CCBL: A new language for End User Development in the Smart Homes

Lénaïc Terrier, Alexandre Demeure, Sybille Caffiau

Institute of Engineering Univ. Grenoble Alpes
{forename.name}@ univ-grenoble-alpes.fr

Abstract. We present Cascading Context Based Language (CCBL), a new programming language for the Smart Home. We build CCBL on the notion of context that express home actions according to the observed states. We describe how CCBL enables users to organize contexts in a concise and predictable way using three mechanisms: 1) The Cascade for specifying device states implicitly, 2) The priority list for ensuring that only one context can access a device at a time and 3) The Allen's interval algebra for enabling orchestration of contexts over time.

Keywords: End User Development, smart homes, CCBL, ECA.

1. Introduction

From a technical point of view, a smart home combines sensors (thermometers, motion sensors…), actuators (lights, sound systems and digital displays) and network-oriented services (synchronized calendar, weather forecast, TV program, email…). As Mennicken et al. [8] state the use of these computing technologies allows "a home either to increase the comfort of their inhabitants in things they already do or enables new functionalities". Usages of smart home aim at easing everyday life chores (automatic vacuum cleaner), at saving energy (heating system optimization, energy consumption tracker), at increasing security (alarm systems), at raising awareness (remote surveillance access) [2], [5]. One challenge is therefore to bridge the gap between technical possibilities and inhabitants' needs.

ECA (Event Condition Action) is the most popular language in use to program home. It allows to express home behavior by a set of rules that independently program actions according to captured events (if Event and Condition then Actions). However, inhabitants may have difficulties to program with ECA. Huang et al. [7] show that users build a wrong mental model for a certain type of ECA rule. Other problems arise when programs become more complex, either in terms of the number of events or conditions that can trigger ECA rules [9] or in terms of the number of rules itself [4].

We explore a new approach centered on the notion of context rather than events. We also explore how Allen's interval algebra [1] can help to structure those contexts in order to make the behavior of complex programs more predictable by end users.

2. Cascading Contexts Based Language (CCBL)

Cascading Context Based Language (CCBL) is a programming language dedicated to programming Smart Homes. In order to design CCBL, we consider general properties that are required by all programming languages. The language should have a **low threshold**: what is simple to mentally model (lit a light) should be easy and simple to express. Moreover, since end-user development addresses non-professional developers, the basics should be easy to learn [3]. On the other hand, simplicity should not mean less expressiveness, the language should have a **high ceiling**: complex tasks should be possible to express. More advanced users should be able to work on complex projects [10]. It is particularly relevant in home automation system where some users need challenges [5].

We centered CCBL on the notion of context rather than the notion of event as it is for ECA. We first make explicit our notion of context before presenting how contexts are ordered by priority so that only one context at a time has access to a device. In a third subsection, we detail how CCBL enables to organize contexts based on Allen's interval algebra.

2.1. Contexts in CCBL

In CCBL, a context can be active or inactive. A context defines a set of actions that the system applies when the system detects that it becomes active (e.g. "*Set volume of the music to off and set lamp A to off*").

There are two types of contexts: State contexts and Event contexts. Event contexts are semantically close to ECA rule: the trigger of an event activates the related event contexts. There is no duration associated with event contexts, once the event is triggered, actions are performed. State contexts are associated with a duration; they have a start and an end. State contexts are semantically close to rules of the type "When condition then actions" as Huang et al. [7] describe them. There are three ways to define a State context, depending on how users express the start and the end of the context:

- A Boolean expression: The context is active while the system evaluate this expression as true. It is inactive when the system evaluate this expression as false. For instance, "*System detects that Martin is at home*" is such an expression.
- A start and an end event: The context is active after that the system detects the start event. It stays active until the system detect the end event. For instance, the start event could be that "*System detects that the door has been opened*" and the end event could be that "*The alarm has been turned off*".
- A Boolean expression and optionally a start and/or an end event: The context is active after either the system detects the start event (if specified) or the system evaluates the Boolean expression as true (if there is no start event specified). The context stays active until either the system detects the end event (if specified) or the system evaluate the Boolean expression as false (in any case). For instance, the starting event could be "*System detects that Martin enters his home*", the Boolean expression could be "*System detects that Martin is on the phone*".

CCBL organizes context hierarchically: A state context may have sub-contexts. Each sub-context can only be active if the parent context is active. For instance, one can consider the sub-context "*System detects that Martin is on the phone*" that apply only when the context "*System detects that Martin is at home*" is active. By construction, CCBL imposes to have one root context, this root context aims to represent the "neutral state" of the smart Home (e.g., lights are off; doors are locked).

2.2. Priority, predictability and cascade

On a complex system with many contexts simultaneously running, it must remain simple to ascertain which context should prevail over the others when both specify actions to apply upon a same device. To do so, whether a context can act upon a device must be independent from the order of execution and of the succession of events. In order to achieve that we set up rules that put into order all the competing contexts.

Several contexts may set the state of a device but only one has access to the device at a time to prevent inconsistencies. Unlike what happen with ECA, it is not the last who speaks who get the access. In CCBL, contexts are **ordered** in a priority list. When several contexts try to modify a device, only the one that has the maximum index in the priority list can do it. If this context becomes inactive, then the next context in the priority list sets the state of the device. The same process happens when a new context that tries to set the device becomes active. The order function used in CCBL is as follow, C1 > C2 means that:

- Either C1 is a descendant context of C2.
- Or C1 and C2 have the same parent context; C1 is indexed after C2.
- Or C1 has an ancestor A1 and C2 have an ancestor A2 such as A1 and A2 have the same parent context, A1 is indexed after A2 in the list of sub-contexts.

Every context specifies either explicitly or implicitly the state of all devices and services. The implicit specification of the state of devices and services is what we call the **cascade mechanism.** We took the inspiration from the Cascading Style Sheet (CSS) language [6] that enables styling of HTML documents.

In order to illustrate the cascade mechanism, let us consider the CCBL program illustrated in Fig. 1. If we suppose that Martin is at home and that he is *on the* phone, then two contexts try to set the state of the lamp: the root context and the one that has the Boolean condition "*System detects that Martin is at home*"). The lamp lights in white because "System detects that Martin is at home" is a descendant context of the root context. If Martin leaves home while still *on the* phone, the lamp will be set to off, as the only remaining context that try to set it will be the root context.

We designed the cascade mechanism to make it easy for users to express what happens when the system exits a context. Devices can never be in an undefined state, at least the root context define what is their state. This way, users can focus on expressing the device states associated with context without having to bother with what happens when a context is over as it is when programming with ECA.

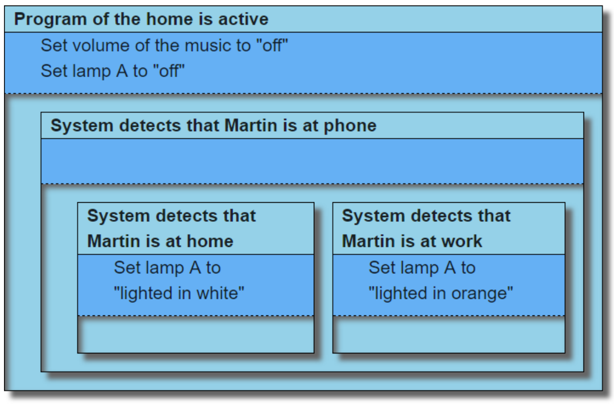
Program of the home is active
Set volume of the music to "off"
Set lamp A to "off"
System detects that Martin is at phone
System detects that Martin is at home
Set lamp A to "lighted in white"
System detects that Martin is at work
Set lamp A to "lighted in orange"

Fig. 1. Illustration of a CCBL program that control lamp A with respect to the fact that Martin is phoning or not and with respect to his location.

2.3. Contexts organization in CCBL

We designed CCBL in order to avoid the pitfalls identified for ECA, one of which is the accumulation of rules without hierarchy. CCBL provides several ways to organize the contexts based on Allen's interval Algebra [1]. In this section, we expose the different context's relationships that CCBL support. For each of them, we discuss the semantic implications in CCBL. We also illustrate how it can help end-users to express behaviors more easily.

SC during C. This relationship aims at supporting the expression of particular cases or exceptions. It expresses that context SC is considered only when context C is active. The semantic is as following: While context C is active, actions specified by C apply to the devices and services. When the context SC becomes also active, then the actions specified by SC override the ones of specified in context C. Vice versa, when the context SC becomes inactive, the actions it specifies do no more override the ones specified by C: The states of devices and services then rolls back to what is specified by C.

A possible scenario that illustrates the relevance of this relationship is the following one: "*When Martin is at home (context C), he wants the music to be played at a normal volume, except when he is talking on the phone (context SC). In this case, he wants the music to be played at a low volume*". When using the "during" relationship, users only have to specify the general context (C: "Martin is at home") that is associated with devices states (the volume of the music is low) and a particular context (SC: "Martin is talking *on the* phone"). CCBL automatically switches from the general context to the specific context, the users do not have to bother with specifying what happens when a context is entered or leaved, they only have to specify what happen when the system detects that a context is active. This way, we tackle the *non-sequitur* problem identified by [Huang 2015].

SC starts C. This relationship is a specialization "SC during C". CCBL considers the SC context only at the beginning of C. The starting event of SC is "when C starts". Therefore, SC can only happens once a time as long as C is active (and at its beginning). Users can define SC by a Boolean expression and/or an ending event.

A possible scenario that illustrates the relevance of this relationship is the following one: "*When Alice is at home (context C), the system plays music from her favorite radio except when she just arrives, then if she has got notifications, the system enumerates them vocally until there is no more notifications or she say stop (context SC)*".

SC finishes C. This relationship is a specialization "SC during C". CCBL consider the SC context only at the end of context C. As neither the duration of SC nor the duration of C can be known in advance, CCBL only allow to define SC using a start event. Obviously, the end event of SC is the end of context C.

A possible scenario that illustrates the relevance of this relationship is the following one: "*When Alice is at home during the morning of a working day (context C), listen to the radio news. If the system detects that she may be late at work (start of context SC), then the system plays a special music until she leaves her home*".

C2 takes place just after C1. This relationship expresses that the system considers C2 only just after C1 becomes unavailable. A possible scenario that illustrates the relevance of this relationship is the following one: "*When Alice is watching TV (global context), when she is on the phone (context C1), the TV is paused. As sometimes she walks through her apartment while talking, she wants that the TV to be kept on pause after the phone call ends while she is not sitting on her sofa (context C2)*".

C2 takes place after C1. This relationship expresses that the system considers C2 only after C1 becomes unavailable, but not necessarily just after. A possible scenario that illustrates the relevance of this relationship is the following one: "*After a soirée, Alice wants to be sure that her friends arrived at home (she always fear car crashes). She uses a lamp to be informed about the position of her friends' car. During the soirée, the lamp is colored in white (context C1). After the soirée, the lamp is lit in orange. When her friends are at home (context C2), the lamp is lit in green*".

Unsupported Allen's relationships. We do not support the relationship of overlapping (context C1 overlaps context C2) as it can only be detected a posteriori and not on the moment. We also do not explicitly support the relationship of equality (C1 happened exactly when C2 happens) as it can easily modeled by expressing a conjunction between the conditions of C1 and C2.

3. Implementation and Future works

We have developed a prototype of CCBL interpreter using the NodeJS[1] environment. This prototype is able to build a coherent system with devices and contexts and to maintain the system in a state depending on the contexts. In order to test the CCBL expressive power, we simulate a Smart Home environment. We have also implemented a simple user interface that allows the user to explore the devices and the contexts to see their status. The source code is available online[2]. In future works,

In future works, we plan to bridge this interpreter to a real home automation system such as OpenHAB[3]. We will also conduct experiment to test whether CCBL logic is more suitable than ECA for end users. Last, we will explore how to represents CCBL program to users in order to enforce their understanding of programs as well as to enable them to edit them efficiently.

4. References

1. James F Allen. Maintaining knowledge about temporal intervals. Communications of the ACM, 26(11):832–843, 1983.
2. AJ Brush, Bongshin Lee, Ratul Mahajan, Sharad Agarwal, Stefan Saroiu, and Colin Dixon. Home automation in the wild: challenges and opportunities. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, pages 2115–2124. ACM, 2011.
3. Margaret M Burnett and Christopher Scaffidi. End-user development. Encyclopedia of HumanComputer Interaction, 2011.
4. Julio Cano, Gwenaël Delaval, Eric Rutten. Coordination of ECA rules by verification and control. 16th International Conference on Coordination Models and Languages, Jun 2014, Berlin, Germany. 16 p., 2014.
5. Alexandre Demeure, Sybille Caffiau, Elena Elias, and Camille Roux. Building and using home automation systems: a field study. In IS-EUD 2015, pages 125–140. Springer, 2015.
6. Håkon Wium Lie, Cascading Style Sheets, Thesis submitted for the degree of Doctor Philosophiœ, Faculty of Mathematics and Natural Sciences, University of Oslo, 2005.
7. Justin Huang and Maya Cakmak. Supporting mental model accuracy in trigger-action programming. In Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing, pages 215–225. ACM, 2015.
8. Sarah Mennicken, Jo Vermeulen, and Elaine M Huang. From today’s augmented houses to tomorrow’s smart homes: new directions for home automation research. In Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing, pages 105–115. ACM, 2014.
9. Chandrakana Nandi and Michael D. Ernst. 2016. Automatic Trigger Generation for Rule-based Smart Homes. In Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security (PLAS '16). ACM, New York, NY, USA, 97-102.
10. Fabio Paternò. End user development: Survey of an emerging field for empowering people. ISRN Software Engineering, 2013, 2013.

[1] https://nodejs.org
[2] https://gitlab.com/ccbl/ccbl-engine
[3] https://www.openhab.org