Self-Reconfigurable Control, Application to Smart Environments

Sébastien Guillet, Bruno Bouchard, Abdenour Bouzouane
{sebastien.guillet1,bruno.bouchard,abdenour.bouzouane}@uqac.ca

Abstract

This work addresses the problem of controlling smart homes dedicated to people with disabilities when their disabilities evolve. These people being usually frail by nature, keeping their environment constantly safe and adapted is crucial. While it is possible to prove the security of such an environment given both its behavioral definition and a patient profile, proving that it will remain secure for all possible profile evolution is a combinatorial problem. A solution to this problem is to define secure execution points on which the control of the environment's behavior can be changed for a new one that is adapted to an evolution of the patient's profile. This paper presents the methodology to implement this solution based on the definition of the environment as a synchronous program containing controllability information. From such a program, we use a synthesis technique named Discrete Controller Synthesis to obtain — when it exists — a control function that will enforce temporal properties on the execution of the synchronous program. A use case is presented, showing a partial model of a smart home, on which security properties are defined to be enforced at runtime. During execution, the patient profile is updated, and a new controller obtained through DCS is integrated to adapt the environment's behavior appropriately.

Introduction

Designing environments dedicated to frail people involves many challenges, like blending unobtrusively into the home environment (Novak, Binas, and Jakab 2012), recognizing the ongoing inhabitant activity (Bouchard, Giroux, and Bouzouane 2007), localizing objects (Fortin-Simard et al. 2012), adapting assistance to the person's cognitive deficit (Lapointe et al. 2012), and securing the environment (Pigot, Mayers, and Giroux 2003).

Given the high degree of vulnerability of people with cognitive deficiencies, securing the house is a primary concern. Indeed, an adequately designed smart home for disabled people should be able to provide both assistance and protection. In (Guillet, Bouchard, and Bouzouane 2013), we addressed one major security aspect regarding failure tolerance: even if a smart home system is usually build to last, it might no be the case for its very own components — lights, screens, sound system, and many other important equipments can fail during the lifetime of the system — in this context, we provided a methodology to automatically obtain the code of a controller component, able to keep the smart home system in safe states for all possible executions: a safe state representing a situation where the system always has at least one way to communicate appropriately with the user (depending on its impairments) even in case of failures on its components.

However, the solution we presented did not take into account the evolution of the system: if a new component is added, or if the user's impairments are updated, a new control system has to be found, but no methodology was defined to include this new controller into the smart home execution without requiring external intervention. Defining such a methodology is the aim of this study which starts by giving the essential notions of the underlying execution model (the synchronous framework), then explains how models defined within this framework can be used to performed validation and synthesis techniques (Discrete Controller Synthesis), and gives the model instrumentation methodology to enforce the capacity of the smart home to be reinitialized at any moment. This methodology is then employed in a case study, using a partial smart home model to show the important steps of the execution when the control system is replaced at runtime.

Synchronous framework: basic notions

Synchronous languages are optimized for programming reactive systems, i.e. systems that react to external events. This section aims at presenting the similarities between a reactive system under control and a controlled smart home, so that a synchronous framework – essentially adopted from (Marchand and Samaan 2000) and (Altisen et al. 2003) – gets justified as appropriate to specify smart home systems.

Execution model

In (Guillet et al. 2012), the execution model of a reactive system under control is depicted. Such a system contains a global execution loop, which starts by taking events from the environment. Then these events get processed by a task (*Reconfiguration controller*), which chooses the system's configuration. Finally, this configuration order gets dispatched

through the system's tasks following its model of computation, and another iteration of the loop can start again. If a system can be represented within this execution model, then the proposition of this work can help to design and formally obtain its *Reconfiguration controller* task.

In (Bouchard, Bouchard, and Bouzouane 2012), guidelines to build the software architecture of a smart home system are presented. Such a software follows a loop-based execution, in which a database containing an updated system state and event values is read and processed by eventual artificial intelligence (AI) modules to transform raw data into high level information. This information can then be used by third party applications. If we add a reconfiguration controller as a third party application in this software architecture, then we obtain the same execution principle presented in (Guillet et al. 2012): in each iteration of the execution loop a controller can be designed to 1) take events and/or high level information provided by the system and its environment, 2) perform a reconfiguration decision, and 3) give this decision back to the system using some of its actuators (i.e. its controllability) before the next iteration.

Designing the aforementioned controller by constraint so that it can be obtained automatically through DCS becomes possible, but it requires the use of formal a model to specify the behavior of the underlying system under control. Behavioral modeling can be performed using various formal representations, e.g. Statecharts, Petri-nets, Communicating Sequential Processes or other ways. The toolset we use in this work – BZR and SIGALI – brings us to define our system in terms of synchronous equations and Labelled Transition Systems.

Synchronous equation

In a declarative synchronous language, semantic is expressed in terms of dataflows: values carried in discrete time are considered as infinite sequence of values, or *flows*. At each discrete instant, the relation between input and output values is defined by an equational representation between flows, it is basically a system of equations: equations are evaluated concurrently in the same instant and not in sequence, the real evaluation order being determined at compile-time from their interdependencies. For example, let x and y be two dataflows such that $x = x_0, x_1, ...$ and $y = y_0, y_1, ...$ Evolution of y over time is given by the following system of equations:

$$\begin{cases} y_0 &= x_0 \\ y_t &= y_{t-1} + x_t \quad \text{if } t \geq 1 \end{cases}$$

In this example, y is defined, amongst others, by a reference to its value at a previous discrete instant. Each declarative synchronous language has a syntax to define such a system. The corresponding BZR program is: `y = x -> pre(y) + x;`, meaning that in the first step, y takes the current value of x, and for all next steps y will take its previous value incremented by x. (Other syntactic features of BZR can be found online [1]). To represent the system execution modes, BZR also allows to define automata, or Labelled Transition Systems, each state encapsulating a set of synchronous equations evaluated only when the state is activated.

[1] http://bzr.inria.fr/pub/bzr-manual.pdf

Labelled Transition System (LTS)

A LTS is a structure $S = \langle \mathcal{Q}, q_0, \mathcal{I}, \mathcal{O}, \mathcal{T} \rangle$ where $\mathcal{Q}$ is a finite set of states, q_0 is the initial state of S, $\mathcal{I}$ is a finite set of input events (produced by the environment), $\mathcal{O}$ is a finite set of output events (emitted towards the environment), and $\mathcal{T}$ is the transition relation, that is a subset of $\mathcal{Q} \times \text{Bool}(\mathcal{I}) \times \mathcal{O}^* \times \mathcal{Q}$, where $\text{Bool}(\mathcal{I})$ is the set of boolean expressions of $\mathcal{I}$. If we denote by $\mathcal{B}$ the set $\{\text{true, false}\}$, then a guard $g \in \text{Bool}(\mathcal{I})$ can be equivalently seen as a function from $2^{\mathcal{I}}$ into $\mathcal{B}$.

Each transition has a label of the form g/a, where $g \in \text{Bool}(\mathcal{I})$ must be true for the transition to be taken (g is the guard of the transition), and where $a \in \mathcal{O}^*$ is a conjunction of outputs that are emitted when the transition is taken (a is the action of the transition). State q is the source of the transition (q, g, a, q'), and state q' is the destination. A transition (q, g, a, q') will be graphically represented by $(q \xrightarrow{g,a} q')$.

The composition operator of two LTS put in parallel is the synchronous product, noted $||$, and a characteristic feature of the synchronous languages. The synchronous product is commutative and associative. Formally: $\langle \mathcal{Q}_1, q_{0,1}, \mathcal{I}_1, \mathcal{O}_1, \mathcal{T}_1 \rangle || \langle \mathcal{Q}_2, q_{0,2}, \mathcal{I}_2, \mathcal{O}_2, \mathcal{T}_2 \rangle = \langle \mathcal{Q}_1 \times \mathcal{Q}_2, (q_{0,1}, q_{0,2}), \mathcal{I}_1 \cup \mathcal{I}_2, \mathcal{O}_1 \cup \mathcal{O}_2, \mathcal{T} \rangle$ with $\mathcal{T} = \{((q_1, q_2) \xrightarrow{(g_1 \wedge g_2)/(a_1 \wedge a_2)} (q_1', q_2')) | (q_1 \xrightarrow{g_1/a_1} q_1') \in \mathcal{T}_1, (q_2 \xrightarrow{g_2/a_2} q_2') \in \mathcal{T}_2\}$.

Note that this synchronous composition is the simplified one presented in (Altisen et al. 2003), and supposes that g and a do not share any variable, which would be permitted in synchronous languages like Esterel.

Here (q_1, q_2) is called a *macro-state*, where q_1 and q_2 are its two *component states*. A macro-state containing one component state for every LTS synchronously composed in a system S is called a *configuration* of S.

Discrete Controller Synthesis (DCS) on LTS

A system S is specified as a LTS, more precisely as the result of the synchronous composition of several LTS. $\mathcal{F}$ is the objective that the controlled system must fulfill, and $\mathcal{H}$ is the behavior hypothesis on the inputs of S. The controller C obtained with DCS achieves this objective by restraining the transitions of S, that is, by disabling those that would jeopardize the objective $\mathcal{F}$, considering hypothesis $\mathcal{H}$. Both $\mathcal{F}$ and $\mathcal{H}$ are expressed as boolean equations. The set $\mathcal{I}$ of inputs of S is partitioned into two subsets: the set $\mathcal{I}_C$ of controllable variables and the set $\mathcal{I}_U$ of uncontrollable inputs. Formally, $\mathcal{I} = \mathcal{I}_C \cup \mathcal{I}_U$ and $\mathcal{I}_C \cap \mathcal{I}_U = \emptyset$. As a consequence, a transition guard $g \in \text{Bool}(\mathcal{I}_C \cup \mathcal{I}_U)$ can be seen as a function from $2^{\mathcal{I}_C} \times 2^{\mathcal{I}_U}$ into $\mathcal{B}$. A transition is controllable *if and only if* (iff) there exists at least one valuation of the controllable variables such that the boolean expression of its guard is false; otherwise it is uncontrollable. Formally, a transition $(q, g, a, q') \in \mathcal{T}$ is controllable iff $\exists X \in 2^{\mathcal{I}_C}$ such that $\forall Y \in 2^{\mathcal{I}_U}$, we have

$g(X, Y) = false$. In the proposed framework, the following function $S_c = \textit{make\_invariant}(S, E)$ from SIGALI is used to synthesize (i.e. *compute by inference*) the controlled system $S_c = S||C$ where E is any subset of states of S, possibly specified itself as a predicate on states (or *control objective*) $\mathcal{F}$ and predicate on inputs (or *hypothesis*) $\mathcal{H}$. The function *make_invariant* synthesizes and returns a controllable system S_c, if it exists, such that the controllable transitions leading to states $q_i \notin E$ are inhibited, as well as those leading to states from where a sequence of uncontrollable transitions can lead to such states $q_i \notin E$. If DCS fails, it means that a controller of S does not exist for objective $\mathcal{F}$ and hypothesis $\mathcal{H}$. In this context, the present proposition relies on the use of DCS to synthesize a controller C, which makes invariant a safe set of states E in a LTS-based system where E is inferred by boolean equations defining a control objective and an hypothesis on the inputs. The controller C given by DCS is said to be *maximally permissive*, meaning that it doesn't set values of controllable variables that can be either true or false while still compliant with the control objective. Actually, the BZR compiler defaults these variables to *true*. Optimization can be done at this level if this type of decision is too arbitrary (Guillet et al. 2012), but it goes beyond the scope of this work, which focuses on security, so the standard decision behavior given by BZR is kept. A smart home system, following the aforementioned execution principle, can now be designed using this framework.

Model instrumentation

When the various components and properties of a system are defined as behavior models (LTS, etc.) and synchronous equations, setting both the controllability and execution constraints enables the use of DCS.

Controllability

Controllability occurs naturally in the smart home domain. In the synchronous model, inputs are received each time the system is triggered, and these can come from both the environment – uncontrollable inputs $\mathcal{I}_U$ (e.g. a button is pressed by a human) – and the system itself – controllable inputs $\mathcal{I}_C$ (e.g. a device is forced to shut down by control system which is part of the execution loop).

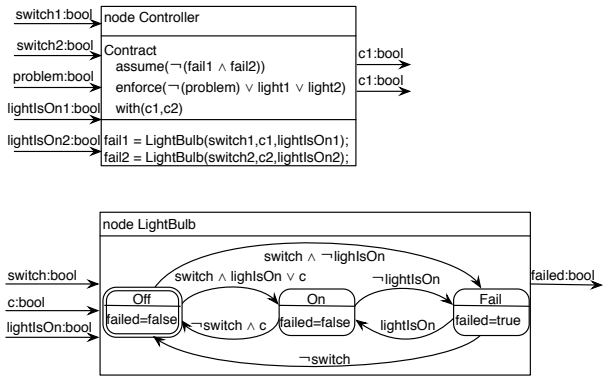

Figure 1: Controllable light bulb model

For example, let's take a system allowing a third party application to control two failure-prone light bulbs so that they can be forced to light up or remaining lit even if their switch is turned off by a human. Figure 1 represents the designed by constraint controller of this small system, instantiating two times the LightBulb node with a boolean variable c representing the aforementioned controllability), which takes amongst others the switches values as uncontrollable inputs $switch1, switch2 \in \mathcal{I}_U$ and the values given by the third party application as controllable boolean inputs $c1, c2 \in \mathcal{I}_C$. The statement $with$, declaring controllable variables, is actually implemented in BZR, which also allows to declare security constraints so that these variables can be valuated accordingly at each instant of the synchronous execution.

Constraints

We consider two types of security constraints expressed as boolean synchronous expressions: 1) *Hypothesis*, which are supposed to remain true for all executions, and 2) *Guarantee*, which are enforced to remain true using controllable variables if and only if the *Hypothesis* stays true form the beginning of the execution.

For example, let's say we want to be sure that, for all possible executions, at least one light bulb is lit up if a problem (uncontrollable information coming from observation) arises: this can be specified using the guarantee $\neg problem \vee light1 \vee light2$ (cf. *enforce* statement). However, the system is not controllable with this rule alone: light bulbs can be in fail mode at the same time while the system receives a *problem*, and thus the guarantee cannot be fulfilled for this specific execution. This situation would be found automatically when applying DCS, which would fail to build a controller.

Now, let's say that the light bulbs can still fail but are supposed to be repaired quickly enough so that they don't fail at the same time. This is an example of fault tolerance: ultimately everything can fail but if there is enough redundancy we can safely state that not everything will fail at the same time. The hypothesis $\neg(fail1 \wedge fail2)$ (cf. *assume* statement) represents this assumption in a synchronous boolean expression. Applying DCS using the BZR toolset on such a model gives back the C code of a controller taking $\mathcal{I}_U$ as inputs and providing the computation of $\mathcal{I}_C$ as outputs so that the system can now be executed, receiving both $\mathcal{I}_U$ and $\mathcal{I}_C$. DCS is able in this example to find automatically the correct controller code so that $c1$ and $c2$ can be valuated to $true$ or $false$ exactly when they should (e.g. when a problem arises, and lights are off, and $light1$ has failed, then $c2$ will be forced to false, etc.). From such a minimal example, we understand how DCS becomes interesting when the system's complexity increases while having to maintain its safety. If we add other failure-prone devices, impairment models, security constraints, etc. both designing and verifying the maximally permissive controller quickly start to be hard without appropriate tools.

Anticipating control evolution

Some modifications about the smart home model can hardly be modeled statically, for example, the evolution of the im-

pairment model associated to the user as defined in (Guillet, Bouchard, and Bouzouane 2013) cannot be known in advance. A solution to this problem would be to define the new model and constraints upon evolution, then apply DCS to obtain a new controller, and stop the current smart home system so that it can be replaced. However, this solution does not take the current system state into account. If the smart home is not in the same state as the one defined in the new model on which we applied DCS, then the new system cannot start securely (because the smart home model and the physical smart home have to react synchronously so they must be in the same corresponding state). This means that someone has to take care of setting the smart home back to its initial state.

Then a question arises : can the patients put the system back into the initial state, given their own impairments ? With the help of the system itself ? And if yes, can it be done at any time in a limited number of steps ?

This gives us a new verification problem: we want to ensure that before the next system modification (ie. impairement model evolution) the user can always put the system back to its initial state, possibly with the help of the system itself, and in a given number of steps (such that DCS will search a solution that does not require an infinite amount of steps, and will fail if such a solution doesn't exist, indicating that external help will be needed to do it).

During the execution, the system will update two variables by LTS, reflecting the amount of steps remaining to reach the initial state: *currentCtrlLimit* and *currentUserLimit*. *currentCtrlLimit* will be decreased each time a controllable transition is taken: a transition is controllable if the controller can always force it to be taken whatever the values of the other inputs. Figure 2 shows the instrumentation of such a state (*UnsafeY2*) of a LTS named Y, having a controllable transition towards the initial state (*CloseToSafeY1* bringing the system closer to *SafeY*). In this transition, *gY2to1* is a boolean equation encapsulating various inputs, and *ctrlY2to1* is a controllable variable, allowing to force this transition. The source state of such a transition has to be instrumented with the encapsulated equations: a boolean *waitForUserY* indicates that in this LTS Y, the state *UnsafeY2* has a controllable transition towards the initial state. In such a state, *currentUserLimitY* remains either the same or gets reinitialized depending on the value of *goSafe*, a global variable indicating the order to set the system in the initial state. In Y, *currentCtrlLimitY* is decreased if 1) the reinitialize order is received, 2) we are not waiting for the user and 3) Y is not in its initial state; else it is set back to 0, so we add this equation globally for Y:

currentCtrlLimitY $=$ if $(\text{goSafe} \wedge \neg(\text{waitForUserY}) \wedge \neg(\text{inSafeY}))$ then $(\text{currentCtrlLimitY} - 1)$ else (0)

Figure 3 shows the instrumentation of a state *UnsafeX2* of a LTS named X, being source of a transition toward the initial state *SafeX*. In such a state, one must be sure that the transition can always be forced by the user (through an uncontrollable input). If their is a conflict (both the user and the controller can force the transition), the designer decides which instrumentation to apply, they are both valid. *UnsafeX2* is instrumented by the indicated equations, which reflect the fact that we are waiting for a user action upon the reception of a reinit order (when *goSafe* is *true*) and the associated *currentUserLimitX* gets decreased each time we enter this state. It has to be noted that because *currentUserLimitX* is decreased only on entering the state, user controlled self transitions are not allowed in the model (because it counts as a user action, however because the state is not entered, *currentUserLimitX* does not get decreased accordingly).

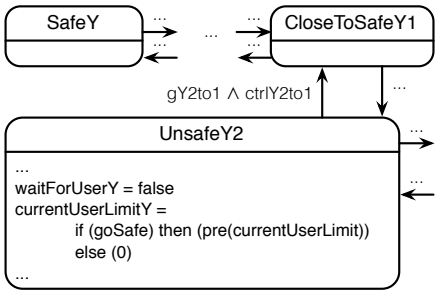

Figure 2: Instrumentation for controllable transitions toward the initial state

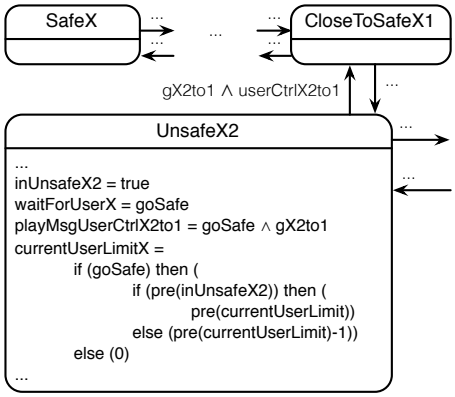

Figure 3: Instrumentation for manual transitions toward the initial state

When this instrumentation is carried through all the system's LTS, the associated defined by constraint controller can declare the following equations and *enforce* statement, meaning that when the number of actions has been reached during a system reinitialization, the initial state much be active (*safe* variable is true):
ctrlLimit $=$ number of allowed controllable steps
userLimit $=$ number of allowed user controlled steps
currentctrlLimit $=$ currentCtrlLimitX $+ ...Y + ...$
currentuserLimit $=$ currentUserLimitX $+ ...Y + ...$
safe $=$ inStateSafeX $\wedge ...Y \wedge ...$
reinitRule $=$ $((\text{ctrlLimit} + \text{currentctrlLimit}) \geq 0) \wedge ((\text{userLimit} + \text{currentuserLimit}) \geq 0) \wedge \neg((\text{ctrlLimit} + \text{userLimit}) == 0) \vee$ safe

A use case of this instrumentation methodology is shown in the next section.

Experiment

This section shows a partial smart home model, defined and instrumented with the previous methodology to ensure that the initial state is always reachable within a given number of steps. DCS is performed on this model, and a controller is obtained. Then a scenario in which the user impairments are modified is played, such that the smart home is forced to be properly reinitialized with the help of both the controller and the user. And finally, a new adapted controller replaces the old one, and the smart home continues its execution without being stopped at any time. Let's focus on two smart home components that will later be indirectly connected by a global constraint to enforce: a kitchen stove and its range hood fan.

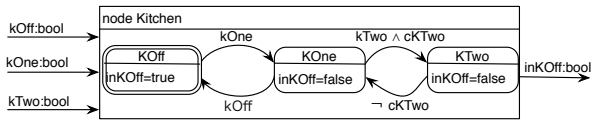

Figure 4: Kitchen model

Figure 4 shows a simplified behavior for the kitchen stove, allowing the activation of two stove burners, the activation of the second being potentially prevented by a controllable variable *cTwo* (which of course must have a physical correspondence: the control system is connected to the kitchen stove and can physically prevent the activation of the second stove if *cTwo* is set to *false* by the controller that will be obtained through DCS).

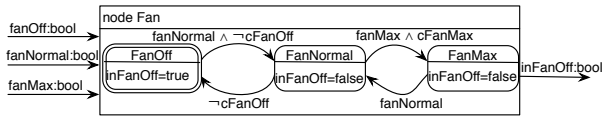

Figure 5: Range hood fan model

Figure 5 represents the behavior model of the range hood fan, which can be forced to go to (and prevented to exit) *FanNormal* with the help of the controllable variable *cFanOff*, and prevented to go to $FanMax$ with $cFanMax$.

(*FanOff*, *KOff*) is the initial state of this partial model. Now we want to enforce that for all possible executions, it is always possible to get back to this state in at most four steps, three of them (at most) involving user intervention. And getting back to this state must be done without jeopardizing the following control constraint: if the user's GDSAPD is above 3 units, then when the Kitchen is powered on, the range hood fan must be activated. This rule is defined by the following equation:

$\text{rule1} = \neg(\text{gdsapd} \wedge \text{inKOff}) \vee \neg(\text{inFanOff})$

Then the global rule to enforce is this rule combined with the generic rule as defined in the previous section :

$\text{enforce}(\text{rule1} \wedge \text{reinitRule})$

where *ctrlLimit* and *userLimit* are respectively set to 1 and 3. The sum of numbers represents the maximum amount of steps to go from any state to the initial state: the transition from *KTwo* to *KOne* is the only one controlled solely by the controller, so *KTwo* is instrumented like in Figure 2; respectively, *KOne*, *FanMax*, and *FanNormal* are instrumented like in Figure 3.

In the following scenario, cf. Figure 6, DCS is performed twice, the model difference being on the user GDSAPD: the first time, it is set to 4 units, and the second time it is set to 3 units. Each time, DCS succeeds, meaning that correct controllers — able to enforce the constraints for all executions — are found. The scenario shows how and when the second controller safely replaces the first one, accordingly to this methodology.

	Steps	1	2	3	4	5	...	6	...	7	...	8	...	9
Inputs	**fanOff**	**1**	1	1	0	0		0		**1**		1		1
	fanNormal	0	0	0	0	0		**1**		**0**		0		0
	fanMax	0	0	0	**1**	1		**0**		0		0		0
	kOff	**1**	0	0	0	0		0		0		**1**		**0**
	kOne	0	**1**	**0**	**0**	0		0		0		0		**1**
	kTwo	0	0	**1**	1	1		1		1		0		0
	goSafe	0	0	0	0	**1**		1		1		1		**0**
Outputs	**cFanOff**	1	**0**	0	**1**	1		**0**		0		**1**		1
	cFanMax	1	1	1	1	1		**0**		0		**1**		1
	cKTwo	1	1	1	1	**0**		0		0		**1**		1
	inFanOff	1	**0**	0	0	0		0		0		**1**		1
	inFanNormal	0	**1**	1	**0**	0		**1**		1		**0**		0
	inFanMax	0	0	0	**1**	**0**		0		0		0		0
	inKOff	1	**0**	0	0	0		0		0		**1**		**0**
	inKOne	0	**1**	**0**	0	**1**		1		1		**0**		**1**
	inKTwo	0	0	**1**	1	**0**		0		0		0		0
	safe	1	**0**	0	0	0		0		0		**1**		**0**

Figure 6: Simulations steps, with controller modification between steps 8 and 9

Step 1: initialisation The system starts in the initial state (*FanOff*, *KOff*), the variable *safe* is still true, and the system is ready to behave under control to various inputs. Both user buttons of the fan and the kitchen are set to *Off*.

Step 2: enforcing fan activation The user decides to activate the first stove burner. Because the user's GDSAPD is equal to four units, the system has to react so that the kitchen does not activate without its range hood fan. So *cFanOff* is automatically set to false by the controller, forcing the transition to *FanNormal*, thus fulfilling $rule1$.

Steps 3 and 4: both components to max The user decides to activate the second stove burner, which is allowed by default (*cTwo* remains *true* as there is no rule to enforce about it). Then the fan is set to *FanMax* which is also allowed by the controller (*cFanOff* goes back to the default value *true* because the fan cannot go back to *FanOff* in one step now).

Step 5: going back to the initial state The system receives a value *true* for *goSafe*, meaning that it has four steps — three of them involving a user action — to go back to its initial state (to fulfill *reinitRule*). The controllable variable *cKTwo* is forced to false, thus disabling the second stove

burner, and setting *currentCtrlLimit* to zero (no more automated steps remaining). From now on, the user has three actions to perform. These actions are indicated by the system with the help of various prompts (screens, lights, etc.). The actual implementation of these prompts is not shown in this study, the reader can refer to (Guillet, Bouchard, and Bouzouane 2013) for more information on how to safely implements the communication system.

Steps 6 and 7: trying to perform an incorrect action The user has to turn off both the kitchen and the fan. But these actions cannot bypass the *rule1* which still holds. Because the following steps involves user actions, these are not the actual next steps, but they can happen whenever, and the controller avoids anything that would set its two *limit* variables below zero (here, it prevents the user to reactivate the second stove burner by keeping *cKTwo* to false). The user decides to disable the fan completely so in step 6, the fan is first set to *FanNormal* (*currentUserLimit* is set to 2) and the controller locks the fan to this mode (*cFanOff* and *cFanMax* being set to *false*). However in step 7, when the user decides to disable the fan, the controller knows that it is forbidden because the kitchen is still activated, but because *cFanOff* is false, the fan remains activated too.

Step 8: initial state is reached Finally the user disables the kitchen completely, which also disables the fan, because the user button modeled by *fanOff* was set to *false*, and the system is set back to its initial state, thus setting *safe* to true. From now on, as long as *safe* remains *true*, the controller can be changed between two execution steps. Let's say that the user impairment model has changed (gdsapd now equals 3 units) and a new controller is synthesized. Then this controller can replace the old one, without powering down the whole system.

Step 9: the home is adapted to the evolution of its user The user decides to activate the first kitchen stove, and now, because the impairment model has been modified, *rule1* always remains *true* accordingly and the fan doesn't have to be activated at the same time any more.

Conclusion

This study has shown first results in using a generic methodology to ensure the ability of the smart home system to be reinitialized by itself and with the help of the user, depending on its impairments, thus without requiring any external help. While the methodology is not limited to smart homes, and can be used in other contexts as long as systems can be defined within the synchronous framework, it is especially useful here, as it solves the problem of modifying the controller when the user health evolves, problem that was not addressed in the previous proposition. Now when DCS is applied, the smart home controller knows 1) how to deal with all possible executions to keep the system safe, and 2) how to take the system back to its initial state so that the controller itself can be safely replaced by a new one, synchronized on this very same initial state. As a perspective, the current methodology could be improved by defining an adequate abstraction level so that smart home designers would not even have to learn about BZR to specify their system. Such an abstraction has for example been proposed in the reconfigurable embedded systems domain (cf. (Guillet et al. 2012)).

References

Altisen, K.; Clodic, A.; Maraninchi, F.; and Rutten, É. 2003. Using controller-synthesis techniques to build property-enforcing layers. In *Proceedings of the 12th European conference on Programming*, 174–188. Berlin, Heidelberg: Springer-Verlag.

Bouchard, K.; Bouchard, B.; and Bouzouane, A. 2012. Guidelines to efficient smart home design for rapid AI prototyping: a case study. In *Proceedings of the 5th International Conference on Pervasive Technologies Related to Assistive Environments*. New York, NY, USA: ACM.

Bouchard, B.; Giroux, S.; and Bouzouane, A. 2007. A keyhole plan recognition model for Alzheimer's patients: first results. *Journal of Applied Artificial Intelligence (AAI)* 21(7):623–658.

Fortin-Simard, D.; Bouchard, K.; Gaboury, S.; Bouchard, B.; and Bouzouane, A. 2012. Accurate passive RFID localization system for smart homes. In *IEEE 3rd International Conference on Networked Embedded Systems for Every Application (NESEA)*, 1–8.

Guillet, S.; de Lamotte, F.; Le Griguer, N.; Rutten, É.; Gogniat, G.; and Diguet, J.-P. 2012. Designing formal reconfiguration control using UML/MARTE. In *7th International Workshop on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*.

Guillet, S.; Bouchard, B.; and Bouzouane, A. 2013. Correct by construction security approach to design fault tolerant smart homes for disabled people. In *EUSPN*.

Lapointe, J.; Bouchard, B.; Bouchard, J.; Potvin, A.; and Bouzouane, A. 2012. Smart homes for people with Alzheimer's disease: adapting prompting strategies to the patient's cognitive profile. In *Proceedings of the 5th International Conference on Pervasive Technologies Related to Assistive Environments*, 30:1–30:8. ACM.

Marchand, H., and Samaan, M. 2000. Incremental Design of a Power Transformer Station Controller using a Controller Synthesis Methodology. *IEEE Trans. Software Engin.* 26(8):729–741.

Novak, M.; Binas, M.; and Jakab, F. 2012. Unobtrusive anomaly detection in presence of elderly in a smart-home environment. In *ELEKTRO*.

Pigot, H.; Mayers, A.; and Giroux, S. 2003. The intelligent habitat and everyday life activity support. In *5th International Conference on Simulations in Biomedicine*. Slovénie: 5 th international conference on Simulations in Biomedicine, avril 2003.