Reconfiguration Control and Decision, 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 optimising the reconfiguration under control of a smart home system dedicated to people with disabilities. The paper presents the transformations to combine both the control and decision parts of the reconfiguration management system and then demonstrates its execution.

Introduction
While the design of smart environments dedicated to people with disabilities 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) or adapting assistance to the person's cognitive deficit (Lapointe et al. 2012), security in the smart home context is a primary concern (Pigot, Mayers, and Giroux 2003).

In (Guillet, Bouchard, and Bouzouane 2013), we have shown how to design secure environments that can be formally controlled to remain in a correct state. In this study we show how the actual control code of such environments can be optimised using transformations to accept a decision module that can optimise a reconfiguration using multiple ones allowed by the controller at each execution step. The first part details the underlying synchronous framework, the second one shows the transformations to accept a decision module, and the third part demonstrates the approach.

Synchronous framework
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, depicting the same execution principle, allowing the use of a Reconfiguration controller task.

Designing such a task (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 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 \( x = x_0, x_1, \ldots \) and \( y = y_0, y_1, \ldots \). Evolution of \( y \) over time is given by the following system of equations:

\[
\begin{align*}
\begin{cases}
  y_0 &= x_0 \\
  y_t &= y_{t-1} + x_t & \text{if } t \geq 1
\end{cases}
\end{align*}
\]

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: 

\[
y = x \rightarrow
\]
\( \text{pre}(y) + x; \), 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.

**Labelled Transition System (LTS)**

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

Each transition has a label of the form \( g \rightarrow a \), where \( g \in \text{Bool}(I) \) must be true for the transition to be taken (\( g \) is the guard of the transition), and where \( a \in O^* \) 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 \( (g, a, q', q) \), and state \( q' \) is the destination. A transition \( (g, a, q') \) will be graphically represented by \((g \rightarrow a \downarrow q')\).

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: \( \langle Q_1, q_{01}, I_1, O_1, T_1 \rangle || \langle Q_2, q_{02}, I_2, O_2, T_2 \rangle = \langle Q_1 \times Q_2, (q_{01}, q_{02}), I_1 \cup I_2, O_1 \cup O_2, T \rangle \) with \( T = \{((q_1, q_2) \xrightarrow{(g_1 \land g_2)/(a_1 \land a_2)}, (q'_1, q'_2))) || (q_1 \xrightarrow{g_1/a_1} q'_1) \in T_1, (q_2 \xrightarrow{g_2/a_2} q'_2) \in T_2 \} \).

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 \( I \) of inputs of \( S \) is partitioned into two subsets: the set \( I_C \) of controllable variables and the set \( I_U \) of uncontrollable inputs. Formally, \( I = I_C \cup I_U \) and \( I_C \cap I_U = \emptyset \). As a consequence, a transition guard \( g \in \text{Bool}(I_C \cup I_U) \) can be seen as a function from \( 2^{I_C} \times 2^{I_U} \) into \( B \). 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 \( (q, g, a, q') \in T \) is controllable iff \( \exists X \in 2^{I_C} \) such that \( g(X, Y) = false \). In the proposed framework, the following function \( S_c = \text{make_invariant}(S, E) \) 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 \( \text{make_invariant} \) synthesizes and returns a controllable system \( S_c \), if it exists, such that the controllable transitions leading to states \( q \in E \) are inhibited, as well as those leading to states from where a sequence of uncontrollable transitions can lead to such states \( q \notin E \). 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 but this type of decision is too arbitrary and the current proposition – which relies on BZR – also proposes a way to integrate a custom decision module, defined by a function interface in C, which can be implemented by the designer. This way, when the controller states that more than one configuration is accessible, this decision module can safely choose one of them to optimize the transitions choices inside \( E \). The actual implementation of such a module goes beyond the scope of this paper so it will be assimilated to a simple random choice.

**Potential transition**

A transition is potential if the current state is its source and at least one authorized values combination for the controllable variables (if there are any) associated to the uncontrollable input values allows its guard to be evaluated to true.

**System equivalence**

Two systems \( S \) and \( S' \) sharing the same states and the same initial state are equivalent with respect to (wrt) an objective \( \mathcal{F} \) and an hypothesis \( \mathcal{H} \) they have in common iff for all correct execution wrt \( \mathcal{H} \):

- objective \( \mathcal{F} \) is guaranteed in both \( S \) and \( S' \);
- every potential transition for each step of execution in a system is also potential in the other.

Two equivalent systems only differ on the final potential transition choice at each step. This choice being always correct wrt \( \mathcal{F} \) and \( \mathcal{H} \).

\(^1http://bzr.inria.fr/pub/bzr-manual.pdf\)
State and configuration accessibility

A state of a LTS is accessible if at least one transition going to it from the current state is potential (which includes by default the case where this state is the current state and no outgoing transition is potential). A configuration, or macro-state of a system, is accessible if each state of its composition is accessible.

Decision integration

Let \( S = \langle Q, q_0, I, O, T \rangle \) be a system defined as a LTS or a synchronuous composition of several LTS, and \( F \) its control objective guaranteed wrt an hypothesis \( \mathcal{H} \). The objective is to build a system \( S' = \langle Q, q_0, I', O', T' \rangle \), equivalent to \( S \) wrt \( F \) and \( \mathcal{H} \), integrating a decision system that makes a choice on accessible configurations at each step.

Modifying inputs and outputs

\( S' \) encapsulates both the inputs and outputs of \( S \) and extends them in order to take into account the configurations accessibility as output and the choice of an accessible configuration as input. Let \( n \) be the number of configurations given by all possible combinations of the states of the LTS of \( S \), and \( m \) be the number of LTS of \( S \). Let \( 1 \leq i \leq n \) and \( 1 \leq j \leq m \).

Let \( D \) and \( P \) be two boolean sets, with \( \text{card}(D) = \text{card}(P) = n \), where \( d_i \in D \) corresponds to the choice given by the decision system where only one configuration is chosen at each step, formally: \( \forall d_i \in D, \{d_i = \text{false} | x \neq 1 \leq x \leq n\} \)

(1)

Each configuration accessibility boolean \( p_i \) is defined by an equation evaluating the potentiality of all transitions going to each state of the configuration \( i \). This equation is true if all concerned states are accessible. So let \( p_i \in \mathcal{P} \), \( p_i = \text{true} \).

The inputs and outputs \( I' \) and \( O' \) of \( S' \) are defined as:

\[
I' = I \cup D, \text{with } I \cap D = \emptyset
\]

\[
O' = O \cup P, \text{with } O \cap P = \emptyset
\]

(1)

(2)

Each configuration accessibility boolean \( p_i \) is defined by an equation:

\[
p_i = \bigwedge_{j=1}^{m} s_{ij}
\]

(2)

with each \( s_{ij} \) defined by:

\[
s_{ij} = \sqrt{(g \land b_j)}(q \longrightarrow q') \in T
\]

(3)

\[
b_q = \begin{cases} \text{true} & \text{if } g \text{ is the current state} \\ \text{false} & \text{otherwise} \end{cases}
\]

(4)

To ensure that, at each step, at least one accessible configuration exists wrt \( F \), this property has to be added as a control objective so that DCS can enforce it. Thus, the control objective \( F' \) of \( S' \) is specified by:

\[
F' = \bigwedge_{i=1}^{n} p_i \land F, p_i \in \mathcal{P}
\]

(4)

Correct decision as input

Now that a way to output the next accessible configurations has been specified, processing the final configuration choice (coming as input from a decision module) should be defined. The decision system is seen as a black box from the point of view of DCS, only its inputs \( P \) and outputs \( D \) are known. However it is necessary to specify some behavior hypothesis from this system in order to give the following formal information to DCS: 1) at least one accessible configuration, given in the previous step, should be given as input (at least one boolean of \( D \) is true); 2) at most one configuration is chosen (at most one boolean of \( D \) is true).

The hypothesis \( \mathcal{H}' \) of \( S' \) extends the hypothesis \( \mathcal{H} \) of \( S \) in order to indicate the previous properties. The \( \text{pre} \) operator, as it is defined in synchronous languages, allows here to refer to the values of configuration accessibility (values of \( P \)) at previous step. Formally, let \( \text{atLeastOne} \) and \( \text{atMostOne} \) be two equations defining respectively 1) the fact that at least one accessible configuration is chosen and 2) at most one configuration is given:

\[
\mathcal{H}' = \mathcal{H} \land \text{atLeastOne} \land \text{atMostOne}
\]

(5)

Thus, \( \mathcal{H} \), \( \text{atLeastOne} \) and \( \text{atMostOne} \) should always be true. Let \( z \) be the identifier of the initial configuration, \( 1 = z, \text{atLeastOne} \) is specified by:

\[
\text{atLeastOne} = (\text{true} \rightarrow \text{pre}(p_z) \land d_z) \lor (\bigvee(\text{false} \rightarrow \text{pre}(p_i) \land d_i), i \neq z)
\]

(6)

which means that at first step, the initial input given by decision sets \( d_z \) to \( \text{true} \), then for the next steps \( d_i \) should be \( \text{true} \) only when \( \text{pre}(p_i) \) or \( \text{pre}(p_z) \) is \( \text{true} \). And \( \text{atMostOne} \) is given by:

\[
\text{atMostOne} = \neg(\bigwedge_{x=1}^{n-1} (d_x \land \bigvee_{y=x+1}^{n} d_y))
\]

(7)

Modifying the LTS transitions

In order to finalize a step execution, it is necessary to modify the LTS transitions of \( S' \), so that they react only on occurrence of inputs coming from the decision system (inputs from the specification of \( S \) are entirely processed by the previous equations defining \( P \)). The LTS should also be modified in order to output, besides its original outputs, a set of boolean values, named \( B \), allowing to identify their current state, which is needed to evaluate \( b_q \) in the specification of a state accessibility.

Let \( S_k \) and \( S'_k \) (with \( 1 = k = m \)) be respectively an LTS of \( S \) and a transformation of \( S_k \) as an LTS of \( S' \). Let \( B_k \) be a set of boolean variables and \( n_k = \text{card}(B_k) = \text{card}(Q_k) \).

The transformation of each \( S_k \) is specified by:

\[
\forall S_k = \langle Q_k, q_{0_k}, T_k, O_k, T_k \rangle, S'_k = \langle Q_k, q_{0_k}, D, O'_k, T'_k \rangle
\]

with \( O'_k = O_k \cup B_k, T'_k \subset (Q \overset{B_{\text{bool}}(I_k)/O'_k}{\longrightarrow} Q), O''_k \)
being a conjunction of $\mathcal{O}'_k$, and $\mathcal{T}'_k$ being defined by:

$$
\mathcal{T}'_k = \{(q_k \text{ } \vdash_{\mathcal{T}'_k} \{ \forall i \in D | q_i^C, q_i^U \in q_i \}, (a_i)_{i \leq 1} \to \{ (q_k \xrightarrow{g/a} q'_k) \}) | q_k \in \mathcal{Q}_k \}
$$

$$
b_{q_j} = \begin{cases} 
true & \text{if } q_{k_j} = q'_{k_j}, q_{k_j} \in \mathcal{Q}_k \\
false & \text{otherwise} 
\end{cases}
$$

Finally, in accordance with the synchronous composition principle, transitions $\mathcal{T}'$ of $S'$ are determined by:

$$
\mathcal{T}' = \{(q \text{ } \vdash_{\mathcal{T}'_k} \{ (A_{i \leq 1})_{i \leq 1} g/a \to q'_k) \}) | (q_k \xrightarrow{g/a} q'_k) \in \mathcal{T}'_k, (q_k, q'_k) \in q \times q' \}
$$

This transformation proposition has shown the way to instrument an equivalent version $S'$ of $S$. This version allows the designer to implement its own decision system without interfering with the control objectives as long as it complies with the control interface, which is:

- a set of booleans $P$ for configuration accessibility as input;
- a set of booleans $D$, containing one and only one final choice (one boolean set to true) as output among accessible configurations given at previous step (so there is a direct correspondence between $D$ and $P$).

**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 $I_U$ (e.g. a button is pressed by a human) – and the system itself – controllable inputs $I_C$ (e.g. a device is forced to shut down by control system which is part of the execution loop).

$$
\begin{array}{c}
\text{lightIsOn1:bool} \\
\text{lightIsOn2:bool} \\
\text{problem:bool} \\
\text{switch1:bool} \\
\text{switch2:bool} \\
\end{array}
$$

![Controller Diagram](image1)

**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 $\in I_U$ and the values given by the third party application as controllable boolean inputs $c1, c2 \in I_C$.

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 $\neg \text{problem} \lor \text{light1} \lor \text{light2}$ (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 $\neg (\text{fail1} \land \text{fail2})$ (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 $I_U$ as inputs and providing the computation of $I_C$ as outputs so that the system can now be executed, receiving both $I_U$ and $I_C$. 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 $c1$ 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.

**Experiment**

This section show an application of the previously defined transformations to a partial smart home model on which DCS is then applied to obtain a controller regarding some security constraints.
Architecture

The smart home system of this experiment is based on the one described in [REF], where a person can use several devices (lights, TV, radio, kitchen stove, etc.) and several prompts can be used to communicate with the person (iPad, Speaker, Light). Figure 2 shows a subset of this model, on which the transformations will be applied. The two automata respectively describe the fact that we can use either the iPad, the Speaker, or the Light to communicate with the user, and the fact that the cognitive load on the user is "high" if some music is played while cooking. As a security measure, when the cognitive load is high, the iPad must not be chosen to communicate with the user, so we want to enforce \( \neg (\text{IPad} \land \text{HighCL}) \).

Applying transformations

To avoid redundancies, Figure 3 only shows the new synchronous equations obtained using the previously defined equations (the numbers in the figure are references to these equations), the transformation of the automata being represented in Figure [REF]. What we obtain is the BZR code of a controller node that takes and provides all original inputs/outputs () and also all new inputs/outputs from and to a decision system that can be plugged using this interface. The controller allows the external decision system to influence its execution while remaining able to enforce \( \neg (\text{IPad} \land \text{HighCL}) \).

Execution

The compilation of the BZR code, involving DCS, provides an executable controller. Figure 4 shows its two first execution steps, illustrating the control and decision principle. The initialisation shows that the very first decision inputs must match the initial state of the controller from the designer definition in Figure 2, then the transformed automata react to be set in "Speaker" and "LowCL". In this first step, the user is not cooking nor listening to music. All controllable variables can remain true which allows the decision system to choose either "Light" or "Ipad". Let’s say the decision to choose the iPad is taken. Then in step two, the decision boolean influence the controller so that it goes to "Ipad" (and remains in "LowCL"). However, in this step, the user is now both cooking and listening to music, so for the next step, the iPad must not be the chosen prompt to communicate. By setting "cIPad" to false, the controller forces to exit the "Ipad" state, but "Speaker" and "Light" can both be accessed, and this choice is provided to the decision system, which for example chooses to use the Light. The decision is then provided at the beginning of state three.

Conclusion

This study has shown first results in using the combination of formal control and decision to manage the reconfigurations of a smart home system. The aim was to show that while the control part allows to enforce the system to remain in a set of correct states, the actual choice of a correct state can be optimised and delagated to a decision function. The transformation of the controller specification so that a decision function can be plugged using an interface has been demonstrated on a proof of concept showing a subset of a smart home system where the choice of a prompt to communicate with the user must both controlled (reduced into a set of correct choices) and decided (from the remaining choices
allowed by the controller).

References


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


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