A Toolchain for Home Automation Controller Development

Peter H. Dalsgaard, Thibaut Le Guilly*, Daniel Middelhede, Petur Olsen*,
Thomas Pedersen, Anders P. Ravn*, Arne Skou*
Department of Computer Science
Aalborg University, Denmark
*{thibaut,petur,apr,ask}@cs.aau.dk

***Abstract*—Home Automation systems provide a large number of devices to control diverse appliances. Taking advantage of this diversity to create efficient and intelligent environments requires well designed, validated, and implemented controllers. However, designing and deploying such controllers is a complex and error prone process. This paper presents a toolchain that transforms a design in the form of communicating state machines to an executable controller that interfaces to appliances through a service oriented middleware. Design and validation is supported by integrated model checking and simulation facilities. This is extendable to controller synthesis. This toolchain is implemented, and we provide different examples to show its usability.**

***Keywords*—Controller; Home Automation; UPPAAL; Model Checking; State Machines;**

I. INTRODUCTION

A diversity of services can be provided by home automation systems. They can control heating systems, gather temperature or humidity data, manage lighting, or manipulate household appliances. The goal of home automation is to coordinate and control these different services in order to create an intelligent environment. The control of the environment is generally accomplished through open loop systems that specify setpoints on devices such as thermostats or light dimmers. The environment thus consist of a set of discrete states representing the different possible configurations of the setpoints. The transitions between the states are triggered either by events coming from the change of a monitored variables or by time. The control of the environment and the configuration of devices can therefore be easily modelled by timed automaton [1].

Based on these observations, we propose a toolchain for designing, validating, implementing and executing controllers for such environments. The controllers are modelled as timed automaton that are used for the execution without any intermediate compilation. The toolchain uses two existing components. Firstly, the HomePort middleware [2], is used to gather information about available services in an Ambient Intelligence (AmI) environment, and to send and receive data from these services. Secondly, the UPPAAL [3] toolbox is used to model and verify controllers, and as an execution engine. To link these two components, two modules have been developed. HomePort2Uppaal gathers necessary information from HomePort about available devices and services, and makes it available in UPPAAL for developing controllers. The UppaalInterpreter enables execution of the control system.

The paper is structured as follows. The components of the toolchain are presented in Section II. Section III describes the process of developing, verifying and executing controllers with the toolchain. In order to demonstrate the applicability of the proposed solution, experiments are presented in Section IV. We continue with a discussion of related work in Section V. Section VI presents future work and Section VII discusses an alternative approach, and comments on a semantic issue with reachability properties. Finally, Section VIII concludes on the paper.

II. TOOLCHAIN OVERVIEW

This section provides an overview of the toolchain, starting with a description of the two existing components, HomePort and UPPAAL. This is followed by a description of the two novel components, HomePort2Uppaal and the UppaalInterpreter. Finally comes a summary of the development and use process.

A. HomePort

HomePort is a middleware implementing a common interface to heterogeneous home automation networks. It facilitates the tasks of the control system by enabling it to access and modify the services of the environment in a unified manner, regardless of the protocol or technology that a specific device is using. Moreover, its REpresentational State Transfer (REST) architecture is easy to interface with. It also provides an event mechanism allowing the control system to receive notifications when the state of a service changes.

HomePort plays two roles in the toolchain: to provide a list of available services and to provide read and write access to these services.

B. Uppaal

UPPAAL is a toolbox for modelling, simulation and verification of real-time systems. It is composed of three main parts: a description language, a simulator, and a model checker. The description language models system behavior as communicating state machines with timers. The simulator enables exploration of possible executions of a model. The model checker can check safety, liveness and reachability properties on a model.

Processes communicate through channels and shared variables. A channel synchronizes a sender and a receiver process. A transition involving synchronization on a channel can only

be taken when both the sender and receiver are able to perform the synchronization. Broadcast channels can also be used to synchronize a sender with multiple receivers. Note that in a broadcast a sender is always able to synchronize on the channel, even if no other process is ready to receive it. To exchange values between processes, synchronous value passing [4] is used. Synchronous value passing is a modelling technique where a sender and a receiver synchronize on a channel and exchange a value by using a shared variable.

Transitions can be guarded by boolean expressions over clocks and discrete variables. A transition with a guard can only be taken when the guard evaluates to true. Transitions can also update the values of variables through update statements and reset clocks. Finally, select statements can be used to non-deterministically assign to a temporary variable a value within a bounded range. The value of the temporary variable can then be assigned to a variable. Note that a select statement actually creates as many transitions as there are values in its range, each transition associating a different value to the temporary variable.

In the toolchain, UPPAAL is first used to model the controller using the description language through a graphical user interface. The model checker is then used to verify properties of the designed model. Finally, the experimental concrete simulator is used to assist in executing the control system. The concrete simulator, as opposed to the symbolic simulator, uses concrete time values rather than intervals.

C. HomePort2Uppaal

The purpose of HomePort2Uppaal is to translate a list of HomePort services to a UPPAAL template that enables controller models to access the services. It starts by creating an empty UPPAAL template, named HomePort. An initial location is added, which will be the only location of this template. Once the basis of the template is created, HomePort2Uppaal establishes communication with a HomePort server and retrieves its service list. Figure 1 shows a simplified example of such a list. For each service in the list, *get*, *set*, and *ev* channels are declared, and the location is populated with corresponding transitions. A *get* channel can be used by a controller model to obtain the value of a service, a *set* channel to update the value of a service, and an *ev* channel to receive a notification when an event occurs on a service. All channels are declared urgent, which means that time cannot pass when a transition involving synchronization on them is active. Moreover, *ev* channels are declared as broadcast channels as several processes may be interested in receiving an event. Having urgent broadcast channels without guards would prevent time from passing, since broadcast channels are always active. Thus we use the guard *evLock == 1* on the event transitions to deactivate them. An "unlock" transition setting the *evLock* variable to 1 is added. This transitions is taken when an event is received from HomePort to enable the event transitions. All event transitions reset the *evLock* variable to 0.

To exchange data between this template and others, one-way unconditional synchronous value passing is used. As *get* and *ev* channels pass data to other processes, they are declared as sending channels (denoted by '!'). The *set* channels are declared as receiving channels (denoted by '?'), as they receive data from other processes.

```
<?xml version="1.0" encoding="utf-8"?>
<devicelist name="services.xml" id="12345">
  <device desc="Thermostat" id="1" location="LivingRoom"
      type="Thermostat">
    <service desc="Temperature" id="0"
        value_url="/Thermostat/1/Temperature/0"
        type="Temperature" unit="Celsius">
      <parameter id="0" max="50" min="0" />
    </service>
  </device>
  <device desc="Window" id="1" location="LivingRoom"
      type="Window">
    <service desc="Sensor" id="0"
        value_url="/Window/1/Sensor/0" type="Sensor"
        unit="Open/Close">
      <parameter id="0" max="1" min="0" />
    </service>
  </device>
</devicelist>
```

Fig. 1. HomePort Simplified Service List

Fig. 2. UPPAAL Template

```
int[0,1] evLock;
urgent chan get_Thermostat_1_Temperature_0;
urgent chan set_Thermostat_1_Temperature_0;
urgent broadcast chan ev_Thermostat_1_Temperature_0;
int[0,50] var_Thermostat_1_Temperature_0_0;
urgent chan get_Window_1_Sensor_0;
urgent chan set_Window_1_Sensor_0;
urgent broadcast chan ev_Window_1_Sensor_0;
int[0,1] var_Window_1_Sensor_0_0;
```

Fig. 3. Global Variables Declaration

The channel names are built by prefixing the name of a service with either get_, set_ or ev_. This enables a reverse translation during the execution phase, in order to retrieve the name of the service a channel is associated with. A global variable is also associated with each service. Its name consists of the name of the service prefixed with var_, and followed by the identifier of the parameter to which this variable is associated. This variable can be used by any template to access the state of the service. Minimum and maximum values of the parameter are used to set bounds on the global variable. A select statement is used to update the value of a service for *get* and *ev* transitions. Thus, during verification, checks a controller for all possible values that a service can deliver. It is also used during the execution to update the value of a service to its actual value in the environment. The template can then be used to model a control system that can access and modify

the state of the services.

Figures 1, 2, and 3 show the results of transforming a HomePort service list to a UPPAAL template. The list of services in Figure 1 contains two devices: a thermostat with a temperature service, and a window with a sensor reporting its state. Given this list, HomePort2Uppaal creates the UPPAAL template shown in Figure 2, and declares the variables shown in Figure 3.

D. UppaalInterpreter

The UppaalInterpreter runs an execution of the control system. The model is first given to a UPPAAL simulator. The UppaalInterpreter then executes it by telling the simulator which transitions to take, by running the algorithm shown in the following pseudo code:

loop
  repeat
    Update available transitions.
    if there is an internal transition **then**
      Choose nondeterministically an internal transition and take it.
    else if an event was received **then**
      Take the "unlock" transition.
      Update available transition.
      Take the corresponding event transition.
    else if there is a Get **or** Set transition **then**
      Perform the request on the HomePort server.
      Take the corresponding transition.
    end if
  until there are no available transitions
  Wait until next transition can be taken **or** an event is received.
end loop

The first step of the algorithm is to retrieve all active transitions from the simulator. It then takes a first active internal transition that can be taken at the current time. An internal transition is a transition that does not involve synchronization with the HomePort process. After taking any transition, the algorithm is restarted, as the set of available transitions is no longer valid. Once all internal transitions that do not involve time passing have been taken, it checks if an event was received from the HomePort server. In this case, the "unlock" transition is taken to activate event transitions. The transition corresponding to the received event and value is chosen among the possible ones created by the select statement used in event transitions. When all events are processed, it takes the first transition involving synchronization with a *get* or *set* channel. When taking such a transition, the corresponding request is performed on the HomePort server. For *get* synchronizations, the transition is chosen based on the value returned by the HomePort server, similar to event synchronizations. Once again, after taking a transition, the algorithm is restarted. Once there are no more transitions that can be taken at the current time, it waits for the minimum time until a transition can be taken again, or until an event is received.

Note that *available transitions* denotes transitions that can be taken at the current time in the environment. The UppaalInterpreter is responsible for synchronizing the UPPAAL time with external time. Therefore, the UppaalInterpreter only takes

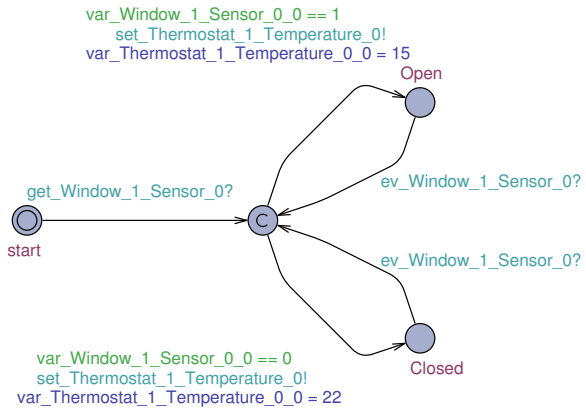

Fig. 4. Temperature Controller

transitions that have a target state in which the time is equal to the external time. The UPPAAL time is updated by specifying a delay when taking a transition, which corresponds to the time since the last transition was taken. The UppaalInterpreter currently uses seconds as a unit, but this could easily be modified to adapt to different time scales.

An execution trace can also be generated and viewed in UPPAAL. Analyzing the trace can be useful for debugging and finding flaws in the controller or in the devices. Figure 4 shows a simple controller for adapting the temperature of a room depending on the state of a window. Figure 5 shows an execution trace of this controller from the UppaalInterpreter interacting with a running HomePort server. The trace is represented as a message sequence chart. Notice that some events are not received by any process, because the event channels are defined as broadcast channels, and can always be taken, even when no process wants to synchronize on them.

E. Development and Use

Figure 6 shows the different components of the toolchain and the different steps to go through from the development to the execution of a control system. The first step is to retrieve the set of services that are available in the environment, corresponding to the arrow labeled 1 in the figure. This is done by the HomePort2Uppaal module making a request to the HomePort web server, which returns an XML file with a description of all the services and information on how to access and use them. HomePort2Uppaal then transforms this list of services into a UPPAAL template providing communication channels and variables for other templates to be able to access and modify the service states. This template can then be opened in UPPAAL, represented by the arrow labeled 2. The actual control system can then be modelled as one or more UPPAAL state machines. The verification and simulation features of UPPAAL serve to check properties of the controller. Once the model of the controller has been constructed and verified, it is exported as a standard UPPAAL XML file.

The process can now move to the use phase, represented in the figure by the arrows labeled 4. The generated XML file is given to the UppaalInterpreter, which will execute the modelled controller, using a UPPAAL simulator. The execution is done by choosing appropriate transitions in the model, and by communicating with the HomePort server to get, set or

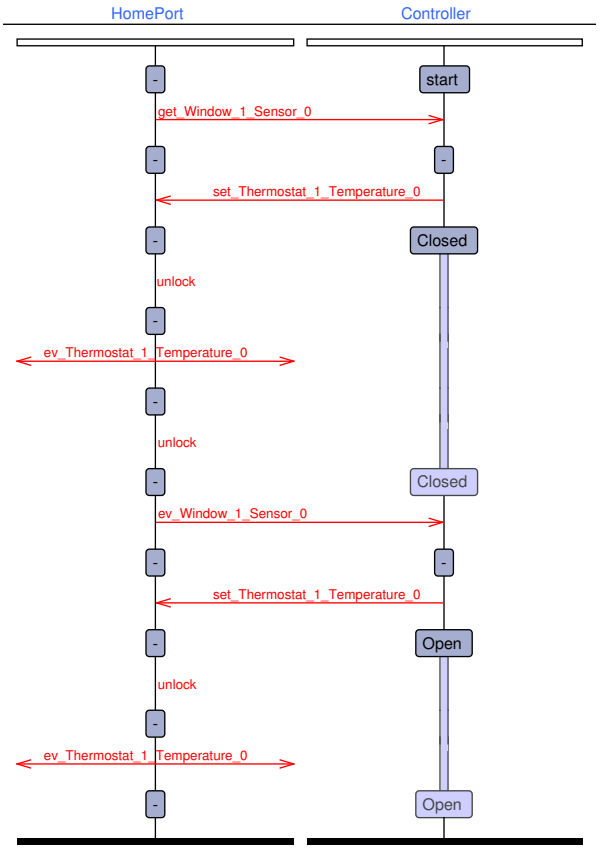

Fig. 5. Trace of an Execution of the Temperature Controller

receive events on the services. It also synchronizes the UPPAAL time with real world time in order to have a faithful real-time execution.

III. DESIGN, VERIFICATION, IMPLEMENTATION AND EXECUTION

Setting up an AmI environment can be viewed as a control problem. Given a set of controllable devices, the problem is to find a controller able to configure the environment in order to satisfy parameters that can for example relate to user comfort or reduction of power consumption. The four steps in this process are presented in this section, using the simple control system example introduced in Section II-D.

A. Design

Dorf and Bishop [5] define the design of a control system as a process composed of seven building blocks, divided into three groups:

- "establishment of the goals and variables, and definition of specifications (metrics) against which to measure the performance",
- "system definition and modeling", and
- "control system design and integrated system simulation and analysis".

The first step is thus to establish the goals of the control system. For our example, the goal is to control the temperature

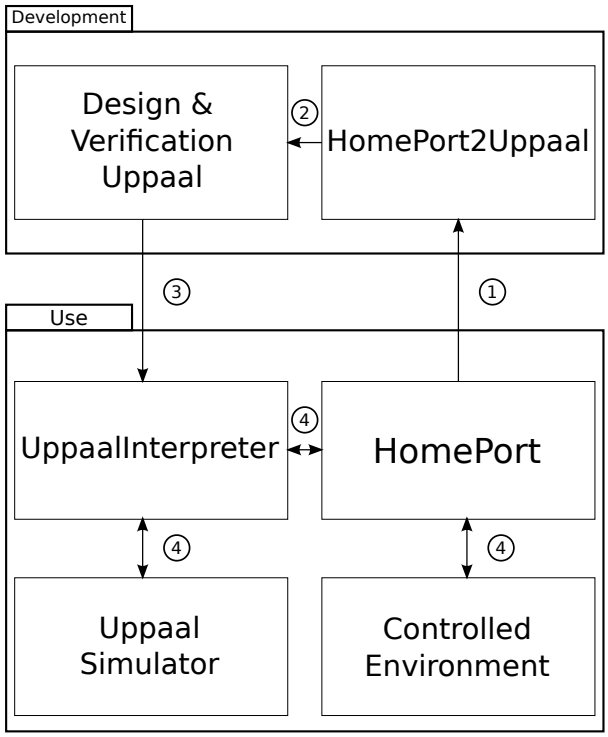

Fig. 6. Development and Use

of the room depending on the state of the window. We then identify the variables to be controlled, here the temperature of the room. Finally comes the specification, namely regulate the room temperature depending on the state of the window. It should not go more than two Celsius degrees under the lowest desired temperature and two degrees above the highest one.

System definition and modelling includes a system configuration that can satisfy the specification. Here we use an open-loop system that uses a thermostat to regulate the temperature, and a window sensor to monitor the state of the window.

The different components of the control system are then modelled. Figure 7 shows models of the thermostat, the window sensor, and the behavior of the variable to be controlled; the temperature. The thermostat simply turns on the heater when the room temperature is less than the setpoint, and turns it off when greater or equal. The window sensor reports the state of the window, and synchronizes with the user model. The temperature is computed depending on the thermostat and window states. We adapt an extremely simple model for the temperature: when the thermostat is on and the window closed, it increases by two degrees per time unit. When the thermostat is off, it decreases by one degree per time unit with a closed window, and by two degrees if the window is open. Finally, the temperature remains unchanged when the thermostat is on and the window open. We also set lower and upper bounds for the temperature: 10°C and 30°C. This temperature model is of course a simplification of a real life scenario, but serves its purpose in this illustrative example.

The next component of the system is a user who interacts with the window, shown in Figure 8. She opens the window between 10 and 15, and closes it between 20 and 24. The user

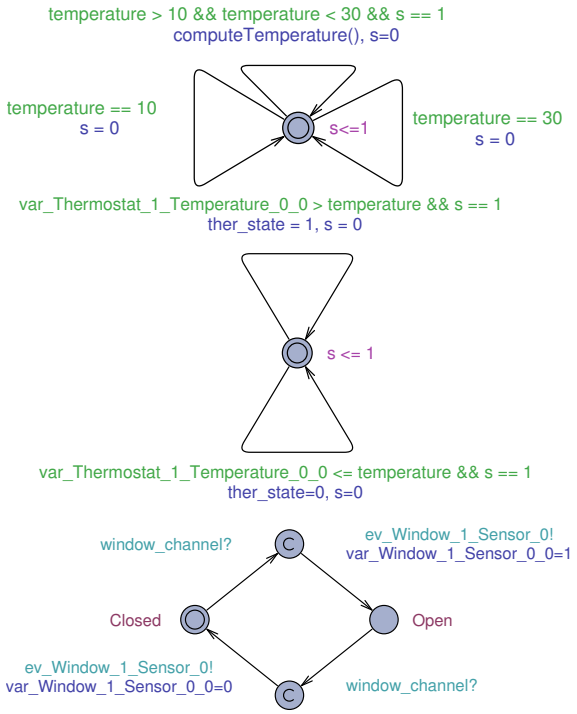

Fig. 7. Temperature and Devices Models

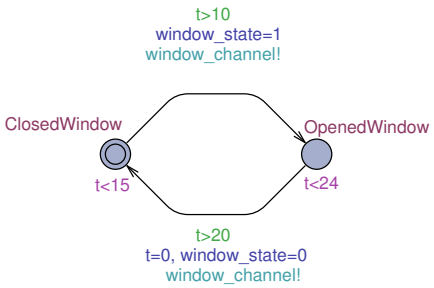

Fig. 8. Model of the user

model synchronizes with the window sensor in order to update the sensor's value.

Finally, the controller is shown in Figure 4. It works as a simple open loop controller setting the desired temperature of the thermostat depending on the state of the window.

Thanks to the UPPAAL Statistical Model Checking (SMC) simulator, we are able to run the simulation of the control system shown in Figure 9. From this simulation we can see that the controller succeeds to fulfill the design goals, and satisfies the specifications. However, simulation is not enough to ensure that the desired properties are satisfied by the control system. In order to ensure it, we use verification.

B. *Verification*

Baier and Katoen [6] say that "system verification is used to establish that the design or product under consideration possesses certain properties". Good controllers must possess certain properties, among which stability is one of the most important.

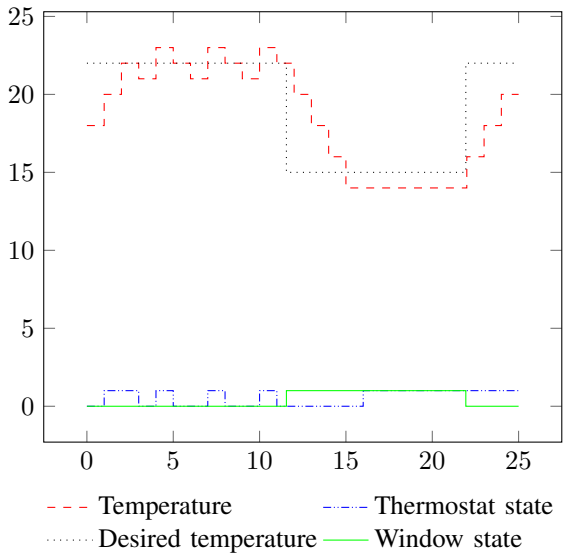

Fig. 9. Simulation of the control system

A stable system is defined by Dorf and Bishop as "a dynamic system with bounded response to a bounded input". Different approaches can be taken to verify this property. In our example, the controller inputs the desired temperature to the thermostats, which has to react with a bounded output. We can thus verify that the controlled temperature stays inside a bounded interval. As we specified in the design, the temperature should not go more than two degrees above the maximum desired temperature and more than two degrees below the minimum temperature. We query the UPPAAL verifier to check that this is satisfied. It is the safety property: A[] (temperature <= 24 and temperature >= 13). A positive answer from the model checker ensure that given this modelled system, the temperature of the room will satisfy the property.

Other properties can also be checked, such as absence of undesirable deadlock states, or reachability of desirable locations.

C. *Implementation and Execution*

After designing and verifying the control system, implementation and execution remain. Depending on the choice of technology for the sensors and actuators devices, it can be necessary to write HomePort adapters to communicate with them. The devices then need to be deployed in the environment, and the HomePort server to be started. When the communication through HomePort is established, HomePort2Uppaal can be used to gather the channel and variable names that need to be put in place of the names used during the design phase. Our example uses the actual names for the design of the controller, thus this is unnecessary. The next step is to get access to a UPPAAL simulator. This can be done either on a local computer inside the environment, or using an external server. Finally, the last step is to run the UppaalInterpreter in order to execute the controller.

IV. EXPERIMENTS

In order to complete the demonstration of the toolchain, we use the HomePort demonstrator. It consists of a home

automation system built into a suitcase, accessible through a HomePort server. The system has three subnetworks, and contains a window sensor, a thermostat, two light switches, a light dimmer, a temperature and humidity sensor, and an access control panel (for more technical details please refer to [2]). Previously, three scenarios were implemented in Java. The purpose of the experiments was to reproduce the scenarios with the presented toolchain. Three scenarios were previously implemented, and we present the design and verification of controllers for two of the scenarios, the third being the temperature controller presented in Section III. Note that the variable names of the models have been shortened in order to improve readability, and that simulation of the control system is omitted since it is really simple.

A. Light Control

The first controller aims at saving energy by switching off the lights when people leave the house, and turn them back on when they come back. The variables to be controlled are the state of the lights, and a variable to monitor occupancy. To detect house occupancy, we create a service toggling occupancy when a correct pin code is entered on the access control panel.

The model of the controller is shown in Figure 10. The house has three lights, two simple lamps connected to a controllable outlet, and a dimmer. At initialization, the controller retrieves the state of the lights, and the occupancy. If the house is unoccupied, the states of the lights are saved and they are turned off. If the house is occupied, the controller just moves to the *atHome* location. After this initialization phase, the controller will react on occupancy events.

The *turnOffAndSave()* function saves the state of the lights and set their values to zero, while the *reload()* function reloads the previously saved value. Also, as only one synchronization per transition is allowed, a separate model synchronizes on the *get* and *set* channels of the services; a simple controller can then synchronize with it in order to update or retrieve the state of the lights. Another solution could have been to create a virtual service controlling all lights.

We then verify properties of the designed controller. The controller must turn off the lights when people are away, thus whenever the controller is in the away location, the value of the light states must be zero. Before performing the verification, we remove the broadcast keyword from the event channels because the verifier would consider taking an event transition always possible for a broadcast. However, in our case the lamp states cannot be modified if no one is at home. Another solution would be to modify the model by handling events in the away location and resetting the value to zero when receiving one. Once the broadcast is removed, the verifier checks the desired property: A[] LightController .away imply (var_Switch_1 == 0 and var_Switch_2 == 0 and var_Dimmer_1 == 0). The property is satisfied. The controller can thus be executed being sure that it will perform the desired control.

B. Burglar Alarm

The second controller monitors the state of the window and the house occupancy and starts the alarm if the window

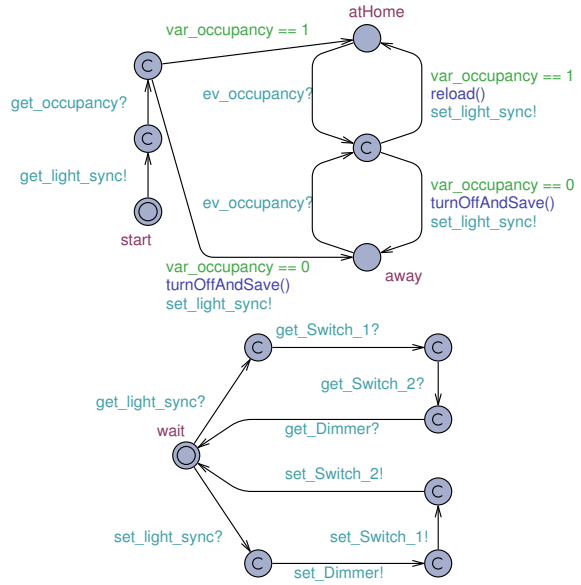

Fig. 10. Light Controller

is opened while the house is empty. The variable to control is the state of the alarm.

The model of the controller is shown in Figure 11. The first step is to get the occupancy of the house. Depending on this state, the controller moves either to the *Home* location or the *Away* location. From the *Home* location, it can only move to the *Away* location, when the pin code is entered, indicating that people are leaving the house. In the *Away* location, if an event indicates that the window has been opened, it triggers the alarm. In this setting, the alarm can only be turned off by entering the correct pin code. Thus, the controller waits for an event on the occupancy state indicating that people returned home before turning it off.

Properties of the modeled controller are then verified. The first property to check is that the alarm cannot be set while the house is occupied: A[] Burglar.Home imply var_alarm == 0. This property is satisfied. The second is to check that if the house is empty and the window is opened, the controller will move to the *Intrusion* location: Burglar.Away and var_window == 1 −−> Burglar.Intrusion. This property actually does not hold. In fact, if the window is open while the occupants leave the house, the alarm is not triggered. This occurs because it is not desirable to trigger the alarm every time people leave the house having a window open. In order to solve this issue, the system could be improved by either informing the user that the house is not secured, or using other sensors to detect intrusion.

C. Composition of Control Components

Once each of the specific control system has been modeled and tested, we combined them into a single system in order to create an AmI environment managing light and temperature, and monitoring intrusion. One important point when composing control components is the risk of interference that can occur when several components try to control the same variable. In fact, this can lead to instability of the control system. One simple way of avoiding this situation is by having

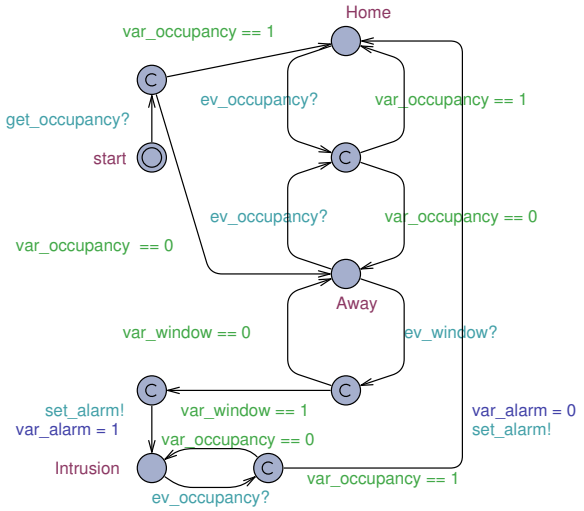

Fig. 11. Burglar Alarm Controller

only one component per control system for each controlled variable. This facilitates verification of stability properties. In our experiments, as each component has its own control variable, we can be sure that no interference will occur when composing the controllers. The composed system can then be executed by the UppaalInterpreter in order to apply the desired control on the environment.

D. Outcomes

The toolchain demonstrates a flexible and efficient path for developing simple controllers for home automation when compared to using classical programming languages. In fact, the example system that previously took two days to create in Java was reproduced in a couple of hours by using the UPPAAL GUI and the template generated by the HomePort2Uppaal module. Also, use of an intuitive GUI for controller development may enable people not familiar with programming languages to construct controllers. Verification of properties of the controllers is also made easy, and helps at insuring that they satisfy their objectives.

V. Related Work

Different alternatives exist for designing and verifying controller models. We first mention Simulink [7], a tool for modelling, simulating and analysing multi-domain dynamic systems based on a graphical data flow programming language. It is able to generate C code from the data flow diagrams for real time implementation. Verification and validation is made through requirements traceability, coverage analysis and modelling standards compliance checking.

Another similar tool is Labview [8], also using a graphical data flow programming language. The graphical code can be compiled into an executable, and tests can be created to monitor and validate the system based on specified constraints.

The SCADE suite [9], a model-based development environment for critical applications, features a graphical interface to design models, a verifier and can generate code. Models are created using a synchronous data flow programming language called Lustre. Verification of safety properties is also done using this language.

Ptolemy II [10] is an open source framework for model-based design and simulations. The design of models is actor-oriented, where actors are software components concurrently executing and communicating via interconnected ports. The model then consists of a hierarchical interconnection of actors. The semantics of the model is not given by the framework, but determined by a software component in the model called a director, implementing the model of a computation. Existing directors include support for process networks, discrete-events, or dataflow among others. Validation of the models is done through simulation.

The proposed toolchain distinguishes itself by the automated generation of templates to communicate with the system, and the fact that controller models do not need to be compiled before being executed. Also, UPPAAL supports formal verification of the models.

The specialization of the underlying computational domain to timed automata may at first seem restrictive. However, it gives the benefits of verification as well as openings for fully automated synthesis through timed game theory [11]. In contrast, Simulink, Labview and Ptolemy II aims at much more general classes of controllers. Thus the computational models that they support are much more expressive; but this comes at the expense of analysis options. Essentially, they provide simulation to validate developed systems. Simulink provides code generation as well; but interfacing the code to the actual system requires a development effort. One has to code suitable S-functions or the like.

A specialization similar to the one proposed here underpins SCADE. The use of synchronous dataflow (Lustre) as the computational model allows verification and synthesis. However, the synchronous model may not fit very well to the essentially distributed, asynchronous nature of home automation controllers. Here the synchronous event based communication supported by the toolchain admits a closer fit.

Note that model checking is a validation option. It is not essential for practical development. Thus we have not discussed other model checkers or analysis tools as this brings us too far from the topic of developing controllers for home automation.

VI. Future Work

Running a UPPAAL server to execute the controller is constraining for small environments. Work is currently being done in collaboration with an industrial partner to translate UPPAAL state machines into lighter state machines that can be directly executed on resource constrained devices, removing the need for a UPPAAL server.

The toolchain could also be further improved to integrate automated controller synthesis. Existing work [12], [13] has shown that controllers can be synthesized using timed game theory. The controller problem is modeled as a timed game and a safe strategy to solve the problem, if one exists, can be generated using UPPAAL-TIGA [11]. Integrating this in the toolchain would further reduce the development time for controllers and ensure that they are safe.

VII. Discussion

The toolchain may be elaborated with test facilities using UPPAAL-TRON [14], [15]. TRON is a tool for online black-box conformance testing for systems specified as timed automata. To perform the tests, it uses a model of the Implementation Under Test (IUT) and a model of the environment. It randomly takes available transitions in the environment model to check that the output produced by the IUT corresponds to the model. To communicate with the IUT, TRON uses adapters that translate channel synchronization between the environment and the model into actual communication with the IUT. One could use the HomePort template generated by HomePort2Uppaal as the system model, the model of the controller as the environment and the HomePort server as the IUT to execute the controller in a manner similar to the UppaalInterpreter. Some modifications would have to be made as TRON only performs tests for a limited amount of time. The reason for developing the UppaalInterpreter instead of using TRON is that the tool and the algorithm are simple and could be implemented for resource constrained devices and be integrated as a component of the HomePort system.

Finally, experiments with the toolchain showed that the interpreter is more deterministic than the verifier. This leads to problems with transitions that are bounded by a time interval. The interpreter always takes such a transition at the lower endpoint of the interval, while the verifier allows it at any point in the interval. This means that, while safety properties are always sound, reachability properties might not be. The verifier may predict more reachable states than the ones visited by the interpreter. For models with deterministic timing behavior, for instance transitions at exact time points, the verifier and the interpreter behave the same. Moreover, choosing the lower endpoint of the interval is a valid interpretation of the transition. We also argue that non-deterministic behavior is not wanted when designing controllers. If needed, it is possible to introduce random delays when taking such transitions.

VIII. Conclusion

This paper presents a toolchain for Home Automation controller development using timed automata, which accelerates and eases the development and execution of controllers. It introduces two novel components developed for the toolchain. HomePort2Uppaal creates a UPPAAL template from a list of services to serve as basis for developing controllers. The UppaalInterpreter enables their execution using a UPPAAL simulator and applies the actual control through a HomePort server. With timed automata and the UPPAAL framework, controller properties can be formally verified, increasing reliability of the control systems. The examples show the benefits of the toolchain both for development time and verification. The toolchain easily adapts to other REST oriented middleware, as only the HomePort2Uppaal component needs to be modified.

As the need for efficient control over appliances and interaction with sensors to improve efficiency of house and building environment is increasing, facilitating their development and verification is essential. The presented toolchain can help to create more sustainable and efficient environments.

Acknowledgment

The research presented in this paper has been partially supported by the EU Artemis project ENCOURAGE[1], the EU FP7 project INTrEPID[2] and the Danish ForskEL project TotalFlex[3].

References

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

[2] T. Le Guilly, P. Olsen, A. P. Ravn, J. Rosenkilde, and A. Skou, "HomePort: middleware for heterogeneous home automation networks," in *5th International Workshop on Smart Environments and Ambient Intelligence 2013 (SENAmI 2013)*, San Diego, USA, Mar. 2013.

[3] K. G. Larsen, P. Pettersson, and W. Yi, "Uppaal in a nutshell," *International Journal on Software Tools for Technology Transfer*, vol. 1, no. 1-2, pp. 134–152, 1997.

[4] 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.

[5] R. C. Dorf and R. H. Bishop, *Modern Control Systems*, 12nd ed. Prentice Hall, 2010.

[6] C. Baier and J.-P. Katoen, *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[7] The Mathworks, Inc. (2013) Simulink. [Online]. Available: http://www.mathworks.com/products/simulink/

[8] National Instruments Corporation. (2013) Labview. [Online]. Available: http://www.ni.com/labview/

[9] P. Abdulla, J. Deneux, G. Stlmarck, H. gren, and O. kerlund, "Designing Safe, Reliable Systems Using Scade," in *Leveraging Applications of Formal Methods*, ser. Lecture Notes in Computer Science, T. Margaria and B. Steffen, Eds. Springer Berlin Heidelberg, 2006, vol. 4313, pp. 115–129.

[10] J. Eker, J. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Sachs, Y. Xiong, and S. Neuendorffer, "Taming heterogeneity - the Ptolemy approach," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 127–144, 2003.

[11] G. Behrmann, A. Cougnard, A. David, E. Fleury, K. Larsen, and D. Lime, "UPPAAL-Tiga: Time for Playing Games!" in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, W. Damm and H. Hermanns, Eds. Springer Berlin Heidelberg, 2007, vol. 4590, pp. 121–125.

[12] J. Jessen, J. Rasmussen, K. Larsen, and A. David, "Guided Controller Synthesis for Climate Controller Using Uppaal Tiga," in *Formal Modeling and Analysis of Timed Systems*, ser. Lecture Notes in Computer Science, J.-F. Raskin and P. Thiagarajan, Eds. Springer Berlin Heidelberg, 2007, vol. 4763, pp. 227–240.

[13] J. Grunnet, J. Bendtsen, and T. Bak, "Automated fault tolerant control synthesis based on discrete games," in *Decision and Control, 2009 held jointly with the 2009 28th Chinese Control Conference. CDC/CCC 2009. Proceedings of the 48th IEEE Conference on*, 2009, pp. 8476–8481.

[14] K. G. Larsen, M. Mikucionis, B. Nielsen, and A. Skou, "Testing real-time embedded software using UPPAAL-TRON: an industrial case study," in *Proceedings of the 5th ACM international conference on Embedded software*, ser. EMSOFT '05. New York, NY, USA: ACM, 2005, pp. 299–306.

[15] M. Mikucionis, K. G. Larsen, and B. Nielsen, "T-UPPAAL: Online Model-based Testing of Real-time Systems: tool demo," in *the 19th IEEE International Conference on Automated Software Engineering*, Linz, Austria, September 24 2004, pp. 396–397.

[1] http://www.encourage-project.eu

[2] http://www.fp7-intrepid.eu/

[3] http://totalflex.dk/In%20English/