A service component framework for multi-user scenario management in ubiquitous environments

Matthieu Faure, Luc Fabresse, Marianne Huchard, Christelle Urtado, Sylvain Vauttier

A Service Component Framework for Multi-User Scenario Management in Ubiquitous Environments

Matthieu Faure*,†, Luc Fabresse†, Marianne Huchard‡, Christelle Urtado*, and Sylvain Vauttier*

*LGI2P / Ecole des Mines d'Alès, Nîmes, France
{Matthieu.Faure, Christelle.Urtado, Sylvain.Vauttier}@mines-ales.fr

†Ecole des Mines de Douai, Douai, France
Luc.Fabresse@mines-douai.fr

‡LIRMM - UMR 5506, CNRS and Univ. Montpellier 2, Montpellier, France
huchard@lirmm.fr

***Abstract*—Software dedicated to ubiquitous environments has to deal with the multiplicity of devices and users. It also has to adapt to frequent changes in its environment. Users can easily access and trigger services provided by different devices but also need to implement complex scenarios, *i.e.*, structured compositions of multiple service. State-of-the-art frameworks do not fully meet the expectation we identified. This is why, we designed the SaS (Scenarios As Services) ubiquitous software: a platform for ubiquitous systems that provides a SDL (Scenario Description Language) to support the creation of tailored user-centric scenarios. Our previous work on the subject did not tackle all distribution and concurrency concerns. In this paper, we present SaS's new features. Using the improved SDL, a user can now describe scenarios that combine services even if all of them are not currently available and will never be at the same time. Moreover, different scenario sharing mechanisms coupled with an access right policy are now included in SaS. SaS is currently implemented in a prototype on top of OSGi.**

Keywords-Ubiquitous environment; service-oriented computing; user-centric; service composition; scenario creation.

I. Introduction

More and more electronic devices (such as smartphones, tablet PCs, etc.) assist us in our daily life. They can interact with their environment and propose various functionalities to users. This is the rise of ubiquitous computing [1][2]. These functionalities can be handled as services, and thus, Service-Oriented Computing (SOC) [3] is a suitable paradigm to design software for ubiquitous environments. Service access and system adaptability to environmental changes are already well handled by execution frameworks. However, to our knowledge, these systems fail to meet user expectations to express their needs as complex scenarios involving multiple services. Based on this observation, we designed the SaS (Scenarios as Services) ubiquitous software [4]. SaS features a service component framework that enables end-users to easily define, control and share scenarios. SaS also proposes an SDL to create scenarios as service compositions.

Besides, ubiquitous environments involve multiple users and devices. Consequently, handling previously unknown device types, sharing information among users and handling control device mobility are challenging issues. First, device types must not be hardwired in the system. It has to be possible to create scenarios with services from specific devices but also from any device of a given type. This capability makes the system more flexible to device change. Second, an access right policy and a process dedicated to sharing scenarios must be specified. Thirdly, handling control device mobility can be seen both as a constraint on the system (that must dynamically adapt to its changing environment) but also as a chance (as the system can benefit from mobility while executing scenarios that involve services that never coexist in a same environment).

The SaS system is twofold. It divides into a scenario description language called SaS-SDL that provides simple means to describe services, scenarios, environments and an execution framework called SaS platform that provides the processes to support the behavior of the ubiquitous software. In this paper, we focus on SaS's new features. The improved SaS-SDL now manages the environment. In addition, SaS handles scenario sharing among selected users, service memorization for future scenario creation and scenario mobility (execution distributed in multiple places and times).

This paper is further organized as follows. Section II introduces service and scenario declaration in SaS-SDL. Section III presents the new feature of SaS-SDL: context management. Then, Section IV describes how the SaS system executes distributed scenarios. Section V is dedicated to the design of our prototype implementation. Related works are discussed in Section VI. Finally, Section VII concludes this paper and draws perspectives.

II. Service and Scenario Declaration with SaS-SDL

In this section, we give an overview of service and scenario declaration (a previous version was presented in [4]) using SaS-SDL, the proposed scenario description language. SaS-SDL enables end-users to create scenarios

that correspond to their needs. Improvements are specifically introduced here, such as: multiple operation selection schemes (from a specific device/service or not), the ability to define and set scenario parameters, the ability to specify the execution type (either in sequence or in parallel) of an action list. Compared to other programming languages for service composition (like BPEL [5]), which are imperative and designed for executable processes, our SDL is a high level language, which is declarative and dedicated to end-users. With this SDL, SaS automatically declares services after they are discovered and then, users can declare scenarios.

A. Service declaration

To be interoperable, SaS does not restrict to a protocol but uses a generic pivot mode to declare services. SaS can be specialized by adding bridges to different protocols (as Frascati [6] and EnTiMid [7] already does). SaS-SDL defines a service by a device (its provider), a name and an operation list. Operations have a return type and can have typed parameters. Users can choose if a service operation to compose comes from a specific device and a particular service or not. To do so, the new version of SaS-SDL features the special word `any`, which enables to elude the provider device or the service name. Only the main elements of the grammar are presented in Listing 1.

```
<service> ::= service <device> <service_name> <op_list>
<op_list> ::= ( <operation> ; )+
<operation> ::= operation <operation_name>([<param_list>])
                : <return_type>
<param_list> ::= <parameter_type> (,<parameter_type>)*

<return_type>::= <type>
<parameter_type> ::= <type>
<device> ::= identifier | any
<service_name> ::= identifier | any
```

Listing 1. Service declaration with the Backus–Naur Form (BNF)

Listing 2 is a *Clock* service declaration example.

```
service clock_Bedroom Clock
operation getTime() : Time;
operation setTime(Time) : void;
```

Listing 2. Service declaration example

B. Scenario declaration

A scenario has a name, some actions and properties. An action can be: (i) an operation invocation, (ii) an alternative (*if - else*), or (iii) a repetition loop.

Listing 3 describes the main elements of a scenario declaration using the BNF notation. With this improved version of SaS-SDL, scenarios have properties, which enable to specify if the scenario is exportable, editable, etc. Moreover, action lists are now executed in sequence by default, however, SaS-SDL enables users to specify some actions to execute in parallel. In addition, users can now leave some parameter values blank at scenario creation. This is represented by the *?* value in SaS-SDL. Such eluded parameters become *scenario parameters* and must be valued by users every time the scenario is invoked.

```
<scenario> ::= scenario <scenario_name> <action_block>
               [<scenario_properties>]

<action_block> ::= { ( <action> )+ } |
 { ( [ [<parallel_exec>] <action_list> <action_list> ] ) }
<action_list>::= ( <action> | <action_block> )+

<action> ::= <op_invocation> ; | <alternative> | <repeat>

<op_invocation> ::= (<device>) <service_name>.
                    <operation_name>([<parameter_list>])
<parameter_list> ::= (<op_invocation> | <parameter_value>)
                  (, (<op_invocation>|<parameter_value>) )*
<alternative> ::= if <cplx_condition> <action_block>
                  [<else_clause>]
<else_clause> ::= else <action_block>
<cplx_condition> ::=  ( <condition>
                       (<log_operator> <condition>)* )
<condition> ::= <op_invocation> <comp_operator>
             ( <op_invocation> | <value> )
<repeat> ::= (while<cplx_condition> | <repeat_value> times)
             <action_block>

<parameter_value> ::= <value> | ?
<parallel_exec> ::= parallel:
<log_operator> ::= and|or|not
<comp_operator> ::= < | <= | > | >= | ==
```

Listing 3. Grammar of the scenario declaration using the BNF notation

Listing 4 illustrates SaS-SDL with a scenario example.

```
scenario night
if ( (any) Clock.getTime() == 6pm and
 (BedroomThermomether) Thermometer.getTemperature() <= 17)
{
(BedroomRadiator) Heater.setValue(7);
}
```

Listing 4. Scenario declaration example

C. Users point of view

SaS integrates a GUI based on our SaS-SDL to facilitate scenario creation for end-users.

1) Service selection: Our GUI presents ordered services in three columns: by device, service and operation. To avoid duplicates, SaS groups services and operations with same name. When users select a device (*resp.* a service), services (*resp.* operations) attached are filtered. It enables users to select a service (*resp.* operation) from a specific device (*resp.* service). In addition, SaS indicates if a service is a scenario.

For users to create conditions on service availability and define alternatives, SaS adds the operation `isPresent` to each service.

2) Scenario creation: When users select a service operation to compose, SaS displays corresponding informations (provider device, service name, operation name and result type) and enables users to enter operation parameters. Users can either provide a fixed value or select another operation result (on which they can apply a basic operation such as +, -, *, /). In case the parameter type is complex, SaS only allows users to select an operation result. Figure 1 represents the GUI *sendMail* service operation, with two parameters (second one is complex).

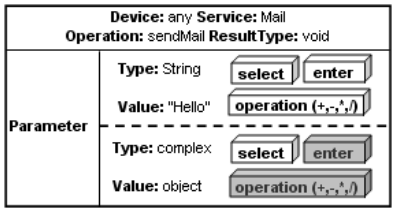

Figure 1. GUI: service operation

SaS provides users with several templates (*i.e.alternative*, *while*, *repeat*, etc.) to create scenarios. Users can combine templates to create the scenario skeleton. Then, users just need to put service operations inside the templates and complete with basic instructions (`and`, `or`, `not`, `<`, `>`, $\leq$, $\geq$, $==$). Figure 2 illustrates a scenario template and Figure 3 shows an example of scenario creation (scenario operations are represented by pictograms to simplify but they actually are similar to those in Figure 1).

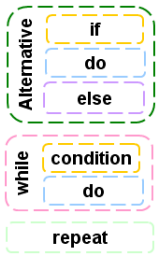

Figure 2. GUI: scenario templates

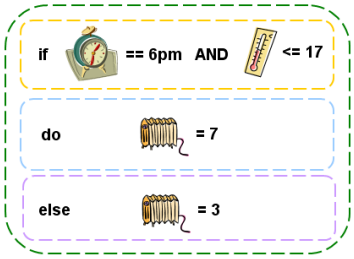

Figure 3. GUI: example of scenario creation

III. Context Management in SaS

As seen, our previous version of SaS enables users to create scenarios. To do so, they dispose of SaS-SDL and its graphical representation. We presented in Section II some improvements for service and scenario declaration. Nevertheless, ubiquitous computing implies users mobility and multiplicity. As defined in [8], two characteristics of ubiquitous system are the *social environment* and the *evolving environment*. A ubiquitous system should therefore provide an access right policy and advanced sharing mechanisms. Moreover, this system has to be adaptive but could also benefit from this changing environment.

A. SaS in ubiquitous environment

Ubiquitous environments involve electronic devices. We define two types of devices: simple devices (such as radiator, light) and control devices (such as laptop, pda) which have an advanced user interface (i.e., touch screen), and can be considered as personal and mobile. A SaS container (which contains all SaS mechanisms handled by SaS ubiquitous software) can therefore only be deployed on a control device to constitute a SaS *system*.

B. Service and System Directories

Every SaS system has a unique identifier. As a SaS system is associated to a unique user, sharing scenarios with select SaS systems is equivalent to define access rights. SaS systems (which might not be always available locally) are permanently indexed into a *system directory*. It makes possible to share scenarios with a system even if it is temporary unavailable (failure, mobility). Such a permanent index is also provided for services by the *service directory*. Users can registers services that they discovered or obtain service declarations from a scenario created by someone else. By this means, scenarios can be defined that include temporarily missing services.

To ease directory browsing, services and systems can be grouped into named categories. These categories are like keywords as a service (resp. a system) can be included into several distinct categories. Browsing by categories diminishes the amount of information to be presented to users. They can also by used to collectively export services (which can be equivalent to providing grouped access rights), see Section IV-B1 for details. Examples of categories might be locations (all services available at *home*) or users (all systems owneb by *kids*).

Listing 5 represents the main elements of the grammar for context management and Listing 6 illustrates how this part of SaS-SDL can be used. Scenarios in the service directory are highlighted to be differentiate from basic services.

```
<sas_system> ::= system<system_id><system_dir><service_dir>

<system_dir> ::= system_directory { (<system_cat>)* }
<system_cat> ::= category <cat_name> [<system_list>]
<system_list> ::= ( system <system_id>  )*

<service_dir> ::= service_directory { (<service_cat>)* }
<service_cat> ::= category <cat_name> [<service_list>]

<service_list> ::= [ services <service_name>
                     (, <service_name>)* ]
```

Listing 5. Context Management with SaS-SDL

```
system pda12
   system_directory {
      category mySystems
         platform Nokia3310
         platform Acer TimelineX
      category family
         platform macintosh
   }

   service_directory {
      category home
         [services TV, wakeUp]
      category office
         [services fax, print]
   }
```

Listing 6. Service and system directories

The class diagram of Figure 4 provides an alternative compact view of SaS-SDL.

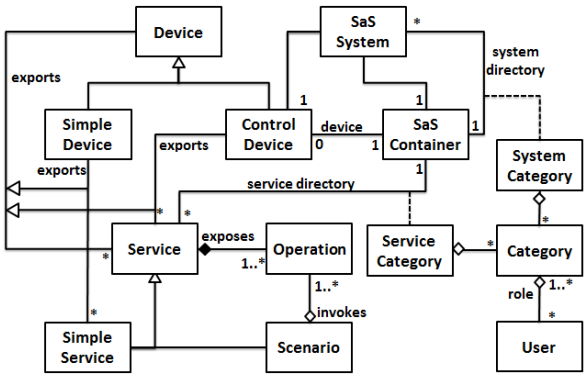

Figure 4. SaS-SDL class diagram

IV. Execution of Distributed Scenarios in SaS

This section presents how the SaS platform supports the execution of distributed scenarios.

A. Scenario execution control

To control scenario execution, SaS handles the *scenarios' life-cycle*. The objective here is threefold:

- provide basic *start, pause, abort and resume* operations for the user to manually control scenario execution,
- provide mechanisms on top of the middleware's detection capability to *dynamically react to detected changes in the environment* (*eg.* unpredictable service unavailability consequent to its failure or mobility),
- provide mechanisms that *take advantage of service and scenario mobility to enrich scenario functionalities* (*eg.* enabling to combine in a same scenario services that will never coexist on a single SaS platform).

Scenario life-cycle. Scenario execution is externally controlled: users can interfere during execution and changes in the environment can trigger compulsory reactions from the platform (*eg.* it is impossible to ignore that a service disappeared while being executed). Therefore, scenario life-cycle needs to be rich enough to encompass specific behavior to dynamically react to many different situations. Scenario life-cycle management is enforced by SaS platform. The state diagram of Figure 5 illustrates the proposed life-cycle. Here, most transitions are initiated by users (except when *finished*, which is automatic) which use the basic *start, pause, resume* and *abort* service operations for the scenario.

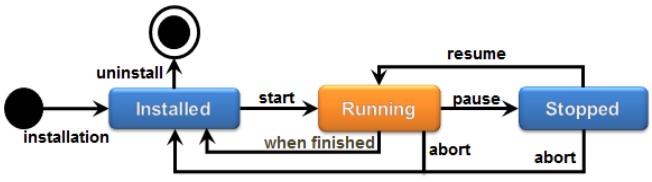

Figure 5. Scenario life-cycle in SaS

Fine, step by step, scenario running. Scenario executions cannot be considered atomic as they involve multiple and distributed service invocations. Moreover, scenario execution can be paused at any time by users or be interrupted at any unpredictable step in case a service disappears.

The *Running* state itself decomposes into a more precise state machine (see Figure 6). SaS considers scenario execution as a succession of steps, and define pre-conditions and post-conditions for each. For example, a pre-condition can be the presence of appropriate services or the execution of a previous step. Post-conditions are threefold: (1) successful execution of the step, (2) a problem occurs (service disappearance or timeout), or (3) interruption by the user. Such capabilities are completed with a logging system that reports scenario step by step execution status. Users can therefore check scenario advancement through the `getScenarioState` operation. Moreover, this enables SaS to retrieve scenario status after an interruption. Transitions are all handled by SaS container.

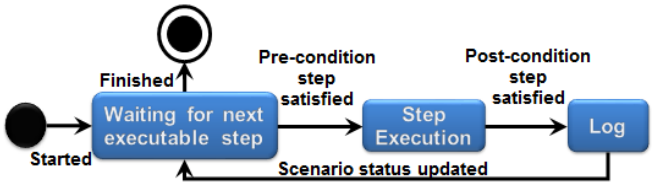

Figure 6. Internal running state diagram

Scenario delayed execution. The step by step running of scenarios has a positive counterpart when considering service mobility. If the user wishes to do so (this option is set at scenario creation), a scenario can be created that comprises operations that are never encountered in a same place at a same time. The user can choose from simultaneous (all services must simultaneously be present) or not. In the latter case, the user has to set a scenario maximum waiting period such as an hour or a day, that limits the duration the scenario might spend waiting for some services to appear.

When the scenario is to be executed, the steps that can be are and the system pauses the scenario until the next step is doable. The satisfaction of the next step precondition will automatically be detected and cause the execution to be resumed. If the device on which the scenario executes has not changed place, this step by step execution might have executed services that are supposed to be present at the same place but not at the same time (*eg.* a service offered by a device that moves with its user such as a mobile phone). If the device on which the scenario executes has changed place (*eg.* the scenario is executed on a device that moves with the user), this step by step execution might have executed services that are supposed to be present at two distinct places (*eg.* a service offered by a device at home and a service offered by another device in a hotel room).

B. Scenario distribution

Registering SaS systems enables users to select who to share scenarios with. Having multiple users generates the need for a scenario access policy. In addition, as devices might fail and scenarios not be shared elsewhere, maintaining scenario availability also lies on SaS's scenario redeployment capability.

1) Scenario access right policies: By exporting their scenarios as services, users can share them. However, users might not want everyone to have the same access to created scenarios. SaS provides two modes for sharing scenarios: *individual*, the scenario is shared with a specific system (which provides access to the system's owner) or *grouped*, the scenario is shared with a whole category of systems (which provides access to the set of these systems' owners). Grouped access mode can be used to designate all systems a given user has access to (*eg.* dad's) or all systems that pertain to another category (*eg.* local network). Three access levels are possible: *private*, the scenario is not shared, *delegated*, the scenario is known and remotely accessible to the systems it is shared with but the owner system possesses the only copy and still executes the scenario, and *copied*, the scenario is copied locally into the system it is shared with and can be executed on the new system autonomously.

2) Scenario redeployment: When a user shuts down his / her platform, the solution to maintaining scenario availability is redeployment. Before doing so, SaS first warns the user if a scenario provided by this platform still is running. User can wait for the end of scenario execution. Otherwise, SaS tries to redeploy the scenario on another platform and transfer its current status and execution advancement. The destination platform is chosen from other available SaS systems registered in the system directory. If none of these systems accept, SaS asks other SaS systems present in the environment. If the scenario has not been redeployed on platform restart, SaS asks the user if scenario execution should be resumed.

V. System Design and Implementation

This section describes the design and implementation of the SaS prototype. It is an ongoing work implemented in Java over OSGi [9] with iPOJO [10]. OSGi is a popular framework that enables to dynamically manage softwares as sets of decoupled modules called *bundles*. iPOJO is a full fledge Service-Oriented Component Model [11] based on OSGi. The main idea is that a component should only contain business logic as in EJB 3.0 [12] (*EJB entities*); SOC mechanisms should seamlessly be handled by the component container as container-managed cross-cutting services. The already implemented parts of SaS are presented in the previous paper [4].

Scenario delayed execution. Depending on execution rules (parallel or sequence), SaS invokes services present as defined in IV-A and register the result necessary for some services (as operation parameter).

As defined in [4], SaS translates a scenario in a succession of Java instructions thanks to Javassist [13]. Instead of implementing the whole scenario as the start operation, this version of SaS implements each action block of the scenario in different methods to enable a stepped and delayed execution. A scenario can now be launched even if all services are not present, and it keeps running until it ends, it is stopped, period of validity finishes or, the platform is closed. Leveraging iPOJO the presence of each service independently. So, when all the services involved in an action block become available, the appropriate method is automatically called.

Sharing scenarios. When users share a scenario with all the available platforms, SaS exports the corresponding service as a remote service.. This way, discovery and distribution can be handled automatically by the last version of OSGi. Instead, if users select some other systems to share a scenario with, SaS uses the *UpdateServiceDirectory* service exported by each SaS platform. It enables to send events (service appearance or disappearance) to selected systems.

VI. State of the Art

This section analyses a representative set of systems that provide a solution for ubiquitous environments and enable scenario creation.

SLCA [14] provides developers with means to compose web services. A composite service contains proxy components bound to involved web services. With **SODAPOP** [15], users specify a goal that the system tries to reach with the available services. The main hypothesis is that each service contains informations about its initial conditions and its effects. **MASML** [16] is a multi-agent system for home automation. Scenarios are defined with an XML syntax and consist of sequences of service operation invocations. Mobile agents are in charge of scenario execution. **SASHAA** [17] is one of our previous work, focused on ubiquitous systems for home automation. It enables end-users to create scenarios with Event - Conditions - Action rules through an appropriate GUI.

The SaS ubiquitous software manages scenario life cycle and provide users with basic *start, pause, resume* and *abort* operations to fully control scenarios, whereas MASML and SASHAA only enables to start and stop scenarios. The SaS system is the only one to to share scenarios with other users. SASHAA, SLCA and MASML handle adaptation to environmental changes, however, scenarios cannot be executed in different times on multiple places. SODAPOP manages the environment by automatically classifying new services according to pieces of information. However, users have no control on this organization. Moreover, SASHAA enables to specify locations for systems but not register services. Table I summarizes this study. Symbol ✓means

that the requirements is fulfilled, - signifies that it is partially accomplished and × represents an absence of solution.

Systems	Scenario Execution Control	Multi User	Adaptability	Context Management
SLCA	×	×	-	×
MASML	-	×	-	×
SODAPOP	×	×	×	-
SASHAA	-	×	-	-
SaS	✓	✓	✓	✓

Table I
SYSTEMS COMPARISON

VII. CONCLUSION AND FUTURE WORK

In this paper, we presented the new mechanisms of our SaS system to manage ubiquitous environments. In addition to enable scenario creation by service composition, the SaS-SDL provides means to organize users' contexts. Users can register services for a future use. SaS can execute scenarios step by step, at different times, on different platforms. Users can also classify surrounding SaS systems and share scenarios according to different access rights. A graphical representation of SaS-SDL enables end-users to benefit from SaS mechanisms.

For future work we want to add semi-automatic service composition to SaS. Learning from existing scenarios, SaS will propose some possible service compositions to the user. SaS will analyze which scenarios are created and used by users and will extract the more frequent services compositions.

ACKNOWLEDGEMENTS

This work is partially supported by the CARNOT M.I.N.E.S Institute (*http://www.carnot-mines.eu/*).

REFERENCES

[1] M. Weiser, "The computer for the 21st century," *Scientific American*, pp. 78–89, 1995.

[2] H. Schulzrinne, X. Wu, S. Sidiroglou, and S. Berger, "Ubiquitous computing in home networks," *IEEE Communications*, pp. 128–135, Nov 2003.

[3] M. P. Papazoglou, "Service-Oriented Computing : Concepts, Characteristics and Directions," in *Proc. of the 4th Int. Conf. on Web Information Systems Engineering*. IEEE, Dec 2003, pp. 3–12.

[4] M. Faure, L. Fabresse, M. Huchard, C. Urtado, and S. Vauttier, "The SaS Platform for Ubiquitous Environments," in *Proc. of the 23rd Int. Conf. on Software Engineering and Knowledge Engineering*, July 2011, pp. 302 – 307.

[5] OASIS, "Web services business process execution language version 2.0," april 2007, [Last consulting: July 2011]. [Online]. Available: http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf

[6] D. Romero, R. Rouvoy, L. Seinturier, and P. Carton, "Service Discovery in Ubiquitous Feedback Control Loops," in *Proc of the 10th IFIP Int. Conf. on Distributed Applications and Interoperable Systems*, ser. LNCS, F. Eliassen and R. Kapitza, Eds., vol. 6115. Springer, Jun 2010, pp. 113–126.

[7] G. Nain, E. Daubert, O. Barais, and J.-M. Jézéquel, "Using mde to build a schizofrenic middleware for home/building automation," in *ServiceWave'08: Networked European Software & Services Initiative (NESSI)*, Madrid, Dec. 2008.

[8] G. Banavar and A. Bernstein, "Software infrastructure and design challenges for ubiquitous computing applications," *Communi. of the ACM*, vol. 45, no. 12, pp. 92–96, 2002.

[9] OSGi Alliance, "OSGi Service Platform Core Specification Release 4," 2005, [Last access: July 2011]. [Online]. Available: http://www.osgi.org/download/r4v40/r4.core.pdf

[10] C. Escoffier and R. Hall, "Dynamically adaptable applications with iPOJO service components," *Proc. of the 6th int. conf. on Software composition*, pp. 113–128, Mar 2007.

[11] H. Cervantes and R. Hall, "Autonomous adaptation to dynamic availability using a service-oriented component model," in *International Conference on Software Engineering (ICSE)*. IEEE, May 2004, pp. 614–623.

[12] Sun Microsystems, "Enterprise javabeans specifications," may 2006. [Online]. Available: http://java.sun.com/products/ejb/docs.html

[13] S. Chiba and M. Nishizawa, "An Easy-to-Use Toolkit for Efficient Java Bytecode Translators," *Proc. of the 2nd int. conf. on Generative programming and component engineering*, pp. 364–376, Sept 2003.

[14] V. Hourdin, J. Tigli, S. Lavirotte, G. Rey, and M. Riveill, "SLCA, composite services for ubiquitous computing," in *Proc. of the Int. Conf. on Mobile Technology, Applications, and Systems*. New York, USA: ACM Press, 2008, pp. 1–8.

[15] J. Encarnaçao and T. Kirste, "Ambient intelligence: Towards smart appliance ensembles," in *From Integrated Publication and Information Systems to Information and Knowledge Environments*. Springer, Dec 2005, pp. 261–270.

[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, Part C*, vol. 37, no. 2, pp. 193–205, Mars 2007.

[17] F. Hamoui, M. Huchard, C. Urtado, and S. Vauttier, "Specification of a component-based domotic system to support user-defined scenarios," in *Proc. of 21st Int. Conf. on Software Engineering and Knowledge Engineering*, July 2009.