# Controller Synthesis for Home Automation

Mathias Grund Sørensen

Department of Computer Science, Aalborg University,
Selma Lagerlöfs Vej 300, 9220 Aalborg Øst, Denmark

**Abstract.** A large challenge in home automation is the construction of control programs to dictate behavior of the system. The design of such a control program is a technically challenging task which is commonly left for the (unaided) end-user. To address this, we present an approach to construct control programs from a high-level specification of desired system behavior using game theory. We provide a complete toolchain based on the HomePort platform and using UPPAAL TiGa for control strategy synthesis. The toolchain is implemented on a Raspberry Pi as a completely automated process from behavior specification in a simple web-app interface to control strategy synthesis and effectuation. We conclude that the approach shows a promising application of game theory in a real-life scenario and that sufficient performance and scaleability for real-life application is achievable, although some work remains on improving performance and extending expressiveness of the approach.

## 1 Introduction

Home automation is currently an area of great interest from industry as well as academia (e.g. [10–12,14]). It has shown the potential to improve various aspects of every-day life, including optimizing energy efficiency, providing quality of life improvements for the elderly and disabled and generally increase accessibility and cooperation between devices in the home. However, with the emerge of affordable network-enabled home appliance devices on the consumer market, many technical challenges arise regarding the configuration of cooperations between such devices. This task is currently either maintained by domain experts or restricted to provide only few predefined functionalities with minimal inter-device cooperation (and general no inter-manufacture compatibility), which results in only limited practical value for the typical end-user.

Some attempts have been made at simplifying this process. HomePort [14] is an example of an open-source software which facilitates inter-device communication through a common web-interface. However, the difficult task of constructing a controller to effectuate behavior in the system is still left to the end-user. Some attempts have been made to simplify this process by providing toolchains to construct controllers in a higher-level modeling framework such as UPPAAL [6], however, this still requires significant domain knowledge to use.

As an alternative approach, it has been suggested to use game theory to synthesize a control strategy from a model of a home automation system and a description of desired system behavior [6]. UPPAAL TiGa [2] is an extension to UPPAAL facilitating synthesis of winning strategies in two-player Timed Games, which can effectively be used to synthesize a control strategy for a home automation system. UPPAAL TiGa has successfully been used to synthesize control strategies in other domains, e.g. to synthesize a control strategy for climate control in a pig stable [13].

However, even when synthesizing control strategies, the task of modeling the system, expressing desired behavior and effectuation of a synthesized strategy is still left as a task for the end-user. This cannot be expected from users that are not domain experts.

### 1.1 Our Contributions

We address the problem by presenting a simple high-level specification logic for behavior in home automation setups and we show how a controller can be automatically obtained using a game theoretic approach. We provide a complete proof-of-concept toolchain for expressing behavior constraints for home automation systems connected through HomePort as well as synthesis and effectuation of control strategies obtained by translation to UPPAAL TiGa. The toolchain is implemented for the Raspberry Pi platform which is generally available at limited cost and could serve as a viable home automation control component.

### 1.2 Related Work

In [6] the authors present a toolchain for generating UPPAAL templates for a HomePort system as well as an interpreter which facilitates control of the actual HomePort system using the UPPAAL simulator. This approach eases the implementation of a controller in UPPAAL, however, the system model and controller must still be implemented manually only using automatically generated communication channel templates, which is not feasible for users that are not domain experts. Also, our experiments show that the approach of controlling the system using a real-time UPPAAL simulation needs some performance improvements to be applicable in larger real-time home automation scenarios.

For behavior specifications in home automation, multiple attempts at mediating logics have been made with various trade-offs between expressiveness and simplicity. In [10] an attempt is made to present a general specification language for smart spaces based on a simple Event-Condition-Action (ECA) concept, originally introduced in [8] and which is commonly adopted for home automation specification languages. However simple, this language provides no direct expressiveness for temporal properties. Instead, they propose a workaround by introducing new components called *timers* which can be set and which will trigger an event when the timer expires. In [15], an attempt is made to include temporal properties in a home automation specification language. This proposed language has a high degree of expressiveness at the cost of simplicity. This is

also a drawback as a mediating logic, as specification and translation becomes very complex. Common for these languages are that they still require significant domain knowledge to use, which we hope to limit as much as possible in our framework.

To synthesize a controller using game theory, UPPAAL TiGa [2] is one of multiple tools available for solving timed safety games. Other such tools include Acacia+ [4] and Unbeast [9]. UPPAAL TiGa is used due to collaboration with developers and access to source-code to obtain ARMv6 Linux binaries, but could be replaced by a similar tool or a dedicated engine.

### 1.3 Outline

The rest of the paper is organized as follows. Section 2 presents our preliminaries. Section 3 describes a home automation system connected through the HomePort platform. Section 4 presents our specification logic, Section 5 explores the logic and provides examples of expressible behavior and Section 6 explores a feature of the specification logic, namely underspecified system behavior. Section 7 provides a translation to Time Game Automata for use with UPPAAL TiGA. Section 8 presents and evaluates the implementation of the automatic toolchain and Section 9 explores a notion of independent subsystems in order to improve scaleability and performance of the approach. Finally, Section 10 provides a conclusion and further work.

**Remark** This paper is based on my pre-specialization project [16]. Section 2 is preserved with only minor changes. Some paragraphs are preserved with no or minor changes in other sections.

## 2 Preliminaries

We now present our preliminaries, in particular Timed Games, and give a formal definition of the synthesis problem.

### 2.1 Timed Automata

Let $X$ be a set of real-valued variables named clocks. A clock valuation over $X$ is a function $v : X \to \mathbb{R}_{\geq 0}$ and $\mathbb{R}^X_{\geq 0}$ is the set of all clock valuations over $X$. We let $v + \tau$, $\tau \in \mathbb{R}_{\geq 0}$, be the valuation $(v + \tau)(x) = v(x) + \tau$ for every clock $x \in X$. We let $v[Y]$ where $Y \subseteq X$ be the valuation $v[Y](x) = 0$ if $x \in Y$ and $v[Y](x) = v(x)$ otherwise.

Let $\mathbb{N}_0 = \mathbb{N} \cup \{0\}$. A clock constraint is a finite conjunction of simple constraints of the form $x \sim n$ where $x \in X, \sim \in \{<, \leq, =, \geq, >\}, n \in \mathbb{N}_0$ and $\mathcal{B}(X)$ denotes the set of all clock constraints.

**Definition 1 (Timed Automaton).** *A Timed Automaton (TA) is a tuple* $(L, l_0, Act, X, E, Inv)$ *where:*

– $L$ *is a finite set of locations,*
– $l_0$ *is the initial location,*
– $Act$ *is a set of actions,*
– $X$ *is a finite set of clocks,*
– $E \subseteq L \times \mathcal{B}(X) \times Act \times 2^X \times L$ *is a finite set of edges,*
– $Inv : L \to \mathcal{B}(X)$ *is a function assigning an invariant to each location.*

A state in the TA is a pair $S = (l, v) \in L \times \mathbb{R}^X_{\geq 0}$ denoting the discrete and continuous part of the state, respectively. The initial state is the pair $s_0 = (l_0, \overline{0})$ where $\overline{0}$ is the clock valuation where all clocks have value 0. From a state $(l, v)$, the TA can either do

– a *delay* transition $(l, v) \xrightarrow{\tau} (l, v')$ where $v' = v + \tau$ for some $\tau \in \mathbb{R}_{\geq 0}$ iff $v, v' \models Inv(l)$, or
– a *discrete* transition $(l, v) \xrightarrow{a} (l', v')$ where $a \in \text{Act}$ iff there exists an edge $(l, g, a, Y, l') \in E$ where $v \models g, v' = v[Y]$ and $v' \models Inv(l')$.

The semantics of the TA is a Labeled Transitions System (LTS) $(S, s_0, \to)$, where $S = L \times \mathbb{R}^X_{\geq 0}$, $s_0 = (l_0, \overline{0})$ and $\to$ is the set of delay and discrete transitions.

A network of TAs is a sequence $A = (A_1, \ldots, A_N)$ of $N$ TAs where if $A_i = (L^i, l^i_0, \text{Act}^i, X^i, E^i, Inv^i)$ then for all $1 \leq i \leq N$ and $1 \leq j \leq N$ where $j \neq i$, $L^i \cap L^j = \emptyset$ and $X^i \cap X^j = \emptyset$. Let $v$ be a valuation of $X^1 \cup \cdots \cup X^N$ and $l^i \in L^i$. We introduce standard broadcast synchronization [3] to allow communication between TAs s.t. $\bigcup_{i=1}^{N} \text{Act}^i$ is partitioned into disjoined sets *Actions* and *Channels* and we use $a!$ and $a?$ to denote broadcasting and synchronizing, respectively, over a channel $a \in Channels$. We also introduce urgent and committed locations s.t. $L^i$ is partitioned into disjoined sets *Standard*, *Urgent* and *Committed*. A network of TAs can thus do[1]:

– A delay transition $(l^1, \ldots, l^N, v) \xrightarrow{\tau} (l^1, \ldots, l^N, v')$ for some $\tau \in \mathbb{R}_{\geq 0}$ iff for all $1 \leq i \leq N$, $(l^i, v) \xrightarrow{\tau} (l^i, v'')$, $l^i \in Standard$ and $v' = v + \tau$.
– A discrete transition $(l^1, \ldots, l^i, \ldots, l^N, v) \xrightarrow{a} (l^1, \ldots, l^{i\prime}, \ldots, l^N, v')$ iff for some $1 \leq i \leq N$, $a \in Actions$, $(l^i, v) \xrightarrow{a} (l^{i\prime}, v'')$ and either $l^i \in Committed$ or for all $1 \leq j \leq N$, $l^j \notin Committed$. For all $1 \leq j \leq N$, if $j = i$ then $v'(x) = v''(x)$ for all $x \in X^j$ else $v'(x) = v(x)$ for all $x \in X^j$.
– A broadcast transition $(l^1, \ldots, l^N, v) \xrightarrow{a!} (l^{1\prime}, \ldots, l^{N\prime}, v')$ iff $a \in Channels$ and for some $1 \leq i \leq N$, $(l^i, v) \xrightarrow{a!} (l^{i\prime}, v'')$ and $v'(x) = v''(x)$ for all $x \in X^i$. For all $1 \leq j \leq N$ where $i \neq j$, if $(l^j, v) \xrightarrow{a?} (l^{j\prime\prime}, v''')$ then $l^{j\prime} = l^{j\prime\prime}$ and $v'(x) = v'''(x)$ for all $x \in X^j$ or else $l^{j\prime} = l^j$ and $v'(x) = v(x)$ for all $x \in X^j$. Also, either $l^i \in Committed$ or for all $1 \leq j \leq N$, $l^j \notin Committed$.

As two-process synchronization will not be used, we only define broadcast synchronization and whenever we refer so synchronizations, these are assumed to be broadcast synchronizations.

[1] This simplified semantics does not account for the case where a receiving transition may block a broadcast transition with an invariant, however, this case will not be encountered in this paper and is thus omitted for simplicity.

### 2.2 Timed Game Automata

We now continue to define Timed Game Automata.

**Definition 2 (Timed Game Automaton).** *A Timed Game Automaton (TGA) is a Timed Automaton* $(L, l_0, Act, X, E, Inv)$ *where the set of actions Act is partitioned into disjoined sets of controllable actions* $Act_c$ *and uncontrollable actions* $Act_u$*.*

Given a TGA $A = (L, l_0, \mathrm{Act}_c, \mathrm{Act}_u, X, E, Inv)$, a run over $A$ is an infinite sequence of states $r = (s_0, v_0), (s_1, v_1), \ldots$ where for all $i \in \mathbb{N}_0$, $(s_i, v_i) \rightarrow (s_{i+1}, v_{i+1})$. Throughout this paper we say that $\bullet$ is a special event which initializes the system to the initial state and only appears in the first term of a timed word. A timed word over a TGA $A = (L, l_0, \mathrm{Act}_c, \mathrm{Act}_u, X, E, Inv)$ is a sequence $\sigma = (t_0, e_0, s_0), (t_1, e_1, s_1), \ldots$ of elements from $(\mathbb{R}_{\geq 0} \times (\mathrm{Act}_c \cup \mathrm{Act}_u) \times L)$ where $t_0 = 0$, $s_0 = l_0$, $e_0 = \bullet$, for all $i \geq 0$ $t_i \leq t_{i+1}$ and there is a run $r = (s_0, v_0), (s_1, v_1), \ldots$ over $A$ s.t. if $t_{i+1} > t_i$ then $v_{i+1} = v + (t_{i+1} - t_i)$. We use $TW(A)$ to denote the set of all timed words over $A$.

We can now continue to define a control strategy for a TGA.

**Definition 3 (Control Strategy).** *Given a TGA* $A = (L, l_0, Act_c, Act_u, X, E, Inv)$*, a (memoryless) control strategy is a function* $f : L \times \mathbb{R}_{\geq 0}^{X} \rightarrow Act_c \cup \{wait\}$ *where wait means "do nothing".*

We define a TGA $A = (L, l_0, \mathrm{Act}_c, \mathrm{Act}_u, X, E, Inv)$ under a strategy $f$, denoted $A \restriction f$, as a restriction to the set of available transitions in the LTS defined by $A$ s.t. for any state $(l, v)$ where $f(l, v) = a$, if $a \in \mathrm{Act}_c$ then $(l, v) \xrightarrow{a} (l', v')$ and otherwise $(l, v) \stackrel{b}{\nrightarrow}$ for all $b \in \mathrm{Act}_c$.

Given a set of winning states and a set of losing states defined by a reachability or safety objective, a control strategy is said to be *winning* if the TGA under this strategy will always reach a winning state and never reach a losing state, no matter which uncontrollable actions are taken.

We can now define the synthesis problem.

**Definition 4 (Synthesis Problem).** *Given a TGA A and a set of winning and losing states, construct a winning control strategy or report it does not exist.*

## 3 Home Automation Systems

Home automation systems consist of a large variety of devices, each providing one or more services. Some of these services provide sensor inputs (e.g. switch clicks, temperature readings, light level readings, movement detection, window state) and are said to be *uncontrollable*, as they only output an observation of the system. Other services also control the state of some physical device (e.g. thermostats, lamps) and are said to be *controllable* as the state of these can be observed, but also changed (programmatically). A device may provide both controllable and uncontrollable services, e.g. a thermostat with a built-in

temperature sensor. Figure 1 shows an example of two uncontrollable services (a switch, top left and a thermometer, bottom) and a controllable service (a lamp, top right) and their state-space.

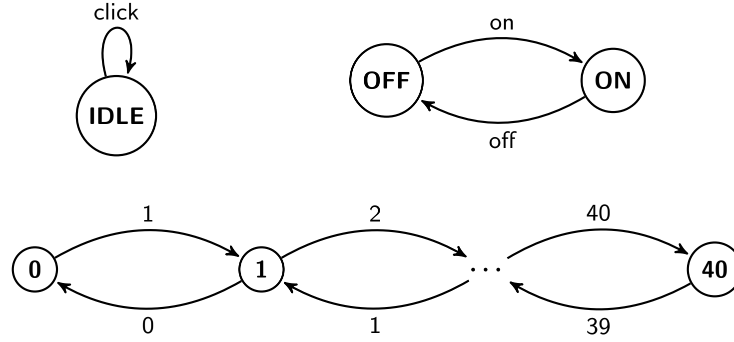


Fig. 1: Example of two uncontrollable services (a switch, top left and a thermometer, bottom) and a controllable service (a lamp, top right) and their state-space.

Home automation allows these services to interact with one another in order to exchange information. This information can be used to make decisions on behavior based on the global state of the system and is used for devices to operate in an optimal manner (e.g. minimizing energy consumption). An example of this is a system which turns off the heat when a window is open. This requires an exchange of information between the window sensor and the thermostat.

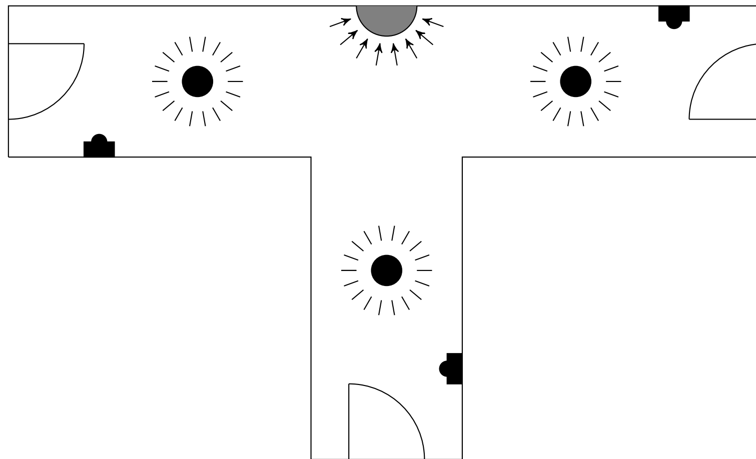

Fig. 2: Home automation scenario. Black boxes are switches, black circles are lamps and the gray semi-circle is a motion sensor.

Figure 2 shows an example of a home automation setup in a hallway. In this scenario, we have seven service; three lamps (controllable), three switches (uncontrollable) and a motion sensor (uncontrollable).

### 3.1 System Behavior

The aim in home automation is for the system of various services to interact and cooperate to meet some desired control objective. Figure 3 provides an overview of the general setup in home automation. A controller has access to information about controllable and uncontrollable services and may use this information to dictate behavior of controllable services (in order to meet the control objective). Dually, the environment is affected by the state of controllable and uncontrollable services and dictates behavior of uncontrollable services (this covers any change to the system that is not controlled by the controller, e.g. user input or physical effects on service states). The behavior of the environment is in general assumed to be unknown as only the effect on uncontrollable services can be observed. In Section 10.1 we consider an extension to the scenario where information about the environment is known.

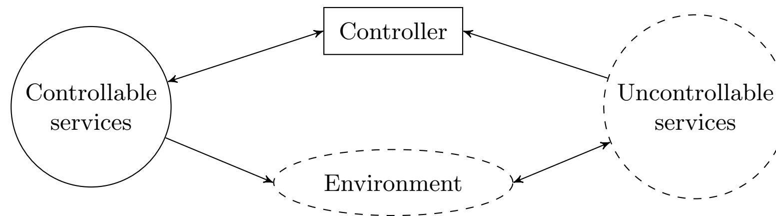


Fig. 3: General setup in home automation.

### 3.2 HomePort

A practical issue in home automation systems is the large variety of incompatible communication protocols used for communication between devices. To overcome this, the open-source project HomePort [14] has been developed. HomePort is a home automation middleware which aggregates home automation devices and provides a common interface for interaction with device services via HTTP requests. This allows for an abstract view of a home automation system as a heterogenous network of services and eliminates the issue of incompatible communication protocols. Besides from observing and changing states of services, HomePort also facilitates monitoring of the system via an event subscription service.

Figure 4 shows the general setup for a system based on the HomePort platform with a centralized controller component. While system control could be

handled in a distributed manner, due to the necessity of middleware software for service interaction, as well as limited computational power and high demands for a minimal power consumption of devices, a centralized controller component is commonly desirable.

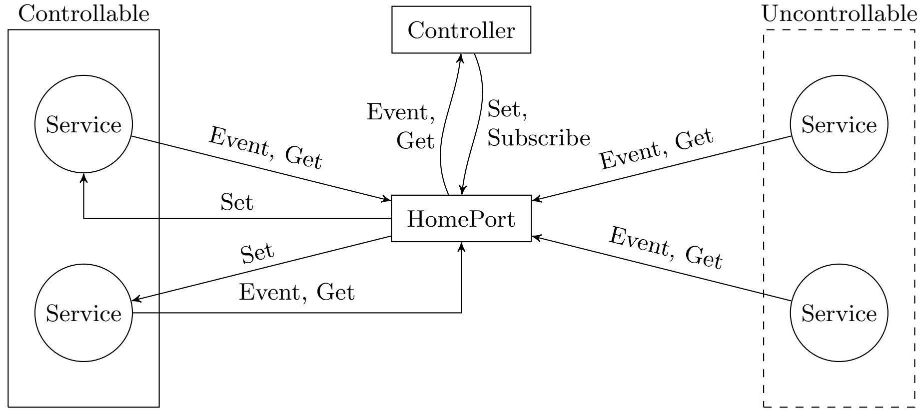


Fig. 4: Specialization of the general home automation setup in Figure 3 based on the HomePort platform. Interaction with the environment is abstracted away, as only state of uncontrollable services is observable in the system.

**State Valuation** HomePort represents service states as an integer value from a predefined interval, e.g. a lamp has the state interval $[0, 1]$ where the state 0 is interpreted as *off* and the state 1 is interpreted as *on* and a switch has the state interval $[0, 0]$ where the state 0 is interpreted as *idle*. In other services, e.g. a thermometer, the interpreted valuation is equivalent to the integer value and is referred to as a *reading*. We denote the non-negative integer valuation of a state $s$ as $\nu(s)$. State valuations define a complete ordering of states by the relation of integers, e.g. for a lamp where $\nu(\mathit{off}) = 0, \nu(\mathit{on}) = 1$, then $\mathit{off} < \mathit{on}$. We will later use this to express relational expression over states.

We notice that due to the discretization of readings, no real-valued data is considered.

### 3.3 System Assumptions

We make some assumption about the behavior of home automation systems:

1. **Instant communication.** We assume communication in a home automation system to be instant, i.e. it takes no time to change or obtain the state of a service. This assumptions is justified by network communication being practically instant and thus that any delay will be negligible.

2. **Non-zero delay between state changes.** We assume that a non-zero delay should always occur between changing the state of a controllable service. This assumption reflects that no physical operation is in fact completely instant in a continuous timed domain.
3. **Non-Zeno behavior.** For similar reasons as above, we assume that the system will not by itself show Zeno behavior, i.e. will not change state infinitely often in a finite amount of time.

These assumptions should be reflected also in a control strategy, that is, should not be violated under any control strategy.

### 3.4 System Definition

We now continue to formally describe a home automation system.

**Definition 5 (Service and System).** *A* service *is a triple $\mathcal{S} = (S, s_0, E)$ where $S$ is a set of service states, $s_0 \in S$ is the initial state and $E$ is a set of edges $E \subseteq (S \times S)$. A* system *is a pair $\mathbb{S} = (\mathbb{S}_c, \mathbb{S}_u)$ where $\mathbb{S}_c = \{\mathcal{S}_1, \ldots, \mathcal{S}_m\}$ is a set of $m \in \mathbb{N}_0$ controllable services, $\mathbb{S}_u = \{\mathcal{S}_{m+1}, \ldots, \mathcal{S}_n\}$ is a set of $n - m \in \mathbb{N}_0$ uncontrollable services s.t. $\mathbb{S}_c \cap \mathbb{S}_u = \emptyset$ and for any $\mathcal{S} = (S, s_0, E) \in (\mathbb{S}_c \cup \mathbb{S}_u)$, $S \cap S' = \emptyset$ for all $(S', s'_0, E') \in (\mathbb{S}_c \cup \mathbb{S}_u) \setminus \mathcal{S}$.*

As we assume all edges that are followed will trigger a corresponding event, we will use the phrase edge and event interchangeably. For a service $\mathcal{S} = (S, s_0, E)$, we refer to the initial state $\mathcal{S}^I = s_0$, the set of states $\mathcal{S}^S = S$ and the set of edges $\mathcal{S}^E = E$.

Consider e.g. the switch and lamp examples from earlier. We can specify these services as follows:

$$\begin{aligned} Lamp &= (\{on, off\}, off, \{(on, off), (off, on)\}) \\ Switch &= (\{idle\}, idle, \{(idle, idle)\}) \end{aligned}$$

A system consisting only of a switch and a lamp would thus be:

$$\mathbb{S} = (\{Lamp\}, \{Switch\})$$

**System Execution** Given a system $\mathbb{S} = (\mathbb{S}_c, \mathbb{S}_u)$ where $\mathbb{S}_c = \{\mathcal{S}_1, \ldots, \mathcal{S}_m\}$, $\mathbb{S}_c = \{\mathcal{S}_{m+1}, \ldots, \mathcal{S}_n\}$ and let $CONF(\mathbb{S}) = \bigtimes_{i=1}^{n} \mathcal{S}_i^S$ be the set of configurations. We say that a state $s \in (s_1, \ldots s_n)$ iff $s = s_i$ for some $1 \leq i \leq n$. We now define the set of infinite timed words $TW(\mathbb{S})$ as the set of sequences $\sigma = (t_0, e_0, c_0), (t_1, e_1, c_1), \ldots$ of elements from $(\mathbb{R}_{\geq 0} \times \mathcal{E}(\mathbb{S}) \times CONF(\mathbb{S}))$ where

- $t_0 = 0$,
- $e_0 = \bullet$,
- $c_0 = (\mathcal{S}_1^I, \ldots, \mathcal{S}_n^I)$ is the initial configuration, and

– $\mathcal{E}(\mathbb{S}) = \{e \in \mathcal{S}^E \mid \mathcal{S} \in (\mathbb{S}_c \cup \mathbb{S}_u)\}$ is the set of events. We also define the set of controllable and uncontrollable events $\mathcal{E}_c(\mathbb{S}) = \{e \in \mathcal{S}^E \mid \mathcal{S} \in \mathbb{S}_c\}$ and $\mathcal{E}_u(\mathbb{S}) = \{e \in \mathcal{S}^E \mid \mathcal{S} \in \mathbb{S}_u\}$, respectively.

We require that for all $i \in \mathbb{N}_0$, $t_i \leq t_{i+1}$ and that if $e_{i+1} = (s, s')$, then $s \in c_i$, $s' \in c_{i+1}$ and for all $s'' \in c_i$, if $s'' \neq s$ then $s'' \in c_{i+1}$ (recall that a configuration contains exactly one state for each service, hence the state of all but $s$ is preserved). We impose some assumptions to the timed words:

1. We maintain an ordering of events that happen at the exact same time s.t. uncontrollable events always occur before controllable events in a timed word if they both occur at the same time. That is, given a system $\mathbb{S}$ and a timed word $\sigma = (t_0, e_0, c_0), (t_1, e_1, c_1), \dots \in TW(\mathbb{S})$, if $e_i \in \mathcal{E}_c(\mathbb{S})$ then for all $j > i$, if $t_i = t_j$ then $e_j \in \mathcal{E}_c(\mathbb{S})$.
2. Based on the second assumption in Section 3.3 we observe that for any $i \in \mathbb{N}_0$, if $e_i \in \mathcal{S}_j^E$ for some $1 \leq j \leq n$ then for all $k \in \mathbb{N}_0$ where $t_i = t_k$, $e_k \notin \mathcal{S}_j^E$.

Returning to our system with a switch and a lamp, one possible (and the initial) configuration of the system is:

$$(Lamp.off, Switch.idle) \in CONF(\mathbb{S})$$

where the dot-notation $\mathcal{S}.s$ in this context means the state $s \in \mathcal{S}^S$ for service $\mathcal{S} \in (\mathbb{S}_c \cup \mathbb{S}_u)$. A possible timed word over the system is:

$$\sigma = (0, \bullet, (Lamp.off, Switch.idle)), (3.14, Switch.click, (Lamp.off, Switch.idle)), \\ (3.14, Lamp.on, (Lamp.on, Switch.idle)), \dots \in TW(\mathbb{S})$$

**Strategy** We can now define a *strategy* for a system $\mathbb{S} = (\mathbb{S}_c, \mathbb{S}_u)$ as a function $\gamma : TW \times \mathbb{N} \to (\mathbb{R}^{\infty}_{\geq 0} \times \mathcal{E}_c(\mathbb{S}))$ from a finite prefix of a timed word to a (possibly infinite) delay and a controllable action. The formal semantics of a system under a strategy $\mathbb{S} \upharpoonright \gamma$ is a restriction to the timed words $TW(\mathbb{S} \upharpoonright \gamma) \subseteq TW(\mathbb{S})$ s.t. for any finite prefix $(\sigma, i) = (t_0, e_0, c_0), \dots, (t_{i-1}, e_{i-1}, c_{i-1})$ of length $i \in \mathbb{N}$ of a timed word $\sigma \in TW(\mathbb{S})$, if $\gamma(\sigma, i) = (t, e)$ then either $e_i = e$ and $t_i = t_{i-1} + t$ or $e_i \in \mathcal{E}_u(\mathbb{S})$ and $t_i \leq t_{i-1} + t$.

**Service Expressions** We facilitate relational expressions over services and integers. Given a system $\mathbb{S} = (\mathbb{S}_c, \mathbb{S}_u)$ where $\mathcal{S} \in (\mathbb{S}_c \cup \mathbb{S}_u)$, let $Rel(\mathbb{S})$ denote the set of relational expressions on the form $\mathcal{S} \bowtie n$, where $\bowtie\ \in \{\leq, <, =, >, \geq\}$ and $n \in \mathbb{N}_0$. The semantics is defined by the satisfaction relation s.t. $c \models \mathcal{S} \bowtie n$ iff the state $s \in \mathcal{S}^S$ in $c$ satisfies $\nu(s) \bowtie n$. In this context, the dot-notation $\mathcal{S}.s$ represents that $s \in \mathcal{S}^S$ is the current state of $\mathcal{S}$ in $c$. The satisfaction relation is extended in the natural way to boolean combinations of relational expressions. We will use these expressions in behavior specifications.

**Data Services** We notice that so far, we have only considered services with a one-to-one mapping to physical device services. However, this is not a requirement. We define *data services* as services that have no effect on and is not affected by a physical device. Given a system $\mathbb{S} = (\mathbb{S}_c, \mathbb{S}_u)$, a data service $\mathcal{S} \in (\mathbb{S}_c \cup \mathbb{S}_u)$ is a special service where $\mathcal{S}^S = \mathbb{N}_0$, $\mathcal{S}^E = \mathbb{N}_0 \times \mathbb{N}_0$ and $\mathcal{S}^I = 0$. In any system, we assume access to an unbounded number of controllable data services. We can e.g. use data services to serve as memory for use in behavior specification as we will see in Section 5.

## 4 Specification Logic

We now continue to present a logic for specifying behavior for a Home Automation system. The logic is designed to be easy to use, compact and readable. It is based on principles similar to the ECA pattern [8]. We recall that in the ECA pattern, events are specified to trigger actions provided some condition is satisfied. We use a similar pattern, but focus on specifications over states rather than actions. We argue that this pattern allows for a more simple and natural specifications in a home automation context, primarily because only the desired states of the system must be specified, rather than (complex) actions and timings to obtain the desired system state. Section 6 explores one benefit of specifications over states rather than actions.

The syntax of the specification logic is designed to aim for human readability by forming (almost) English sentences. It is also designed to be easy to specify from a point-and-click user-interface as we will see in Section 8 (in contrary to commonly used textual interfaces).

### 4.1 Specification Logic

The specification logic is defined by the following syntax:

$$\begin{aligned}
\varphi &::= \mathit{true} \mid \mu \mid \neg\varphi \mid \varphi \ \wedge \ \varphi \\
\psi &::= \textbf{on } E \textbf{ provided } \varphi \textbf{ satisfy } \varphi \textbf{ starting in } [s,e] \textbf{ for } d \textbf{ unless } E' \mid \\
&\qquad \varphi \mid \psi \ \wedge \ \psi
\end{aligned}$$

where $s, e \in \mathbb{N}_0$, $s \leq e$, $d \in \mathbb{N}^\infty$, $\mu \in \mathit{Rel}(\mathbb{S})$ and $E, E' \subseteq \mathcal{E}(\mathbb{S})$ are sets of events. For simplicity, we define $\varphi_1 \vee \varphi_2$ as $\neg(\neg\varphi_1 \ \wedge \ \neg\varphi_2)$ and by

$$\begin{aligned}
&\textbf{on } E \textbf{ provided } \varphi \\
&\quad \textbf{satisfy } \varphi_1 \textbf{ starting in } [s_1, e_1] \textbf{ for } d_1 \ \wedge \\
&\quad \textbf{satisfy } \varphi_2 \textbf{ starting in } [s_2, e_2] \textbf{ for } d_2 \\
&\ \textbf{unless } E'
\end{aligned}$$

we understand

$$\begin{aligned}
&\textbf{on } E \textbf{ provided } \varphi \textbf{ satisfy } \varphi_1 \textbf{ starting in } [s_1, e_1] \textbf{ for } d_1 \textbf{ unless } E' \ \wedge \\
&\textbf{on } E \textbf{ provided } \varphi \textbf{ satisfy } \varphi_2 \textbf{ starting in } [s_2, e_2] \textbf{ for } d_2 \textbf{ unless } E'
\end{aligned}$$

The point-wise semantics for the logic is defined over a system $\mathbb{S}$ s.t. for any run $\sigma = (t_0, e_0, c_0), (t_1, e_1, c_1), \dots \in TW(\mathbb{S})$, the satisfaction relation $\sigma, i \models \psi$, meaning the suffix of $\sigma$ starting at term $i$ satisfies $\psi$, is defined inductively as follows:

$\sigma, i \models true$

$\sigma, i \models \mu$ iff $c_i \models \mu$

$\sigma, i \models \neg\varphi$ iff $\sigma, i \not\models \varphi$

$\sigma, i \models \psi_1 \ \wedge \ \psi_2$ iff $\sigma, i \models \psi_1$ and $\sigma, i \models \psi_2$

$\sigma, i \models$ **on** $E$ **provided** $\varphi$ **satisfy** $\varphi'$ **starting in** $[s, e]$ **for** $d$ **unless** $E'$ iff

- $e_i \notin E$, or
- $\sigma, i-1 \not\models \varphi$, or
- $e_i \in E$ and $\sigma, i-1 \models \varphi$ and there exists a number $k$ s.t. $e \geq t_k - t_i \geq s$ and for all $j \geq k$ either
  a) $\sigma, j \models \varphi'$, or
  b) $t_j - t_k \geq d$, or
  c) there exists a number $l$ where $i < l \leq j$ s.t. $e_l \in E'$

We say that $\sigma \models \psi$ iff $\sigma$ is infinite and $\sigma, i \models \psi$ for any suffix $i \in \mathbb{N}_0$. We define that $TW(\mathbb{S}, \psi) = \{\sigma \in TW(\mathbb{S}) \mid \sigma \models \psi\}$. We also introduce the following notations for a specification $\psi = \psi_1 \ \wedge \ \psi_2 \dots$:

- We refer to $\psi_i$ on the form **on** $E$ **provided** $\varphi$ **satisfy** $\varphi'$ **starting in** $[s, e]$ **for** $d$ **unless** $E'$ as a rule, $Prop(\psi_i) = \varphi'$ as the proposition and $Prec(\psi_i) = \varphi$ as the precondition. We denote the set of all rules $Rules(\mathbb{S})$.
- We refer to $\psi_i$ on the form $\varphi$ as an invariant and to $Prop(\psi_i) = \varphi$ as the proposition and define that $Prec(\psi_i) = \emptyset$. We denote the set of all invariants $Invariants(\mathbb{S})$.

### 4.2 Control Strategy Synthesis Problem

Given a system $\mathbb{S}$ and a specification $\psi$ we can now define the control strategy synthesis problem.

**Definition 6 (Control Strategy Synthesis Problem).** *Given a system* $\mathbb{S}$ *and a specification* $\psi$*, the control strategy synthesis problem* $CSSP(\mathbb{S}, \psi)$ *is to find a control strategy* $\gamma$ *s.t.* $TW(\mathbb{S} \upharpoonright \gamma) \subseteq TW(\mathbb{S}, \psi)$ *or report it does not exist.*

## 5 Specifications

We will now present some specifications for our example-system consisting of a switch and a lamp and for the hallway scenario in Figure 2.

### 5.1 Examples

**Simple** A basic behavior for our example-system is for the lamp to toggle on and off when the switch is pressed. We can specify this property as follows.

```
# Turn on lights
on Switch.click provided Lamp.off satisfy Lamp.on
 starting in [0,0] for ∞ unless Switch.click

# Turn off lights
on Switch.click provided Lamp.on satisfy Lamp.off
 starting in [0,0] for ∞ unless Switch.click
```

**Timer** One could also image that it would be of interest to turn on the lamp when the switch is clicked and then to automatically turn it off again after a period of time. We can specify this property as follows.

```
# Turn on lights for 20s
on Switch.click provided true satisfy Lamp.on
 starting in [0,0] for 20 unless ∅

# Turn off lights
on Switch.click provided true satisfy Lamp.off
 starting in [20,20] for ∞ unless Switch.click
```

What should be noticed from this example is that the rule for turning off the lamp is not similar to the rule for turning it on, an issues which makes it nontrivial to provide a complete specification for how the system should behave. The problem becomes even more urgent in the following example.

**"Follow Me"** Consider now the hallway scenario, which consists of three switches, three lamps and a motion sensor like we saw in Figure 2. Alternatively to the previous examples of behavior, which could easily be extended to the larger system, one might rather want for the lamp closest to the clicked switch to be turned on first and then wait before turning on the lamps further away. The idea is that the light then "follows" the user as she walks through the hallway, rather than keeping all the lights on at all times (and thereby saving energy). We can specify this property as follows.

```
# Turn on lights
on Switch1.click provided true
   satisfy Lamp1.on starting in [0,0] for 10 ∧
   satisfy Lamp2.on && Lamp3.on starting in [8,8] for 12
 unless ∅
```

```
on Switch2.click provided true
   satisfy Lamp2.on starting in [0,0] for 10 ∧
   satisfy Lamp3.on && Lamp1.on starting in [8,8] for 12
unless ∅

on Switch3.click provided true
   satisfy Lamp3.on starting in [0,0] for 10 ∧
   satisfy Lamp1.on && Lamp2.on starting in [8,8] for 12
unless ∅

# Turn off lights - complicated!
```

While the rules for turning on the lamps are relatively simple to specify, the corresponding specification for when to turn off the lights again is not trivial and consists of a number of cases that must be considered correctly and completely. A different approach is to use game theory to synthesize the dual behavior of returning the system to a default state (here assumed to be when the lamps are off). This approach will be explored in Section 6 and thus we will not concern ourself with turning off the lamps in the remaining examples.

**Motion Sensor** Alternatively to using the switches, one might want to use the motion sensor output to turn on lamps (in this case for 10 seconds) when movement is detected. The motion sensor service can be described similarly to a switch:

$$MotionSensor = \{\{idle\}, idle, \{(idle, idle)\}\}$$

where the edge is followed every time motion is detected, triggering the event *motion*. We can now specify this property as follows.

```
on MotionSensor.motion provided true satisfy Lamp1.on ∧
Lamp2.on ∧ Lamp3.on starting in [0,0] for 10 unless ∅
```

**Motion Sensor with Override** Finally, one might wish to be able to change the behavior of the system e.g. at day and night, so that in the day, the more energy efficient "Follow Me" lights are used, but in the night, the motion sensor output should activate the lights. We can implement this changing of operation mode using data services. We assume that we want to double-click to change the behavior of the system. We use two data services; *CC* (Click Count) which we will use to detect double-clicks and *SD* (Sensor Disabled) to store if the motion sensor is disabled ($SD = 1$) or enabled ($SD = 0$). We can specify the enabling and disabling of the motion sensor as follows.

```
# Count number of clicks
on Switch.click provided true
satisfy CC ≥ 1 starting in [0,0] for 1 unless ∅
```

**on** *Switch.click* **provided** *CC.1*
**satisfy** $CC = 2$ **starting in** $[0, 0]$ **for** 1 **unless** $\emptyset$

# Toggle sensor
**on** *CC.2* **provided** *SD.0*
**satisfy** *SD.1* **starting in** $[0, 0]$ **for** $\infty$ **unless** *CC.2*

# Motion sensor
**on** *MotionSensor.motion* **provided** *SD.0* **satisfy** *Lamp1.on* $\wedge$ *Lamp2.on* $\wedge$ *Lamp3.on* **starting in** $[0, 0]$ **for** 10 **unless** $\emptyset$

The specification can be combined with one of the other specifications (with added precondition $SD.1$) to specify behavior when the motion sensor is disabled. These examples show how behavior generally can be expressed using only few, simple rules, and that more complex rules can be implemented with the aid of data services.

### 5.2 Patterns

We notice that when specifying behavior for a larger system, some specifications will likely have recurring patterns. In particular we notice two kinds of patterns:

*Service Groups* Specifications commonly impose an identical behavior for multiple services, e.g. if we apply the simple light behavior example to the hallway, triggers will be click events from all switches and the proposition will be to keep all lamps turned on. We can in this case introduce an abstraction to specify behavior about a group of services based on e.g. service types rather than by listing explicit service names.

*Equivalent Rules* Commonly similar specifications are desirable in multiple rooms. For instance, the simple light behavior might be desirable in most rooms in a house. For easy specifications, a rule should be applicable to multiple rooms using general service groups.

We propose using specification patterns to provide more general and reusable specifications. The game theoretic approach ensures adaption of general specifications to a given system. Generality of rules using patterns also facilitates redistribution and migration of specifications between systems, allowing end-users to obtain more complex behavior specifications with a minimum of effort from a repository of specification patterns.

## 6 Underspecified Behavior

In the previous examples of specifications, we exploit the fact that the system will return to a default state (where the light are off). We notice that this

allows us to specify only a subset of the actual behavior and avoid complex case handling for correctly returning the system to a default state. We refer to this as underspecification of the desired system behavior.

We argue that traditionally, unintentionally underspecified behavior specifications are an obvious source of bugs in controller behavior. For instance, consider the "Follow Me" lights example in Section 5. While it is easy to specify that lamps should turn on, making sure lamps are turned off under the right conditions is less trivial and could easily lead to missed cases and thus incorrect behavior.

The proposed specification logic in combination with game theory has the great benefit of eliminating this issue, as only the desired state (the lights should be on) is specified and additional practical issues (the lights should be turned off) is automatically computed in the control strategy synthesis process. Not only does this approach allow for smaller and less complex specifications, it also ensures that all cases are handled correctly, eliminating a great source of controller behavior bugs.

To support underspecified behavior for a system $\mathbb{S}$ and a specification $\psi$, it is however not sufficient to solve the control strategy synthesis problem $\mathit{CSSP}(\mathbb{S}, \psi)$; while this will find *some* strategy satisfying the specification, there is by default no preference to one state of a service (lamp turned off) to another state (lamp turned on). Consequently, there is no incentive for the system to return to a particular state once a rule is no longer active. As an example, we see in the presented examples (without explicit rules to turn off the lamps) that a control strategy satisfying all specifications is to turn on the lamps and never turn them off again! This is clearly not desirable, so to circumvent this we introduce a notion of service state costs.

### 6.1 Service State Cost

Given a system $\mathbb{S} = (\mathbb{S}_c, \mathbb{S}_u)$ where $\mathit{States} = \{\mathcal{S}^S \mid \mathcal{S} \in (\mathbb{S}_c \cup \mathbb{S}_u)\}$, we introduce a cost-function $\mathcal{C} : \mathit{States} \to \mathbb{N}_0$ assigning a cost to each service state. The cost of a configuration $c = (s_1, \ldots, s_n)$ is thus $\mathcal{C}(c) = \sum_{i=1}^{n} \mathcal{C}(s_i)$. If the cost is not explicit specified for a state $s$, we assume $\mathcal{C}(s) = \nu(s)$.

Recall from Section 3.2 that for lamps, $\nu(\mathit{off}) = 0$, $\nu(\mathit{on}) = 1$ and for switches as well as the motion sensor, $\nu(\mathit{idle}) = 0$. We then assume the cost-function is

$$\mathcal{C}(s) = \nu(s).$$

We can now see e.g. that for the hallway scenario:

$$
\begin{aligned}
\mathcal{C}((&\mathit{Lamp1.on}, \mathit{Lamp2.on}, \mathit{Lamp3.on}, \mathit{Switch1.idle},\\
&\quad \mathit{Switch2.idle}, \mathit{Switch3.idle}, \mathit{MotionSensor.idle})) = 3\\
\mathcal{C}((&\mathit{Lamp1.on}, \mathit{Lamp2.on}, \mathit{Lamp3.off}, \mathit{Switch1.idle},\\
&\quad \mathit{Switch2.idle}, \mathit{Switch3.idle}, \mathit{MotionSensor.idle})) = 2\\
\mathcal{C}((&\mathit{Lamp1.off}, \mathit{Lamp2.off}, \mathit{Lamp3.off}, \mathit{Switch1.idle},\\
&\quad \mathit{Switch2.idle}, \mathit{Switch3.idle}, \mathit{MotionSensor.idle})) = 0
\end{aligned}
$$

### 6.2 Cost-Based Local Optimality

We can now express local optimality as a task of finding a winning strategy which always selects the configuration that satisfies the propositions of all invariants and of all active rules with minimal cost. That is, a winning strategy $\gamma$ has minimal cost if for any other winning strategy $\gamma'$ and for any timed word $\sigma = (t_0, e_0, c_0), \ldots, (t_i, e_i, c_i), \cdots \in TW(\mathbb{S} \restriction \gamma)$, it holds for any $i \in \mathbb{N}_0$ that if

- $\gamma(\sigma, i) = (t_{i+1}, e_{i+1})$ where $t_{i+1} \neq \infty$, and
- $\gamma'(\sigma, i) = (t'_{i+1}, e'_{i+1})$ where $t'_{i+1} \neq \infty$,

then $\mathcal{C}(c_{i+1}) \leq \mathcal{C}(c'_{i+1})$.

The cost of states thus defines a default system configuration, which the system will go to when no rules are active, namely the configuration with the lowest cost. Returning to the example-system, the default configuration must then be the configuration where the lamp is in the *off*-states. If we consider again the specifications in Section 5, we see that a strategy $\gamma$ with minimal cost will always turn off the lamps automatically. Relying on this makes explicit specifications for when to turn off lamps unnecessary.

## 7 Control Strategy Synthesis

We now present a translation from the control strategy synthesis problem to the synthesis problem for TGA in form of a UPPAAL TiGa model. UPPAAL TiGa uses solid lines to represent edges with controllable actions and dashed line to represent edges with uncontrollable actions. Initial locations are indicated with a circle and urgent locations are indicated with a rounded *U*. A full introduction to UPPAAL TiGa can be found in [2].

### 7.1 Translation

The translation consists of three main templates; one to represent services, one to represent rules and one to enforce correct behavior of the system and synthesized controller. Table 1 shows global variables used in the templates, their type and a description of their semantics. We will use $TGA(\mathbb{S}, \psi)$ to refer to the TGA defined by the translation presented below for a system $\mathbb{S}$ and a specification $\psi$.

| **Name** | **Type** | **Description** |
|---|---|---|
| Variables | | |
| x | clock | Clock for the control component. |
| cost | integer | The sum of service state costs for the current configuration of the system. |
| activeRules | array of integers | Given a rule id, returns integer representing the number of active rule templates. |
| controllable_events | selection | Set of controllable events. |
| Channels | | |
| event | array of broadcast channels | Broadcasts event, either from state change or by command of the control component. |
| Functions | | |
| ruleSatisfied | bool | Evaluates to true if the configuration of the system satisfies a given $\varphi$ expression. |
| calcCost | integer | Returns sum of service state costs for the provided configuration of the system. If no argument is provided, the current configuration of the system is assumed. |
| validState | bool | Returns true iff a given configuration satisfies $Prop(\psi)$ for all rules where $activeRules > 0$ and for all invariants. If no argument is provided, the current configuration of the system is assumed. |
| currentReplica | integer | Given a rule id, returns the number of the next available replica of the rule template to use. |
| updateCurrentReplica | void | Given a rule id, updates the current replica to the next available replica. |

Table 1: Global variables and functions used in translation.

**Service Templates** Service templates encode the states and edges of services. Figure 5 shows the general template structure for a controllable service (left) and uncontrollable service (right).

Controllable service templates have a location for each state and edges connecting locations corresponding to edges connecting states in the service. Edges are equipped with broadcast synchronizations on corresponding events and when followed, *cost* is updated to match the cost of the new configuration. Broadcasts for controllable services are performed by the control component.

Uncontrollable service templates are similar to controllable services with a few exceptions: The edge is uncontrollable and broadcasts a corresponding event, leaving firing of uncontrollable events to the environment. The edge is also equipped with the guard $x > 0$ which, in combination with the control component, assert the first assumption to timed words in Section 3.4, that there is an ordering of events in timed words so that uncontrollable events always occur before controllable events in a timed word if they both occur at the same time.

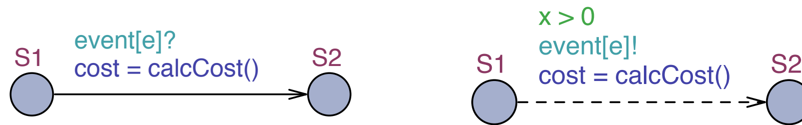


Fig. 5: Service templates.

**Control Component Template** Figure 6 shows the control component template. The template can broadcast controllable events to change the state of controllable services. The template consists of a location *ENVIRONMENT* where the environment may fire uncontrollable events and a location for each controllable services. Once the edge going out from the *ENVIRONMENT* location is followed, clock $x$ is reset, effectively preventing any more uncontrollable events to occur at this time (due to the guard on all edges of uncontrollable services), enforcing the first assumption to timed words in Section 3.4.

In the following locations, one controllable events may be broadcast for the corresponding controllable service or choose not to change anything (the *tau* action). This enforces the second assumption to timed words in Section 3.4 of only changing the state of a controllable service once at any time. As the intermediate locations are urgent, the template must return to the *ENVIRONMENT* location before time can progress. Uncontrollable events are reenabled as time progresses.

**Rule Template** Because UPPAAL TiGa synthesizes only memoryless strategies (action depends only on the state in contrary to actions depending on a prefix of a timed word), it is necessary to encode the state of active rules in the

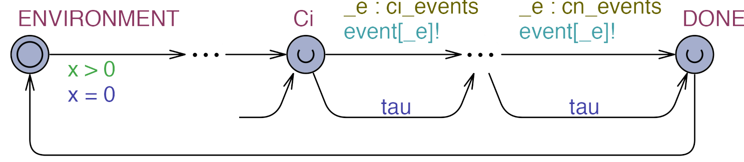


Fig. 6: Control component template.

TGA state. Figure 7 shows a UPPAAL TiGa template $R(id)$ for representing a rule:

$$\textbf{on } E \textbf{ provided } \varphi \textbf{ satisfy } \varphi' \textbf{ starting in } [s, e] \textbf{ for } d \textbf{ unless } E'$$

with id *id*. We assume initially an unbounded number of replicas of $R(id)$ s.t. a new replica is used for each occurrence of a triggering event, implemented using a unique *replica* value for each replica and the *currentReplica* function. In Section 7.2 we explore a finite bound for the required number of replicas.

Each replica has a private clock $y$. The template is initially in the location *IDLE*. When $e \in E$ occurs while $\varphi$ is satisfied, $y$ is reset and the next available replica of this rule goes to the location *ENABLED*. After $s$ time units, the transition to *ACTIVE* is enabled, at it must be followed at latest after $e$ time units due to the invariant. When followed, $y$ is reset and *activeRules* is incremented for the rule (effectively enforcing the proposition $\varphi'$ to be satisfied, as showed later in Fact 1). The template remains in the *ACTIVE* location for $d$ time units, after which it must follow the lower edge back to *IDLE*, resetting $y$ and decrementing *activeRules* for the rule.

From the *ENABLED* and *ACTIVE* location, the rule may also be cancelled by some $e \in E'$. From the *ENABLED* location, the clock is simply reset and the template returns to the *IDLE* location. From the *ACTIVE* location, the template goes to the *CANCELLED* location, resets $y$ and decrements *activeRules* for the rule. As *CANCELLED* is urgent, the template must return to the *IDLE* location before time can progress.

**Synthesis Problem** Given a translation using the templates above, a control strategy can be found by solving the corresponding safety game (expressed in UPPAAL TiGa syntax) for a system with one controllable and one uncontrollable service, where *c_service_states* are the set of states for the controllable service and *u_service_state* is the current state of the uncontrollable service.

```
control : A[]  ((x>0) imply (validState() &&
  forall(s : c_service_states)
    validState(s, u_service_state)  imply
      cost <= calcCost(s, u_service_state)))
```

In general, a nested forall-loop is added for each controllable service. The property asserts that 1) whenever $x > 1$, *validState* must return *true* and 2) for all

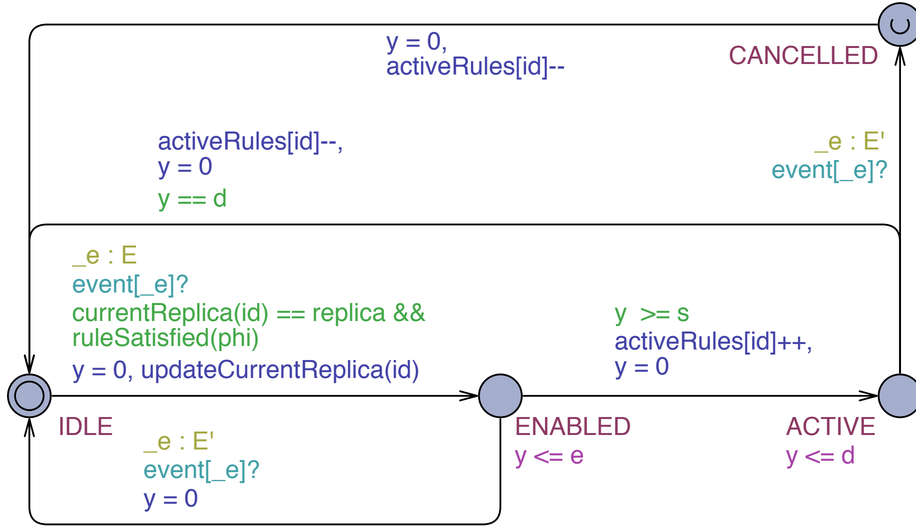


Fig. 7: Rule template $R(id)$ for rule with id $id$.

possible configurations of controllable services, no configuration with lower cost results in *validState* returning *true*. The latter enforces local optimality.

The *validState* function is implemented so that it returns *true* iff the proposition of all invariants and all rules where *activeRules* $> 0$ are satisfied. Let us e.g. consider a modified subset of the "Follow Me" hallway scenario specification with an added invariant:

$\neg(Lamp2.on \;\wedge\; Lamp3.off)$

**on** *Switch1.click* **provided** *true*
  **satisfy** *Lamp1.on* **starting in** $[0,0]$ **for** $10 \;\wedge$
  **satisfy** *Lamp2.on* **starting in** $[8,8]$ **for** $12$
**unless** $\emptyset$

We recall that the internal conjunction in the rule actually represents two rules. Let the id of the first of these rules be 0, the id of the second be 1 and assume a function *state* which returns the valuation of the current state of a given service (according to the valuation function). Then this specification results in the following *validState* function logic.

```
if(!(!(state(Lamp2) == 1 && state(Lamp3) == 0))){
  return false;
}
if(activeRules[0] > 0 && !(state(Lamp1) == 1)){
  return false;
}
```

```
if(activeRules[1] > 0 && !(state(Lamp2) == 1)){
    return false;
}
return true;
```

No further elaboration is provided for the construction of the *validState* function, as it trivially follows from structure of propositions and we simply postulate the following fact.

*Fact 1 (Correctness of* validState*).* Let $\mathbb{S}$ be a system, let $\psi = \psi_1 \wedge \ \ldots \ \wedge \psi_n$ be a specification and let $\sigma' = (t_0, e_0, c_0), \ldots, (t_i, e_i, c_i)$ be a finite prefix of a timed word $\sigma = (t_0, e_0, c_0), \ldots, (t_i, e_i, c_i), \cdots \in TW(\mathbb{S}, \psi)$. Then *validState* returns true for a configuration $c_i \in CONF(\mathbb{S})$ iff $c_i \models Prop(\psi_j)$ for all $1 \leq j \leq n$ where $\psi_j$ is active in $c_i$ for $\sigma'$.

### 7.2 Correctness

We now turn our attention to correctness of the presented translation. We first notice that the translation relies on an unbounded number of rule template replicas of a given rule template $R(id)$. However, we need a finite bound for the number of required replicas to solve the synthesis problem for TGA. Lemma 1 provides such finite bound.

**Lemma 1 (Finite Number of Rule Template Replicas).** *Let $\mathbb{S}$ be a system, let $\psi = \psi_1 \wedge \ \ldots \ \wedge \psi_n$ be a specification and let $TGA_F(\mathbb{S}, \psi)$ denote the translation above, but with only* $\mathrm{e}+\mathrm{d}$ *replicas of $R(i)$ for rule $\psi_i =$ **on** $E$ **provided** $\varphi$ **satisfy** $\varphi'$ **starting in** $[s, e]$ **for** $\mathrm{d}$ **unless** $E'$. Then there is a winning control strategy for the synthesis problem of $TGA(\mathbb{S}, \psi)$ iff there is a winning control strategy for the synthesis problem of $TGA_F(\mathbb{S}, \psi)$.*

*Proof.* We show this by showing that it holds in both directions.

**"$\Rightarrow$":** It follows from Fact 1 that *validState* returns true iff the propositions of all invariants are satisfied and the propositions of all rules where $activeRules > 0$ are satisfied. We notice that *activeRules* for a rule *id* is incremented exactly once when following the edge to the ACTIVE location and is subsequently decremented exactly once when leaving the ACTIVE location. As *activeRules* for *id* is not changed anywhere else, more replicas can only increase the frequency of $activeRules > 0$. As $TGA_F(\mathbb{S}, \psi)$ is equivalent to $TGA(\mathbb{S}, \psi)$, only with less replicas of $R(id)$, it follows that $TGA_F(\mathbb{S}, \psi)$ cannot increment *activeRules* for rule *id* more commonly than $TGA(\mathbb{S}, \psi)$ and thus cannot impose more restrictions in *validState* than those imposed in $TGA(\mathbb{S}, \psi)$.

**"$\Leftarrow$":** To show this direction we will first make two observations:

*Observation 1* Consider the problem of finding a winning control strategy $f$ for $TGA(\mathbb{S}, \psi)$. If no such strategy exists, it means that the environment has a winning strategy $f'$ s.t. eventually *validState* returns false for a non-singular period of time for any $\sigma \in TW(\mathbb{S} \upharpoonright f')$ as a result of a specific sequence of uncontrollable events. We notice also that only rules with non-singular, non-negative integer duration can be specified, i.e. a rule is always active for at least 1 time unit. We can now observe that it must be possible to rearrange every choice by the environment such that uncontrollable events occur only at integer points and preserve a non-singular period of time where *validState* returns false.

*Observation 2* We first notice that in a rule template $R(id)$, *activeRules* for $id$ is incremented on the edge going to the ACTIVE location, is decremented on all edges going out of the ACTIVE location and is not changed anywhere else. We now make the observation that the actual number of replicas in the ACTIVE location is irrelevant with respect to the *validState* function, as the function only considers if $activeRules > 0$ or not i.e. if there is at least one replica in the ACTIVE location.

*Conclusion* It follows from the first observation that for a given rule $\psi_i =$ **on** $E$ **provided** $\varphi'$ **satisfy** $\varphi$ **starting in** $[s, e]$ **for** $d$ **unless** $E'$, $R(i)$ can be triggered at most $e + d$ times before the replica used for the first invocation must have returned to the IDLE location and can be reused. It follows from the second observation that at most one replica should be used at any time $t$. This leads to the trivial bound that $e + d$ replicas are sufficient for $R(id)$ to find a winning strategy for the environment if one exists with unboundedly many replicas. □

As it is of crucial importance to performance to minimize the bound for the required number of replicas, we continue to postulate the following improved replicas bound.

*Claim 1 (Bound for Number of Replicas).* Lemma 1 holds with only $\lceil \frac{e}{d} \rceil + 1$ replicas of $R(i)$ for rule $\psi_i =$ **on** $E$ **provided** $\varphi'$ **satisfy** $\varphi$ **starting in** $[s, e]$ **for** $d$ **unless** $E'$.

We can now continue to show soundness of the translation.

**Theorem 1 (Soundness of Translation).** *Let* $\mathbb{S}$ *be a system and* $\psi$ *be a specification, then* $TW(TGA(\mathbb{S}, \psi) \upharpoonright f) \subseteq TW(\mathbb{S}, \psi)$ *for any winning strategy* $f$ *in* $TGA(\mathbb{S}, \psi)$.

*Proof.* We show this by structural induction on $\psi$.

$\psi = true$*,* $\psi = \mu$*,* $\psi = \neg\varphi$*, or* $\psi = \varphi_1 \wedge \varphi_2$*:* It follows from Fact 1 that *validState* returns true for a configuration $c \in CONF(\mathbb{S})$ iff $c \models Prop(\psi)$. We require in the safety property that in any state where $x > 0$, *validState* must return true, thus this case must be satisfied.

$\psi$ = ***on** E **provided** $\varphi$ **satisfy** $\varphi'$ **starting in** $[s, e]$ **for** d **unless** E':* Let $id$ be the id of $\psi$. We assume the abstraction of unbounded many replicas $R(id)$ and show this using a case analysis. Soundness for the bounded case follows from Lemma 1.

Assume some $e \in E$ occurs at time $t$ in a configuration $c \models \varphi$. Consider first the flow where the rule is not cancelled. This means, due to the select statement, that one replica of $R(id)$ goes from the IDLE location to the ENABLED location and resets the local clock $y$. After $s$ time units, the edge to the ACTIVE location becomes enabled. Due to the invariant, the edge must be followed at latest after $e$ time units. When the edge is followed, the local clock $y$ is reset again and *activeRules[id]* is incremented. The replica stays in the ACTIVE location for exactly $d$ time units, after which it must return to the IDLE location, resetting $y$ and decrementing *activeRules[id]*. This means that the rule must be in the ACTIVE location for $d$ time units, starting in the interval $[t + s, t + e]$, coinciding with the semantics of $\psi$. Assuming *activeRules[id]* is initially 0, we see that $activeRules[id] > 0$ whenever some replica is in the ACTIVE location, as *activeRules[id]* is always incremented once when entering the location and always decremented once after leaving the location before returning to the IDLE location. We know from Fact 1 that $\varphi'$ is enforced by the *validState* function when $activeRules[id] > 0$, i.e. when some replica is in the ACTIVE location. Thus, we can conclude that $\varphi'$ must be enforced whenever a replica is in the ACTIVE location, as the objective ensures that *validState* returns true whenever $x > 0$, and that a replica is in the ACTIVE location exactly when $\varphi'$ should be satisfied according to the semantics (disregarding canceling).

Consider now the case where the rule is cancelled by some event $e' \in E'$. As events are broadcast, all replicas are canceled. If in the ENABLED location, the replica returns to the IDLE location, resetting $y$. If in the ACTIVE location, the replica goes to the CANCELLED location. This location is urgent and allows for the control component to change the configuration of the system before *activeRules[id]* is decremented and the replica returns to the IDLE location. This location is necessary, as $x > 0$ if $e'$ belongs to an uncontrollable service, thus directly decrementing *activeRules[id]* and returning to the IDLE location would result in a violation of cost optimality in the objective. Introducing the intermediate location allows for the control component to change the configuration of the system before *activeRules[id]* is decremented and thus to react to the uncontrollable event. Before time can progress, all replicas must return to the IDLE location and consequently $activeRules[id] = 0$ and $\varphi'$ is no longer enforced by the *validState* function □

We now continue to examine completeness of the translation. To do so, we will start by considering an issue regarding completeness with services where the state-space is not completely connected. Consider a new service, which is a lamp with three states:

$$AdvLamp = (\{off, low, high\}, off, \{(off, low), (low, high), (high, low), (low, off)\})$$

and the system:

$$\mathbb{S} = (\{\mathit{AdvLamp}\}, \{\mathit{Switch}\})$$

We see that the state-space of the lamp is not completely connected, as it cannot go directly from the *off* state to the *high* state. If we recall the second assumption to timed word in Section 3.4, we restrict that only one edge can be followed in any service at any time. This means that it is not possible to go from the *off* state to the *high* state instantly. Consequently, if we assume cost function $\nu(\mathit{off}) = 0, \nu(\mathit{low}) = 1, \nu(\mathit{high}) = 2$, the following specification is not satisfiable with the current definition of local cost optimality:

| **on** *Switch.click* **provided** *true* **satisfy** *AdvLamp.high*<br>**starting in** $[1, 1]$ **for** 10 **unless** $\emptyset$ |
|---|

This is a consequence of local cost optimality dictating that the system should stay in the *off* state whenever *AdvLamp.high* is not enforced, thus the assumption of only one state change at any time prevents the service from going instantly to the *high* state and the local cost optimality prevents the service from going to the *low* state in advance. We notice that without local cost optimality, it is possible for the service to remain in the *low* state and thus be able to go directly to the *high* state.

Strictly speaking, the conclusion that no winning control strategy exists for the specification is correct with respect to our assumptions and local cost optimality. However, the specification could also reasonably be expected to have a winning control strategy. Therefore, we will leave ourself to a claim of correctness for systems with only completely connected services and leave the proof as well as adaptation to the unconnected case for further work. We notice that because HomePort only supports completely connected services, this is of no practical concern for our implementation.

*Claim 2 (Completeness of Translation with Completely Connected Services).* Let $\mathbb{S}$ be a system where all services are completely connected and let $\psi$ be a specification. If there exists a control strategy $\gamma$ s.t. $TW(\mathbb{S} \upharpoonright \gamma) \subseteq TW(\mathbb{S}, \psi)$ then there exists a strategy $f$ s.t. $TW(TGA(\mathbb{S}, \psi) \upharpoonright f) \subseteq TW(\mathbb{S}, \psi)$.

## 8 Implementation

We provide a proof-of-concept implementation of a completely automated controller synthesis toolchain, available[2] at `http://homeio.net/concept`. Figure 8 shows the toolchain of the implementation. The entire toolchain — including strategy synthesis by UPPAAL TiGa — is executed on a Raspberry Pi (Model B), a cheap (35€), low-powered credit-card sized computer featuring a 700 MHz ARMv6 processor and 512 MB of RAM equipped with a WiFi dongle for wireless

[2] Only source code for individual components. UPPAAL TiGa for ARM not available for licensing reasons.

interaction with the system. Figure 9 shows a picture of the setup. To the left in the picture, the Raspberry Pi is seen. The Raspberry Pi is here connected to a demo unit (box below) via the GPIO pins (grey cable). The following sections elaborate on individual steps in the toolchain.

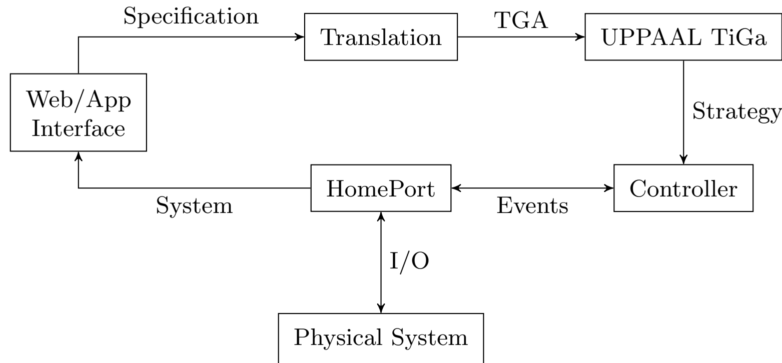


Fig. 8: Toolchain for automated controller synthesis.

**Interface** Interaction with the system happens through a simple web-interface in form of a *web app*. Screenshots from the interface are available in Figure 10. The app automatically detects changes to the underlying home automation system and allows easy configuration and management of devices. A rule editor facilitates easy point-and-click management of behavior specifications. Given a specification, a control strategy is automatically synthesized and a corresponding controller is constructed and invoked.

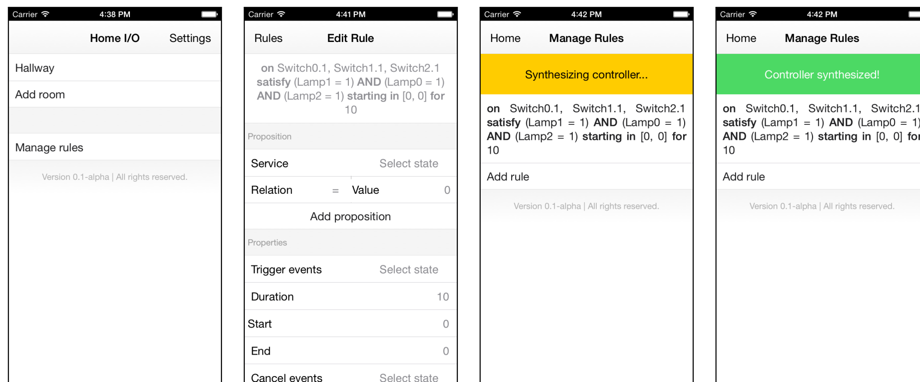


Fig. 10: Interface of the toolchain. The interface is optimized for point-and-click interaction and can be used directly on a smartphone.

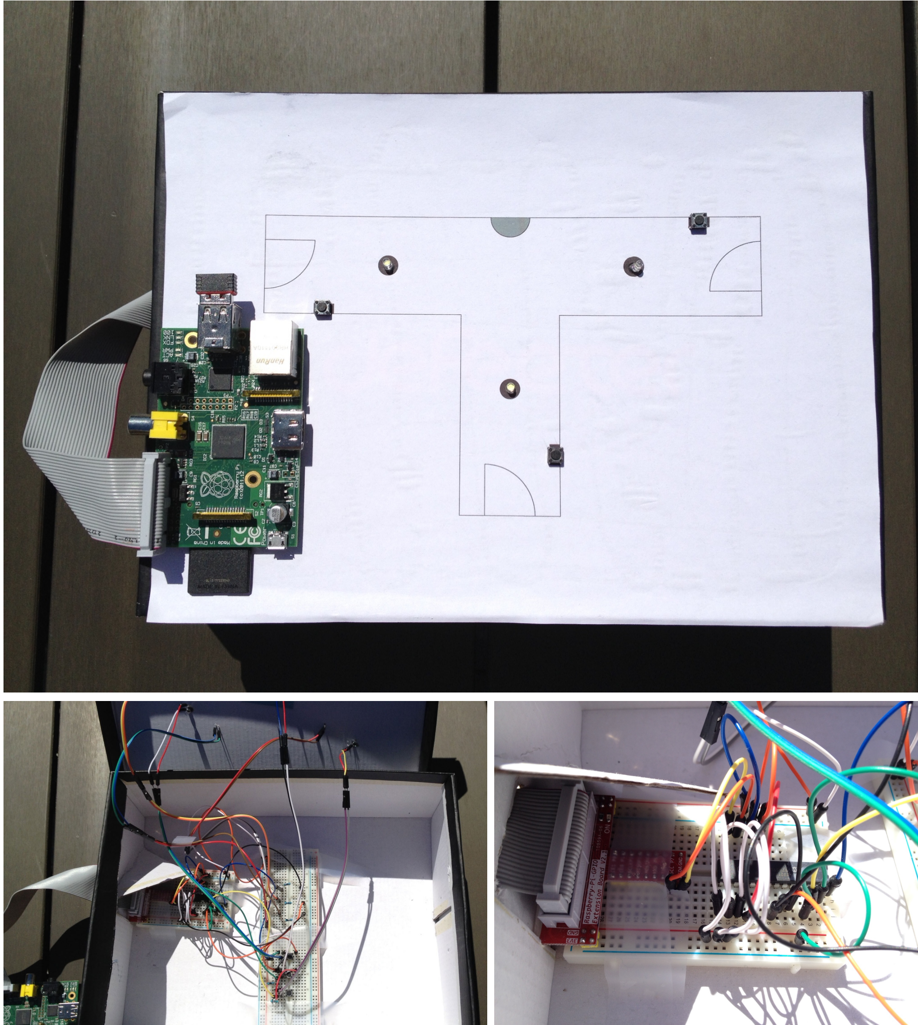

Fig. 9: Top: Toolchain executed on Raspberry Pi connected to a demo unit (box below) via the GPIO pins (grey cable). Bottom: Internal wiring of demo unit.

**Translation & Strategy Synthesis** Based on the system description and specification provided from the interface, translation to UPPAAL TiGa models is performed in PHP. The model is automatically provided to UPPAAL TiGa, which has been compiled for the ARMv6 processor and is executed directly on the Raspberry Pi. The synthesized strategy is used to dictate behavior using the controller.

**Controller** The controller is a small control program, written in C++ for portability. Provided with a system description, a specification and a strategy, the controller monitors the underlying HomePort system and effectuates behavior according to the strategy, effectively providing an implemented control program.

### 8.1 Performance

We provide a performance evaluation of the proof-of-concept toolchain implementation. Table 2 shows synthesis time for some of the example specifications from Section 5. As we see, simple specifications (the simple and timer examples) can be synthesized in reasonable time directly on the Raspberry Pi. However, we also see that larger specifications ("Follow Me" example) are unfeasible to synthesize, even on laptop-grade hardware (MacBook Pro equipped with a 2,3 GHz Intel Core i5 processor and 8GB of RAM). Due to the complexity of the synthesis problem, a different approach is required to achieve scaleability of the method. We explore in the next section an approach of decomposing the synthesis problem into smaller subproblems, in order to obtain greater scaleability.

| Model | Raspberry Pi | Laptop |
|---|---|---|
| Simple | 5.56s | 0.15s |
| Timer | 5.16s | 0.16s |
| Follow Me | 🕐 | *OOM* |

Table 2: Synthesis time on a Raspberry Pi and on a laptop for the control strategy synthesis problem for some of the example specifications from Section 5 expressed for the hallway scenario. 🕐 means more than 15 minutes, *OOM* means out of memory (4GB limit).

## 9 Compositional Control Strategy Synthesis

Due to the complexity of the synthesis problem for TGAs, achieving sufficient scalability for real-world application is non-trivial. To address this, we present here a method to exploit independence in the system and specifications. This allows for decomposition of the control strategy synthesis problem into multiple

subproblems. To do so, we need to know how to find such semantically correct decompositions. We here provide an overview of how such decompositions can be found.

### 9.1 Syntactic Decomposition

On the syntactical level, we notice that rules and invariants with propositions containing conjunctions can be decomposed into multiple separate rules or invariants. Consider the following specification $\psi$ for a system with one switch and two lamps:

**on** *Switch.click* **provided** *true* **satisfy** *Lamp1.on* $\wedge$ *Lamp2.on*
**starting in** $[s, e]$ **for** $d$ **unless** $\emptyset$

Clearly, we can construct an equivalent specification $\psi'$:

**on** *Switch.click* **provided** *true* **satisfy** *Lamp1.on*
**starting in** $[s, e]$ **for** $d$ **unless** $\emptyset$

**on** *Switch.click* **provided** *true* **satisfy** *Lamp2.on*
**starting in** $[s, e]$ **for** $d$ **unless** $\emptyset$

Invariants are decomposed in a similar manner. We refer to this step as syntactic decomposition. We see that syntactical decomposition can sometimes be used to construct more, but smaller, rules and invariants. We will explore how this is beneficial in the following section and we assume from now on that all rules and invariants are syntactically decomposed.

### 9.2 Independent Subsystems

We now want to clarify the relationship between services. Consider e.g. a home automation setup with multiple rooms. If specifications do not utilize informations between the rooms, surely it should be possible to consider each room independently. That is, we can say that services in the rooms are independent of one another and that the rooms are two independent subsystems. We will now generalize this approach to identify independent subsystems of services. To do so, we consider first intuitively dependencies for uncontrollable and controllable service.

*Uncontrollable Services* We notice that the behavior of uncontrollable services by definition cannot be restricted by rules or invariants, so clearly an uncontrollable service is always independent of all other services.

*Controllable Services* We notice that the behavior of controllable services is restricted by rules and invariants, so controllable services must be dependent if they are referred to in the same rules.

We can now continue to formalize this intuition. Let $\psi =$ **on** $E$ **provided** $\varphi$ **satisfy** $\varphi'$ **starting in** $[s, e]$ **for** $d$ **unless** $E'$ be a rule. We define $Serv(\psi)$ be the set of services s.t. $\mathcal{S} \in Serv(\psi)$ iff

- $e \in \mathcal{S}^E$ for some $e \in (E \cup E')$, or
- $\mathcal{S}$ is referred to in $\varphi$, or
- $\mathcal{S}$ is referred to in $\varphi'$.

The definition is extended to invariants in the intuitive way. We can now continue to define an independent subsystem.

**Definition 7 (Independent Subsystems).** *Let $\mathbb{S} = (\mathbb{S}_c, \mathbb{S}_u)$ be a system and let $\psi = \psi_1 \wedge \ldots \wedge \psi_n$ be a specification. A subset $\overline{\mathbb{S}} \subset \mathbb{S}$ with a corresponding set of rules $\Psi$ is an independent subsystem of $\mathbb{S}, \psi$ iff*

- *For all $\psi_i \in \Psi$, if $\mathcal{S} \in Serv(\psi_i)$ then $\mathcal{S} \in \overline{\mathbb{S}}$.*
- *For all $\mathcal{S} \in (\mathbb{S}_c \cap \overline{\mathbb{S}})$, if $\mathcal{S} \in Serv(\psi_i)$ then $\psi_i \in \Psi$.*

We can use this notion of independent subsystems to split $CSSP(\mathbb{S}, \psi)$ into multiple subproblems $CSSP(\overline{\mathbb{S}}, \Psi)$. Consider again the example with two lamps and a switch. We see that this example can be split into two independent subsystems:

$$\overline{\mathbb{S}}_1 = (\{Lamp1\}, \{Switch\}), \Psi_1 = \{\psi_1\}$$
$$\overline{\mathbb{S}}_2 = (\{Lamp2\}, \{Switch\}), \Psi_2 = \{\psi_2\}$$

Thus we can obtain a control strategy for the control strategy synthesis problem $CSSP(\mathbb{S}, \psi)$ by solving $CSSP(\overline{\mathbb{S}}_1, \Psi_1)$ and $CSSP(\overline{\mathbb{S}}_2, \Psi_2)$ independently.

### 9.3 Performance

If we return to the performance analysis in Section 8.1, we can apply decomposition to the evaluated cases. Table 3 shows synthesis time for the evaluated cases in Section 8.1 extended with decomposition, synthesized sequentially or all in parallel (assuming sufficient processing cores). We see that decomposition introduces a small overhead due to extra parsing of files etc., resulting in slightly worse sequential performance on the small timer example. When considering the larger “Follow Me” example, however, we see that decomposition greatly improves scaleability and performance of the approach and concretely allows us to synthesize a control strategy for the “Follow Me” example in very reasonable time on laptop-grade hardware.

We conclude that using decomposition allows us to synthesize control strategies for more interesting specification, although even subsystems quickly become to large to be synthesized directly on the Raspberry Pi. We also see that decomposition greatly improves scaleability of the approach and is crucial to extend the scenario to a full home automation specification. Finally, we see that synthesizing strategies for independent subsystems in parallel is a great way to achieve

| Model | Ra. Pi | Laptop | Dec., Ra. Pi | Dec., laptop | Dec., laptop, parallel |
|---|---|---|---|---|---|
| Simple (1 subs.) | 5.56s | 0.15s | 5.56s | 0.15s | 0.15s |
| Timer (3 subs.) | 5.16s | 0.16s | 6.30 | 0.39s | 0.13s |
| Follow Me (3 subs.) | ⏲ | *OOM* | ⏲ | 11.25s | 3.75s |

Table 3: Synthesis time on a Raspberry Pi and on a laptop for the control strategy synthesis problem for some of the example specifications from Section 5 expressed for the hallway scenario. Subs. = number of independent subsystems, ⏲ means more than 15 minutes, *OOM* means out of memory (4GB limit).

practical applicability of the method and that using a cloud-based service for synthesis would be an obvious way to facilitate this. We also emphasize that decomposition can be supplemented with additional means to reduce the size of the synthesis problem, thereby improving performance even more.

## 10 Conclusion & Further Work

We have presented a complete framework for using game theory to synthesize an executable controller for a home automation setup. We have presented a simple specification logic and shown how patterns can be used to express general and portable specifications, potentially limiting end-user interaction to simply selecting desired system behavior from a repository of patterns. We have presented an efficient translation to Timed Game Automata in form of UPPAAL TiGa models and have explored approaches to improve synthesis performance and scaleability by exploiting decomposability of systems and specifications into independent subsystems. We have provided a completely automated proof-of-concept toolchain implementation, which automates the entire process from specification of behavior in a simple web-app interface to executable controller. The entire toolchain — including synthesis by UPPAAL TiGa — is executed on a Raspberry Pi, a low-cost, low-performance open source hardware device, directly applicable in a home automation setup.

We conclude that using game theory as a backend technology in home automation provides some great benefits, in particular by facilitating underspecified behavior specifications. We also conclude that acceptable performance for real-life applications is achievable through decompositionality, although using a remote device or cloud service for UPPAAL TiGa execution will be desirable in large-scale setups in order to fully exploit parallel synthesis for independent subsystems.

### 10.1 Further Work

We have already mentioned several topics of further work throughout the paper. We explore here five additional areas of further work in form of possible extensions to the work presented in this paper.

**Service Models** In Section 5 we saw a specification for how to detect double-clicks using data services. While this solution is possible, one should be careful with how the click event affects different rules. Consider in conjunction with the double-click specification the following rule.

| **on** *Switch.click* **provided** *SD.1* **satisfy** *Lamp1.on* $\wedge$ *Lamp2.on* $\wedge$ *Lamp3.on* **starting in** $[0, 0]$ **for** $\infty$ **unless** *Switch.click* |
|---|

This specification says that when the motion sensor is disabled, the switch toggles the lamp states. When double-clicking to toggle the motion sensor with this specification, we trigger the *Switch.click* event twice and thus also toggle the light twice. This is not desirable, as we would probably want the lamps not to react until it has been determined if the click is a single- or double-click. While this behavior is obtainable by replacing the trigger event *Switch.click* with the event $CC.0$, we argue that this approach will make specifications less intuitive. An alternative approach is to implement the double-click on the service level. Thus, for the switch, we can add an intermediate service model in conjunction with the regular switch model. A such intermediate model is shown in Figure 11. The intermediate model synchronizes with click broadcast events by the switch and broadcasts itself single- or double-click events. This approach thus moves the concern of detecting double-clicks from the specification level to the service level, as specifications can now be expressed using the new single- and double-click events.

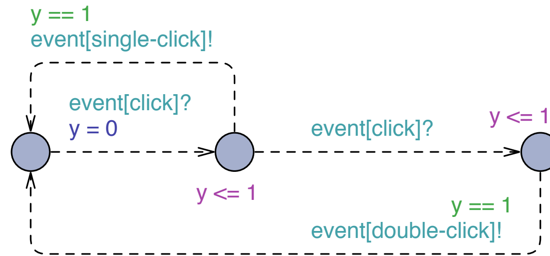


Fig. 11: Intermediate service model which synchronizes on click events and broadcasts single-click or double-click events.

**Known Environment** So far we have assumed no information about the environment. This means that we can express only properties for controllable services, as uncontrollable services are assumed to behave randomly. This is however not always the case. Consider a home automation scenario for a floor heating system. The system consists of two controllable valves which regulate the water flow and a thermometer which provides readings of the room temperature, i.e.

$$
\begin{aligned}
\mathit{Valve1}, \mathit{Valve2} &= (\{on, \mathit{off}\}, \mathit{off}, \{(on, \mathit{off}), (\mathit{off}, on)\}) \\
\mathit{Thermometer} &= (\{i \mid 0 \leq i \leq 40\}, 20, \{(i, i+1), (i+1, i) \mid 0 \leq i < 40\}) \\
\mathbb{S} &= (\{\mathit{Valve1}, \mathit{Valve2}\}, \{\mathit{Thermometer}\})
\end{aligned}
$$

where *Valve1* and *Valve2* control the valves for a floor heating system and *Thermometer* measures the temperature in the room. Without knowledge of the environment, we can only specify rules for the state of the valves based on the thermometer reading, e.g. the following specification:

> **on** *Thermometer*.0, . . . , *Thermometer*.18 **provided** *true*
> **satisfy** *Valve1.on* && *Valve2.on* **starting in** [0, 0] **for** ∞
> **unless** *Thermometer*.19
>
> **on** *Thermometer*.0, . . . , *Thermometer*.21 **provided** *true*
> **satisfy** *Valve1.on* **starting in** [0, 0] **for** ∞
> **unless** *Thermometer*.22

However, a more intuitive specification would be to specify an invariant directly on the (uncontrollable) thermometer, e.g.

> *Thermometer* ≥ 19 && *Thermometer* ≤ 22

If we assume knowledge of the environment, we can support this kind of expressions over uncontrollable services by deducing relations between controllable and uncontrollable services. We do however need to relax the satisfaction relation as well as the local optimality criteria in order to introduce this extension.

**Relation to Absolute Time** In the presented logic, rules specify a time interval for rule activation relative to the time of the triggering event. In some cases, however, it might be desirable to specify absolute time intervals. Consider the situation where an electric car is plugged in and should charge during the night (rather than at a specific time interval after the car is plugged in). This would allow us to express specifications such as (assuming uncontrollable service *Charger* and controllable service *Battery*):

> **on** *Charger.connect* **provided** *true*
> **satisfy** *Battery.charge* **starting in** [*6pm*, *4am*] **for** *3hours*
> **unless** *Charger.disconnect*

**Expected Cost Optimal Strategies** With the presented approach, time intervals provide some slack for when to activate a rule. In a real-world scenario, utility costs fluctuate over time, thus this slack could be exploited to select the best timing for activating the rule, based on the expected utility costs. While

this could be directly achieved by introducing utility costs in the model (resulting in Priced Timed Games), the problem unfortunately becomes undecidable with only three clocks [5].

Alternatively, statistical model checking (SMC) combined with reinforced learning can be used to approximate expected cost optimal strategies from the most permissive strategy, which is obtainable in UPPAAL TiGa. This approach is successfully applied to the Priced Timed Markov Decision Process formalism in [7]. This allows us to reduce utility costs while still behaving within the constraints of the specification, e.g. in the car charging example, this would allow the car to start charging whenever electricity is expected to be cheapest between 6pm and 4am (so that the car is always fully charged by 7am).

**Discrete Time Synthesis** Finally, we notice that it has previously been shown that for some problems, explicit methods outperform the symbolic methods utilized by UPPAAL (see e.g. [1]). It is an interesting area of further work to evaluate whether explicit methods are able to improve scaleability and performance in this scenario, as well as to explore semantical equivalence between the continuous time semantics of UPPAAL TiGa and the discrete time semantics of explicit methods for this problem.

## References


1. Mathias Andersen, Heine Gatten Larsen, Jiří Srba, Mathias Grund Sørensen, and Jakob Haahr Taankvist. Verification of liveness properties on closed timed-arc petri nets. In Antonín Kučera, ThomasA. Henzinger, Jaroslav Nešetřil, Tomáš Vojnar, and David Antoš, editors, *Mathematical and Engineering Methods in Computer Science*, volume 7721 of *Lecture Notes in Computer Science*, pages 69–81. Springer Berlin Heidelberg, 2013.
2. Gerd Behrmann, Agnes Cougnard, Alexandre David, Emmanuel Fleury, KimG. Larsen, and Didier Lime. Uppaal-tiga: Time for playing games! 4590:121–125, 2007.
3. Gerd Behrmann, Re David, and Kim G. Larsen. A tutorial on uppaal 4.0, 2006.
4. Aaron Bohy, Véronique Bruyère, Emmanuel Filiot, Naiyong Jin, and Jean-François Raskin. Acacia+, a tool for ltl synthesis. In P. Madhusudan and SanjitA. Seshia, editors, *Computer Aided Verification*, volume 7358 of *Lecture Notes in Computer Science*, pages 652–657. Springer Berlin Heidelberg, 2012.
5. Patricia Bouyer, Thomas Brihaye, and Nicolas Markey. Improved undecidability results on weighted timed automata. *Inf. Process. Lett.*, 98(5):188–194, June 2006.
6. Peter H. Dalsgaard, Thibaut Le Guilly, Daniel Middelhede, Petur Olsen, Thomas Pedersen, Anders P. Ravn, and Arne Skou. A toolchain for home automation controller development. Accepted to SEAA 2013.
7. Alexandre David, Peter Gjøl Jensen, Kim Guldstrand Larsen, Alex Leagy, Didier Lime, Mathias Grund Sørensen, and Jakob Haahr Taankvist. On time with minimal expected cost!, 2014. Manuscript.
8. U. Dayal, B. Blaustein, A. Buchmann, U. Chakravarthy, M. Hsu, R. Ledin, D. McCarthy, A. Rosenthal, S. Sarin, M. J. Carey, M. Livny, and R. Jauhari. The hipac project: combining active databases and timing constraints. *SIGMOD Rec.*, 17(1):51–70, March 1988.

9. Rüdiger Ehlers. Unbeast: Symbolic bounded synthesis. In ParoshAziz Abdulla and K.RustanM. Leino, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 6605 of *Lecture Notes in Computer Science*, pages 272–275. Springer Berlin Heidelberg, 2011.
10. Manuel García-Herranz, Pablo Haya, and Xavier Alamán. Towards a ubiquitous end-user programming system for smart spaces. 16(12):1633–1649, jun 2010.
11. Apple Inc. Homekit, June 2014. https://developer.apple.com/homekit/.
12. Nest Labs/Google Inc. Nest, June 2014. https://nest.com/.
13. Jan Jakob Jessen, Jacob Illum Rasmussen, Kim G. Larsen, and Alexandre David. Guided controller synthesis for climate controller using uppaal tiga. In *Proceedings of the 5th international conference on Formal modeling and analysis of timed systems*, FORMATS'07, pages 227–240, Berlin, Heidelberg, 2007. Springer-Verlag.
14. T. Le Guilly, P. Olsen, A.P. Ravn, J.B. Rosenkilde, and A. Skou. Homeport: Middleware for heterogeneous home automation networks. In *Pervasive Computing and Communications Workshops (PERCOM Workshops), 2013 IEEE International Conference on*, pages 627–633, 2013.
15. Geetika Singla, Diane J. Cook, and Maureen Schmitter-edgecombe. Incorporating temporal reasoning into activity recognition for smart home residents. In *Proceedings of the AAAI Workshop on Spatial and Temporal Reasoning*, 2008.
16. Mathias Grund Sørensen. Towards automated controller synthesis in home automation. 2013. DAT9 Semester Project.