Safe and Automatic Addition of Fault Tolerance for Smart Homes Dedicated to People with Disabilities

Sébastien Guillet, Bruno Bouchard and Abdenour Bouzouane

Abstract In this chapter, we discuss a project of the LIARA laboratory that introduces a methodology to design and control smart homes dedicated to people with disabilities. In this context, this project aims at improving the security of the environment through a design methodology involving formal synthesis techniques.

Keywords Smart home · Security · Fault tolerance · Discrete controller synthesis

1 Introduction

Ubiquitous computing, making us more connected to our environment and other people, is challenging the way we live through many different means, ranging from anticipating our needs to securing our environment and automating routine physical tasks. Contributions to ubiquitous computing has lead the scientific community to the smart home era [1], which involves a wide range of these means to liberate us from usually hard and repetitive work at home and to help us live more independently.

Enhancing independence is actually the core concept of smart homes dedicated to disabled people. For example, such a house can be designed to help a human resident suffering from a cognitive deficit to complete his activities of daily living (ADL) [2] without the need of additional human assistance. De-signing this kind of smart homes involves many challenges, including blending unobtrusively into the home environment [3], recognizing the ongoing inhabitant activity [4], localizing

S. Guillet (✉) · B. Bouchard · A. Bouzouane
LIARA, Université du Québec À Chicoutimi, 555 Blvd Université EST, G7H2B1 Chicoutimi, QC, Canada
e-mail: sebastien.guillet1@uqac.ca
URL: http://liara.uqac.ca

B. Bouchard
e-mail: bruno.bouchard@uqac.ca

A. Bouzouane
e-mail: abdenour.bouzouane@uqac.ca

K.K. Ravulakollu et al. (eds.), *Trends in Ambient Intelligent Systems*,
Studies in Computational Intelligence 633, DOI 10.1007/978-3-319-30184-6_4

objects [5], adapting assistance to the person's cognitive deficit [6], and securing the environment [7].

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. However, even if a smart home system is usually build to last, it might not be the case for its very own components [8]: lights, screens, sound system, and many other important equipment can fail during the lifetime of the system. In this context, providing a viable security strategy over time requires to take failures into account due to their high probability and potential harmful consequences if not taken seriously [9].

To operate properly over time, the main concern of a fault tolerant smart home system is, upon detection of a failure, knowing how to "react appropriately". Let's suppose that a detected random failure affects an arbitrary component, is the system still able to provide both protection and adequate assistance with respect to the person's disabilities?

Answering such a question usually requires to solve a non-trivial combinatorial problem: a smart home is supposed to be composed of many dynamical components (electrical shutters, lights, ventilation systems, etc.), each one having several exclusive execution modes (opening, opened, on, off, disabled, failed, etc.) which can be observed using sensors; These components are concurrently executed and their execution modes can be influenced upon reception of events which can be external to the system (e.g. the user pushes a button) and/or internal (e.g. a security system prevents a hair dryer from powering on because it is too hot). Here lies the complexity: ensuring that the system will respect a security property (e.g. being able to provide assistance even if a component fails) requires to verify that this property holds for each accessible combination of execution modes.

Now let's introduce the notion of *controllability*, which happens when a component offers an interface so that a control system—a program named controller—can send events to constrain its behaviour. Controllability is very common in the context of ubiquitous computing—smart home is no exception—where almost every component provides such an interface so that it can be adapted to a situation. A system is said to be controllable with respect to a temporal property whether given its dynamicity and controllability, it exists a controller able to constrain the system such that the temporal property holds for all possible executions.

Applied to smart homes for disabled people, a smart home has the capacity to undertake a security constraint[1] over time if and only if a control system can be proven to keep the system in execution modes complying with the constraint. But even if a system under control can be proven correct using verification techniques, its controller is not guaranteed to be interesting. For example, let's take a controllable component which can be prevented to start, and a security constraint such that this component must not be started when temperature is above 50 °C; now let's

[1]Or, to be more general, a "quality of service constraint".

build a controller which always disables this component; then the system under control can be verified to be correct, however we understand that the implemented controller should be more *permissive* when temperature is lower than 50 °C.

Basically, in presence of controllability, the designer of a fault tolerant smart home system has to face two non-trivial problems: building a permissive controller and verifying the system under control. It happens that these two problems are the specialty of a formal technique named Discrete Controller Synthesis (DCS) [10]: given a system's dynamicity, controllability, and temporal constraints, if a control solution exists then DCS is able to provide automatically a controller which is both correct by construction and maximally permissive, meaning that it valuates control events tied to control interfaces of dynamic components only when the system has to be constrained.

In [11], we made a contribution giving concepts on representing the behavior of a smart home system dedicated to disabled people. DCS was shown to be applicable using this representation to create a controller designed to keep the smart home in a correct state. The contribution of the present study relies on the definition of a design methodology around these concepts, and shows examples on how a smart home system can be specified, so that DCS can be applied to solve concrete fault tolerance related problems in smart homes for people with impairments.

The paper is organized as follows. Section 2 presents the related work about security in the smart home domain and justifies the choice of DCS over other formal techniques to provide a solution for the controllability problem. Section 3 describes the synchronous framework which serves as a foundation for modeling and applying DCS. Section 4 explains how to de ne a smart home model using this synchronous framework. Section 5 shows the application of DCS on such a model. Section 6 details the experiments we conducted using a partial model of our own smart home equipment. The model is kept partial so that both DCS application and controller execution remain easy to follow step by step. Finally, Sect. 7 concludes the paper and outlines future developments of this work.

2 Related Work

The literature on which this study is based can be divided into three major domains. The first one is smart home modeling [7, 12, 13]: this work aims to give a framework to represent key aspects of a smart home (*dynamicity*, *controllability*, and *temporal constraints*) so that formal techniques such as DCS can be applied; these aspects are generic and need concrete definitions for the smart home context. The second domain is smart home security [2, 6, 9, 14–18]: what makes a smart home secure, especially a smart home for disabled people? What are the techniques employed to provide some form of security in this context? Finally, the third domain is formal techniques [11, 19–22]: failing to provide a correct smart home behavior for all its possible executions could easily have harmful consequence for a

vulnerable inhabitant, so how do we prove a smart home to be secure? And what is the best technique to apply given the smart home properties?

2.1 Smart Home Modeling

Research projects related to modeling of smart homes for disabled people usually share many concepts based on representing: the smart home elements (devices, doors, lights, etc.) with their positions and execution modes, the person itself (its state of mind, behavior, position, cognitive profile, etc.), and the global execution model (how the smart home is supposed to run and process events in order to provide both assistance and security using artificial intelligence).

In [7, 12], Pigot et al. present respectively (1) a meta-model containing generic knowledge of a smart home system for elders suffering from dementia and (2) a corresponding model showing cognitive assistance and telemonitoring concepts. These works detail a pervasive infrastructure and applications to provide assistance to elders with cognitive deficiencies using two kinds of interventions: one operating inside the home to help the person to complete its ADL in case of difficulties, and another one establishing communication outside the home to send message to caregivers, medical teams or families.

In [13], Lat et al. give an overview of an ontology-based model of a smart home dedicated to elderly in loss of cognitive autonomy. The ontological architecture is partitioned into seven sub-domains: (1) Habitat, describing the home structure (rooms, doors, windows, etc.); (2) Person, which can describe the patient itself (medical history, behavior, etc.) and the various persons supposed to interact with the patient and/or the habitat (medical actor, habitat-staff, friend, etc.); (3) Equipment, which defines the various home appliances; (4) Software, de-scribing reusable software modules of the smart home system; (5) Task, detailing the observable tasks that the patient, the personal, and house itself, can per-form; (6) Behavior, regrouping life habits and critical physiological parameters; (7) Decision, related to the smart home adaptation behavior.

These works constitute the foundation of Sect. 4, which will synthesize and show how to represent their ideas into a formal synchronous model, so that security properties can be set and verified.

2.2 Smart Home Security

Due to its importance, security in the context of smart homes—especially those dedicated to disabled people—has been widely covered in the literature. Methods employed to secure a smart home target three main layers: (1) Fault tolerance, as a smart home system is supposed to experience failures through its lifetime; (2) Smart

sensing technology, so that ADL can be monitored accurately; (3) Appropriate smart home behavior depending on the patient deficiencies.

Fault tolerance Failures in a smart home system may occur on several levels [9, 23]: a smart home is typically a set of hardware and software components communicating together, so failures can happen either happen at hardware, software or communication level.

Sensors, actuators, displays, speakers, lights, etc. are traditional failure-prone smart home hardware components. They wear over time, can be damaged, can go down if they are battery powered, can cease to communicate because of limited signal strength, can operate incorrectly because of a manufacturing defect, etc. A single failure at this level can compromise the smart home security.

A smart home system also typically contains multiple software components running together (operating systems, artificial intelligences, controllers, etc.), including commercial applications (i.e. trusted black boxes). Unless formally checked against security requirements, few assumption should be made about applications. Even software verified by competent and credible experts can contain bugs. The malfunction in the control software in Ariane 5 Flight 501 is an example of such a bug, which remained undetected through several human-driven verification processes [24].

Communication between hardware and software components happens through wired and wireless channels. Communication failures are mainly caused by low signal strength (e.g. two mobile wireless devices communicating together get separated by a too long distance) or heavy traffic. They are not really hardware or software related, but can be (wrongly) perceived as such because affected components cease to communicate and become unavailable, making these failures important to detect.

When a hardware, software or communication component failure is detected, two common responses are (1) using an equivalent component (redundancy) [17] and (2) executing the system in a degraded mode, allowing it to work correctly through failures using a safe subset of its functionalities [25].

These methods will be used in Sects. 4 and 5 as a base to build a fault tolerant smart home.

Smart sensing Increasing a smart home robustness also involves an effective sensing system. Identifying ADL [4, 15], locating a person or mobile components [5], recognizing the mood [26], etc. are examples of smart sensing features that can be integrated into a smart home.

Section 4 takes the presence of these kinds of high level sensors (artificial intelligences) into account, so that security rules can be based on their information.

In [17], Bouchard et al. give guidelines to integrate and execute artificial intelligence modules into a generic smart home system. We will take advantage of these guidelines to model a system that will comply with the same execution principles.

Impairment adaptation Knowing how to adapt a smart home for disabled people to their impairments is a sensitive and complex problem largely discussed in the literature. Smart homes usually contain technological devices aiming to provide adapted cognitive assistance—or *prompts*—when needed. Typical prompts can be based on sounds, music, spoken messages, photos, videos, lights, etc. Implementing an adequate prompting system is actually the core concept or impairment adaptation [6, 7, 15, 16].

In [6, 17], the authors provide experimental results on prompt efficiency according to cognitive pro les. Section 4 shows how to represent these relations, so that security properties can be defied for the prompting system.

Combined together, all these security layers bring a new question. If the house is equipped with redundant critical equipment, if the prompting system can be adapted in accordance with the severity and characteristics of the patient's impairments, and if the context (ADL, mood, position, etc.) can be accurately monitored: how do we prove that, in case of a failure, the smart home system can still provide adequate assistance if the failure impacts the prompting system or the way ADL can be monitored?

Proving it for every allowed failure, every possible execution, every context, etc. is essentially a combinatorial explosion problem that is very di cult to solve without appropriate tools. This is where verification techniques come in.

2.3 Formal Methods

Many research work contribute to formal modeling and verification of user's in-teractions, hardware/software components and control algorithms in the smart home domain [27–29]. However, formal verification suppose that a complete system can be modeled before being applied. In the modeling methodology proposed in [29], a modeling step named "control algorithm modeling" is explicitly required. This step is about the definition of a module which, given (1) the system current configuration, (2) incoming message from the system or its environment, and (3) control rules, makes a reconfiguration decision and sends triggering messages to the associated devices for performing the required operations.

This step is precisely the part that is di cult to design because of the combinatorial problem we are facing in this context. This is the reason why we are more interested into an alternative method, DCS, which is able to both build the control part automatically and perform formal verification of the system.

Regarding smart homes, a smart home system can be considered as a specialization of autonomic computing systems [30], which adapt and reconfigure themselves through the presence of a feedback loop. This loop takes inputs from the environment (e.g. sensors), updates a representation (e.g. Petri nets, automata) of the system under control, and decides to reconfigure the system if necessary. This consideration is detailed in Sect. 3. Describing such a feed-back loop can be done in

terms of a DCS problem. It consists in considering on the one hand, the set of possible behavior of a discrete event system [31], where variables are partitioned into uncontrollable and controllable ones. The uncontrollable variables typically come from the system's environment (i.e. "inputs"), while the values of the controllable variables are given by the synthesized controller itself. On the other hand, it requires a specification of a control objective: a property typically concerning reachability or invariance of a state space subset. Such a programming makes use of reconfiguration policy by logical contract. Namely, specifications with contracts amount to specify declaratively the control objective, and to have an automaton describing possible behavior, rather than writing down the complete correct control solution. The basic case is that of contracts on logical properties i.e., involving only Boolean conditions on states and events. Within the synchronous approach [19], DCS has been defined and implemented as a tool integrated with synchronous languages: SIGALI [20]. It handles transition systems with the multi-event labels typical of the synchronous approach, and features weight functions mechanisms to introduce some quantitative information and perform optimal DCS.

One of the synchronous languages it has been integrated with is BZR [22], which is used in this work; BZR actually includes a DCS usage from Sigali within its compilation. The compilation yields (if it exists) the code of a correct-by-construction controller (here in C language), which can itself be compiled to be executed into the smart home system.

Based on the synchronous characteristics of a smart home system, Sect. 3 sets the synchronous context and notations so that they can be applied to smart home modeling in order to perform DCS.

DCS has already been successfully applied in various domains, e.g. adaptive resource management [32], reconfigurable component-based systems [33], reconfigurable embedded systems [11], etc. However, DCS in the context of smart homes has not been seen until very recently, where it was introduced in [34] and [35]. Both studies show preliminary results on how DCS could be applied to secure a smart home, [34] having a general point of view, and [35] a specific one regarding fault tolerance. They have a common perspective to show results with more types of objectives and adaptive control in order to go beyond a demonstration of DCS applicability and really show its relevance and efficiency in this context. This study takes this perspective into account to give a contribution on actual usage of DCS to solve concrete smart home problems—related to fault tolerance—through a modeling methodology using BZR.

3 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 [36, 37]—gets justified as appropriate to specify smart home systems.

3.1 Execution Model

In [11], the execution model of a reactive system under control is depicted, cf. Fig. 1. 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 [17], guidelines to build the software architecture of a smart home system are presented, cf. Fig. 2. 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.

Immediately, we can see similarities arising from such an architecture com-pared to the reactive system execution model. If we add a reconfiguration controller as a third party application in this software architecture, then we obtain the same execution principle presented in Fig. 1: 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

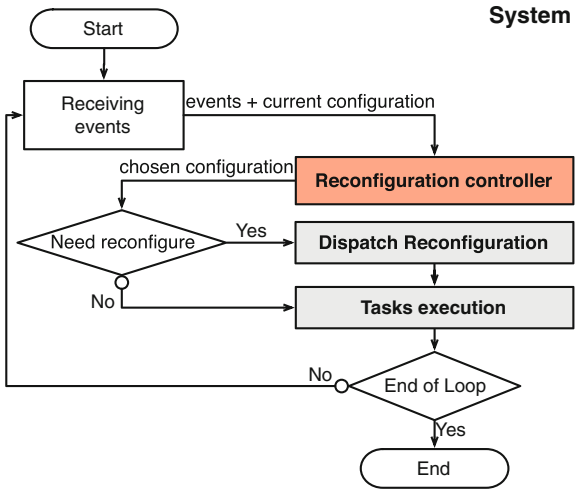


Fig. 1 Configuration processing flowchart

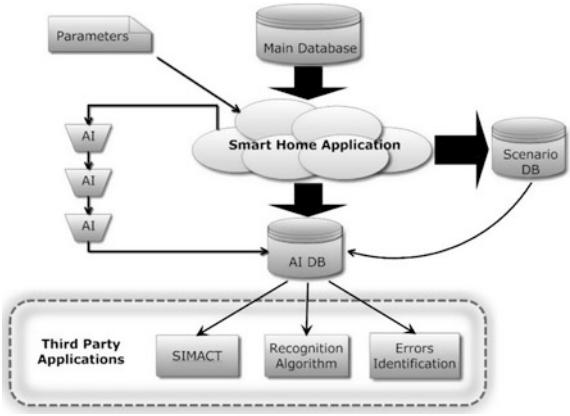


Fig. 2 Smart home software architecture

(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. Behavioural modeling can be performed using various formal representations, e.g. State charts, Petri-nets, Communicating Sequential Processes or other ways. The toolset we use in this work—BZR and SIGALI—brings us to de ne our system in terms of synchronous equations and Labelled Transition Systems.

3.2 Synchronous Equation

In a declarative synchronous language, semantic is expressed in terms of data flows: 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 data flows 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:

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

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 de ne 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[2]). To represent the system execution modes, BZR also allows to de ne automata, or Labelled Transition Systems, each state encapsulating a set of synchronous equations evaluated only when the state is activated.

3.3 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 {true, false}, then a guard g belong to $g \in \text{Bool}(\mathcal{I})$ 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 $\mathcal{T} = \left\{ \left((q_1, q_2) \xrightarrow[\wedge]{(g_1} g_2) / (a_1 \wedge a_2) (q'_1, q'_2) \right) | (q_1 \xrightarrow[/]{g_1} a_1 q'_1) \in \mathcal{T}_1, \ (q_2 \xrightarrow[/]{g_2} a_2 q'_2) \in \mathcal{T}_2 \right\}$. Note that this synchronous composition is the simplified one presented in [37], 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.

3.4 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 $\mathcal{H}$.

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

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 = *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 [11], 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.

4 Smart Home Model

From the various smart home presentations found in the related work, a smart home system for people with disabilities can be abstracted as a hierarchy of hardware and software components (dynamic or not), sensors, and effectors distributed among several interconnected rooms, helping a person with impairments to perform ADL. Showing how to specify all these features within a synchronous model is the aim of this section.

4.1 Dynamic Components

The top component of the hierarchy is the system itself. In accordance to the synchronous execution model, let S be the LTS of the system, taking inputs $\mathcal{I}$ from

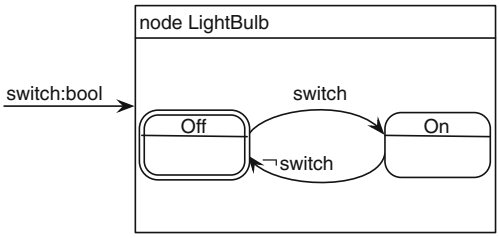


Fig. 3 Simple light bulb model

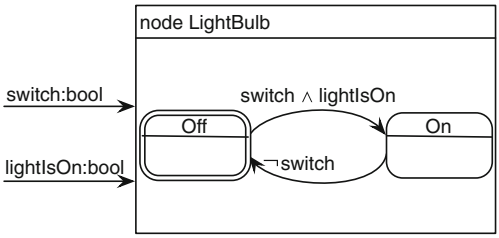


Fig. 4 Observable light bulb

its effectors (buttons, touchscreens, controllable interfaces, etc.) and producing outputs $\mathcal{O}$ from its sensors (low level sensors, AI, any device producing notifications, etc.) each time it is triggered.

The smart home system is usually built upon several components, which can in turn be defined as LTS or LTS compositions if they are dynamic (i.e. they have multiple exclusive running modes) or as a set of synchronous equations if they have only one execution mode. Some components may be redundant and should not be specified more than once. For this case, BZR provides a *node* construct, in which LTS and synchronous equations can be defined to be instantiated. Figure 3 shows the graphical representation of such a node for a light bulb behavior definition.

Representing or not a component must be decided upon the following principle: if a component is concerned by a security rule, or if it can directly or indirectly influence a component concerned by a security rule, its behavior must be defined in the synchronous model. Moreover, if a behavior is modeled, it must also be observable. Regarding the example of the light bulb from Fig. 3, if its corresponding switch is set to ON or OFF,[3] then the bulb is supposed to respectively light up or shut down. This abstraction can work for a system with a relatively short life and built with new light bulbs. However, in the context of smart homes, a light bulb may fail at some point. In this model, the light bulb failure is not observable, so it does not correspond to reality. Being able to observe such a failure requires another component, like an appropriate sensor represented in Fig. 4 by the boolean variable *lightIsOn*. To keep track of the failure, it can be represented as an execution mode, cf. Fig. 5.

[3]The state of the switch is itself supposed to be known by the system.

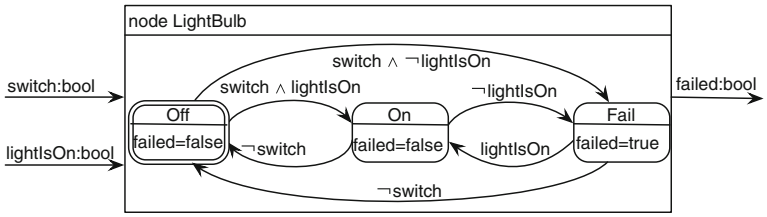


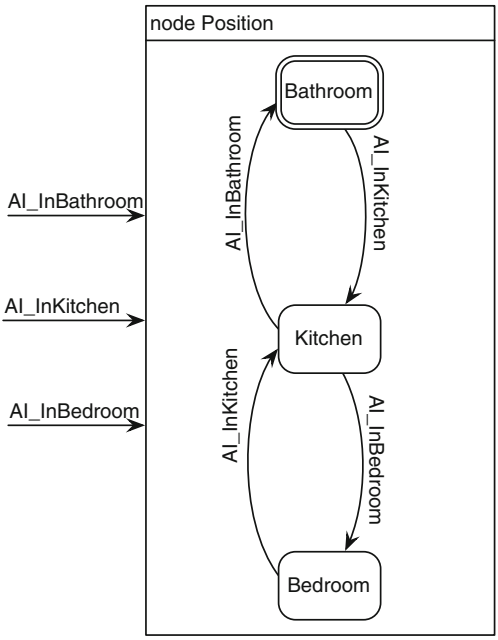
Fig. 5 Light bulb failure model

4.2 Person

Any person interaction with the system can be observed through its various types of sensors. However, in the very specific context of smart homes for disabled people, some characteristics of the person's behaviors and impairments can also influence security rules, and thus, have to be both observable and represented in the synchronous model.

As shown in the related literature, usual observable properties about a person are its position, mood, ADL and impairments. Position can be trivially defined as a LTS depending on how rooms are interconnected in the house. Let's suppose there are three rooms: a kitchen, connected to a bathroom and a bedroom. If sensors can determine the current position of a person, then the position evolution over time can be modeled by the LTS shown in Fig. 6. Observable behaviors in a smart home

Fig. 6 Position model



system are usually defined as a set of scenarios containing multiple steps and conditions to go from a step to another, so as to be processed by AI—in combination of events coming from the system—which can infer which step of which scenario the person is currently doing. Such a representation for scenarios makes them easy to be defined as LTS. And because a scenario can be aborted at any time by the person, modeling a scenario can follow the same principles presented for the observable failure of the light bulb. Figure 7 shows a LTS example representing the act of making coffee, evolving from step to step using AI notifications. Finally, mood and impairments are usually represented by boolean or numerical attributes, so they can be represented using synchronous equations. Evaluation of impairments for example, can come from various assessments such as the Global Deterioration Scale for Assessment of Primary Degenerative Dementia (GDSAPD) [38] which allocates a number between 1 and 7 depending on the cognitive decline (7 being very severe). We could also add additional disabilities such as "blind" or "deaf" which can be associated to booleans, cf. Fig. 8. It should be noted that this impairment model cannot evolve as it does not take inputs to influence the person's

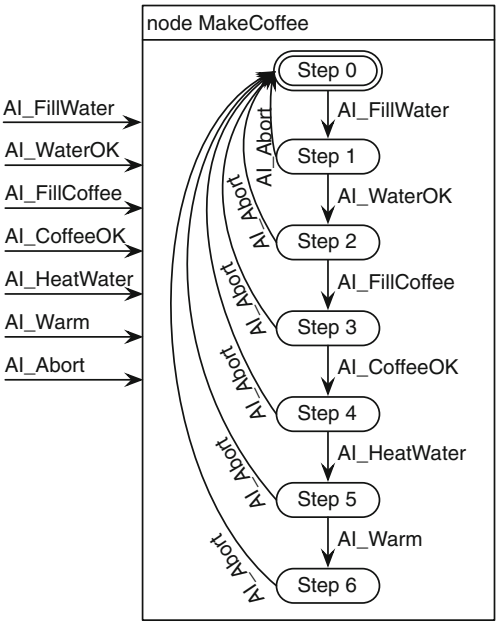


Fig. 7 ADL "Make Coffee"

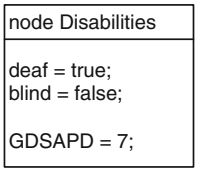


Fig. 8 Impairment model

profile, so in the case this person is diagnosed with additional problems, this model should be changed accordingly, and recompiled. But this evolution could be represented with LTS and inputs as usual.

Using all these specifications, a smart home model can be completed by specific properties required by DSC, namely: designation of controllability within the model, and security constraints definition.

5 Applying DCS

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.

5.1 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).

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 9 represents the designed by constraint controller of this small system, instantiating two times the LightBulb node (modified compared to Fig. 5 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.

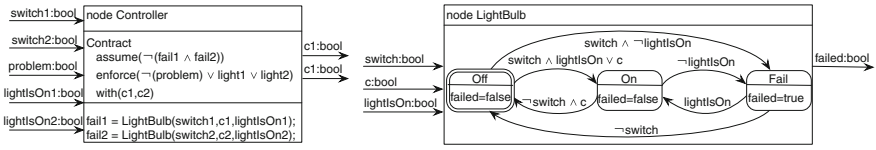


Fig. 9 Controllable light bulb model

5.2 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 *light*1 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 in-creases 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.

6 Experiment

This section shows the application of DCS on the model of a smart home system to address various errors coming from the user's behavior or the system itself (failure of its components). The examples are built incrementally, i.e. they can be merged together into a model of a system on which DCS can be applied to synthesize a controller guaranteeing all user/component safety properties. They show four types of control behavior: (1) adaptation and (2) usage limitation to anticipate a user problem (known disabilities and potential behavior errors), and (3) adaptation and (4) usage limitation to anticipate components related problems (hardware failure). These types of control behavior are an answer to the smart home fault tolerance problems identified in [9].

6.1 Base Model

Before describing the aforementioned scenarios, let's represent the base model of a smart home system (cf Fig. 10). This model contains the elements concerned (directly or indirectly) by security constraints. Their behavioral definitions are given as a set of automata and synchronous equations following the BZR concepts. The model also specifies inputs and outputs, and follows the synchronous execution definition: each execution step consumes all inputs and computes all outputs through the specified equations and automata, the actual organization of computations inside a step being solved by the synchronous compiler. This specific view of the smart home system (i.e. its control related information) con-stitutes the designed-by-constraint definition of the controller that we will try to synthesize.

Point ① of this model represents the main node—the *controller node*—centralizing all incoming events and all necessary outputs. The first four inputs (*fail* and *repair*) are related to the failure and repair events of specific elements named islands, coming from a previous implementation of fault tolerance [17]. Islands are independent system monitoring several sensors and effectors. Here we consider two of them, instantiating (cf. Point ③) the generic *node island* definition given in point ⑦. They manage respectively (1) an iPad, a speaker, and (2) a light bulb (cf. point ③).

Beyond island events, the controller node also receives a notification when smoke is detected (*smoke*), when the kitchen room is in the dark (*dark*), when the radio is activated (*pushBtnRadio*), when the range hood fan in the kitchen is activated (*pushBtnRHF*), when the person receives a telephone call (*telephoneCall*), and when the stove burners 1 and 2 of the kitchen stove are set to then ON position (*activateSB1*, *activateSB2*).

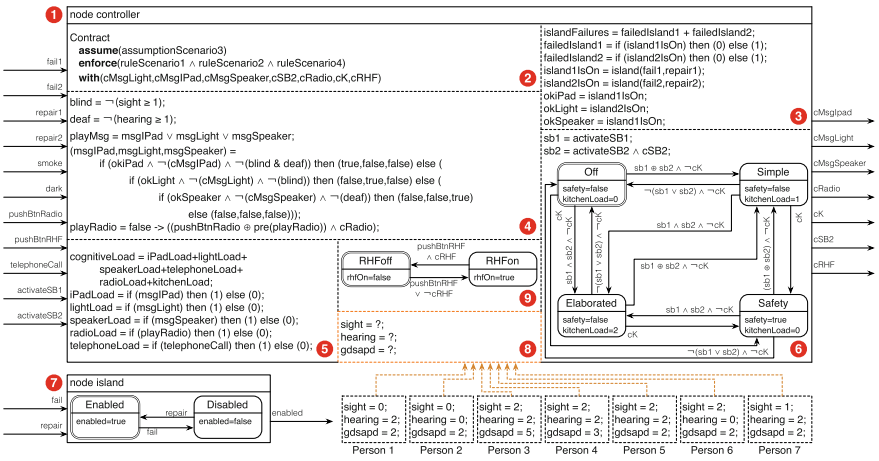


Fig. 10 Graphical representation of the defined by constraint controller

Point ⑥ shows a simplified behavior for the kitchen stove, allowing the activation of the two aforementioned stove burners, and being able to go in a safety mode if some problem is detected. When this mode is activated, the stove burners are set to OFF and the controller is notified (through the variable named *safety*) for actions to be derived.

Point ⑧ represents an abstract and simplified impairment model, which will be filled with a person actual data; here we will see what happens with seven persons given their information about their *sight* and *hearing* (integers from 0 to 2, meaning respectively "completely impaired" to "normal"), and also on their *GDSAPD* evaluation (integer from 1 to 7, meaning respectively "normal" to "severely mentally disabled"). These information will influence how the system should communicate—choosing between the iPad, the speaker, or the light—and when to communicate or not—*playMsg* being true or false—cf. point ④.

This communication with the user will also take care of the cognitive load: communicating with the person using the iPad, etc. increments the load by one unit, cf. point ⑤; if the kitchen stove is in use, this will also increase the load by one or two depending on the number of activated stove burners (cf. *kitchenLoad* from point ⑥); and finally, receiving a telephone call and listening to the radio are also considered as a cognitive loads. Thus, *telephoneLoad* and *radioLoad* (cf. point ⑤) increments the load by one depending on their activation (respectively influenced by the *telephone* input and the *playRadio* equation from point ④).

Finally, automata from ⑨ and ⑩ represent the activation behavior of the range hood fan and the kitchen light.

Now the model just needs a contract (which will be detailed in the next parts), so that DCS can be applied to eventually obtain an executable controller. This contract is given in point ② where, first, we make assumptions about the uncontrollability (*assumptionScenario3*), i.e. we define synchronous equations representing some key parts of the environment's behavior—this helps DCS to eliminate the verification of events combinations and sequences that are not supposed to happen—. And second, we specify rules that must remain true for all possible executions (*ruleScenario1,2,4*), with the help of seven controllable variables representing here the actual controllability of the system (i.e. they represent the real interface proposed by the smart home so that it can be influenced through the use of a computing system). These controllable variables are valuated internally by the synthesized controller (obtained through DCS) and provided as outputs, so that the system can react at each control step. When these variables are forced to false, *cMsg*(*IPad*/*Light*/*Speaker*) indicate which prompting system has to be used (respectively the iPad, the light, and the speaker), *cRadio* can force the radio to turn off, *cK* can set the kitchen stove in a safety mode, *cSB2* can prevent the use of the second stove burner and *cRHF* can activate the range hood fan.

We will now incrementally set four types of safety rules to tolerate errors or to adapt upon dangers, and detail their effects when the smart home system under control is used by people with different disabilities and different levels of impairments.

6.2 Scenario 1: User Assistance (Adapted Prompting)

Description and objective This first scenario aims at showing the added value of DCS when designing a controller to adapt the way the smart home communicates with the user. Guaranteeing the safety of the controller (with respect to rules) is made through the verification aspect of DCS, just like in classical formal verification algorithms (e.g. Model Checking); but knowing if such a controller actually exists is addressed by the very specific aspect of DCS: synthesis from constraints. Let's specify a first constraint, for example we want to be sure that when smoke is detected in the kitchen (*smoke* variable is *true*), then the range hood fan activates and a prompt information is provided through an adapted media (iPad, speaker or light-bulb). To reflect this in the model, we set the following equation for *ruleScenario1*:

$$\text{ruleScenario1} = \neg(\text{smoke}) \vee (\text{rhfon} \wedge \text{playMsg}) \quad (1)$$

We now take the cases of persons 1 and 2: will the smart home be controllable, i.e. will it be able to cope with this constraint for all possible execution? Let's apply DCS with each user profile on this specification—containing the base model, the user profile, and this first rule (the other ones being set to *true* for the moment)—and let's simulate a scenario where the user activates the two stove burners to start cooking some food, but smoke gets detected when the food is starting to burn.

Comments on scenario execution DCS fails when applied with the profile of person 2. This person is actually both blind and deaf, so there is no appropriate communication media in case of a problem (e.g. when smoke is detected). Technically: variables *msg*(*IPad*/*Speaker*/*Light*) can never be set to *true* what-ever the values given to the controllable variables *cMsg*(*IPad*/*Speaker*/*Light*) because *blind* and *deaf* are *true*; thus *playMsg* can never be *true*. This leads to the fact that *ruleScenario1* can be *false* if *smoke*—an uncontrollable variable given as input—is *true*, which is of course not permitted (*ruleScenario1* has to be enforced to *true* for all executions). Because of this, the smart home system is not controllable for this first rule and this is the reason why DCS fails, meaning that the system has to be reworked (e.g. by adding adapted medias), before person 2 can actually use it safely.

However, DCS succeeds when applied with the profile of person 1, meaning that a controller has been found so that *ruleScenario1* will always remain *true* for all possible execution. Figure 11 shows the simulation results, highlighting important events. It has to be noted that only the relevant events and steps are represented:

Fig. 11 Execution of scenario 1 for person 1

		Person 1					
		Step 1	Step 2	Step 3	Step 4	Step 5	Step 6
Inputs	smoke	0	0	**1**	1	**0**	0
	activateSB1	**1**	1	1	1	1	1
	activateSB2	0	**1**	1	1	1	1
	pushBtnRHF	0	0	0	**1**	0	**1**
Out.	cMsgSpeaker	1	1	**0**	0	**1**	1
	cRHF	1	1	**0**	0	**1**	1

missing inputs are *false*, missing outputs are *true*, and a new step is represented only when something changes from the previous one (new inputs/outputs or internal state modification).

- Steps 1 and 2: Person 1 activates respectively the stove burners 1 and 2 to start cooking.
- Step 3: Some smoke gets detected by a sensor that valuate the *smoke* input to *true*; the controller reacts by setting the controllable variables *cMsgSpeaker*, *cK* and *cRHF* to *false*, which has the following effects:
  - the range hood fan activates
  - a message about the smoke is played using the speaker; physically, the system is able to select the appropriate message and media, knowing that *smoke* is true and *cMsgSpeaker* is forced to *false*.

  This way, *ruleScenario1* remains true.
- Step 4: The person tries to deactivate the range hood fan, however, because smoke is still detected, the fan has to stay activate; deactivation is actually prevented by forcing cRHF to remain false in this step, avoiding to take the transition from step *RHFon* to *RHFoff*.
- Step 5: No smoke is detected anymore, and no controllable variable has to be forced to *false*; this has the following effect: the message about smoke is stopped being played.
- Step 6: the person tries to deactivate the range hood fan, which is this time permitted because *cRHF* is not forced to *false*.

This scenario has shown the interest of using DCS in this context to both verify that the smart home is able to adapt to a person's disabilities and generate a controller to manage the smart home adaptation behavior.

6.3 Scenario 2: User Error Prevention (Simultaneous Devices Usage Limitation)

Description and objective This second scenario shows the advantage of using DCS to put limitations on how the various devices in a smart home can be used, depending on the user's profile. As an example of such a case, we will focus here on the compound cognitive load, due to simultaneous device usage by a person, and show how the smart home system gets configured to prevent a cognitive overload. Let's specify a second constraint, such that the cognitive load cannot exceed 2 and 3 units when the person's GDSAPD is respectively evaluated to 5 and 3. Regarding the adaptation possibilities, the radio can be turned o automatically and the kitchen stove cannot be used on *Elaborated* mode (only the stove burner 1 can be activated at most) to reduce the cognitive load when it is necessary. We set this as the following equation for *ruleScenario2*:

$$\text{ruleScenario2} = (\neg(\text{gsdapd} \geq 5) \vee (\text{cognitiveLoad} \leq 2)) \wedge (\neg(\text{gdsapd} \geq 3) \vee (\text{cognitiveLoad} \leq 3)) \quad (2)$$

Like in the example given in the first scenario, if the smart home cannot be adapted to a person (due to disabilities) for all possible executions, DCS will fail to find a controller. So we will take the cases of persons 3, 4 and 5—for which DCS succeeds—and simulate a scenario where the person is listening to the radio, then activates the stove burners 1 and 2 to start cooking, but then receives a telephone call (Fig. 12).

Comments on scenario execution Fig. 12 shows the simulation results, where we can see how the smart home gets adapted to cope with the differences in abilities to deal with cognitive load between the three persons. Person 5, having a GDSAPD lower than 3, should be able to deal with a high cognitive load, but it is not the case for persons 4 and 3, and this is why they experience some limitation when using some devices simultaneously in this controlled smart home.

- Step 1: Persons 3, 4 and 5 start listening to the radio; cognitive load is then set to 1 unit, which is correct for everyone.
- Step 2: Persons 3, 4 and 5 activate the first stove burner; this increments the cognitive load by 1 unit, which is now the acceptable limit for person 3, having a GDSAPD equal to 5 units.
- Step 3: Person 4 and 5 activate the second stove burner, now the cognitive load is set to 3 units, the acceptable limit for person 4 (having a GDSAPD equal to 3 units); but when person 2 tries to activate the second stove burner by setting the *activateSB1* switch to "on" (*true*), the radio goes o because the controller forces *cRadio* to be *false*, thus keeping the cognitive load below the acceptable limit, as required by *ruleScenario2*. It is interesting to note here that preventing the second stove burner to start by forcing *cSB2* to be *false* would also have been a correct response from the controller. The order in which the controllable variables are set actually depends on the order they are declared in the BZR program. Here *cSB2* is declared before *cRadio*, so when a value is asked for *cSB2* in a step, the value of *cRadio* is not decided yet, and in this example *cSB2* has no reason to be forced to false, because there is still a solution to comply with the rules (i.e. by setting *cRadio* to false). Inverting the declarations of *cSB2* and *cRadio* would have let the radio "on" and prevented the use of the second kitchen stove for person 3 in this step.

		Person 3				Person 4				Person 5			
		Step 1	Step 2	Step 3	Step 4	Step 1	Step 2	Step 3	Step 4	Step 1	Step 2	Step 3	Step 4
Inputs	pushBtnRadio	**1**	0	0	0	**1**	0	0	0	**1**	0	0	0
	telephoneCall	0	0	0	**1**	0	0	0	**1**	0	0	0	**1**
	activateSB1	0	**1**	1	1	0	**1**	1	1	0	**1**	1	1
	activateSB2	0	0	**1**	1	0	0	**1**	1	0	0	**1**	1
Out.	cRadio	1	1	**0**	0	1	1	1	**0**	1	1	1	1
	cSB2	1	1	1	**0**	1	1	1	1	1	1	1	1

Fig. 12 Execution of scenario 2 for persons 3, 4 and 5

– Step 4: a telephone call is received in this step, which increments the cognitive load by one, and this cannot be prevented (there is no controllability on receiving telephone calls for this smart home, at least in the model we have defined). Person 5 can receive the call whilst continuing to cook and to listen to the radio. However letting the cognitive load going to 4 units is not permitted for person 4, so the smart home has to react to not let this happen: this is why the radio gets deactivated (and not the stove burner 2 for the same reason explained in step 3 for person 3). Finally, this telephone call impacts the smart home usability for person 3 to keep the cognitive load to an acceptable level: the controller forces *cSB2* to be false here, thus deactivating the second stove burner and keeping the cognitive load to 2 units.

This scenario has shown an example on how the person's static profile can be used to prevent user errors (here by keeping the cognitive load below an adapted level) by configuring the smart home with the help of DCS which provided a corresponding smart home controller.

6.4 Scenario 3: Component Failure (Redundancy)

Description and objective This third scenario shows the application of DCS to solve a problem that could arise from a previous implementation of fault tolerance in our smart home system: in our architecture presented in [17], we "did install industrial grade material […] to avoid hazardous situations [for example where] the resident cannot turn on the light due to a system failure."; thus we connected our various sensors and effectors to four independent fault-tolerant islands so that "if a block falls, only the [connected equipments] will be affected". Sensors and effectors are critical safety elements, so if their connected island can fail, do we have enough redundancy? To generalize, having enough redundancy in a given system means that, for all possible execution of this system and in case of a failure, there is always a solution to keep it running correctly; it means here that the smart home system remains adapted to the person's impairments. In order to keep the base model small and visually clear, we consider a simplification of our own redundancy implementation where only islands can fail but not the other devices (lights, sensors, etc.). Island failure is a kind of uncontrollable event (the system can do nothing to prevent this), so it cannot be represented as a control rule. However we will assume here that the case where two islands are disabled in the same step should never happen, but we still want to tolerate one and only one failure at most. This hypothesis can be represented in the assume part of the contract, by giving the following equation to *assumptionRule3*:

$$\text{assumptionRule3} = \text{islandfailures} \leq 1 \tag{3}$$

In this scenario, we will see that having an island failure may have different impacts on the smart home behavior regarding who is actually living in. It starts when the island number 1 fails. Then the user activates the two stove burners to start cooking, but smoke gets detected which triggers a message (using an appropriate media) and the user deactivate the stove burners. At some point, no more smoke is detected and the island number 1 gets repaired. Then the user re-starts cooking, but smoke gets detected again and an new message has to be communicated. We take persons 1 and 6 for this scenario and apply DCS (Fig. 13).

Comments on scenario execution As shown in Fig. 13 for person 1, DCS does not find an appropriate controller. Indeed, if island number 1 fails, then the speakers cannot be activated (cf. Point ③, *island1IsOn* being *false* means *okSpeaker* is *false* too, and then *msgSpeaker* cannot be *true*); however this is the only acceptable communication media when a person is blind (but not deaf) which is the case of person 1 (cf. Point ④, if *blind* is *true*, then *msgLight* and *msgIPad* cannot be *true*); so if *msg*(*IPad*/*Speaker*/*Light*) are all false, then *playMsg* becomes false and this can be problematic: if a message has to be communicated to the user because of a problem, such as a smoke problem as specified in ruleScenario1, there is no media available and because this situation cannot be prevented by any available controllability then DCS fails, meaning that the redundancy implementation has to be reworked.

For person 6 however, DCS succeeds and the steps results of the aforementioned scenario can be seen in Fig. 13. It has to be noted that if both islands fail at the same time, then no media can be selected anymore, and in case of a smoke problem this would violate *ruleScenario1*. But DCS ignores this case, as we have defined that a double island failure should not happen at the same time in the assume part of the contract.

- Step 1: The scenario starts when island number 1 fails; this disables the use of the iPad and the speaker as communication devices.
- Steps 2 and 3: Person 6 activates the two stove burners to start cooking.
- Step 4: Smoke gets detected from the environment (this enables the range hood fan but it is not relevant in this scenario), and a message has to be communicated in order to keep *ruleScenario1* to true. Person 6 being deaf, the iPad cannot be used as it requires to have correct sight and hearing (cf. Point ④); so only the

		Person 6							
		Step 1	Step 2	Step 3	Step 4	Step 5	Step 6	Step 7	Step 8
Inputs	fail1	**1**	0	0	0	0	0	0	0
	repair1	0	0	0	0	0	0	**1**	0
	smoke	0	0	0	**1**	1	**0**	0	**1**
	activateSB1	0	**1**	1	1	**0**	0	**1**	1
	activateSB2	0	0	**1**	1	**0**	0	**1**	1
Outputs	cMsgLight	1	1	1	**0**	0	**1**	1	1
	cMsgIPad	1	1	1	1	1	1	1	**0**
	cMsgSpeaker	1	1	1	1	1	1	1	1

Fig. 13 Execution of scenario 3 for person 6

light remains, and this is why *cMsgLight* is forced to *false* by the controller, this way a message about the smoke problem can be communicated through the light.
- Step 5: Person 6 sets the two stove burners' switches to "off".
- Step 6: No smoke is detected, so the controller does not continue to keep *cMsgLight* to *false* (having *smoke* to *false* keeps *ruleScenario1* to *true*) thus the message about smoke can be stopped.
- Step 7: Island 1 has been repaired and the two stove burners are switched "on" again.
- Step 8: Smoke is detected anew: This time, the message about smoke gets communicated through the iPad, because (1) island number 1 is operational and (2) the controllable variable *cMsgIpad* is declared after *cMsgLight*, explaining why *cMsgLight* remains true (not forced by the controller), because *cMsgIpad* can always be forced to false after (which is the case in this step), thus keeping *ruleScenario1* to true.

Usage of DSC in this scenario is especially powerful: instead of trying to define redundancy generically for all types of users, we can reduce costs by defining more or less redundant component combinations for different users and applying DCS to guarantee that a specific redundant installation is safe for a specific user profile. Moreover, adjusting hypothesis on the components' quality is simply a question of setting a boolean equation in the assume part. For example, if we have five islands monitoring our components, and want to tolerate a maximum of three failures at the same time and see if our system is still stable, we just have to set (*islandfailures* ≤ 3) in the *assume* part and apply DCS to know if the smart home is actually stable (controllable) in this context and get the associated controller.

6.5 Scenario 4: Component Adaptation (Degraded Mode)

Description and objective An alternative solution to cope with hardware failures, besides using redundancy, is to modify the way the remaining operational components can be used. If we take the islands failures example, and want to cope with a double failure—which creates a communication problem when smoke is detected -, then we can program a controller to degrade what can create smoke (i.e. the kitchen stove by setting *cK* to *false*), and either remove *ruleScenario1* or assume that *smoke* remains *false* when *Safety* mode is active. But it would mean that the kitchen stove would be forced to remain in *Safety* mode for all possible executions just because smoke could happen, which is not acceptable. Instead, we want to find an example showing how DCS can be useful to build a controller helping to anticipate a hardware failure by modifying the remaining active components in the case where no redundancy is available. Let's say we only have one lightbulb in the kitchen. At night, if the lightbulb fails—and whatever its controllability (i.e. the lightbulb can be switched "on" or "off" automatically)—then the kitchen goes

completely dark. Now let's build an new safety rule such that, if the user has limited sight, the kitchen stove cannot be used during the night when the lightbulb is not activated. Of course, because the light bulb has no redundancy, we cannot create a rule such that when the kitchen is used during the night, then the light should go "on" if sight is limited: this model would not comply with the reality if we consider that the lightbulb can fail (and thus cannot go "on" at some undetermined moment). So instead, we use a light sensor providing the value of a variable named *dark*—indicating if there is enough light in the room (*false*) or not (*true*)—and this value explicitly impacts the way the kitchen can be used through the following synchronous equation attributed to *ruleScenario4*:

$$\text{ruleScenario4} \;\; = \neg(\text{sight} == 1) \vee (\neg\text{dark}) \vee (\text{kitchenLoad} == 0) \qquad (4)$$

This means that if the user's sight is evaluated to 1 (visually impaired but not blind), and if the kitchen is in the dark, then the smart home has to adapt itself so that the kitchen stove cannot be used (it is either in *Off* or *Safety* mode, the only modes where *kitchenLoad* equals 0).

We now define a scenario where we start at night, the kitchen is in the dark, and the user activates the stove burner 1; then the user presses the light button (this event is not given to the controller because no information about the actual lightbulb activation can be safely derived from it), but the lightbulb fails in the following instant.

Depending on the user's profile, different control results happen when this scenario is played, as shown is Fig. 5 for persons 7 and 5 (for which DCS succeeded). This example relates to Scenario 1 regarding the explicit constraint definition on environment events (smoke value directly impacts the activation of the range hood fan), to Scenario 2 regarding the dynamic adaptation to multiple users profiles, and to Scenario 3 regarding the hardware failure example (islands can fail, their failures impact the system's behavior) (Fig. 14).

Comments on scenario execution Following Fig. 14, person 7 being visually impaired, the smart home system adapts itself consequently. However, person 5 having a good sight, the smart home does not interfere with the kitchen stove usage, whatever the light condition.

		Person 6							
		Step 1	Step 2	Step 3	Step 4	Step 5	Step 6	Step 7	Step 8
Inputs	fail1	**1**	0	0	0	0	0	0	0
	repair1	0	0	0	0	0	0	**1**	0
	smoke	0	0	0	**1**	1	**0**	0	**1**
	activateSB1	0	**1**	1	1	**0**	0	**1**	1
	activateSB2	0	0	**1**	1	**0**	0	**1**	1
Outputs	cMsgLight	1	1	1	**0**	0	**1**	1	1
	cMsgIPad	1	1	1	1	1	1	1	**0**
	cMsgSpeaker	1	1	1	1	1	1	1	1

Fig. 14 Execution of scenario 4 for persons 7 and 5

- Step 1: Both users are in the kitchen, in the dark.
- Step 2: Both of them press a button to activate the stove burner 1, but the actual activation is prevented for person 7: the controller forces *cK* to *false*, which prevents the kitchen stove to go in *Simple* execution mode where *kitchenLoad* would be equal to 1 instead of 0, thus violating *ruleScenario4*.
- Step 3: They both activate the lightbulb, and the light sensor reacts by setting *dark* to *false*. This allows the actual activation of the stove burner 1 for person 7.
- Step 4: The kitchen returns in the dark, as indicated by the *dark* value (*true*), even if neither person 7 nor 5 actually touched the light switch to turn it off: the lightbulb has failed and the controller reacts on the *dark* value—instead of the actual light switch position—for person 7 by setting the *cK* controllable variable to *false*, thus placing the kitchen stove to its safety execution mode.

In the context of usage limitation to anticipate components related problems, this example shows again the advantage of being able to define the system's controllability by constraint, instead of giving its actual implementation: here the kitchen stove is actually controllable in the sense that an internal system (the controller) can act on it to prevent the activation of its stove burners, but the kitchen model does not have to define explicitly under which conditions it has to react; the actual control implementation (valuation of the controllable variables) is obtained by synthesis given the global constraints defined by the programmer in the model of the system's components. States or behaviors of one or several components (e.g. the light and impairment profile) can have an impact on the controllability of other components (e.g. the kitchen stove) without requiring to define explicitly this controllability.

6.6 Evaluation

Our approach is compared in Fig. 15 to the most relevant ones already discussed in the "related work" section of this document. Beforehand, we had implemented some redundancy mechanisms in our own smart home test lab [17], so that not all sensors/effectors would be controlled by a single machine (an Island) because it would have consisted in a single point of failure. However, we could not be exactly sure that for every susceptible failure, the use of redundant elements (islands) would

	Smart home context	Smart home modeling	Fault tolerance	Disability context	Adaptation to a person	Methodology, concrete scenarios and examples	Verification	Synthesis
Dumitrescu et al. 2010			x				x	x
Bouchard et al. 2012	x		x	x	x	x		
Chetan et al. 2005	x		x	x				
Corno et al. 2013	x	x					x	
Lapointe et al. 2012					x	x		
Zaho et al. 2012	x	x					x	x
Guillet et al. 2013	x	x	x	x	x		x	x
This study	x	x	x	x	x	x	x	x

Fig. 15 Comparison to other approaches

be sufficient to keep the smart home safe for a particular person (as several sensors and/or effectors not managed by the new island would have been disabled). The way we connect our sensors and effectors to multiple islands could actually be safe for a person in case of a failure but not for another one with different disabilities, and this could be hard to find and verify without appropriate tools able to solve this combinatorial problem. Redundancy without verification was indeed not sufficient.

As we were trying to solve this failure problem, we compared our approach to closely related ones regarding smart homes for disabled people, and learned—especially from [9]—that failures could be of multiple types in this context and redundancy itself was not the only solution to address them. This is why we became interested in the more general problem of fault tolerance for this kind of smart homes, and we would base our use cases on their studies to show how the different types of failures could be addressed. But unlike these approaches, we would complete ours by verifying it.

Designing smart home models models such that they can be verified has been done several times. One of the most related approaches regarding modeling and verification is [29], where a smart home is modeled using a formal representation (State Charts), and safety properties defined so that formal verification tools can be employed to guarantee that these properties are ensured for all possible execution. However, we already discussed the problem of modeling the entire system to apply verification. This why we kept the formal modeling approach, but took an approach based on synthesis to address this combinatorial problem by solving it automatically from constraints, instead of trying to find (and verify) a complete solution manually. Our expertise with synthesis techniques comes from previous work in the domain of reconfigurable hardware architectures, where DCS was proven useful to build formal reconfiguration controllers. Still in the hardware context, DCS was also employed to carry out computations on failure-prone processors, giving us inspiration on how to actually use DCS to manage fault tolerant smart homes.

Usage of synthesis techniques in the smart home context is very recent. First results can be seen in [34, 35] which present the use of DCS respectively from a generic point of view (in the context of the Internet of Things) and from a specific one regarding fault tolerance. However, being preliminary, they both lack of concrete use cases, implementation and results. This proposal makes a contribution over them by giving these elements: it presents a detailed methodology to create smart home controllers by synthesis, using BZR for smart home elements modeling, and shows use cases with realistic scenarios that we tested in our lab to demonstrate the relevance and advantages of using DCS for addressing the various fault tolerance problems (identified in [9]) that may occur in a smart home dedicated to disabled people.

7 Conclusion and Perspectives

Safety and security services are essential requirements for many pervasive computing systems. This is especially true for smart homes dedicated to people with disabilities, where security constraints prevail. They represent a pervasive systems category where safety is actually a very critical property: the person living in such a house is usually frail and is not supposed to be able to cope with errors; implication of failures can range from user annoyance to hazardous situations.

Correct adaptation behavior—so that the smart home remains safe whatever the conditions of execution—is both difficult to design and verify. While verification has been addressed multiple times, uses of synthesis techniques in this context to guarantee a safe behavior (employing formal verification) while simplifying the design (which is derived from constraints) are still rarely encountered and lack of examples showing how they can be used to solve practical problems. In this context, this proposal makes a contribution by providing a design methodology, relying on DCS, and backed by scenarios examples, to build smart home controller systems guaranteeing safety properties.

The results validating the proposal present both modeling and executions parts for different scenarios. They especially focus on fault tolerance as a safety property, and show how to deal with four types of typical control needs in this context: adaptation and usage limitation for users problems and components failures. With these results, obtained by rigorous experiments (real scenarios, executed in our smart home test lab), we demonstrated that the synchronous paradigm (on which BZR is based) and DCS tools (such as Sigali) are a relevant to design and compute the controller of a smart home system, in the context of fault tolerance.

In the end, the proposed methodology allows us to solve a simple but crucial question: can this smart home be adapted to this person, for every failure situation that can be derived from its model? A negative answer implies that a safety constraint (defined in the model) can be violated, and this cannot be prevented: the smart home itself has to be modified (by adding more redundancy, removing dangerous elements or executions modes, etc.) because no correct adaptation controller exist. However, a positive answer to this question automatically gives back the code of a correct control system, to be connected (inputs and outputs) and executed within the corresponding smart home so that it can actually be adapted dynamically.

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. For example, such an abstraction has been implemented in the reconfigurable embedded systems domain (cf. [11]) to allow designers to specify reliable reconfiguration controllers using only a UML profile (high level abstraction); models built with this profile could be transformed into a synchronous representation based on BZR to make DCS applicable transparently, thus giving back the executable code of their specified-by-constraints controllers.

References

1. Ramos, C., Augusto, J.C., Shapiro, D.: Ambient intelligence: the next step for artificial intelligence. Intell. Syst. IEEE **23** (2008)
2. Carberry, S.: Techniques for Plan Recognition. User Model. User-Adap. Inter. **11**, 31–48 (2001)
3. Novak, M., Binas, M., Jakab, F.: Unobtrusive anomaly detection in presence of elderly in a smart-home environment. In: ELEKTRO (2012)
4. Bouchard, B., Giroux, S., Bouzouane, A.: A keyhole plan recognition model for Alzheimer's patients: first results. J. Appl. Artif. Intell. (AAI) **21**, 623–658 (2007)
5. Fortin-Simard, D., Bouchard, K., Gaboury, S., Bouchard, B., Bouzouane, A.: Accurate passive RFID localization system for smart homes. In: IEEE 3rd International Conference on Networked Embedded Systems for Every Application (NESEA), pp. 1–8 (2012)
6. Lapointe, J., Bouchard, B., Bouchard, J., Potvin, A., Bouzouane, A.: 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, pp. 30:1–30:8. New York, NY, USA, ACM (2012)
7. Pigot, H., Mayers, A., Giroux, S.: The intelligent habitat and everyday life activity support. In: 5th International Conference on Simulations in Biomedicine, avril 2003. Slovenie (2003)
8. Bulow, J.: An economic theory of planned obsolescence. Q. J. Econ. **101**, 729–749 (1986)
9. Chetan, S., Ranganathan, A., Campbell, R.: Towards fault tolerance pervasive computing. Technol. Soc. Mag. **24** (2005)
10. Ramadge, P.J.G., Wonham, W.M.: The control of discrete event systems. Proc. IEEE **77**, 81–98 (1989)
11. Guillet, S., de Lamotte, F., Le Griguer, N., Rutten, É., Gogniat, G., Diguet, J.P.: Designing formal recon guration control using UML/MARTE. In: 7th International Workshop on Reconfigurable Communication-centric Systems-on-Chip (Re-CoSoC), pp. 1–8 (2012)
12. Pigot, H., Lefebvre, B., Meunier, J.G., Kerhervé, B., Mayers, A., Giroux, S.: The role of intelligent habitats in upholding elders in residence. In: 5th International Conference on Simulations in Biomedicine, pp. 497–506 (2003)
13. Latfi, F., Lefebvre, B., Descheneaux, C.: Ontology-based management of the tele-health smart home, dedicated to elderly in loss of cognitive autonomy. In: Workshop on OWL: Experiences and Directions, pp. 1–10 (2007)
14. Augusto, J.C., Nugent, C.D.: Smart homes can be smarter. In: In Designing Smart Homes—The Role of Artificial Intelligence (2006)
15. Patterson, D.J., Kautz, H.A., Fox, D., Liao, L.: Pervasive computing in the home and community. In Bardram, J.E., Mihailidis, A., Wan, D. (eds.) Pervasive Computing in Healthcare, pp. 79–103. CRC Press (2006)
16. Mihailidis, A., Boger, J., Canido, M., Hoey, J.: The use of an intelligent prompting system for people with dementia. Interactions **14** (2007)
17. Bouchard, K., Bouchard, B., Bouzouane, A.: 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 (2012)
18. Bouchard, K., Bouchard, B., Bouzouane, A.: Discovery of topological relations for spatial activity recognition. In: Proceedings of the IEEE Symposium Series on Computational Intelligence (SSCI 2013), pp. 1–8 (2013)
19. Benveniste, A., Caspi, P., Edwards, S.A., Halbwachs, N., Le Guernic, P., de Simone, R.: The synchronous languages 12 years later. Proc. IEEE **91**, 64–83 (2003)
20. Marchand, H., Bournai, P., Borgne, M.L., Guernic, P.L.: Synthesis of discrete-event controllers based on the signal environment. Discrete Event Dyn. Syst. **10**, 325–346 (2000)

21. Dumitrescu, E., Girault, A., Marchand, H., Rutten, É.: Multicriteria optimal dis-crete controller synthesis for fault-tolerant real-time tasks. In: Workshop on Dis-crete Event Systems, WODES'10, pp. 366–373. Berlin, Germany (2010)
22. Delaval, G., Marchand, H., Rutten, E.: Contracts for modular discrete controller synthesis. In: Proceedings of the ACM SIGPLAN/SIGBED 2010 Conference on Languages, compilers, and Tools for Embedded Systems, pp. 57–66. New York, NY, USA, ACM (2010)
23. Kilgore, C., Peitz, M., Schmid, K.: System Requirements Document for Safe Home. Research report, Iowa State University (2004)
24. Le Lann, G.: The Ariane 5 flight 501 failure—a case study in system engineer-ing for computing systems. Technical report, REFLECS—INRIA Rocquencourt (1996)
25. Jaygarl, H., Denner, A., Pham, N.: Software requirements and specification document for smart home notification and calendering system. Research report (smart home project, Iowa State University), pp. 1–45 (2008)
26. Picard, R.W.: A ective computing. Technical report (1995)
27. Schmidtke, H.R., Woo, W.: Towards ontology-based formal verification methods for context aware systems. In: Proceedings of the 7th International Conference on Pervasive Computing, pp. 309–326. Springer, Berlin, Heidelberg, (2009)
28. Corno, F., Sanaullah, M.: Formal verification of device state chart models. In: 7th International Conference Intelligent Environments (2011)
29. Corno, F., Sanaullah, M.: Modeling and formal verification of smart environments. Secur. Commun. Netw. (2013)
30. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. IEEE Comput. (2003)
31. Cassandras, C.G., Lafortune, S.: Introduction to Discrete Event Systems. Springer, Berlin (2006)
32. Delaval, G., Rutten, E.: Reactive model-based control of reconfiguration in the fractal component-based model. CBSE (2010)
33. Bouhadiba, T., Sabah, Q., Delaval, G., Rutten, E.: Synchronous control of reconfiguration in fractal component-based systems—a case study. In: Proceedings of the International Conference on Embedded Software. EMSOFT (2011)
34. Zhao, M., Privat, G., Rutten, É., Alla, H.: Discrete control for the internet of things and smart environments. In: 8th International Workshop on Feedback Computing, In conjunction with ICAC (2013)
35. Guillet, S., Bouchard, B., Bouzouane, A.: Correct by construction security approach to design fault tolerant smart homes for disabled people. In: EUSPN (2013)
36. Marchand, H., Samaan, M.: Incremental design of a power transformer station controller using a controller synthesis methodology. IEEE Trans. Softw. Eng. **26**(8), 729–741 (2000)
37. Altisen, K., Clodic, A., Maraninchi, F., Rutten, É.: Using controller-synthesis techniques to build property-enforcing layers. In: Proceedings of the 12th European conference on Programming, pp. 174–188. Springer, Berlin, Heidelberg (2003)
38. Reisberg, B., Ferris, S.H., Crook, T.: Signs, symptoms and course of age-associated cognitive decline. In Corkin, S., Davis, K.L., Growden, J.H. (eds.) Aging: Alzheimer's Disease: A Report of Progress, pp. 177–181. Raven Press (1982)