SAASHA : a Self-Adaptable Agent System for Home Automation

Fady Hamoui, Christelle Urtado, Sylvain Vauttier
LGI2P / École des Mines d'Alès
Nîmes, France
≺First≻.≺Last≻@mines-ales.fr

Marianne Huchard
LIRMM, UMR 5506, CNRS and UM2
Montpellier, France
Huchard@lirmm.fr

Abstract

This paper proposes SAASHA, a Self-Adaptable Agent System for Home Automation that provides end-users with the capacity of defining custom scenarios to act on their environment. The proposed system adapts dynamically to the environment without any expert intervention. It is composed of two types of component-based software agents: Graphical User Interface Agents that constitute the system's front-end and Device Control Agents that control the devices from the environment and implement user-defined scenarios. SAASHA seamlessly avoids scenario conflicts and automatically recovers from device failures.

1 Introduction

Home environments are composed of networks of domestic electronic devices controlled by home automation systems. Each device provides services described by a set of operations and emits events that reflect the change of its state. Home automation systems implement scenarios that coordinate the actions of various devices in their environment. Let us, for example, consider an environment that contains a shutter, a radiator, a light and a clock. The scenario "agreeable temperature in the evening" is defined by "after 7:00 PM, if the living room temperature is below 17°C, the shutter should be closed, the light turned on and the radiator turned on at power 6". There is a strong need for home automation systems that adapt to various environments and to environment changes [9]. To do so, we advocate that home automation systems must possess the following qualities [2, 3]:

• *Configurability*. The system must be configurable to fit the needs of both its administrators (that might need to parameterize the system's architecture) and its end-users (that must be able to directly control devices, execute predefined scenarios and define new custom scenarios) [4].

• *Context-awareness*. The system must be aware of both its internal state (failure of its constituents) and changes in its environment (device addition or removal).

• *Autonomic reconfiguration*. The system must be able to reconfigure itself automatically to adapt to system-related or environment-related changes. It must also be capable of seamlessly managing conflicts in service executions without human intervention [5].

• *Dynamic adaptation*. The system must be capable to dynamically perform configuration and reconfiguration operations without disrupting its running [5].

In this paper, we present SAASHA, a Self-Adaptable Agent System for Home Automation. The article is structured as follows. In section 2, we present the general architecture of SAASHA. Sections 3 and 4 describe agents' inner architecture and the processes for defining and realizing scenarios. In section 5, we present the technologies used to implement our system. In section 6, we compare our system with existing proposals. Finally section 7 concludes and draws several research perspectives.

2 The SAASHA home automation system

The architecture of the SAASHA system consists of a set of component-based agents. Agents are active distributed entities which perceive information from their environment and react to events by running behaviors. They collaborate to realize complex scenarios. A software component [14] is a decoupled reusable programming unit, which can be assembled to other components through interfaces. These component assemblies can be dynamically modified by adding, removing or replacing components or by changing the connections between them. We define the internal structure of our agents using such an architecture based on components in order to benefit from the capability to dynamically adapt their behavior. SAASHA agents are of two kinds: Graphical User Interface Agents (GUIAs) and Device Control Agents (DCAs). GUIAs are responsible for the system's interactions with end users and system administrators. They provide GUIs to define scenarios and to configure DCAs. DCAs perform device detection and control, in order to execute scenarios. Administrators launch GUIAs

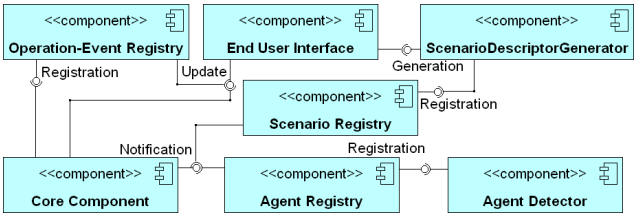

Figure 1. Internal architecture of a GUIA.

that connect to their networked environment. When GUIAs detect DCAs, they register them in their local registries and update their DCA configuration GUI. Administrators can launch a DCA from the GUI. Once launched, DCAs detect available agents and devices and register them in their local registries. The optional configuration of a DCA consists in specifying its control scope (device and service type, location). If a DCA is not parameterized explicitly, it controls all the detected devices (default setting). Once setup is complete, DCAs generate device control components accordingly. For each device, a DCA generates and dynamically deploys a control component based on the descriptors provided by the device. Control components supply three types of interfaces; *sensor interfaces* hold operations that retrieve values measured by a device in the environment, *actuator interfaces* hold operations that perform actions on a device, *event interfaces* provide events emitted by a device and register component subscriptions to these events. Once control components are generated, DCAs send information messages that GUIAs use to present a list of the available services. Users can define scenarios through a dedicated end-user GUI. The GUIA registers the defined scenario and generates a scenario descriptor. This descriptor is sent to all the DCAs involved in scenario realization and to all the GUIAs for logging purposes. Depending on their role (further details in Sect. 4), DCAs can then generate and deploy coordination components that dynamically adapt the agents' internal architectures to implement scenarios.

3 Graphical user interface agents and scenario definition

Figure 1 shows the internal architecture of GUIAs. The *Core Component* is responsible of both the communication between agents and of the implementation of GUIA's behavior. Agents are automatically detected by the *Agent Detector* component. User-oriented services provided by DCAs are registered in a local service registry managed by the *Operation-Event Registry* component. When a user defines a scenario, the *Scenario Descriptor Generator* component generates a descriptor. The scenario definition GUI and the scenario definition process are detailed in [6]. Each scenario is defined as an ECA (Event / Condition / Action) rule [8] which combines in its event, condition and action clauses event, sensor and actuator services provided by DCAs. To help users select the services, SAASHA provides two mechanisms:

- *Context restriction*. Users can choose to browse services from the whole house or restrict to a given room.
- *Device selection modes*. Services might be provided by several devices of the same type, SAASHA provides three device selection modes:
  - *all devices* of a chosen type in the specified context.
  - *any device* of a chosen type from the specified context.
  - *a specific device* from the specified context.

4 Device control agents and scenario implementation

The core architecture of a DCA as is presented in the upper part of Figure 2. The *Core Component* is responsible of the communication between agents. Available devices are detected by the *Device Detector* component. Control and coordination components are generated by the *Component Generator* component. Scenarios can be implemented in two ways, either centralized (see Sect. 4.1) or distributed (see Sect. 4.2). When implementing and deploying a scenario, the centralized approach is preferred to the distributed one when possible as it requires no message passing between DCAs. This choice is made by GUIAs.

4.1 Centralized scenarios implementation

The DCA in charge of a scenario generates and deploys a corresponding coordination component. This coordination component is dynamically bound to the control components prescribed by the scenario. Once assembled prop-

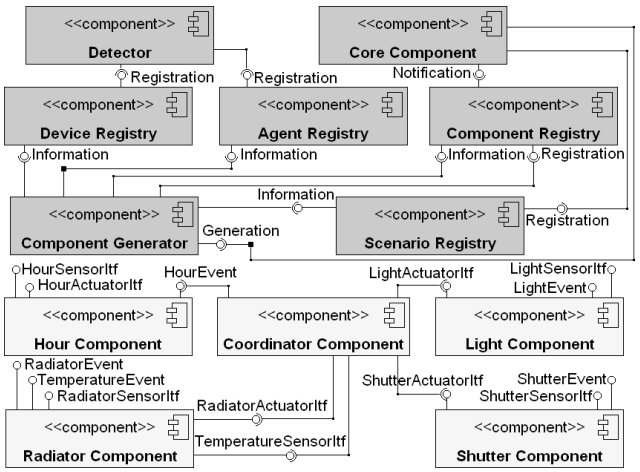

Figure 2. Internal architecture of a DCA.

erly, the coordination component waits for event notification. When an adequate event is received, it checks whether the condition is satisfied. If so, it triggers the execution of actions. The DCA architecture that implements our example scenario in a centralized way is shown in the lower part of Figure 2.

4.2 Distributed scenarios implementation

Figure 3 shows how distributed DCAs can cooperate to implement a scenario. DCAe is only responsible for both capturing the event and coordinating scenario execution. DCAc is responsible for verifying the condition. When the event is raised, DCAe sends a message (1) to DCAc to make it check the condition. When the condition is satisfied, DCAc sends a message (2) to DCAe to notify that the condition is satisfied. Then, DCAe sends request messages (3) to all the DCAs in charge of actions (DCAc, DCAa1 and DCAa2). In the general case, the condition clause of a scenario can be expressed as a conjunction of several conditions. This is why DCAe (that controls the event) has been chosen as a central coordination point.

4.3 Scenario conflict avoidance

An action conflict is a situation where a scenario executes an action on a device very soon after an opposite action has been executed on the same device. To prevent conflicts, SAASHA blocks opposite actions during a configurable remanence period. There nonetheless are two exceptions, for which the opposite action is not blocked; if an opposite action is *executed by a user*, or if an opposite action is to be *executed by a predominant scenario*. Predominant scenarios manage emergency situations and must always be executed.

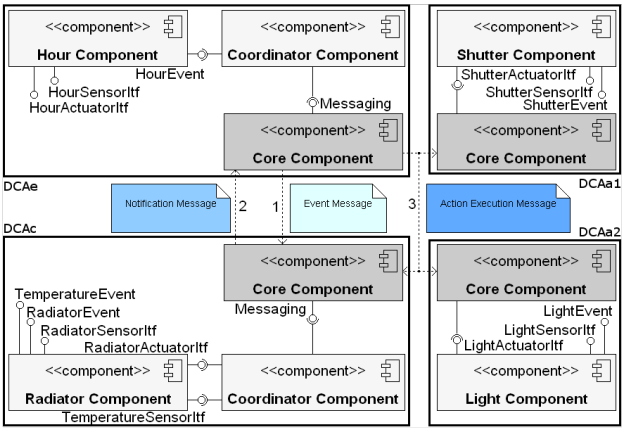

Figure 3. Example scenario implementation.

4.4 Device failure recovery

SAASHA reconfigures itself transparently according to device selection modes when a device fails or disappear. In *all devices* mode, there is no need for reconfiguration. The operation is not executed by the faulty device which makes nearly no difference. In *any device* mode, the system can automatically replace the faulty device by an other device from the same context. If no alternative device is found or if the scenario is to be by a *specific device* only, the system suspends scenario execution.

5 Prototype implementation & experiments

A prototype of SAASHA has been implemented using OSGi[1] and UPnP[2] technologies. The complete meta-model is presented in [6]. Agents are implemented as UPnP devices. This provides them with a communication service to receive messages from other agents. An agent life cycle control service that enables system administrators to activate, stop or dismiss agents. DCAs provide an extra setting service that enables system administrators to configure them via GUIAs. UPnP makes multi-agent system management easier by providing a mechanism for the automatic configuration of agents' communication resources and for the automatic detection of other agents. Components in agents' internal architectures are implemented as OSGi bundles managed thanks to an OSGi platform. The *Device Detector* component is implemented as a UPnP control point which detects all the available devices. A control component is a control point that handles a single UPnP device, invokes service operations and can subscribe to events emitted by the device's services. A prototype of the SAASHA multi-agent system has been deployed and tested on a set of OSGi Felix[3] containers. It includes a GUIA and a set of DCAs controlling virtual devices (reused from the Cybergarage[4] project). Various scenario deployment tests have been successfully conducted.

6 Discussion

Home automation systems can be classified into three categories, regarding their architectures. Centralized systems [1, 11, 5, 7, 10] are driven by unique control points. Distributed systems [9, 12] are composed of intelligent devices. Federated systems [16, 13, 15] are commonly designed as multi-agents systems. The qualities established in Sect. 1 are used to compare SAASHA with representative home automation systems.

[1] OSGi Alliance, http://www.osgi.org
[2] UPnP forum, http://www.upnp.org
[3] http://felix.apache.org/site.index.html
[4] http://www.cybergarage.org/

• *Configurability*: No system offers full system setting capabilities. SAASHA partly fulfills this quality by allowing administrators to set agent perception but, for now, they cannot choose scenario deployment strategies.

• *End-user orientation*: Systems have predefined scenarios the implementation of which depends on the context. Only [12] proposes a scenario definition GUI but it is directed to expert-users. These scenarios also are restricted to sequences of services. [1, 11] simply enable users to trigger services. SAASHA enables users to define complex custom scenarios based on events and conditions.

• *Context-awareness*: All the reviewed systems are aware of device appearance or disappearance (environment), but only two of them [5, 7], apart from SAASHA, detect component failure (system). Even federated systems are not aware of agents' failure.

• *Autonomic reconfiguration*: When a device fails (environment), centralized and federated systems propose a reconfiguration which amounts to uninstall components and deploy others. In distributed systems, device failure stops a set of scenarios but no reconfiguration is proposed for recovery. Finally, no existing system except SAASHA manages conflicting scenarios (system). However, SAASHA does not yet handle agent's failure.

• *Dynamic adaptation*: Scenario implementation (scenario) is performed dynamically in most systems. It consists in the dynamic deployment of a set of components. In distributed systems, it is done by adding rules. Finally, unlike SAASHA, systems do not dynamically integrate new system settings (system).

7 Conclusion and perspectives

This paper presented SAASHA, a Self-Adaptable Agent System for Home Automation that enables users to define scenarios to orchestrate services and events provided by devices from the environment. SAASHA agents connect to the environment and adapt themselves by generating and assembling control components (that control devices) and coordination components (that orchestrate control components to implement user-defined scenarios). Our solution offers a simple means to express user scenarios that relies on the system's "intelligence" to automate the adaptation and deployment agents. Finally, SAASHA handles conflicting scenarios and device failures automatically. Preliminary experiments were conducted on a small set of agents and virtual devices. Perspectives are to graphically simulate a home environment to conduct larger scale tests and demonstrate easier. We will also need to manage agents' failure recovery. Furthermore, we plan to develop a GUIA taxonomy to set access rights to services (*eg.* for parental control).

References

[1] S. Berger, H. Schulzrinne, S. Sidiroglou, and X. Wu. Ubiquitous computing using sip. In *13th Int. Wkshop on NOSSDAV*, pages 82–89, New York, USA, 2003. ACM.

[2] A. Bottaro, A. Gerodolle, and P. Lalanda. Pervasive service composition in the home network. In *IEEE 21st Int. Conf. on AINA*, Niagara Falls, Canada, pp 596–603, May 2007.

[3] J. Bourcier, A. Chazalet, M. Desertot, C. Escoffier, and C. Marin. A dynamic-SOA home control gateway. In *IEEE Int. Conf. on SCC*, Chicago, USA, pp 463–470, Sept. 2006.

[4] M. Burnett, S. K. Chekka, and R. Pandey. FAR: An end-user language to support cottage e-services. In *Proc. IEEE Int. Symp. on Human-Centric Computing Languages and Environments*, Stresa, Italy, pp 195–202, May 2001.

[5] G. Grondin, N. Bouraqadi, and L. Vercouter. MaDcAr: An abstract model for dynamic and automatic (re-)assembling of component-based applications. In *9th Int. Symp. on CBSE*, Västerås, Sweden, LNCS, 4063:360–367, June 2006.

[6] F. Hamoui, M. Huchard, C. Urtado, and S. Vauttier. Specification of a component-based domotic system to support user-defined scenarios. In *21st Int. Conf. SEKE*, Boston, USA, pp 597–602, July 2009.

[7] V. Hourdin, J.-Y. Tigli, S. Lavirotte, G. Rey, and M. Riveill. SLCA, composite services for ubiquitous computing. In *Int. Conf. on Mobile Technology, Applications, and Systems*, New York, USA, pp 1–8, 2008. ACM.

[8] J.-Y. Jung, J. Park, S.-K. Han, and K. Lee. An ECA-based framework for decentralized coordination of ubiquitous web services. *Inform. & Soft. Tech.*, 49(11-12):1141–1161, 2007.

[9] T. Kirste. Ambient intelligence: Towards smart appliance ensembles. In *From Integrated Publication and Information Systems to Information and Knowledge Environments*, pages 261–270. Springer, 2005.

[10] G. Nain, E. Daubert, O. Barais, and J.-M. Jézéquel. Using MDE to build a schizophrenic middleware for home/building automation. In *1st Europ. Conf. on ServiceWave*, pages 49–61, Madrid, Spain, 2008.

[11] T. Nakajima and I. Satoh. A software infrastructure for supporting spontaneous and personalized interaction in home computing environments. *Personal Ubiquitous Comput.*, 10(6):379–391, 2006.

[12] M. Nakamura, H. Igaki, H. Tamada, and K. ichi Matsumoto. Implementing integrated services of networked home appliances using service oriented architecture. In *2nd Int. Conf. on SOC*, New York, USA, pp 269–278, Nov. 2004. ACM.

[13] M. Son, D. Shin, and D. Shin. Design and implementation of the intelligent multi-agent system based on web services. In *IEEE 7th Int. Conf. on WAIMW*, Washington, USA, 2006.

[14] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 2002.

[15] M. Vallée, F. Ramparany, and L. V. cou ter. Using device services and flexible composition in ambient communication environments. In *1st Inter. Wkshop on RSPSI*, http://www.igd.fhg.de/igd-a1/RSPSI/papers/RSPSI-Vallee.pdf, 31/05/2010, 8 pages, Dublin, Ireland, May 2006.

[16] C.-L. Wu, C.-F. Liao, and L.-C. Fu. Service-oriented smart-home architecture based on OSGi and mobile-agent technology. *IEEE Trans. on SMC*, 37(2):193–205, 2007.