, few things are more frustrating than computer systems that do not work the way we want them to, or do not provide the functionality we expect. In fact, such systems quickly attract our attention as being unreliable or poorly designed, and often cause frustrated user experiences in stark contrast to Weiser's vision of "calm" ubiquitous computing. For machines to fit well into human environments, the ensemble of machines and environments must be made technically reliable and functionally transparent.

Ensuring reliability and transparency of intelligent environments is a complex task due to their dynamic and distributed nature, and the number of sub-systems involved. There has been a lot of research trying to increase the reliability of different aspects of intelligent environments, and in particular in using formal verification techniques to do so. For example, Augusto and Hornos [6] show how to formally model intelligent environments and use model checking to validate their properties. Another example is Coronato and Pietro [13] who show how to use the ambient calculus for designing ambient intelligence applications. The overview of Corno and Sanaullah [12] of formal modeling and verification of Smart Environments also illustrates well the growing attention for formal methods in this area.

There are different aspects of dependability and reliability of reactive systems. One aspect is the reliability of physical components that can be used to compute mean time to failure and the impact of failures on system functionality, for example [23]. Another is the correctness of the control algorithms being applied to the environment that implement the intelligent part of the system. If the objectives of the control system differ from the expected ones, or if the control specifications do not satisfy the requirements, there will be a mismatch between the expected and actual behavior of the system. To ensure the correctness of control specifications, we previously proposed a toolchain to facilitate the development of reliable home automation controllers [14] based on timed automata [3]. However, due to the complexity of the formalism used, this toolchain, similarly to other formal

✉ Thibaut Le Guilly
thibaut@cs.aau.dk; leguilly.thibaut@gmail.com

[1] Department of Computer Science, Aalborg University, Aalborg, Denmark

approaches, is difficult to use by users who are not familiar with such formalisms. Hence, while helping make the intelligent environment technically reliable, it does not help make it functionally transparent for the user.

Another lesson learned from past research is that a smart home needs to evolve with the needs and requirements of their inhabitants, adapting to new routines and habits. This has been highlighted by, amongst others, Davidoff et al. [15] who discuss the dynamic nature of activities in American households. They show that common household activities are a complex mix of interactions between people, requiring varying equipment and devices, which can easily fail if elements are missing. Therefore, a completely ubiquitous system to control a smart home is likely to have difficulties coping with the fast changing behavior of users. Making the smart home even smarter, for example, by implementing a learning control algorithm, could lead to uncertainty for users on who actually has control over the house, if it is even feasible [36]. In response, Rogers [32] has advocated a more user-engaged approach where "technology is designed to enable people to do what they want" and thus helps them to understanding and stay in control of the system.

From a system perspective, ideally users would specify an environment that fits their needs, and update these specifications as they evolve. However, as pointed out by Christopher Alexander [2], determining what makes a design fit its environment is usually quite difficult. He argues that instead fit between form and context is easier to describe by the absence of misfit. While this observation was originally made in relation to the architectural design of physical homes, we believe that this also applies to smart ones. This is illustrated by Brush et al. [11] who showed that smart home users often do not know exactly what to expect from the system when acquiring it, and that their preconceptions often turn out to be unfulfilled. It is, for example, easier to specify that the temperature of a room should not go above a certain value, than it is to specify what the temperature should be for several possible parameters. Enabling specification of misfits, or what a system should not do, thus appears to be easier for users. Moreover, by constraining the possible scenarios with these misfits, users could safely explore developing scenarios without having to worry about them potentially leading to undesired configurations of the system. Letting users set the constraints themselves could also make it easier to update or adapt to changes. With the scenarios, specifying what the system should do, on one hand, and the constraints restricting them on the other, the objective is thus to ensure that no specified scenario can lead to violation of a constraint. Here the question is thus how to empower users to detect possible misfits created while specifying scenarios using formal methods and the verification capabilities that they provide over intelligent systems. The objective is also to design a system that can be deployed in a concrete smart home system, and able to validate *online* scenarios and newly added constraints, without requiring specific insights on the smart home environment. In our previous work, we proposed a system based on timed automaton, enabling users to specify scenarios and constraints, and ensuring that the former do not violate the latter [24]. This paper extends this contribution by exploring the details of the system and the verification process underlying it, and presents an updated user interface, with a new design reflecting lessons learned from previous experiments. The main contributions of the paper are as follows:

- Executable models and a verification approach of smart home scenarios in the form of timed automata that can be verified for freedom of undesirable control commands defined as constraints.
- An abstraction layer for the models and constraints in the form of event condition action (ECA) rules, making them accessible to end users.
- A system enabling online validation of newly added scenarios against constraints, and execution of valid models.
- A user interface providing a graphical representation of ECA rules to let user specify both scenarios and constraints.
- An evaluation of the interface to determine if users are able to understand the notions of constraints, and if they could improve the understanding that user has of conditions in ECA rules compared to the previous version.

The paper is organized as follows. In Sect. 2, we present the modeling approach to smart spaces that enables the specification of constraints and scenarios, and their formal verification. The ECA rules used to facilitate the specifications of scenarios and constraints are also described, and their semantics in terms of timed automata is provided. Sect. 3 presents the components of the system enabling specification and verification, and the redesigned user interface enabling users to input specifications and constraints. This updated user interface was evaluated in a completely new user study that consisted of a usability test, described in Sect. 4. Related work is discussed in Sect. 6, and a conclusion as well as further work is given in Sect. 7.

2 Modeling and verification of smart home controllers

This section introduces the models and verification techniques that serve as a basis for the system presented in this paper.

2.1 System model

The first step in modeling a system is to identify its components. In this paper, a smart home is viewed as a control

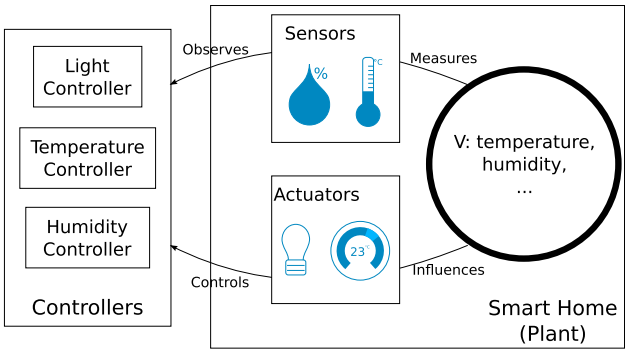


Fig. 1 Illustration of a smart home as a control system

system, represented as a plant, the system to be controlled, and a set of controllers. The plant is represented by a set of controllable variables that define its state space. It also contains a set of sensors making the variables observable, and a set of actuators providing control capabilities over them. An illustration of this view is shown in Fig. 1. Disturbances are ignored here for simplicity but can be added as shown in [30].

The components of the system can formally be represented as a network of timed automata with discrete variables [14]. A timed automaton is a directed graph with a set of nodes called locations, and a set of edges connecting these nodes. It also contains a set of clocks evolving simultaneously that can be evaluated and reset. Discrete variables are also used to facilitate the representation of component parameters. The state of such a timed automaton is composed of a location, a valuation of the clocks and a valuation of the discrete variables. Timed automata locations and edges can be decorated, respectively, with invariants and guards over clocks and discrete variables. A location invariant must hold as long as the location is part of the state. An edge can only be taken when its guard is satisfied by the current clock and variable valuations. A network of timed automata is a parallel composition of timed automata that can communicate through channels to synchronize and exchange information (more details in [1,3,8,14]). This formalism fits well with intelligent environments as it makes it possible to represent the parallel computations of each components, as well as their dynamics and interactions. The system model developed in this paper is as follows.

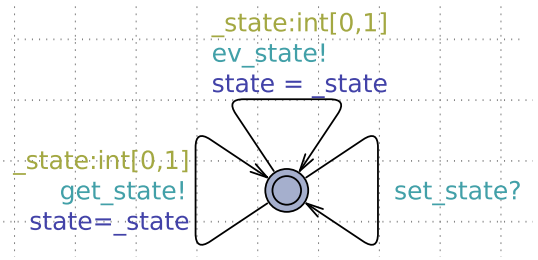
A sensor is represented by a discrete variable, and two communication channels. The GET channel enables other components to query the latest value measured by a sensor. The EV channel enables other components to be notified when the value measured by a sensor changes, and to obtain its value.

An actuator is represented by a discrete variable representing its current set point, and three communication channels. The GET and EV channels are similar as for sensors. The SET channel enables components to update the state of an actuator.

A controller is a timed automaton interacting with the sensors and actuators, reading sensor or actuator values, waiting for events or updating set points.

Plant variables are in this paper represented directly through sensors and actuators. The reason is that their behavior depends on the concrete environment in which a system is deployed, making it difficult to provide a generic model for them. Therefore, only their respective domains are specified through the discrete variable of sensors and actuators. Their behavior evolves non-deterministically within their domains similarly to actuator variables. It is rational to do so in this context because most actuators already implement embedded supervisory controllers taking only set point values as input, as thermostats for example, and verifying their correctness is not the objective here. This also takes into account the non-deterministic behavior of the user interacting with actuators (the objective is to restrict the action of the system, not of the user). Note that the same models have been used in combination with environment specifications in [14], which could thus also be done in the proposed process and tool. The choice of not doing it in this work aims at obtaining a generic approach that can be easily applied in any smart home, without requiring environment specification, which can be difficult to obtain.

A timed automaton representation of an actuator model is shown in Fig. 2. A sensor representation is similar apart from the absence of a SET? synchronization edge. The GET and EV transitions send information. To do so they use a select statement (in the first line) that randomly initializes a temporary variable within a given interval. These transitions are thus non-deterministic as they both represent a set of transitions each assigning a particular value to the variable, among which one is picked at random. This reflects the non-deterministic behavior of plant variables in the model. The value of the temporary variable is then assigned to the state of the sensor or actuator. Note that the GET and EV channels in the associated edges are followed by an exclamation mark, indicating that they provide information, while the SET channel is followed by a question mark indicating that it receives information. A component providing (receiving) information is called a sender (receiver). The three channels are declared as *urgent*, indicating that time cannot pass while they can synchronize. It ensures that when a query to a sensor or actuator is made, the obtained value corresponds to its current state, thus communication delays are ignored. EV and SET channels are also declared as broadcast, indicating that a single sender can synchronize with multiple receivers. This enables sensors and actuators to notify several controllers when their state is updated, and the actions of controllers to be observable externally, which will be useful for verification. Note that locations can also be declared urgent (with a capital U in their center), to indicate that time cannot pass while part



(a)

```
int[0,1] state;
urgent chan get_state;
urgent broadcast chan set_state;
urgent broadcast chan ev_state;
```

(b)

Fig. 2 Model of an actuator. **a** A timed automaton representing an actuator. **b** Variables and channel declarations

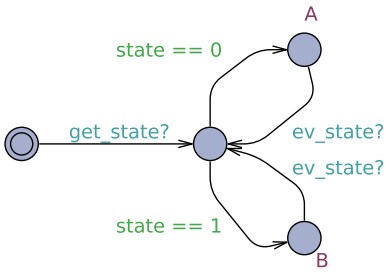


Fig. 3 A timed automaton representing a controller

of the system state. Committed locations (with a capital C in their center) also prevent time delays but also ensure that the next transition taken in the network moves an automaton out of one of them.

An example of a controller is shown in Fig. 3. It first queries the value of the actuator, and picks a transition based on the value of its state variable. It then waits for an event from the actuator and repeats. As the variables representing the states are declared globally, controllers can read and write variables directly.

2.2 Model checking

Model checking is a decision procedure which given a formal representation of a system $\mathcal{M}(s)$ and a set of formal requirements R returns *yes* if $\mathcal{M}(s)$ satisfies R (written $\mathcal{M}(s) \models R$) and returns no, often with a counter example, if $\mathcal{M}(s) \not\models R$. In this paper, it is used to validate that a smart home model in the form of a network of timed automata, satisfies a given set of properties. To do so the UPPAAL [8] model checker is used. It is a toolbox for modeling, simulating and verifying networks of timed automata extended with discrete variables. Requirements are expressed in a temporal extension of the computational tree logic (CTL), a branching time logic. The model checking is performed on a tree of states where each

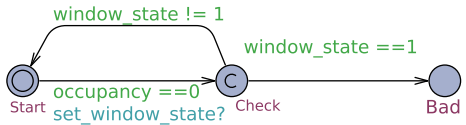


Fig. 4 Example of an observer automaton

path represents a possible execution of a given network. The outer modalities A and E apply to a tree and mean "for all traces" and "for some traces", while $\Box$ and $\Diamond$ apply to a trace and mean "for all states" and "for some states". Two types of formulae can be defined: state formulae that represent properties of states, and path formulae that quantify state formulae over traces. Three types of properties can be expressed:

> Reachability properties check whether a state formula φ is satisfied by some reachable state, written $E\Diamond\phi$.
> Safety properties check that something bad (represented by a state formula φ) will never happen. An example is $A\Box\varphi$ that expresses that all reachable states should satisfy φ.
> Liveness properties check that something good will eventually happen.

Without specifying the behavior of plant variables, liveness properties checking that controllers lead the plant to "good" states cannot be used. The analysis focuses instead on safety properties, ensuring the absence of undesirable control actions from the controllers, represented by synchronization transitions between controllers and actuators. However, actions cannot be directly expressed in the logic. To detect undesirable actions, *observer automata* are used to represent the properties to verify. An observer automaton monitors the actions from controllers, and under specified condition moves to a *Bad* location. Note that using observer automata is a common approach, used for example in [9,18,34]. Composing such an observer with the model enables checking for reachability of bad locations given a set of controllers, thus detecting misbehaviors, using the query $A\Box\neg Obs.bad$, indicating that for all states in the tree of traces, the location $Obs.bad$ is not part of it. An example of an observer is shown in Fig. 4. It moves to a committed location as soon as it observes a SET action from a controller to a window while the variable occupancy is equal to zero, and moves to a bad location if right after the variable window_state is equal to one. The observers are used to constrain the specification of scenarios to well-behaved ones. Note again that if environment specifications are included in the model, liveness and safety properties on plant variables could be performed, but was not the objective in the presented work. Observers can also be used to express more complex properties over the control actions of the system, for example, including temporal restrictions.

2.3 Event condition action language

Modeling the control of a smart home using timed automata can be done in multiple ways. To facilitate their specification, we propose to abstract the controller models using event condition action (ECA) rules. This was initially proposed by Augusto and Nugent [7] that defined such a language to perform temporal reasoning using active databases applied in a smart home. In this paper, an ECA rule is composed of an event, a set of conditions and a set of actions:

An event is a predicate on a sensor or actuator state that causes the system to check the condition when its evaluation changes from *false* to *true*;
Conditions are composed of a set of predicates on sensor and actuator states;
Actions are composed of a set of control commands on actuators sent if the conditions hold.

Each language element of an ECA rule is considered as a block. An ECA rule thus consists of a sequence of such blocks. Three types of blocks exist, corresponding to events, conditions and actions. An event block is composed of:

A device (identifier) on which the event is defined;
A predicate on the state of the device, composed of a relational operator in $<, \leq, >, \geq, =, \neq$ and a value;
A duration, corresponding to the length of time (possibly null) for which the predicate should evaluate to true before the event represented by the block is triggered.

A condition block is identical to an event block, except that no duration is defined. An action block contains only a device and a value to which the device state should be set.

An example of such a scenario is as follows:

When $temperature < 18$
If $window = 0$
Then $thermostat = 21$

It is triggered when the variable $temperature$ goes under 18, and if the variable $window$ is equal to zero, it sets the actuator $thermostat$ to 21. A time duration can be added to events, indicating that its predicate should hold for a given amount of time before the condition is evaluated. Note that more complex scenarios could be defined, such as considering sequences of events, or time specifications for condition and actions. As the objective is to let the user define scenarios, it was, however, kept simple. A scenario thus includes only one event, but can contain several conditions and actions.

The semantics of the different language elements in the form of timed automata is shown in Fig. 5. Note that an interesting aspect of the proposed semantics is that it can be interpreted to apply the specified control in the environment. A single event, shown in Fig. 5a, is triggered by a synchronization on an EV channel from a sensor or actuator.

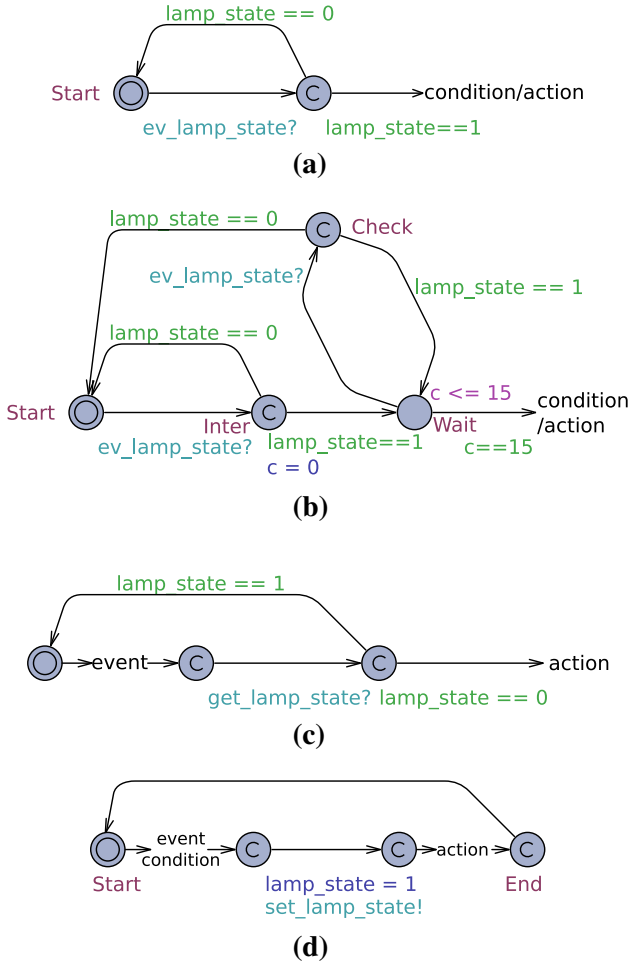


Fig. 5 Semantics of the language elements. **a** Simple event. **b** Simple event with duration. Note that c is a clock. Its value evolves with time passing. **c** Condition. **d** Action

If the updated value matches the event predicate, the conditions are evaluated. The illustrated example represents a lamp being turned on. A single event with duration, shown in Fig. 5b works similarly, but waits for a specified amount of time before evaluating the conditions. After checking the event predicate, the automaton moves to a *Wait* location and initializes a clock c. The invariant on the *Wait* location, and the guard on the edge going to the condition block ensures that it can only proceed to the condition part after a specified amount of time (in the example 15 time units). If during that time interval a state change is observed (using an EV channel), invalidating the event predicate, the scenario is reset, and the conditions and actions will not be evaluated. The illustrated example represents an event where a lamp is turned on, and not turned off again for 15 time units. A condition block, shown in Fig. 5c, obtains the current value of a sensor or actuator using a GET channel and evaluates its predicate based on the returned value. If true, the next condition or action is evaluated, otherwise the scenario is reset. Finally,

```
function ECAToTA(eca)
    initLoc = newInitialLocation()
    lastLoc = initLoc
    for all event : eca.events do
        newLoc = createCommittedLocation()
        edge = createEdge(lastLoc, newLoc)
        addEventSync(edge, event.device)
        edge = createEdge(newLoc, lastLoc)
        addGuard(edge, negate(event.relOp),
        event.value)
        lastLoc = newLocation()
        edge = createEdge(newLoc, lastLoc)
        addGuard(edge, event.relOp,
        event.value)
        if event.duration > 0 then
            c = newLocalClock()
            addInv(newLock, c, event.duration)
            addUpdate(edge, "c = 0")
            newLoc = createCommittedLocation()
            edge = createEdge(lastLoc, newLoc)
            addEventSync(edge, event.device)
            edge = createEdge(newLoc, lastLoc)
            addGuard(edge, event.relOp,
            event.value)
            edge = createEdge(newLoc, initLoc)
            addGuard(edge, negate(event.relOp),
            event.value)
        end if
    end for
    for all cond : eca.conditions do
        ...
    end for
    for all act : eca.actions do
        ...
    end for
end function
```

Fig. 6 Algorithm to transform an ECA rule to its executable TA representation

an action, shown in Fig. 5d, is a control command sent to an actuator using a SET channel.

The translation from an ECA rule to TA is done by transforming each block of the rule to its corresponding TA semantics. Figure 6 shows how this is done for events, which are the most complex elements to transform. First the initial location of the model is created (*Start* in Fig. 5b). A committed location is then added (*Inter* in Fig. 5b), linked by an edge to the initial location. This edge is decorated with a synchronization transition on the EV channel of the device associated with the event block. This represents the trigger for the TA model. Another location (*Wait* in Fig. 5b) is then added to the model, linked to the previous one by an edge decorated with a guard corresponding to the event predicate. An edge also links back the *Inter* location to the *Start* one with a guard representing the complement of the event predicate. This edge ensures that the model goes back to its initial state during execution, but has no influence on the verification phase. If the event includes a duration, a local clock is declared. This clock is reset to zero in the edge between the *Inter* and *Wait* locations. An invariant is added to the *Wait* location, corresponding to the duration of the event. A committed location is added (*Check* in Fig. 5b), linked to the *Wait* location by an edge decorated with a synchronization on the EV channel of the device associated with the event. This synchronization will thus be triggered if the device changes state while in the *Wait* location. Two edges are then added. The first one links the *Check* location to the initial one, decorated with the complement of the event predicate, indicating that the event predicate does not hold anymore. The second one links the *Check* location back to the *Wait* location, to continue waiting until the duration has expired. A new location is finally added, which will correspond to the first one of the following block (which could be another event or a condition), decorated with a guard on the local clock. If the event does not include a duration, the *Wait* location corresponds to the first one of the next block.

The transformation of conditions and actions is essentially similar to the one for events. For conditions, the GET channel is used instead of the EV to retrieve the current value of the device associated with the condition. Again, this is necessary for the execution of the model, but not for verification. Actions make use of the SET channel to notify of an update of the state of their associated device (used for execution), using an update statement to do so.

To simplify constraint specification, they are also abstracted by language constructs. Here we consider only simple constraints composed of a condition, and a forbidden action under this condition. An example is:

If $occupancy = 0$
Then no controller should change $window\_state$ to 1

This example is shown in Fig. 4. Again, more complex constraints could easily be defined, involving time for example. They were kept simple for usability purposes. The condition and action blocks of the constraints are similar to that of scenarios, except that the action block contains a predicate over its value. This enables defining forbidden actions

```
function ConstToTA(const)
    initLoc = newInitialLocation()
    loc = newCommittedLocation()
    bad = newLocation()
    edge = createEdge(initLoc, loc)
    addGuard(edge, const.condition.relOp)
    addSetSync(edge, const.condition.device)
    edge = createEdge(loc, initLoc)
    addGuard(edge, negate(const.action.relOp))
    edge = createEdge(loc, bad)
    addGuard(edge, const.action.relOp)
end function
```

Fig. 7 Algorithm to transform a constraint to its TA representation

such as *no controller should change thermostat to* >20. The algorithm used to transform constraint from their ECA representation to TA is shown in Fig. 7. After creating an initial location (*Start* in Fig. 4), a committed location is created (*Check*), with an edge between both. This edge is decorated with an input synchronization on a SET channel to observe an update on the service specified in the action part of the constraint. It is also decorated with a guard, corresponding to the predicate contained in the condition part of the constraint. Thus, this edge will be taken whenever a scenario updates the service specified in the action block, while the predicate of the condition block is satisfied. Two edges are then added from the *Check* location. One to the *Bad* location, decorated with a guard that corresponds to the action predicates, another to the initial location, decorated with a guard that corresponds to the complement of the action predicates. This ensures that a constraint model only moves to the *Bad* location when the observed action led the service associated with the action to an undesired state under the specified condition.

Having a language for specifying both scenarios and constraints, it can be used in a concrete system to interact with users.

3 HomeBlock

HomeBlock is a system that allows users to specify scenarios and constraints on their smart home environment. It enables them to safely create scenarios while being sure that they do not violate any predefined constraint. To do so it uses the models and verification techniques presented in the previous section. An overview of the system is provided in Fig. 8. This section details its components and functions going through the different step of modeling, specifying, verifying and executing scenarios and constraints.

3.1 Obtain plant information

The first step taken by the HomeBlock system upon start up is to retrieve information about available sensors and actuators in the environment. In the current setting, the HomePort [22] middleware is used to obtain it. This middleware offers a common interface to multiple subnetworks, in the form of a web interface available through a REST API over HTTP. Sensors and actuators can be accessed through devices that represent physical entities, containing services that encapsulate their functionalities. HomePort provides meta-information about each device and service, such as location in the house, human readable description, or unit. The state of services can be queried, actuators can be updated, and event notifications can be received using the *server sent event* mechanism. HomePort is thus an abstraction layer for the concrete smart home system.

3.2 Build plant model

As discussed in Sect. 2, a model for each sensor and actuator is needed, both for verification and execution purposes. Essentially this step consists of converting the meta-information provided by HomePort into a timed automata model as shown in Fig. 2 [14]. The information used for this conversion is the minimum and maximum value that the service state can take, as well as if it represents an actuator or sensor service. In practice, to improve system performance and reduce the model complexity, sensor and actuator models are only added when used in a scenario or constraint specification. Thus, every time a new service is used by a scenario or constraint, its information is retrieved from HomePort. A global variable is then declared, with a domain corresponding to the maximum and minimum values that can be taken by the service. GET and EV channels are also added to the global declarations, as well as a SET channel for actuator ser-

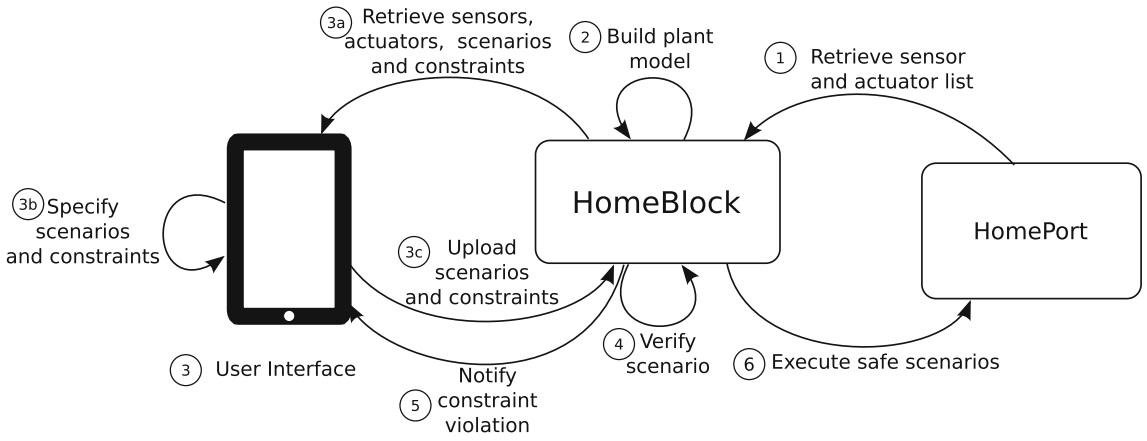


Fig. 8 Overview of the HomeBlock system

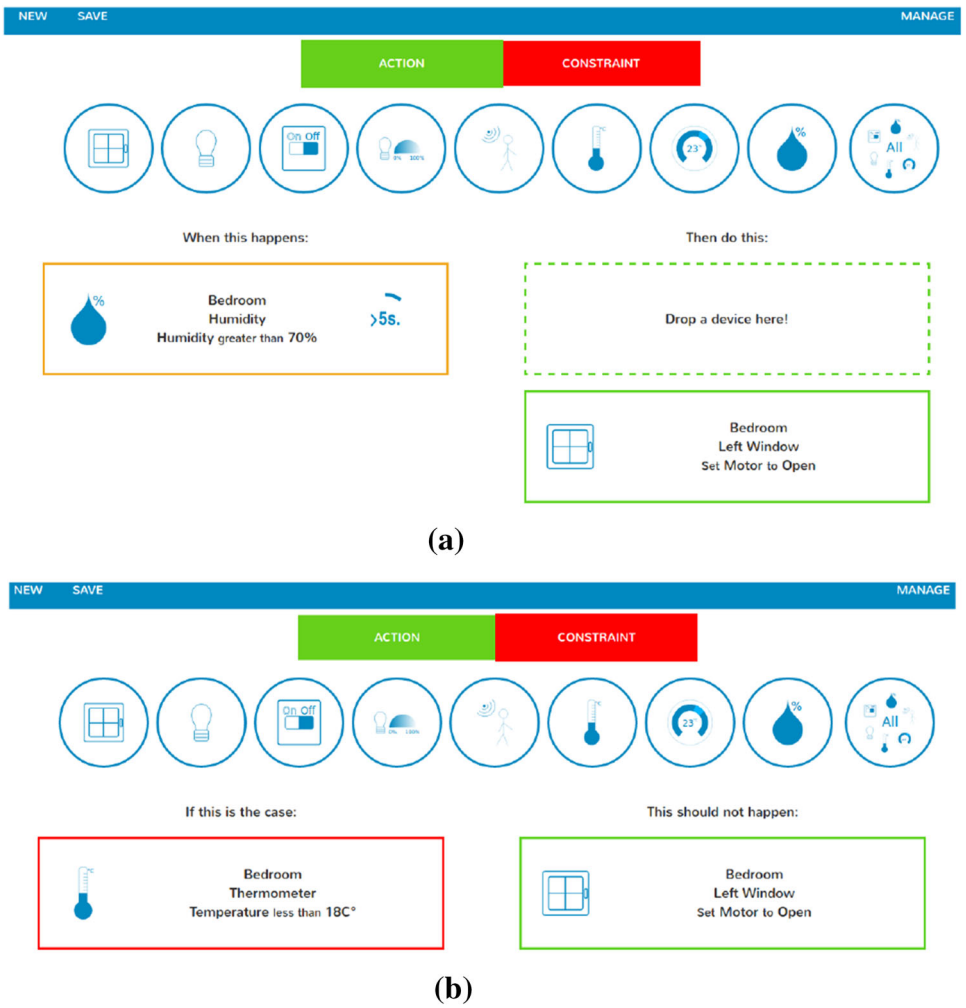


Fig. 9 HomeBlock GUI. **a** Interface to create a control scenario. **b** Interface to create a constraint

vices. Finally, a model of the service is created, as shown in Fig. 2, with an edge for each synchronization channel to enable scenarios to use them.

3.3 User interface

To enable users to specify scenarios and constraints, the HomeBlock system provides a Graphical User Interface (GUI) in a form of a web application, optimized for tablets. It is an improved version of the one presented in [24]. It was redesigned to improve its usability, and errors were fixed to make sure that they do not disturb subjects during experiments. Screen shots of the updated design are provided in Figs. 9 and 10. Figure 9 shows the screen enabling specification of scenarios and constraints, respectively. It is divided into four parts. The first top part consists of a menu with the following choices:

New to reset the display to an empty scenario/constraint (depending on the current screen);
Save to save the current scenario/constraint;
Manage to access to the management window shown in Fig. 10b.

The second part enables switching between scenario specification and constraint specification. Note that in this updated version, it was chosen to refer to control scenarios as *actions*, to try to make the difference with constraints clearer, and reduce user confusion. The third part consists of a set of icons, each representing a specific type of service, that can be dragged and dropped into an empty block. Note that when a sensor service is dragged, the empty action block is grayed out to indicate that it cannot be associated with it. The fourth part is divided into columns, each containing a set of blocks representing an element of a scenario or constraint. When an icon is dropped into an empty block, the modal window shown in Fig. 10a is displayed to the user. It enables configuring the block, choosing the specific device and service to associate the block with, as well as its predicate or control command.

Usage example To understand the use of this user interface, an example scenario is proposed. Let us assume that a smart

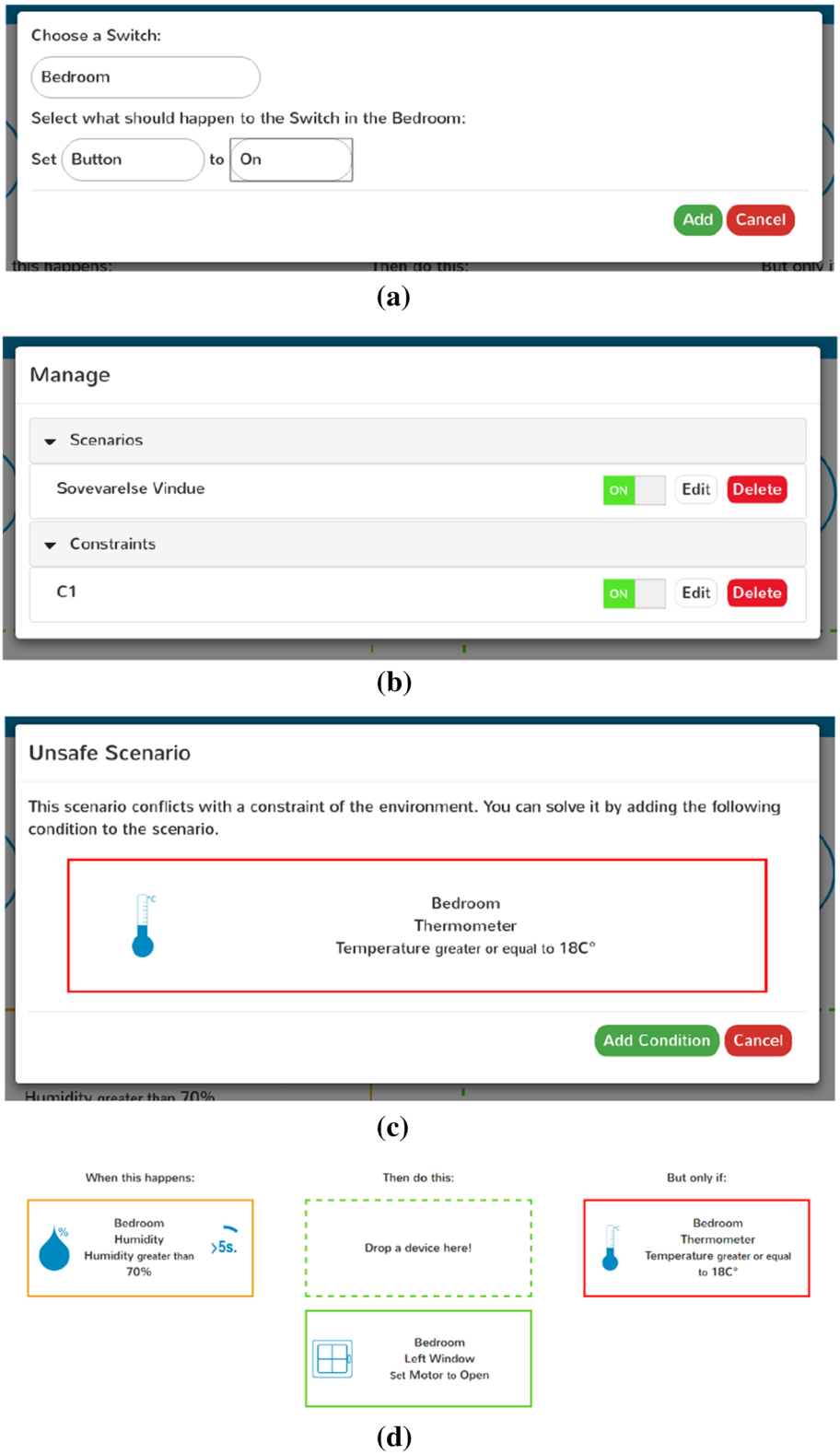


Fig. 10 HomeBlock GUI. **a** Modal window to build a scenario or constraint block. **b** Modal window to access saved scenarios and constraints. **c** Modal window to notify user of an inconsistency between a scenario and a constraint, and proposing a fix for it. **d** Updated scenario after adding the proposed condition

home user is concerned that the system could open the window of the living room located on the ground floor while the room is unoccupied, potentially allowing thieves to enter the house. Using the HomeBlock system, a constraint expressing that "*When* the living room is not occupied, *then do not* open the living room window" could be defined. At a later time, a member of the household might want to define a scenario allowing fresh air in the living room when the temperature becomes greater than 20°C. However, this scenario can potentially violate the previous constraint, since the temperature could increase while the room is unoccupied, leading the system to open the window. HomeBlock will thus inform the user of this potential violation, and propose to add a condition to the scenario specifying that the window should only be open if the room is occupied.

3.3.1 Retrieve sensors, actuators, scenarios and constraints

The display of service icons as well as the management window needs information about available services as well as saved scenarios and constraints. This information is provided to the GUI by the HomeBlock system in JSON format through a simple HTTP API. It ensures the decoupling between the verification engine and the user interface, making the back-end reusable easily with different GUI.

3.3.2 Specify scenarios and constraints

The GUI allows specifying scenarios and constraints by building ECA rules presented in Sect. 2.3. Based on previous experiences, it was decided to disallow users to directly specify conditions inside scenarios. This is because users often confuse between the event and condition parts. Conditions are instead proposed by the system when a scenario under specification violates a constraint, as shown in Fig. 10c (the careful reader will have noticed that the scenario in Fig. 9a is inconsistent with the constraint in Fig. 9b). If the user accepts the condition, a column is added to the right of the actions, containing a list of conditions that restrict the execution of the scenario. The updated scenario is shown in Fig. 10d.

3.3.3 Upload scenarios and constraints

Every time a valid scenario—composed of at least one event and one action—is modified or saved, an updated version is sent to HomeBlock for verifying its consistency against active constraints. The first step taken by the system is to transform the scenario, received in the form of an ECA rule, to TA, as shown in Sect. 2.3. If it contains a service that is not part of the system model yet, it is also added to it as presented in Sect. 3.2. Constraints are uploaded upon saving, and are also transformed to their TA representation as shown in Sect. 2.3. At the moment, running scenarios are not checked against newly added constraints, but it would be a technical matter to enable this function.

3.4 Verify scenario

Once a scenario is transformed into its TA representation, it is verified using the model checking procedure presented in Sect. 2.2. To reduce the verification time, a scenario is only checked against the constraints that it can potentially violate. This means that it is only checked against a constraint that contains a service that is updated by one of its action block. In fact, as a constraint consists of an observer reacting to updates of services, if a scenario does not update the service associated with a constraint, it cannot violate it. This is to reduce the state space of the model to be checked, and has drastically improved the performance of the system compared to the previous version. The performance and limitations of the verification approach will be discussed in Sect. 5.1.

3.5 Notify constraint violation

When an inconsistency is found between an uploaded scenario and a constraint, the system first creates a condition that makes the scenario consistent with the constraint, by negating the constraint condition. Again, as a constraint can only be triggered by observing a certain action under a given condition, by restricting the execution of a scenario to states that do not satisfy the constraint condition, it will not be triggered. The server returns a 409 error code to the upload request, indicating an inconsistency with an already established rule, together with the created condition. This is then used to propose the user to add a condition in the screen shown in Fig. 10c.

3.6 Execute safe scenarios

When a scenario is saved and does not violate any constraint, it is executed by a UPPAAL timed automaton interpreter [14]. An overview of this component is shown in Fig. 11. It essentially performs a live simulation of the system, synchronizing external time (seconds in the current implementation) with the model clocks. For that purpose, a *ticker* model is introduced to the system that simply consists of a clock that is reset after one time unit. Its transition is taken every second to ensure the consistency of time in the model with external time. It also performs the mapping between the actions expressed by the GET, SET and EV channels with the concrete services, through the HomePort middleware. When a scenario does a GET synchronization, the interpreter retrieves the state of the corresponding service and updates it accordingly in the model using the service models. When a SET synchronization is performed, the interpreter updates the state of the corresponding service to the one specified by the scenario.

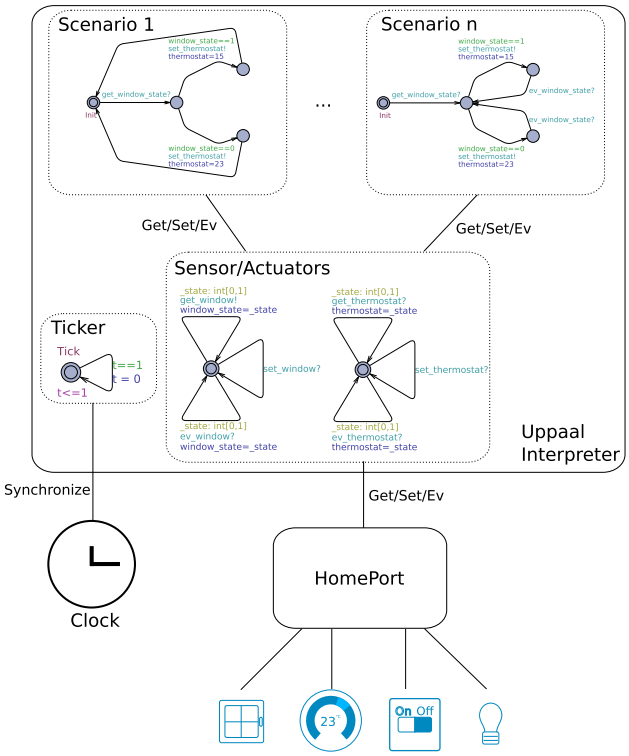


Fig. 11 Overview of the UPPAAL interpreter that executes the scenario models

Finally, when it receives an event from a service through HomePort, it takes the corresponding synchronization action, possibly triggering some scenarios, and updates the state of the corresponding device in the model accordingly, using the service model. Scenarios can be dynamically added and removed from the interpreter. The constraints are not part of the execution as they are only used during the verification process.

4 Evaluation

A lab-based usability study was conducted to evaluate the interaction with the system through the GUI. The primary objective of the study was to understand how novice users with none or little understanding of home automation would embrace the iteration of the system. The think-aloud protocol was used during this evaluation. A pilot evaluation was first conducted to test the tasks and evaluation setup. For the actual evaluation, we recruited seven participants (1 female) between the age of 25 and 44, with mixed educational backgrounds and professions, as shown in Table 1. Six of them lived in apartments while one participant lived in a residential house. One participant had previously interacted with and tried a smart home system, but none of them owned a system themselves. Note that the objective of the evaluation was to obtain qualitative results on the usability of the system through the GUI. Considering this objective, the number of participant is adequate [29].

4.1 Procedure

Each participant took part in the evaluation individually in one session, which lasted approximately 1 h. A session was divided into four parts. First, the participants were briefed about the evaluation and asked to fill out a demographic questionnaire. Second, they were given a short introduction to the system. Third, the actual evaluation was conducted, during which they had to solve assigned tasks (for approximately 45 min). Participants were instructed to think-aloud while solving the tasks and interacting with the system. One of the authors of this paper acted as test monitor and was present in the room during the evaluation. A standard desktop computer with keyboard and mouse was used. Finally, they were debriefed and asked for comments on their experience with the system.

Participants were instructed to solve three tasks:

The first task involved creating three actions (control scenarios) involving changes of temperature and light settings when given events occurred.
The second task was divided into two subtasks where the objective was to create restrictions.

Table 1 Participants demographic

ID	Gender	Age	Profession	Residential status	Owns SH system	Used SH system before
1	Male	25	Business student	Apartment	No	No
2	Female	29	Psychologist	Apartment	No	No
3	Male	28	Sales assistant	Apartment	No	No
4	Male	29	Physiotherapist	Apartment	No	No
5	Male	28	Toolmaker	Apartment	No	No
6	Male	44	CEO at small software company	House	No	Yes
7	Male	26	Medical student	Apartment	No	No

The third task was divided into several subtasks, e.g., one with multiple control actions on different devices, two that violated previously created restrictions, and one that edited a restriction.

During the evaluation, the system was configured to operate in a virtual environment with a predefined set of devices that the participants could interact with just as if the system was installed in a real house. The evaluation was conducted in the participants' native language (Danish). For this reason the GUI was translated. With the objective of the test in mind, the tasks were created to guide participants through the main features of the system.

4.2 Data collection and analysis

The interaction between the participants and the system was recorded on video including what they said and articulated. The analysis was based on observations from the videos, where usability issues experienced by participants were identified. Elements or parts of the system that were found to be interesting or improved in this iteration were also written down. Finally, findings and usability issues were categorized.

4.3 Findings

All participants articulated that the system was interesting and they could easily see themselves using it at home. It was observed that the time for completing the various tasks varied among participants, ranging from a total of about 30 min to almost an hour. All participants did, however, complete all tasks, some of them with help from the test monitor. This is an interesting point, given that they had little knowledge about home automation prior to the evaluation. During the evaluation some general trends were observed.

Differentiating actions and constraints All the participants had difficulties understanding the difference between actions and constraints. Several participants expressed that they thought that when creating a constraint, they expected that the house performed some action to account for it. For example, if the restriction was "the windows should never open if the temperature is below 18° in the bathroom" they expected that the windows would automatically close if it had been opened by an action when it was above 18°. However, this is not the case, as they would need to create a separate action to handle this. Potentially this could lead to undesirable or unexpected behavior of the system.

The specified tasks in our evaluation were deliberately formulated so that it would include either "create the condition" or "create the action". Even so, some participants were unsure if they should create a constraint or an action, perhaps because they did not notice the constraint or action parts in the task formulation. Sometimes participants began to create a constraint even though it was supposed to be an action. But many participants realized after some time what they were expected to do. One of the participants stated that "it is because they look like each other", referring to that both are displayed as boxes with the drag and drop functionality.

Some participants mentioned that they were unsure if an action would override a previously specified constraint, but after trying this a couple of times they became more confident. Some participants proposed to add hierarchies between different constraints and actions. The idea was that some devices could violate a certain constraint, but others could not. For example, if a constraint prevents the specification of an action opening the bathroom window when the temperature is below 17°, a user might want to add an exception to that rule when the humidity is higher than a given threshold.

Trial and error Several of the participants used a trial and error approach in their interaction with the system. In the beginning of the session, they were quite uncertain on the purpose of some GUI components. They interacted with the GUI exploratively to try to find their meaning. Sometimes they dragged devices around the interface to get a better understanding of their function, e.g., "I will just try to drag this here to see what it does". This way of creating actions did seem to help them to solve the tasks. Some of them pointed out that new users might experience a learning curve, which might prevent them from fully utilizing the system. It was, however, anticipated that users might need some time to get accustomed to the interface and the logical way of thinking.

The trial and error approach also seemed to cause frustration in some situations. When a device is dragged to a box, a pop-up appears (as shown in Fig. 10a), where the user should first choose a specific device, that is, if it is the one in the kitchen, bedroom, bathroom, etc. If the device is not chosen, the rest of the drop-down boxes is disabled. All participants failed to see that they had to choose the device first and, therefore, clicked around to see if they could choose some value in the other drop-down boxes. Most participants noticed the relevant drop-down box after a significant amount of time. This could be due to graphical elements needing an improved design.

Representing devices as icons Several participants were occasionally confused about or misunderstood the icons representing device types. Some icons fit what participants expected while others were confusing. The most common mistakes were the window icon interpreted as an oven, the humidity sensor as a water control, and the occupancy sensor as a burglar alarm. However, most participants quickly learned what the icons represented by trying out. A solution to this issue could be to add textual description under the icon to help to identify their meanings.

The "All" icon that was intended to help users when they were in doubt of which device type to chose was not used that

frequently. This icon was intended to represent any type of device in case a user could not identify the one he was looking for in the list of icons. It was, however, misunderstood. Only two of the seven participants used this icon, thinking that it could be used as a wildcard to turn everything off for example. This was relevant during one task where they had to turn off the light in the bathroom when they left it. They quickly learned that this was not the case and proceeded to try out another icon. Some users expressed that it would have been nice with a tool tip or something similar so they would have an idea of what they were controlling. It seems that a way of controlling several devices with one block element could be relevant for further development.

Another common confusion relates to icon similarity. As an example, most participants found it difficult to distinguish between the thermometer and thermostat icons. They knew that both of them had something to do with temperature, but they could not tell the difference. In one case, this resulted in a participant selecting a thermostat instead of a thermometer, which led to a wrongly implemented scenario. The difference is that one measures the temperature (thermometer), while the other sets it (thermostat). One participant stated, "maybe you could divide or group icons depending on what they do", to be able to better distinguish between them.

Identifying violations When participants created actions that violated previously created restrictions, a pop-up window proposes to add a condition to the action, as shown in Fig. 10c. A few participants already knew that they were making an action that would violate a constraint and anticipated the display of the pop-up. However, several were surprised when experiencing this, but after reading the explanation on the screen they all understood its meaning. After adding the proposed condition to the action, it seemed to make more sense to them in the context of the already created and violated constraint.

Reflecting natural behavior and language Many participants were inclined to connect the lights in the system to switches. When solving a task that specified that they should "turn off the light in the hallway when they turned off the switch in the bedroom", they instantly dragged the switch icon to the action box. When asked about it, some of them said that this seemed natural. While the switch can be connected to the hallway light, it can also be connected to other devices. Some participants did, however, successfully connect the light with the switch, obtaining the expected action.

During the evaluation, the participants would normally read out loud the task description when creating an action or a constraint, and sometimes participants realized mistakes that they were about to make. They would choose the perceived appropriate devices and then read the full scenario out loud. During the tests, most participants found some parts of the GUI confusing. Especially when selecting specific devices and their properties, the language used seemed to confuse them. An example was using the presence sensor to detect if someone is in the room, resulting in the text "Presence sensor is present".

5 Discussion and lessons learned

This section discusses the limitations of the system, and the findings of the user evaluation.

5.1 HomeBlock system

Using formal methods to validate scenarios provides both advantages and limitations. A first consideration is on the scalability of the verification task.

Scalability The scalability of the system is mainly impacted by the number and domain size of plant variables used during verification. To determine the limitations, experiments were conducted to evaluate it. The experiments consisted in defining for each run a scenario and a constraint against which it was verified. The constraint used a single variable in its condition and action blocks, while the scenario used a single variable for the event and condition blocks, and an increasing number of variables for the action blocks. A timeout was set to 1 min, considering that the system should be responsive enough to be usable through a GUI. The results of the experiments are shown in Table 2. It shows that as expected the verification time depends both of the number of variables involved in the model and their domain size. For binary variables, the system can verify scenarios involving more than 10 variables. For variables with larger domain, timeouts appear when having between 9 variables for a domain of size 5–4 for a domain of size 50.

In practice, the scalability of the system is acceptable considering the application domain of smart homes. First, many devices and appliances take binary values, such as lights or switches for example. Variables with larger domains often represent plant variables such as temperature or humidity. As illustrated by the results of the experiments, issues can arise when several of them are involved in a single scenario. This can be solved by abstracting the domain into intervals, and reducing the domain to keep only its relevant parts. For example, humidity can be abstracted into four intervals, 0–25 %, 25–50 %, 50–75 % and 75–100 %. This was actually done in the user interface to improve usability, as having to chose between 100 values is not practical. More ad hoc approaches can further enhance the scalability, like for example splitting a scenario with several actions, but were not applied here.

Advantages and limitations The proposed modeling approach and the overall system offer advantages, but has also inherent limitations. On the positive side, the possibility to execute the

Table 2 Scalability experiments

Domain size	Number of variables									
	1	2	3	4	5	6	7	8	9	10
1	221	256	328	445	500	496	439	496	609	615
5	152	213	313	403	461	1182	5906	39699	DNF	DNF
10	251	310	323	571	3462	DNF	DNF	DNF	DNF	DNF
20	196	288	499	5285	DNF	DNF	DNF	DNF	DNF	DNF
50	215	392	5648	DNF	DNF	DNF	DNF	DNF	DNF	DNF

Results are expressed in milliseconds. Timeout was set to 1 min, time after which the verification is considered as DNF (did not finish)

models ensures correct execution of the control defined. The verification capabilities offered by TA is also an advantage, even if it has limited scalability as shown in the previous paragraph. The proposed model checking approach, using observer automata, has the advantage of not requiring any specification of the environment, which is often specific to a smart home, and thus costly to obtain. The system thus could be easily deployed in any smart home. The connection to the HomePort middleware also makes it possible to concretely execute the specified scenarios.

Apart from scalability, the system also has some limitations. If the model checking approach facilitates its applicability to different environments, it is at the cost of not being able to verify properties over the plant variables. Another limitation is the fact that feature interactions are currently not taken into account in the system. This could be done by applying the model checking procedure described in [19]. The ECA language also restricts the expressiveness of specifications. This is, however, intentional as the objective was to reduce the complexity of TA and abstract them to make them understandable and usable by end users. The presented system also considers only single services, without considering complex ones that can be defined as a combination of sensors or actuators. It could, however, be extended to such systems in different ways. A first approach could be to consider such a complex service similarly to simple ones, enabling restriction on the actions that are performed on them. A second approach could be to specify complex services in terms of the simple ones as a timed automata model. Here, however, scalability issues could arise when composing many simple services.

5.2 Evaluation

From the usability study, some general lessons can be drawn. First, even though the design of the GUI was reworked compared to the previous version, there were still many elements that confused the users, and that would need to be improved. This includes many graphical elements, in particular device icons, but also device names and text provided in modal windows. They created disturbances for the users, removing the focus from the actual tasks to perform. A possible solution here could be to conduct two separate evaluations. One could focus on obtaining feedback only on the visual and textual aspects of the GUI to try to reduce such disturbance. Another could then be conducted to evaluate the usability and understanding of the concepts behind it. However, this would require more resources, which were not available.

A successful improvement compared to the previous version was to remove the possibility of manually adding condition blocks. In fact, during the previous experiments reported in [24], users had much trouble understanding their meaning. Only adding them when needed seemed to make their meaning much easier to understand. However, while in the previous experiments users easily understood the difference between the actions and constraints screen, it was not the case here. This might be linked to the fact that by removing the condition column from the default view, the two screens now look very much similar. An improved design might thus solve this issue. A legitimate question is also if the block layout chosen for the interface is the best approach for interacting with users. Using a wizard approach for example could help to reduce its complexity and make the user feel more confident.

A last interesting point is in the expectancy from some users that specifying constraints would trigger some actions from the system to make sure that they are not violated. This is not part of the system functionalities at the moment, but could be added in future versions. An idea could be to use a game theoretic approach, as for example explored in [33].

6 Related work

The research presented in this paper can be compared to several other contributions in the domain of reliable intelligent environments and GUI for smart homes.

Augusto proposed in [5] to use simulation and verification techniques to increase the reliability of intelligent environments. It shows how model checking can be used for that purpose, using the SPIN and UPPAAL model checkers. SPIN is also proposed to be used in a methodology to

model, simulate and verify intelligent environments in [6]. This methodology includes different modeling steps to represent different aspects of the system, and details on how to apply it. Another approach to formal modeling and verification of intelligent environments is proposed by Benghazi et al. [10]. They use a timed version of Communicating Sequential Processes [20] as modeling formalism to represent system behavior. The modeling approach and formalism used in these papers are essentially similar to those used here. However, here the objective is to abstract the complexity of the formalism, to automate the verification process and make it usable by end users.

Another application of model checking to intelligent environment was proposed by Liu et al. [27]. They propose a modeling framework to help detect errors in the development of such environments. If the application of formal methods can be compared to the work presented of this paper, the HomeBlock system is targeted at end users, and aims at abstracting the complexity of formal models. The developed models are, moreover, executable and the model checking is performed in a running smart home setting.

The formal mirror models proposed by Loke et al. [28] propose an online validation approach for intelligent environment scenarios. Devices, their effects on the environment as well as scenarios are represented as Petri-Nets to validate the correctness of the behavior obtained by their composition. The approach is essentially similar to the one proposed here on the modeling and verification part. However, we have also focused on abstracting the complexity of the formalism through a GUI, and tested the applicability and usability of the concept with an experiment. The integration with the existing HomePort middleware, as well as with the UPPAAL model checker, verifying and executing scenarios also makes our approach more concrete. Note, however, that contrary to them, we do not consider environment effects in this paper.

The use of ECA rules in the context of intelligent environments was pioneered by Augusto and Nugent in [7]. As already mentioned, this work was used as an inspiration to develop the abstraction layer to the timed automata models. García-Herranz et al. [17] also used ECA rules, together with a context aware middleware to enable specification of intelligent environment scenarios. An interesting feature that they used was wildcards that enable specifying scenario blocks over set of similar device types rather than single ones. Timing is introduced in the rules through the use of timers that can be attached to each block element. They also developed a GUI, but do not provide any user experiment on it, neither on the use of time in scenarios. A more technical use of ECA rules is proposed in [31]. They provide a very expressive set of rules, defining their semantics in terms of Metric Temporal Logic [4]. A GUI enables users to specify scenarios using these rules. Here, the use of timed automata makes it easy to perform formal verification of scenarios using the UPPAAL tool, and the objective was to provide a simple set of rules rather than an expressive one to improve usability and understanding by non-expert users.

There has also been a tremendous amount of work on developing GUI enabling users to specify smart home scenarios that inspired the development of the one presented in this paper. In particular, we mention Lee et al. GALLAG smartphone application [25], and Microsoft HomeMaestro [21]. Both are based on "If-Then" rules, which correspond to an event and an action. The interest of these approaches is that they are tangible, meaning that rules can be created by interacting with devices. This could be an interesting feature to add to the presented interface.

An evaluation of simple and complex event action rules (called trigger action by the authors) was conducted in [35]. They show that in general these rules are expressive enough to build realistic scenarios in smart home, and that users are in general able to learn quickly to build complex scenarios.

A noticeable work of translating ECA rules to timed automata models has been done by Ericsson [16]. The timed automata semantics is used as a base for the development of the Rule and Events eXplorer (REX) tool, also using UPPAAL as a model checker for verifying properties. The difference in the presented work is that the models developed for HomeBlock aimed at being executed in a smart home and thus include further constructs.

Note that a main contribution in this paper is to enable specification of safety requirements in the system by users, which to our knowledge was not considered previously. This can help to improve the reliability of such rule-based systems, and help users defining safe and secure scenarios. In addition, compared to several researches on formal methods for intelligent environment, a concrete system is proposed that has the potential to be used at runtime and not only during the design phase of the system. Finally, the verification approach using observer automata also makes the approach more generic as it does not require specific information about the smart home environment, even though it restricts the verification capabilities.

7 Conclusion and future work

This paper introduced a formal framework based on timed automata for modeling smart home environments and scenarios that can be formally verified against formal requirements representing constraints on possible control commands sent to the system. Among other features, it enables automatic extraction of device models, automated verification of scenarios as well as their execution. An abstraction of this framework in the form of ECA rules was then introduced, to reduce its complexity and improve its usability. The translation from the ECA language to timed automata was detailed,

which is the core of the abstraction. A concrete implementation of this framework was then introduced, to show its feasibility and applicability. As part of this implementation, a GUI enabling specification of scenarios and constraints was presented. It was used in a qualitative user study that aimed at evaluating the usability of the approach and the GUI itself, providing useful results for future research in rule-based control of smart home and associated user interfaces.

Several directions for future work are considered. First, the formal framework will be improved. In this paper, the environment is abstracted through sensors and actuators, ignoring the dynamics of its variables. This makes it impossible to verify liveness properties, e.g., that desired behavior will eventually be observed given a given set of scenarios. This is a useful thing to do to reduce complexity of the models, and thus their state space, rendering the model checking problem highly tractable. It was also in a desire to focus on safety properties that we thought of higher importance for users, and to provide a general framework applicable to any smart home. To add environment variables, but maintain reasonable model complexity, a possible approach could be to use Statistical Model Checking [26]. The concrete implementation of the system will also be worked upon, trying to develop new features useful to the end users. The GUI will also be updated based on the results obtained through the user experiment. The objective is in particular to show the applicability and usability of formal methods in intelligent environment, using abstraction layers to reduce their complexity.

Acknowledgments The work presented in this paper is supported by the European Artemis project Arrowhead and the Danish ForskEL project TotalFlex. The authors would also like to thank Professor Anders P. Ravn for his comments on early versions of this work.

References

1. Aceto L, Ingólfsdóttir A, Larsen KG, Srba J (2007) Reactive systems: modelling, specification and verification. Cambridge University Press, New York
2. Alexander C (1964) Notes on the synthesis of form. Harvard University Press, Harvard
3. Alur R, Dill DL (1994) A theory of timed automata. Theor Comput Sci 126(2):183–235
4. Alur R, Feder T, Henzinger TA (1996) The benefits of relaxing punctuality. J ACM 43(1):116–146
5. Augusto JC (2009) Increasing reliability in the development of intelligent environments. In: Proceedings of the 5th international conference on intelligent environments, IOS Press, pp 134–141
6. Augusto JC, Hornos MJ (2013) Software simulation and verification to increase the reliability of intelligent environments. Adv Eng Softw 58:18–34
7. Augusto JC, Nugent CD (2004) The use of temporal reasoning and management of complex events in smart homes. In: ECAI, Citeseer, vol 16, p 778
8. Behrmann G, David A, Larsen K (2004) A tutorial on Uppaal. In: Bernardo M, Corradini F (eds) Formal methods for the design of real-time systems, lecture notes in computer science, vol 3185. Springer, Berlin, pp 200–236
9. Belarbi M, Babau JP, Schwarz JJ (2004) Temporal verification of real-time multitasking application properties based on communicating timed automata. In: Eighth IEEE international symposium on distributed simulation and real-time applications, 2004. DS-RT 2004, pp 188–195
10. Benghazi K, Hurtado MV, Hornos MJ, Rodríguez ML, Rodríguez-Domínguez C, Pelegrina AB, Rodríguez-Fórtiz MJ (2012) Enabling correct design and formal analysis of ambient assisted living systems. J Syst Softw 85(3):498–510
11. Brush AB, Lee B, Mahajan R, Agarwal S, Saroiu S, Dixon C (2011) Home automation in the wild: challenges and opportunities. In: Proceedings of the SIGCHI conference on human factors in computing systems, ACM, New York, NY, USA, CHI '11, pp 2115–2124
12. Corno F, Sanaullah M (2013) Design-time formal verification for smart environments: an exploratory perspective. J Ambient Intell Humaniz Comput 5(4):581–599
13. Coronato A, Pietro GD (2010) Formal design of ambient intelligence applications. Computer 43:60–68
14. Dalsgaard P, Le Guilly T, Middelhede D, Olsen P, Pedersen T, Ravn A, Skou A (2013) A toolchain for home automation controller development. In: 2013 39th EUROMICRO conference on software engineering and advanced applications (SEAA), pp 122–129
15. Davidoff S, Lee M, Yiu C, Zimmerman J, Dey A (2006) Principles of smart home control. In: UbiComp 2006: ubiquitous computing, lecture notes in computer science, pp 19–34
16. Ericsson A (2009) Enabling tool support for formal analysis of eca rules. PhD thesis, Linkping UniversityLinkping University, Department of Computer and Information Science, The Institute of Technology
17. García-Herranz M, Haya PA, Alamán X (2010) Towards a ubiquitous end-user programming system for smart spaces. J UCS 16(12):1633–1649
18. Gruhn V, Laue R (2005) Specification patterns for time-related properties. In: 12th international symposium on temporal representation and reasoning, 2005. TIME 2005, pp 189–191
19. Guilly T, Olsen P, Pedersen T, Ravn AP, Skou A (2016) Software technologies: 10th international joint conference, ICSOFT 2015, Colmar, France, July 20–22, 2015, Revised Selected Papers, Springer International Publishing, Cham, chap Model Checking Feature Interactions, pp 307–325
20. Hoare CAR (1978) Communicating sequential processes. Commun ACM 21(8):666–677
21. Karagiannis T, Athanasopoulos E, Gkantsidis C, Key P (2008) Homemaestro: order from chaos in home networks. Tech. rep, Microsoft Research
22. Le Guilly T, Olsen P, Ravn A, Rosenkilde J, Skou A (2013) HomePort: middleware for heterogeneous home automation networks. In: 2013 IEEE international conference on pervasive computing and communications workshops (PERCOM workshops), pp 627–633
23. Le Guilly T, Olsen P, Ravn AP, Skou A (2015a) Modelling and analysis of component faults and reliability. In: Petre L, Sekerinski E (eds) From action system to distributed systems: the refinement approach. Taylor & Francis, Abingdon
24. Le Guilly T, Smedegard JH, Pedersen T, Skou A (2015b) To do and not to do: constrained scenarios for safe smart house. In: 2015 international conference on intelligent environments (IE), pp 17–24
25. Lee J, Garduño L, Walker E, Burleson W (2013) A tangible programming tool for creation of contextaware applications. In: Proceedings of the 2013 ACM international joint conference on pervasive and ubiquitous computing, UbiComp'13, pp 391–400
26. Legay A, Delahaye B, Bensalem S (2010) Statistical model checking: an overview. In: Barringer H, Falcone Y, Finkbeiner B,

Havelund K, Lee I, Pace G, Rou G, Sokolsky O, Tillmann N (eds) Runtime verification, lecture notes in computer science, vol 6418. Springer, Berlin Heidelberg, pp 122–135
27. Liu Y, Zhang X, Dong JS, Liu Y, Sun J, Biswas J, Mokhtari M (2012) Formal analysis of pervasive computing systems. In: 2012 17th international conference on engineering of complex computer systems (ICECCS), pp 169–178
28. Loe SW, Smanchat S, Ling S, Indrawan M (2008) Formal mirror models: an approach to just-in-time reasoning for device ecologies. Int J Smart Home 2(1):15–32
29. Nielsen J (1993) Usability engineering. Morgan Kaufmann Publishers Inc., San Francisco
30. Pedersen T, Le Guilly T, Ravn A, Skou A (2015) A method for model checking feature interactions. In: Proceedings of the 10th international conference on software engineering and applications, pp 219–228
31. Qiao Y, Wang H, Zhong K, Li X (2006) Visual event-condition-action rules with temporal events. In: Eighth real-time linux workshop, p 275
32. Rogers Y (2006) Moving on from Weisers vision of calm computing: engaging ubicomp experiences. In: Dourish P, Friday A (eds) UbiComp 2006: ubiquitous computing, lecture notes in computer science, vol 4206. Springer, Berlin, pp 404–421
33. Sørensen MG (2014) Controller synthesis for home automation. Master's thesis, Department of Computer Science, Aalborg University
34. Thums A, Schellhorn G (2003) FME 2003: Formal methods: international symposium of formal methods Europe, Pisa, Italy, September 8–14, 2003. Proceedings, Springer, Berlin, Heidelberg, chap Model Checking FTA, pp 739–757
35. Ur B, McManus E, Pak Yong Ho M, Littman ML (2014) Practical trigger-action programming in the smart home. In: Proceedings of the SIGCHI conference on human factors in computing systems, ACM, New York, NY, USA, CHI '14, pp 803–812
36. Valiente-Rocha P, Lozano-Tello A (2010) Ontology and SWRL-based learning model for home automation controlling. In: Ambient intelligence and future trends-international symposium on ambient intelligence (ISAm I 2010), advances in intelligent and soft computing, vol 72, pp 79–86
37. Weiser M (1991) The computer for the 21st century. Sci Am 265(3):94–104