To Do and Not To Do: Constrained Scenarios for Safe Smart House

Thibaut Le Guilly*, Jacob H. Smedegård, Thomas Pedersen*, Arne Skou*
Department of Computer Science
Aalborg University
Aalborg, Denmark
*{thibaut,tp,ask}@cs.aau.dk

***Abstract*—A smart house is a complex system, and configuring it to act as desired is difficult and error prone. In this paper we extend a previously developed framework based on timed automata for designing safe and reliable home automation scenarios to make it easier to use. To do so we abstract it with an Event-Condition-Action language to create intelligent scenarios, and constraints that prevent scenarios with undesirable behaviors to be applied. This language is itself abstracted by a graphical user interface that enables the creation of scenarios by manipulating graphical blocks representing elements of the language. We have designed and implemented a prototype system to test our approach, and we report on a qualitative user study that was conducted.**

I. Introduction

The technical complexity of intelligent environments leads to two main issues: reliability and usability. Weiser mentions at the end of his seminal paper [1] that the machine should fit the human environment. In order to fit, it is essential that such systems behave predictable and according to their specifications. This has led to a growing interest in the application of formal software engineering methods to ubiquitous environments. The use of modeling tools such as the Spin model checker [2], or the ambient calculus and logic [3] to support the development of these environments, are among the many recent examples [4] of this interest. Following a similar desire to increase reliability, we introduced in [5] a toolchain to facilitate the development of home automation controllers modeled as timed automaton, and enable their execution. This toolchain enables the precise specification and verification of home automation systems, ultimately increasing the trust in complex intelligent systems. However, hiding the complexity of such systems is essential to make them accessible to users.

Designing a specification for a home automation controller is a task that requires the active participation of the end-user over time and is not a single up-front investment. Users do not know exactly what to expect from a home automation system before acquiring it [6] and preconceived ideas about automation scenarios often do not hold once in actual use. Davidoff et al. [7] highlight the dynamic nature of activities in a regular American household. They show that these activities are a complex mix of interactions between people, that require varying equipments and devices, and can easily break down when pieces of the puzzle are missing. They further stress the need for understanding periodic changes, exceptions and improvisation.

From these observations, we draw that a home automation system must allow for dynamic changes of its configuration, and enable the users to create, modify and mix scenarios to fit with the current activity. Here a possibility is to use non-intrusive systems that learn from inhabitants behavior and adapt to their actions [8]. However, such systems can lead to uncertainty for the user as to whether he or the system is in control. Here we prefer Rogers' [9] vision of engaged living, where *"technology is designed to enable people to do what they want"*. We thus want a system that enables the user to provide a full specification of the system over time through experimentation and discovery. However, it is not always easy for the user to decide which scenarios would fit his needs and his environment. Moreover, some scenarios, if wrongly specified, can easily lead to an undesired configuration of the house, and thereby to frustrated users. Here, the analogy with the notion of *fit* from Alexander [10] teaches us that it is often easier to decide what does not fit an environment, or what is undesirable. Therefore, enabling the user to constrain the control of the environment with a list of undesired behaviors makes it possible for him to safely experiment and discover. Here, formal methods, and Timed Automata (TA) [11] in particular, provide a good framework for specifying and verifying scenarios against constraints. We recall our previous work on modeling and verifying home automation controllers with TA in Section II. While TA are useful for performing model checking and verifying models against requirements, they are too complex to be handled by regular users and need to be abstracted away.

The first step in this abstraction is to use a higher level language to define the scenarios. We borrow from Augusto and Nugent [12] the use of a temporal Event-Condition-Action language to model such scenarios. Here we propose to reuse this language to define constraints which represent actions that the system should not perform given certain conditions. Such a language can easily be translated to TA, which makes it possible to verify that the scenarios created by the user do not violate the constraints. We discuss this in Section III. However, this language still does not provide an easy way to model the scenarios for users. To solve this issue, we propose a user interface based on the notion of blocks that abstract this language. We present this interface and its associated backend in Section IV, and a qualitative experiment we conducted with it in Section V.

We note that there has been much research on automating smart houses, based on scenarios, user interfaces or machine learning. We thus review the most related to our approach in

Section VII. Finally, the work presented in this paper opens up to different opportunities for future work that we present in Section VI before concluding in Section VIII.

Our contributions in this paper is summarized as follow:

- we provide a semantics for an Event-Condition-Action (ECA) language for smart houses in the form of TA, making it possible to verify the scenarios against undesired states of the house;
- we propose to reuse the ECA language to model undesired configuration of the house;
- we have implemented a working prototype system and an associated user interface;
- we report on findings from a user study relevant for the construction of GUI on top of an ECA language, the modeling of scenario by users and the use of constraints on the scenarios.

II. Background

The system presented in this paper is built upon a toolchain that facilitates the design, verification and execution of home automation controllers modeled as TA. For a better understanding, we recall the basics of this toolchain and refer the reader to the paper presenting it [5] and the UPPAAL tutorial [13] for more information. As depicted in Figure 1, it is composed of four components. The first one, Homeport [14], is a REST oriented middleware that provides a common interface to devices belonging to different networks. It is used in the toolchain as the interface to access and control the services provided by the home automation system. The second main component is UPPAAL [13], a toolbox for modeling, simulating and verifying systems modeled as networks of TA. In the toolchain, UPPAAL is used for the creation and validation of the controllers, and its simulator is used as part of their execution. The Uppaal GUI allows for creation of TA and presents an interface to the simulation and verification engine of Uppaal. The two remaining components create the link between the services provided by the home automation system through HomePort and the UPPAAL models. The HomePort2Uppaal tool is used to create a UPPAAL model for the services, which controllers can interact with to get or set their states. An example of such a model can be seen in Figure 2. Finally, the UppaalInterpreter is the one executing the controllers. It takes as input a UPPAAL system, runs it through the UPPAAL simulator, and chooses the appropriate transitions in the models that correspond to the actual state of the environment. This toolchain thus supports the design, verification and execution of controllers in a smart house. The use of TA provides the possibility to verify that the control models satisfy desired properties. However, the use of the UPPAAL interface for the design makes it difficult to use by non expert users. A more accessible interface, abstracting the complexity of the TA, is thus necessary to attract a wider range of users. A first step in abstracting the complexity is to think about scenarios in the house rather than controllers.

III. Home Automation Scenario

In order to define the notion of scenarios in a home automation setting, we first need to define the environment upon which scenarios will be defined.

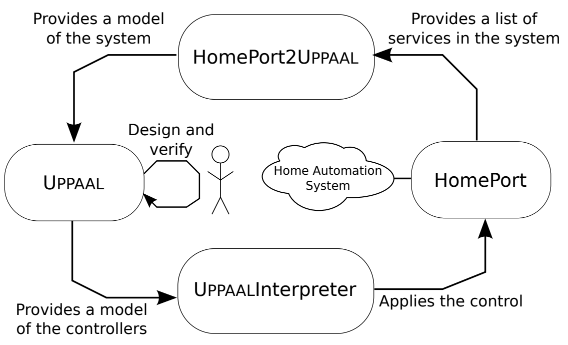

Provides a model of the system
HomePort2UPPAAL
Provides a list of services in the system
Design and verify
UPPAAL
Home Automation System
HomePort
UPPAALInterpreter
Provides a model of the controllers
Applies the control

Fig. 1. Overview of the toolchain components

A. Home Automation Environment

A home automation environment is composed of elements divided into three categories. The first category is sensors, which provide observations on environment variables, such as temperature, movement, or luminosity. They report the values of these variables, and their changes. The second category is actuators, which provide control over the environment. They can be set to different states and report on their current state. Finally the last category is the one of controllers, coordinating the actions of actuators based on the observations provided by sensors and the state of actuators. As we will see, scenarios are part of the controllers category.

B. Rule Based Scenarios

A scenario is a controller, modifying the state of the system when observing specific changes in the environment. We can add to this definition that some scenarios will only modify the system when in a specific state. This can be easily expressed using ECA rules, that were first introduced for active databases management [15] and later used in Smart Homes context [12]. This early work defines the event part as *"The event that triggers the rule (i.e. causes* the system *to evaluate the rule's condition"*. Applied to home automation, an event is a modification of the observable state of the environment, therefore a change in a measurement by a sensor or the modification of the state of an actuator. The condition is then defined as *"A collection of queries that are evaluated when a rule is triggered by its event"*. In home automation, the queries that compose the conditions perform relation checks on the current observable state of the environment, a measurement reported by a sensor or the state of an actuator at the time the query is evaluated. Finally an action is "what *is executed when the rule is triggered and its condition is satisfied"*. In home automation, that corresponds to the modification of the state of one or more actuators, for example opening a window or turning off a light.

The most complex part of ECA rules is the event part. It can be primitive (a single event) or composite (a sequence of events). It becomes even more complex when adding temporal relations between the events, giving the possibility to define many types of temporal events [16]. As stressed in [17], this complexity needs to be hidden or at least reduced in the user interface to be manageable by non expert users. To do so, we limit ourselves to single events, and single events with duration. It enables us to reduce the complexity but still keeping acceptable expressiveness. In fact, if complex events are interesting to model advanced scenarios, they are difficult to represent, and therefore even more difficult to define for non expert users. Expressing them in the form of TA would however be possible for future studies.

a) Single Event: This is the most simple type of event. It is composed of only one event, and does not embed any temporal information. It can be used to define simple scenarios such as *"when I press a button, turn on a lamp"*.

b) Single Event with Duration: In the language defined in [12], a scenario such as *"when I press a button for more than ten seconds"* needs to be expressed as an event *"press a button"* and a condition expressing the absence of an event *"stopped pressing a button"* for ten seconds. In order to simplify scenario definitions, we propose the notion of a single event with duration to represent a service that changes state and keeps it for a minimum period of time. It enables modeling such scenarios with a reduced complexity and in a more natural way. Note that the TA representation of this language element reuses the definition of [12], as we will see later.

C. ECA Semantics

To ensure the correct execution of scenarios, and enable the use of the underlying toolchain, we provide a semantics for each type of language elements using TA. To facilitate the understanding of the models, we take as an example a lamp. For a more detailed explanation of this model, we refer the reader to [5]. The model of this lamp is shown in Figure 2. In this example, the state of the lamp is modeled by a variable, `lamp_state`. It has two states, 0 (off) and 1 (on). To enable controllers to access and modify its state, the model of the lamp has three channels. In UPPAAL, channels are used to synchronize and pass values between processes. When a process (in particular here a scenario) wants to get the state of the lamp, it synchronizes with the lamp process on the `get_lamp_state` channel. During the synchronization, the temporary variable `_lamp_state` is assigned the value of the current lamp state, fetched from the physical lamp device. This value is then stored in the global variable `lamp_state` for the synchronizing process to access it. The "!" symbol means that the lamp is the sender and the "?" symbol that it is the receiver. The `set_lamp_state` channel is used when a process wants to update the state of the lamp. Again value passing is used to transfer the value through the variable `lamp_state` and notify the lamp model. Finally, the `ev_lamp_state` channel is used to notify interested processes when the state of the lamp changes. It is similar to the get channel, except that it is broadcast, meaning that several processes can receive an event notification.

Figure 3 shows the assigned semantics of the different language elements described in Section III-B. The model of a single event, Figure 3a, waits for an event notification from the lamp and checks whether the value corresponds to the one specified. This block expresses the event *"when the lamp is turned on"*. The other type of event, the simple event with duration, Figure 3b, is composed of a single event, followed by a location that checks that the state does not change for a specified amount of time. The depicted example expresses the event *"when I turn on a lamp for more than 15 time units"*. Note that the variable c represents a clock, whose value

```
int [0,1] lamp_state;
chan get_lamp_state;
chan set_lamp_state;
chan broadcast ev_lamp_state;
```

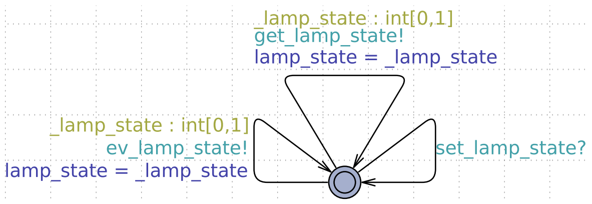
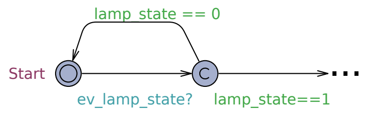
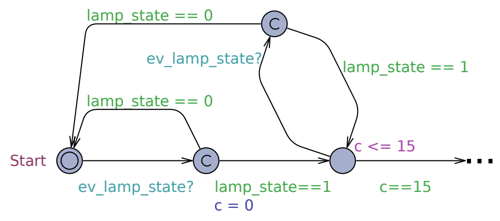


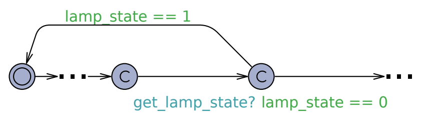
Fig. 2. Model of a lamp. The upper part is the declaration of the variables representing its state, and the three channels that can be used by controllers to get, set or receive events on its state. The automaton represent the transitions corresponding to these three actions.



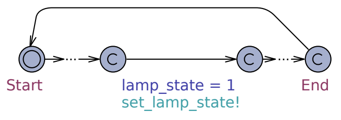
(a) Simple Event



(b) Simple Event with Duration. Note that c is a clock. Its value evolves with time passing.



(c) Condition



(d) Action

Fig. 3. Models of the Blocks. Note that the three dots indicates that other blocks can be inserted there.

increases with time passing, reset at the moment the event is observed. This clock is used to check whether an event on the state of the service of interest is observed within a time interval, in this example 15 time units. A condition, Figure 3c, gets the state of a service and checks that it has the desired value. This example expresses the condition *"the lamp is off"*. An action, Figure 3d, sets the state of the service to a specified value. In this case, *"turn on the lamp"*. Note that conditions are optional, that both conditions and actions can be combined in sequences and that the model always return to its initial location, so that it can be triggered again.

D. Scenario Constraints

In order to leverage the model checking possibilities of the TA models, we need to have requirements to check them

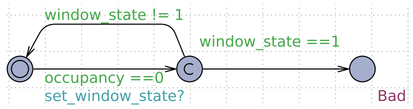


Fig. 4. An example of a scenario constraint.

against. Here we propose these requirements to be a set of constraints set by the user which represent actions that should not be performed by the system when the house is in a particular state. A simple example here is *"if the room is not occupied, do not open the window"*. Therefore a constraint is composed of a condition, that represents a specific state of the house (in our example the house being empty), and an action that *should not* be performed by the system (in our example opening the window). If the user later on defines a scenario such as *"when the humidity goes above 50%, then open the window"*, the system will detect that the previous constraint is violated (the humidity can increase while the house is empty) and warn the user of the conflict. As we will see later, the system is capable of generating a condition that makes the scenario acceptable, and suggest it to the user. In this case transforming the scenario into *"when humidity goes above 50%, if the house is not empty, then open the window"*.

In order to model constraints, we use observer automata, that observe if the undesired behavior can occur in the system, given the set of scenarios. An example of such a constraint model is shown in Figure 4. Here the first edge listens for a transition updating the state of the window (a `set` transition) while the house is not occupied. If this update led to the window being open, the observer moves to the *Bad* location, otherwise it goes back to its initial state. By checking the reachability of the location labeled as *Bad*, using the CTL formula $A\Box\neg Constraint.Bad$, we can verify that no scenario can perform the undesired action. Note that it would be easy to model more complex constraints, involving several conditions and/or several actions, or even temporal relations between those. However, we restrict ourselves to simple constraints to evaluate if they can be understood by users. We now move on to discuss the implemented system.

IV. HomeBlock

The HomeBlock system is composed of two parts, a backend and a GUI.

A. HomeBlock Backend

The backend is a modified version of the toolchain that was previously developed, which is used to verify scenarios against constraints, gather information about the environment and execute the control. The main modification consists of the inclusion of a web server. This web server provides services to get, add, remove scenarios and constraints on the fly in ECA form and transform them into TAs to be verified and executed. Secondly, a Graphical User Interface (GUI) that facilitates the creation and management of scenarios and constraints. The interaction flow between the GUI and the backend is depicted in Figure 5. The backend first provides the interface with the set of available devices and their services in the environment, as well as the already existing scenarios and constraints. The user then creates or modifies a scenario or a constraint and

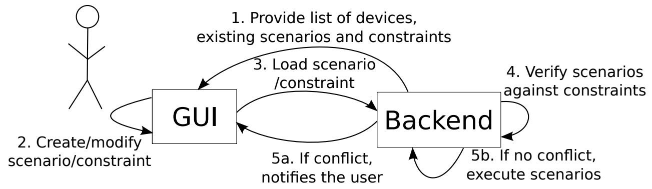

1. Provide list of devices, existing scenarios and constraints
3. Load scenario /constraint
4. Verify scenarios against constraints
GUI
Backend
2. Create/modify scenario/constraint
5a. If conflict, notifies the user
5b. If no conflict, execute scenarios

Fig. 5. Interaction flow between the backend and the GUI

saves it, sending it to the backend. The backend then checks for possible conflicts between scenarios and constraints, e.g. whether the *bad location* of a constraint is reachable. If this is the case, the backend proposes the user the appropriate condition for solving the conflict. If no conflict is found, the backend (re-)executes the scenarios.

However, in order to verify the absence of conflicts, we need the scenarios and constraint specifications. As writing scenarios based on the ECA language is still complicated for the average non-programmer, we turn to visual programming that fits well with the concept of ECA.

B. HomeBlock GUI

The HomeBlock GUI consists of a visual programming tool for scenario creation. It is developed using HTML5 technologies making it usable on any device with a web browser[1]. Our focus is to foster experimentation through rapid iteration and in-situ programming. We abstract the underlying ECA language into a simple drag and drop block-based programming language. The design is focused on minimizing the mental load on users and allow them to select, order and experiment as they please when designing. The main screen of the application, used to create scenarios, is composed of three parts, as shown in Figure 6. The top part consists of a menu to access the functionalities of the application and set a name for the scenario. The second part consists of a list of device categories, each represented by an icon showing a graphical representation of these categories. In fact, a modern household with smart devices can contain a large number of devices, which makes a full list of concrete devices impractical. The last icon, the question mark, act as a wildcard for the entire set of devices, in case the user is not sure of which category the device he is looking for belongs to. Finally, the last part consists of the graphical representation of the scenario itself. This last part is itself divided in three parts, one for each element of the ECA language. The user can drag and drop device icons into one of the empty blocks to instantiate a new event, condition or action in the scenario. When dropping the device, the user is prompted with the popup window shown in Figure 7. Here he can select in order, the desired device (limited to the devices corresponding to the type that was dropped), the specific service in the device, a logical operator (is, greater than, less than, ..., only if the service can take more than two values) and a value. When creating an event, he can optionally choose a minimum duration for which the event should hold (thus creating a simple event with duration). The block can then be added to the scenario, and appears as *instantiated* in the scenario view, as shown in Figure 8.

[1]The source code of the interface is available at: https://github.com/Tibo-lg/HomePortInterface

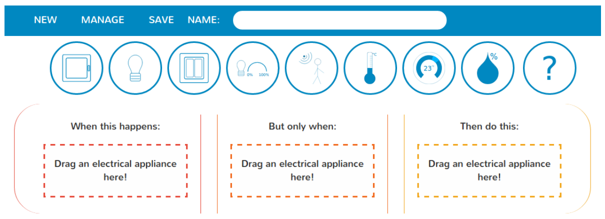
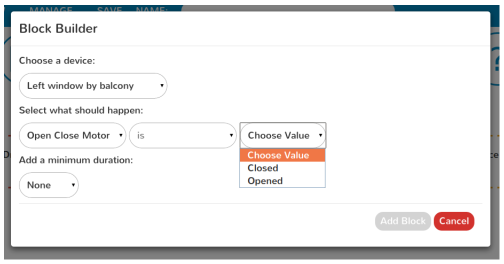

NEW
MANAGE
SAVE
NAME:
?
When this happens:
But only when:
Then do this:
Drag an electrical appliance here!
Drag an electrical appliance here!
Drag an electrical appliance here!

Fig. 6. Main screen of the interface used to build scenarios. The menu enables the user to navigate between the different functionalities of the applications. The device type icons can be drag and dropped to create new blocks.


Block Builder

Choose a device: Left window by balcony

Select what should happen: Open Close Motor · is · Choose Value (Choose Value / Closed / Opened)

Add a minimum duration: None

Fig. 7. Popup window used to build blocks. Here this window is the one used to create events. Windows for creating conditions and actions differ slightly.

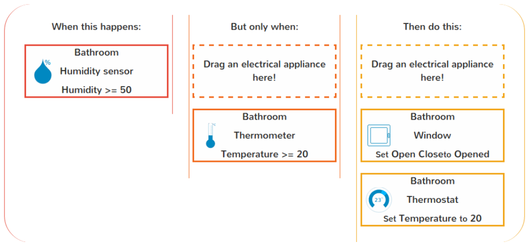


Fig. 8. An example of a completed scenario.

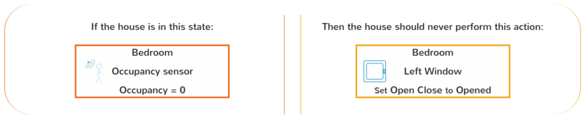
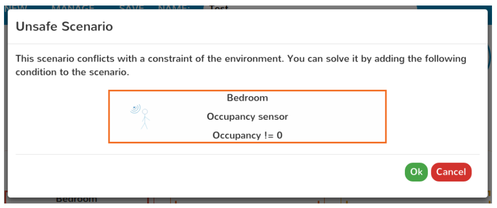

If the house is in this state:
Bedroom
Occupancy sensor
Occupancy = 0
Then the house should never perform this action:
Bedroom
Left Window
Set Open Close to Opened

Fig. 9. Example of a created constraint.

Creating constraints is done in a similar manner, except that here only a condition and *undesired* action under that condition are specified. An example of this screen is shown in Figure 9. When the user tries to upload a scenario that violates a constraint, the popup window showed in Figure 10 appears, proposing him to add a condition that will make the scenario acceptable. In this example, the user tried to create a scenario opening the bedroom window when the temperature increases above 22°C. This violates the constraint of Figure 9. Therefore the system proposes to add the condition "if the room is not empty" to make this scenario acceptable.

The last point we mention about this GUI is the possibility of using templates. When a single scenario can be applicable to several rooms or set of devices in the house, re-programming it for each of them is time consuming and error-prone. Saving templates enables the re-use of scenarios, potentially even to share scenarios between users, as it is the case for the HomeMaestro system [18]. An example of a template can


Unsafe Scenario

This scenario conflicts with a constraint of the environment. You can solve it by adding the following condition to the scenario.

Fig. 10. Example of the popup window warning the user of a conflict and proposing him to add a condition to the scenario to solve it.

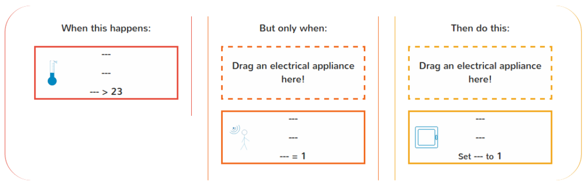


Fig. 11. An example of a template to be filled in by the user.

be seen in Figure 11. This template models a scenario which opens the window of a room when the temperature gets higher than 23°C and the room is occupied.

The interface also enables the user to manage scenarios and constraints, by easily activating or deactivating them, editing them or deleting them.

V. Experiments

This section describes a small user study conducted to gauge the reception of the interface in users with no prior experience with home automation. Our study is exploratory with a focus on understanding the users mental model when tasked with designing automation rules.

A. Design

We performed semi-structured interviews with 5 participants of varying ages and professions listed in Table I. Each session was 30-45 minutes during which we asked them to solve 4 tasks using our prototype interface. Each session was recorded, transcribed and analyzed for common patterns. Each task consisted of specifying a small automation scenario set in a virtual house. Since a virtual house does not allow for in-situ programming, we created floor plans and furnished 3D snapshots from within the house showing the placement and type of smart devices installed in the house. This enables participants to immerse themselves in the concepts and for us to elicit responses from them in regards to their mental model as to causal relationships, naming and implicit relations. Participants were selected such that non of them had any experience with programming or living with smart technologies. Non of the participants had any special interest in computers, but $U1$, $U3$ and $U4$ were comfortable with using them in general. The sessions were conducted in Danish as all participants where native Danes with secondary English language skills.

We chose a bias towards experienced home owners, consequently leading to a higher mean age. The tasks (Table II) are designed to elicit responses to different aspects. Tasks 1 and 2 are used to understand the mental model of the user, and if they can, after designing the first scenario, recreate a slightly

TABLE I. UNIQUE ID, AGE AND OCCUPATION OF PARTICIPANTS.

ID	Age	Gender	Occupation
$U1$	50	F	Head of Division
$U2$	71	M	Mechanical Engineer
$U3$	46	F	Head of Division
$U4$	20	F	Student of Medicine with Industrial Spec.
$U5$	65	F	Retired Receptionist

modified version of it, phrased differently. Task 3 requests the user to create a constraint on the scenarios. Finally Task 4 asks the user to create a more complex scenario, that violates the constraint created in Task 3, triggering an error message from the system.

TABLE II. CONTENT OF THE DIFFERENT TASKS.

Task number	Task description
1	When the switch by the dining table is turned on, turn on the lamp over the table.
2	Turn on the dimmable lamp over the sofa table at 50% when the switch by the hallway is turned on.
3	The light should never turn off in the bathroom while someone is in there.
4	When the switch at the main entrance is turned on for more than 5 seconds, turn off the light in the bathroom, bedroom and hallway and lower the temperature in the living room to 17 degrees.

B. Findings

When analyzing the transcriptions and video recording from the sessions, several trends emerged.

1) Identification of routines: When described an automation scenario, all participants were able to discover patterns in their own lives that could be transformed into similar scenarios. These scenarios were often representations of specific fixed rituals, such as going from room to room to turn off electrical appliances left on by the family when leaving home in the morning, or protecting the home. “In my own house, I know that when it rains then the windows should close” ($U1$).

2) Translating into scenarios: All of the participants were able to understand the iconography. After identifying how the abstract device icons could be transformed into specific ones, they were all comfortable with dragging the icon around, dropping them into the empty boxes. In general the participants quickly learned the iconography of the floor plan. In an in-situ setting this could be mimicked by real-world device renderings in the UI for improved correspondence without a digital floor plan. We note however that the naming had an important impact on the ability to identify the devices. For example, two different naming schemes were applied to two switches. All participants were able to quickly identify the switch named “Switch by the dining table”, whereas the other switch, named “Switch by the hallway” located right beside the opening to the hallway was deemed much more confusing by the participants. There was a general consensus that a naming scheme tied to the furniture and or general usage for the area was more appropriate.

When translating a task into a scenario, participants were at first goal oriented, focusing on the action to be performed. Participants generally preferred to fill out the scenario in a left to right manner, which led to confusion when designing the scenario for Task 1. Moreover, except for $U4$, they were not able to decode the labels until presented with concrete example. This resulted in misinterpretations, the event field being thought as the one for the action. For example, some participants tried to fit the time duration of the event field to their mental model of action. “In 5 seconds, the light will turn on.”($U2$). “When the button is on, then it will turn off within a minute” ($U1$, even though 30 seconds was selected). In general, we note that $U4$ showed much stronger skills in translating tasks to ECA, with a strong focus on reading and parsing the accompanying labels to each input field.

3) Programming by example: After experimenting and creating a correct scenario for Task 1, the participants were able to reapply the pattern for Task 2 and 4, noting the similarities in structure between the tasks in the follow-up interview. $U3$ even observed how the specific pattern of turning on a lamp would be perfect for templating, a feature implemented, but not experimented during the study. In general programming by example, using templates or predefined scenarios, appears to be suitable for the participants and would minimize the current reliance on labels, reducing the programing task to mapping scenarios to concrete devices.

4) Required vs optional: All participants noted difficulties in understanding which fields were required and which were optional in modal boxes (Figure 7). Most participants at first tried to select as many options as possible, while trying to make up a reason and meaning for a specific setting. Similar problems could be observed with regards to condition field in scenarios, where leaving it empty felt as something was missing. “It looks like it is missing something in the middle, but I have no idea what it should be. ”, “(why do you believe that it should be used?) because it is there.” ($U4$). Emphasizing the difference between events, conditions and actions could help, possibly separating conditions from the causal relationship between events and actions.

5) Designing constraints and identifying bad scenarios: All of the participants felt comfortable with designing constraints in the system, with a clear understanding of how they were different from the scenarios in Tasks 1 and 2. Both $U4$ and $U5$ suggested restrictions on when the light should turn off when asked to think up an example of a constraint before being asked to design Task 3. During Task 3, both $U1$ and $U4$ noted that while they understood the usefulness of constraints, they found that this particular one would not fit with their daily routines or did not take into account pets roaming the house. This highlights the importance for inhabitants to understand the automation of their living space and enable them to act upon it.

Linking the error message triggered by the design of the scenario in Task 4 with the constraint designed in Task 3 proved to be more challenging. When faced with the message stating that the scenario violated a constraint, the reactions varied. While all were able to decipher the message and apply the change to the scenario, not all were able to link it to the previously designed constraint. $U4$ was however immediately able to identify the link between the constraint and the error. At the same time, he was able to understand the use of conditions in scenarios “ oohhh! Now I understand the box in the middle!” ($U4$). After this realization, $U4$ was able to identify a number of scenarios in which conditions were necessary and used correctly. Highlighting the ’why’ when presenting the error is a first step to solve this issue, making it explicit as to where this constraint comes from. When constraints have been designed months or years in advance, it would not be fair to expect even the most observant user to remember the exact reason for adding a constraint. One solution could be to add name, description and the visualization of the specific constraint.

VI. Discussion and Future Work

In light of the experiments that were conducted, a number of ideas and possibilities for improvement emerged. Apart from improving the graphical representation to be more intuitive, a deeper integration of templating and programming by example could be helpful to help ease users into scenario creation. Suggesting templates based on installed devices is another possible extension of this. Another approach to be tested (more goal oriented compared to our sequential programming approach), would be a wizard supporting the user in the creation of scenarios. The notion of time in the interface is also an interesting point. We initially thought that the event was the easiest part to put timing constraint. It seems however that having time in actions could also be an interesting path to explore. Finally, it would be interesting to do an in depth user study going into detail as to how specific groups (such as programmers and non-programmers for example) approach scenario creation.

Another type of improvement that is foreseen is on the backend and the utilization of the verification engine. Verification could be used for checking feasibilities of scenarios, detect conflicts between them, or support the user in debugging scenarios by finding the cause of undesired behaviors. Some work also needs to be done in improving the verification time. We noticed in fact that some devices with large input values, such as dimmers for example, lead to long verification time. Reducing or abstracting these values into intervals could significantly improve the verification. Compositional verification is also another direction that will be explored to improve this step of the process.

VII. Related Work

The work presented in this paper follows a trend in applying formal method to reason about reliability of intelligent environments. Apart from the works already mentionned in the introduction [2], [3], [4], we can mention the work done in [19]. In this work the authors formalize a Smart Home environment using Petri-Nets which enables a formal analysis of the environment. This is similar to the work that we have previously done in [5] using TA.

We divide the rest of the related work into three categories. Firstly we discuss research done on the use of scenarios to build specifications for intelligent environments. Secondly we describe research projects that have developed user friendly programming interfaces for home automation, be it tangible, block building or textual interfaces. Lastly we mention machine learning approaches that solve complexity issues by learning from the user.

A. Rule Based Scenario

The work of Augusto and Nugent [12] is the first noticeable appearance of the concept of Event-Condition-Action (ECA) in smart homes, in which an event triggers an action if a certain condition is met. They bring together Active Databases concepts and Temporal Reasoning to define an ECA language with temporal constrains. They provide a detailed explanation of the expressive power of the language and a formal definition of the different elements. We reuse their concept of ECA language to model scenarios and constraints and translate them into TA. However the language we developed is intentionally less expressive in order to keep complexity at an acceptable level of simplicity for the user. We also note that our system does not record events, which implies that language elements in a single scenario must be related in a sequential order.

To enable flexible and simple programming of indirect control in smart environments, Garcia et al. [17] make use of a context-aware middleware combined with an ECA language. We note that their work is rather general as their focus is on all types of smart environments and not only on home automation. The ECA language they use is designed to isolate the complexity of the scenario, and flexibility is provided by the use of wild cards that represent sets of entities instead of single entities. They conducted a user study to evaluate the language over two groups, those with programming experience and those without. The study shows that both groups can easily differentiate events from conditions in the language, but that natural language does not make the distinction clear for non programmers. This shows that ECA languages can be understood by people, and that graphical representation might help non technical people to use them. In order to remove time complexity, they introduce timers that replace time relations between events. However, the user study did not take into account timed scenarios, so there is no confirmation that timers are a good way of reducing time complexity for the users. Moreover, the UI provided as examples are not discussed in details and are not evaluated.

Qiao et al. [16] present visual ECA rules with temporal events. They first define an ECA language extended with Metric Temporal Logic (MTL) that allows them to express more complex scenarios. They then propose a visual representation for these rules and the relations between events, conditions and actions. They finally provide a GUI aimed for users to design ECA rules. If the ability to use MTL within the ECA rules can be appealing to experienced user, the added complexity might confuse the ones not used to temporal logic. A similar remark can be made on the GUI, which is rather technical and seems directed toward experienced users.

B. Tangible programming tools

Drawing on observations on context-aware applications, Lee et al. [20] present a smartphone app, GALLAG, for tangible programming in a home automation setting where device actions are programmed by recording user actions. The focus of Lee et al. is mainly on small and simple applications that provide reminders and audible rewards when the user performs actions in a fake household setting. The approach is similar to that of HomeMaestro [18]. Both solutions make use of a simple *If-then* relationship between input and output, however GALLAG only allows for tangible programming to be used in the *If*. Neither solution provides support for temporal relationships, nevertheless Lee et al. do provide ways to virtually add temporal requirements.

C. Machine learning approaches

A different approach to creating scenarios and interaction in a smart home is to use machine learning techniques, taken by the Nest thermostat[2]. This thermostat learns from the user behavior to generate personalized heating and cooling schedule to improve comfort and energy consumption. It also provides a smart phone and web-based access to the schedule and to control the thermostat in real time, giving raise to a better interaction compared to classic thermostats. User experiences with this intelligent thermostat have been reported in [21]. Although the results cannot be directly generalized to all learning systems, and more studies are needed to understand and improve user interactions with such systems, they provide interesting insights on users' reactions and expectations. They first show a frustration from the users due to a lack of control and understanding of how the system generates the schedule. In some cases, that led to the user overriding the automated schedule with a manual one. Facilitating the creation of intelligence by the user itself might thus improve its collaboration with the system. Even though the intelligent features of the thermostat did not satisfy all the users, they generally found it more enjoyable to use than a classic thermostat, mostly due to its interactive design and the smart phone access to it. This shows that it is essential to provide well designed interfaces that enable the user to have a sense of control over the system.

In this paper we have focused on enabling users to program their own environment to help them understand it, and propose them to tell the system what it *should not* do. We note that during our survey of related work we have not found any proposal for constraining the control to allow users to experiments safely with creating scenarios.

VIII. Conclusion

In this paper, we have presented a framework enabling the specification of scenarios and constraints preventing users from defining scenarios leading to undesired configuration of the environment. The specification language is based on an ECA paradigm and we provided a semantics of this language in the form of TA, making possible a formal verification of scenarios with respect to the constraints. We have then presented a prototype implementation of this framework composed of a backend that performs the translation of the language and the verification, and a user interface aimed at facilitating the design of the scenarios. We finally presented a user study that provides interesting findings for developing user interfaces on top of an ECA languages, simplifying scenario specifications and the understanding of constraints for restraining scenarios. This opens up for further work and experiments, and applications of formal methods to intelligent environments.

Acknowledgments

The authors would like to thank Jesper Kjeldskov and Anders P. Ravn for valuable conversations throughout the writing of this paper. This research is partially founded by the Danish project TotalFlex and the European Projects Arrowhead and Intrepid [3].

[2]http://nest.com

[3]totalflex.dk, arrowhead.eu, fp7-intrepid.eu

References

[1] M. Weiser, "The computer for the 21st century," *Scientific american*, vol. 265, no. 3, pp. 94–104, 1991.

[2] J. C. Augusto and M. J. Hornos, "Software simulation and verification to increase the reliability of intelligent environments," *Advances in Engineering Software*, vol. 58, pp. 18 – 34, 2013.

[3] A. Coronato and G. D. Pietro, "Formal design of ambient intelligence applications," *Computer*, vol. 43, pp. 60–68, 2010.

[4] F. Corno and M. Sanaullah, "Design-time formal verification for smart environments: an exploratory perspective," *Journal of Ambient Intelligence and Humanized Computing*, pp. 1–19, 2013.

[5] P. Dalsgaard, T. Le Guilly, D. Middelhede, P. Olsen, T. Pedersen, A. Ravn, and A. Skou, "A toolchain for home automation controller development," in *Software Engineering and Advanced Applications (SEAA), 2013 39th EUROMICRO Conference on*, 2013, pp. 122–129.

[6] A. B. Brush, B. Lee, R. Mahajan, S. Agarwal, S. Saroiu, and C. Dixon, "Home automation in the wild: challenges and opportunities," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '11. New York, NY, USA: ACM, 2011, pp. 2115–2124.

[7] S. Davidoff, M. Lee, C. Yiu, J. Zimmerman, and A. Dey, "Principles of smart home control," in *UbiComp 2006: Ubiquitous Computing*, ser. Lecture Notes in Computer Science, 2006, pp. 19–34.

[8] P. Valiente-Rocha and A. Lozano-Tello, "Ontology and SWRL-based learning model for home automation controlling," in *Ambient Intelligence and Future Trends-International Symposium on Ambient Intelligence (ISAmI 2010)*, ser. Advances in Intelligent and Soft Computing, 2010, vol. 72, pp. 79–86.

[9] Y. Rogers, "Moving on from Weisers vision of calm computing: Engaging ubicomp experiences," in *UbiComp 2006: Ubiquitous Computing*, ser. Lecture Notes in Computer Science, P. Dourish and A. Friday, Eds. Springer Berlin Heidelberg, 2006, vol. 4206, pp. 404–421.

[10] C. Alexander, *Notes On The Synthesis Of Form*. Harvard University Press, 1964.

[11] R. Alur and D. L. Dill, "A theory of timed automata," *Theoretical Computer Science*, vol. 126, no. 2, pp. 183 – 235, 1994.

[12] J. C. Augusto and C. D. Nugent, "The use of temporal reasoning and management of complex events in smart homes," in *ECAI*, vol. 16. Citeseer, 2004, p. 778.

[13] G. Behrmann, A. David, and K. Larsen, "A tutorial on Uppaal," in *Formal Methods for the Design of Real-Time Systems*, ser. Lecture Notes in Computer Science, M. Bernardo and F. Corradini, Eds. Springer Berlin Heidelberg, 2004, vol. 3185, pp. 200–236.

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

[15] D. McCarthy and U. Dayal, "The architecture of an active database management system," *SIGMOD Rec.*, vol. 18, no. 2, pp. 215–224, 1989.

[16] Y. Qiao, H. Wang, K. Zhong, and X. Li, "Visual event-condition-action rules with temporal events," in *Eighth Real-Time Linux Workshop*, 2006, p. 275.

[17] M. García-Herranz, P. A. Haya, and X. Alamán, "Towards a ubiquitous end-user programming system for smart spaces." *J. UCS*, vol. 16, no. 12, pp. 1633–1649, 2010.

[18] T. Karagiannis, E. Athanasopoulos, C. Gkantsidis, and P. Key, "Homemaestro: Order from chaos in home networks," Microsoft Research, Tech. Rep., 2008.

[19] S. W. Loe, S. Smanchat, S. Ling, and M. Indrawan, "Formal mirror models: an approach to just-in-time reasoning for device ecologies," *International Journal of Smart Home*, vol. 2, no. 1, pp. 15–32, 2008.

[20] J. Lee, L. Garduño, E. Walker, and W. Burleson, "A tangible programming tool for creation of context-aware applications," in *Proceedings of the 2013 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, ser. UbiComp'13, 2013, pp. 391–400.

[21] R. Yang and M. W. Newman, "Learning from a learning thermostat: Lessons for intelligent systems for the home," in *Proceedings of the 2013 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, ser. UbiComp'13, 2013, pp. 93–102.