Towards scenario creation by service composition in ubiquitous environments

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

Towards Scenario Creation by Service Composition in Ubiquitous Environments.

Matthieu Faure[a], Luc Fabresse[a], Marianne Huchard[b], Christelle Urtado[c], Sylvain Vauttier[c]

[a] *Ecole des Mines de Douai, Douai, France – Matthieu.Faure@mines-douai.fr, Luc.Fabresse@mines-douai.fr*
[b] *LIRMM - UMR 5506, CNRS and Univ. Montpellier 2, Montpellier, France – huchard@lirmm.fr*
[c] *LGI2P / Ecole des Mines d'Alès, Nîmes, France – Christelle.Urtado@mines-ales.fr, Sylvain.Vauttier@mines-ales.fr*

Abstract

Ubiquitous computing provides a large access to different functionalities through electronic devices; however these devices are not always thought to function in a set. This limits their use. Scenario creation by service composition makes it possible for end-users to specify their needs more precisely. In Service-Oriented Computing, some frameworks (such as OSGi) provide mechanisms to develop components that provide services; however they are not adapted to end-users and limited for service composition.

This is why we developed a new user-centric system. Placed into an appropriate user electronic device, it discovers the available services around him and orders them to provide a better access through a GUI. Users can thus compose services to create a scenario thanks to a graphic adaptation of our ADL. Once created, this scenario can be registered and shared among users. A prototype (still in development) implements our system and demonstrates its feasibility. It can be improved by adding some recovery strategies.

Keywords: ubiquitous computing, service-oriented computing, service composition, scenario creation, OSGi, user-centric system

1. Introduction

With the rise of ubiquitous computing [1, 2], we are surrounded by electronic devices (such as smart phones or TVs) that propose a huge amount of services through public or private networks. According to the OASIS organization, *"a service is a mechanism to enable access to one or more capabilities"* [3]. In service-oriented computing (SOC [4]), and specially in home automation [5, 6, 7], efforts have been made to facilitate the use of these electronic devices through their services. However, it is still not easy for end-users that have no specific technical knowledge to fully benefit from the services proposed by they surrounding devices as shown in Figure 1. Indeed, each device provides its own set of services and users are often limited to use a single service at a time. However, users usually want to describe *scenarios* which involve multiple services and devices. A scenario is a combination of different services and it should be easily described by end-users since they are not experts. Allowing end-users to directly describe scenarios would be a first improvement and a step to go in the direction of ambient intelligence [8]. In addition, users might want to have access to their scenarios on several control devices (PDA, mobile phone or laptop). Moreover, several users can be in the same environment and eager to share scenarios. Scenarios should therefore be exported in the environment to be shared and re-used.

In this paper, we propose the SaS (Scenario as Services) system specifically designed to enable end-users to describe and use scenarios in a high level manner. Users describe scenarios through a dedicated GUI (Graphical User Interface). This GUI presents a classified and filtered view of available services and provides tools to easily *compose* services from a set of widely spread devices. Then, a scenario is registered for future use or sharing among end-users. Since we enable hierarchical composition of services, the SaS system integrates scenarios as regular services. We also propose a prototype of SaS , currently implemented on top of OSGi [9].

The remainder of this paper is structured as follows. Section 2 introduces the context of this work dedicated to open and distributed environments. Section 3 presents an overview of our proposition. Section 4 describes the creation of scenarios. Section 5 presents the architecture of our prototype built on the top of OSGi. Related work is presented and discussed in Section 6. Section 7 concludes this paper and draws some perspectives to this work.

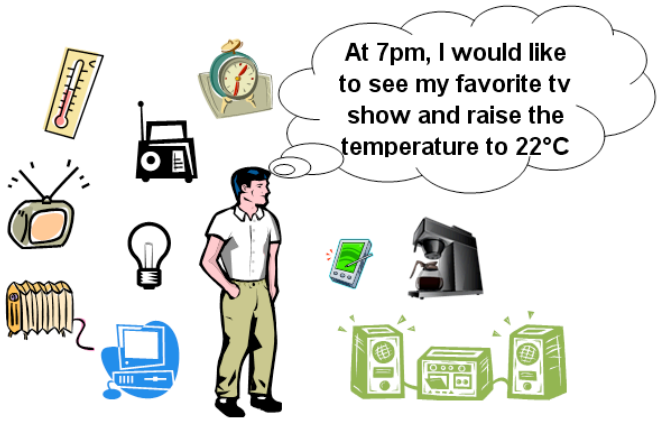

Figure 1: User's main issue

2. Context of Service-Oriented Computing

As a solution to develop and maintain applications distributed on several devices, SOC proposes to build applications out of interconnected *services*. OSGi [9] is a popular SOC framework that is widely adopted by industry for developers to create *bundles* (Java components).

In this paper, we use a specific terminology for the different elements that compose our system. A *device* is an electronic object (such as a computer). It is usually represented by a single OSGi bundle that contains a driver for controlling the device. A bundle publishes *services* (such as *Time*) which are implementations of Java interfaces. Each *service* provides one or more *operations* (such as *getTime* or *setTime*). They are the 'capabilities' of the OASIS definition. End-users need to be able to use these *operations*.

The OSGi platform manages the bundles' life cycle and the dependencies between them. Initially designed for a single Java virtual machine, the latest version (4.2) of OSGI now supports distribution (RFC 119 [10]). Our proposition uses this latest version because our system is intended to compose services controlled by a set of distributed OSGi platforms.

However, OSGi programming is difficult and limited regarding service composition which is the base of scenario creation. iPOJO [11] proposes an improvement to these issues. This framework is based on OSGi and follows the Service-Oriented Component Model [12]. The main idea is that a component should only contain business logic as in EJB 3.0 [13] (*EJB entities*); SOC mechanisms should seamlessly be handled by the component container as container-managed cross-cutting services. iPOJO also provides an Architecture Description Language (ADL) which handles how component instances should be connected together to meet their dependencies (*ie.* connect required interfaces to provided interfaces). This ADL is used to set up the component assemblies that provide the services. However, it is impossible to specify the relationships between the components. The ADL is limited to the declaration of service invocations, imports or exports.

Although all of these mechanisms are needed and useful for programmers, they are not usable by end-users. Indeed, they do not provide any means to define usage scenarios, *ie.* the services that are to be invoked to meet end-user requirements. The main objective of our system is to help users specify their needs precisely and to promote an advanced use of services by giving them some easy to use tools.

3. Overview of SaS

Our system is named SaS for *Scenario As Services*. Its purpose is to help end-users create scenarios by composing services and export the resulting scenarios into the environment, for future use.

For solving this, several steps are necessary. First, the system needs to discover the services, order them and display them. Then, users should be able to compose services to define a scenario. Finally, SaS creates this scenario and makes it available as a new service into the environment.

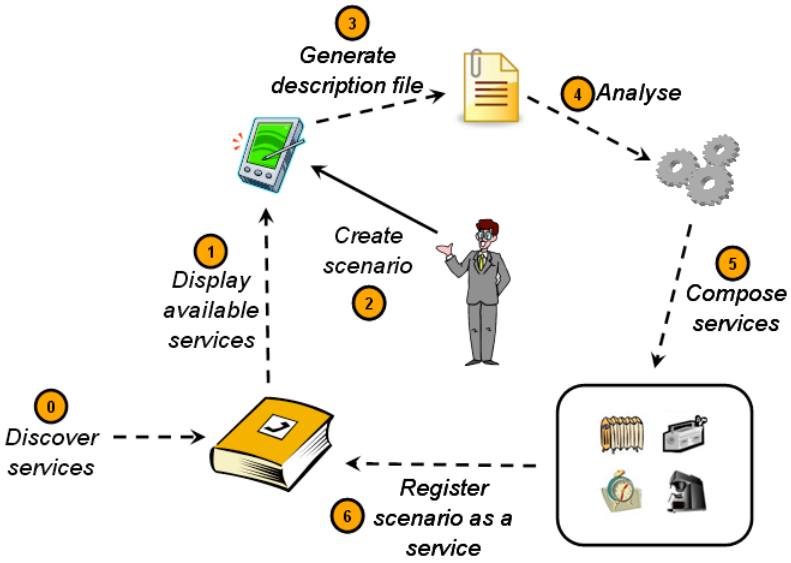

Figure 2: Overview of the proposed SaS scenario creation and reuse cycle

All these steps define an user-centric cycle, illustrated by Figure 2:

0. The system (placed into a user device) discovers the services available in its neighboring environment.

1. SaS classifies services depending on their providers (bundles) and services (interface names). It then displays them.

2. Users can compose several available services to create a scenario. This is possible through a dynamically adaptive graphical user interface.

3. The created scenario is translated into an XML descriptor file. It then becomes easily transmissible and can be shared with other platforms and users.

4. Next, SaS analyzes the scenario. It extracts information about the different services involved and how they are composed.

5. The system creates a bundle with the involved services and a generated *manager*. This *manager* handles the services according to the previously made user choices.

6. Finally, the *manager*, which is in the composite, registers the scenario as a new service into the environment. It becomes accessible from other devices and shared among end-users. Moreover, it can be composed into a new scenario.

4. Scenario creation

This section describes the main part of our system: scenario creation. It consists in three parts: service selection, scenario construction by service composition and scenario deployment.

As exposed in section 3, the first functionality of the system is discovery. Some protocols already exist that do so (eg. UPnP [14], SLP [15], Jini [16]) along with different extra functionalities. To be as interoperable as possible, our SaS system does not prescribe the use of a particular discovery protocol. SaS uses the API of distributed OSGi instead, which can be implemented by any of these protocols. Once the available services are discovered, they can be listed and ordered by SaS to enable service selection.

4.1. Service Selection

Every service proposes one or more operations (for example, the service *light* can offer two operations: `getValue` and `setValue`). To define a scenario, users always select operations but the name of the service and the identity of the provider device does not always matter. For example, to print a document a person can choose his favorite printer, have access to the specified service and select the appropriate operation. Alternatively, he can directly select the `print` method, no matter which printer provides it. So we need to enable users to select operations depending on the provider device or on the service name or else, just by the operation name.

Therefore we present the available services by three different ways: by device type (*eg.* the list of available printers), by service name (*eg.* the *printService* service) or directly by operation name (*eg.* the `print` operation). If users select a device, services provided by this device are then proposed to choose from. If users select a service, operations that compose this service are then proposed to choose from. Moreover, some different devices can propose identical services (*ie.* implement the same interface), sometimes with additional operations. This is why SaS groups these services and displays all the available operations (even if they are not provided by every device that proposes this service). When users choose an operation, the system automatically selects the device which proposes that service and provides that operation.

4.2. Service Composition

With the available service list and their operations, users can compose several operations to create a scenario. This is possible using an appropriate graphical interface based on our ADL.

4.2.1. Presentation of the ADL

This ADL is inspired by BPEL (Business Process Execution Language for Web Services) [17] and implemented in XML. BPEL is a standard for composition in Web Services. We base our ADL on this standard because it is powerful and well designed; however, it is too complex for our needs and hardly understandable for end-users. Moreover, because of their complexity, BPEL engines are too heavy for small devices. This is why we propose a new ADL, simple and tailored to scenario creation.

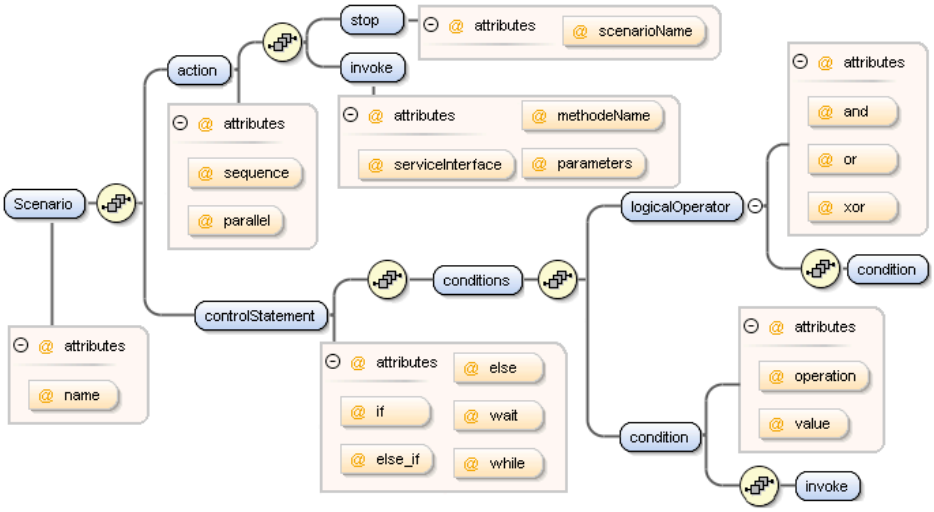

Figure 3: ADL for scenarios

We define a scenario as a succession of *actions*. An *action* can be an invocation of a service operation or a stop of a scenario. Invocation of a service operation includes *serviceInterface*, *operationName*, *parameters* and *resultValue* if needed. Parameters have a name, a type and a value. Users can directly choose to set this value or define it as the result of another operation. To combine these *actions*, the system proposes five *control statements*: `if`, `else_if`, `else`, `wait`, `while`. Except for `else`, all *control statements* include *conditions*. *Conditions* are defined by one or

more *conditions* separated by *logical operators* (`and, or, xor`). A *condition* is the evaluation of a service operation result. Evaluation is done with an operator ($=, <, >, \leq, \geq$) and a value to compare to. Figure 3 illustrates the different elements of our ADL.

4.2.2. The graphical user interface

Even if this ADL is simple and declarative, SaS must propose a more user-friendly view than the XML representation. This is why SaS provides a graphical representation of the ADL. It represents service operations by little boxes with inputs for parameters and outputs for results. Users do not manipulate the ADL directly but compose service operations with basic instructions (based on *actions* and *control statements* of our ADL): `do, stop, when, if/else_if/else, and, or, xor` and operators: `+, -, *, /`. For parameters entries, users can binf outputs to inputs or choose specific values. In addition, users can store a result value for future use.

Thanks to a transformation process, the scenario is then automatically transcribed in our ADL. Figure 4 shows the scenario transcription with a simplified version of the GUI. This schema does not represent the different ways to select services (by device, service name or operations). With the list of available services, users compose some of them separated by specific words. Then, it provides values for parameters. When the description is finished, SaS transcribes the scenario into an XML description file.

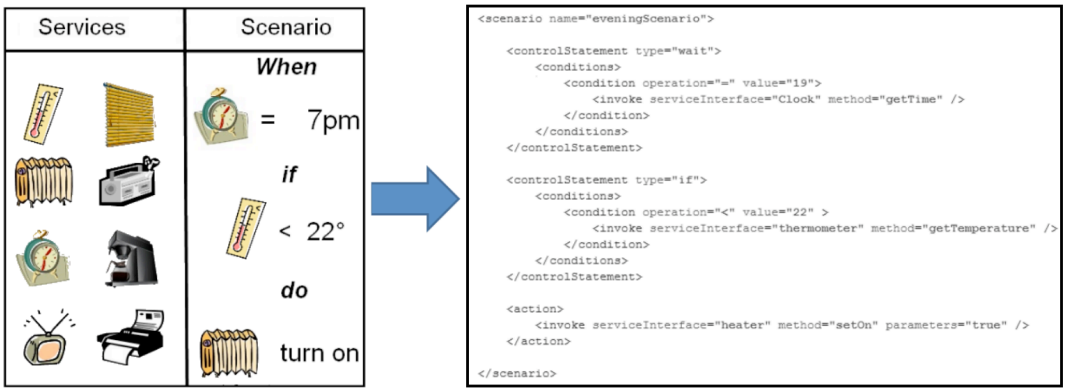

Figure 4: Scenario Transcription: from the GUI to the XML description file

Even if this solution is practical, the definition of a logical structure is not accessible for all end-users. They need an easier mechanism to create scenarios. This is why we also propose some already defined scenario templates for which users only have to select involved services. In addition, users can create their own predefined scenario templates and share them. The logical structure of a scenario template can also be extracted from a previously created scenario. Users only have to change services or modify parameter values to customize it.

4.3. Scenario deployment

Once the scenario is defined, the system analyses it to create a composite that manages its deployment. This composite includes services chosen in the scenario and a *Scenario Manager*.

There are two ways to include services into the composite depending on users choices. If a service has been chosen regarding its provider, the system instantiates the service from the specific component. Otherwise, the service is instantiated from any of its available provider components (an optimized selection scheme is a perspective). If the provider disappears, the system will instantiate another implementation of the same service (if available).

In addition to the involved services, the system adds a *Scenario Manager* to the composite. This manager has two roles: manage the different services and export the scenario as a new service in the system. SaS creates it by analyzing the scenario description so it represents the scenario inside the composite. Once created, the *Scenario Manager* registers the scenario as a new service with three operations: `start`, `stop` and `getDescriptor`. The scenario can then be used as a service and, as such, composed into a new coarser grained scenario. The new service

does not describe the functioning of the scenario: it hides services and their interactions inside the composite. This guarantees encapsulation. Reflexion is nonetheless provided (composites are not black boxes but grey boxes) as the `getDescriptor` operation provides access to the scenario description file. With this file, the user GUI can display a visual transcription of the scenario.

Figure 5 illustrates scenario export with an example. The scenario is simple and placed in a home automation environment: at 6pm, close the main door and set the thermostat at 7. There are three services: one is defined from a specific device (the main door), one is an event (the clock) and the other does not depend on a specific device (the thermostat). SaS analyses the created scenario and recognizes that a service is specific to a given device (the main door service). It instantiates the service provided by the bundle of this specific device. SaS creates a *Scenario Manager* with the information that is needed to handle the involved services. The scenario is then registered as a service. When this service is called, the scenario starts. The *Scenario Manager* waits for the event. Once the desired event occurs, the other services are activated.

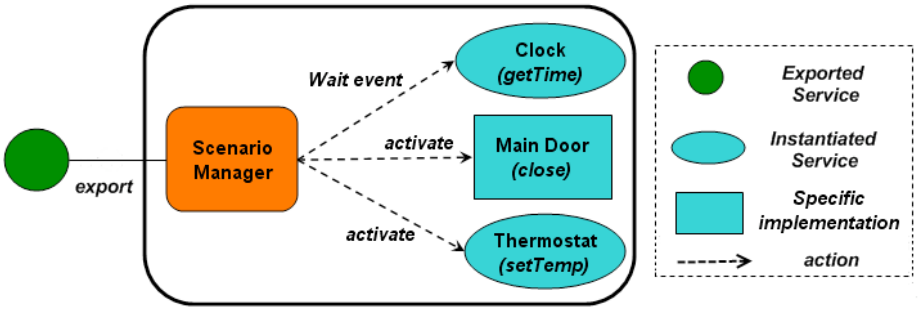

Figure 5: Scenario Export

If a service provider disappears, SaS automatically tries to found another service that corresponds to the same needs (implements the same interface with the same parameters) and instantiates it into the composite. Otherwise, it displays a message to the user and the scenario is automatically disabled until an appropriate service appears.

5. Prototype

This section describes our prototype implemented in Java over OSGi with iPOJO. This first version is not distributed and still under development.

5.1. Model

Three components compose SaS . Each of them is packed in its own bundle: it is safer and easier to update. We can see these components over the OSGi platform in Figure 6.

- Service Listener

  This bundle obtains the available service list from the OSGi context. It orders this list depending on providers and services. During the functioning, it listens to different events (disappearance or appearance of a service) to dynamically update the list.

- GUI

  The GUI is platform and operating system specific (PDA, mobile phone, Android, iOS,...). It provides a graphical adaptation of our ADL. It enables users to see the available services and compose them.

- Scenario Creator

  It creates scenario bundles which are composed of the selected services and a scenario manager. This latter manages services inside the composite and exports the scenario as a service.

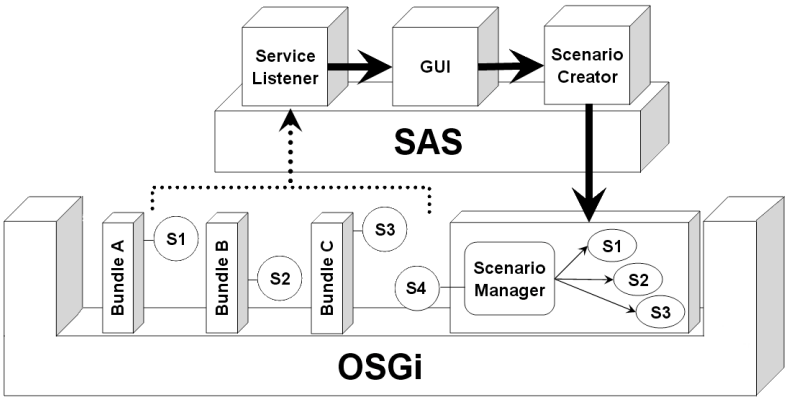

Figure 6: SaS system over OSGi

5.2. Implementation of the different system functions

In this subsection, we go deeply into technical aspects of each system's functionality.

5.2.1. Service selection

SaS orders the available services before presenting them to users. This is done by the *ServiceListener*. It first retrieves a list of the available services from the directory provided by the OSGi framework. Then, it registers as a *ServiceEventListener* to this directory. Thus, it is informed of any appearances, disappearances or service changes and can maintain an up-to-date list of available services. This list gives access by introspection to the provider devices and to the operations provided by the services. Finally, SaS classifies all services to order fast and easy ways of selection (*eg.* `getServicesByDevice, getOperationsByService`). With this, we enable users to directly select an operation or to retrieve it from a specific device or service.

5.2.2. Service composition

The GUI displays available devices, services and operations in three columns. If a specific device is selected, SaS reduces the list of services (*eg.* to the ones provided by this device). Then by selecting a service, SaS does the same with the provided operations.

At this point, users can select the operations they want to compose. If they previously have selected a device or a service, the GUI asks if they want this specific operation or any implementation of it. The selected operation appears in the GUI as a box with inputs for parameters and an output for the result (if it is relevant). Users can select an operation result as the parameter of another operation by connecting inputs and outputs. The system automatically detects if the types are compatible. If not, it refuses the connection. In addition, the GUI provides a toolbox with basic instructions (`do, stop, when, if/else_if/else, and, or, xor`) and operators (`+, -, *, /`) to define logical sequences of actions.

5.2.3. Scenario transcription

The scenario is shared among users by exporting it as a service. However, users might want to retrieve the description of the scenario. This is why SaS transcripts the user defined scenario into an XML file. To do this, it needs to first extract the different blocks that compose the scenario by analyzing user instructions. A block is a succession of operation invocations inside a *control statement* (`when, if, else_if, else, do, stop`). A block can also be composed of sub-blocks. Then, inside blocks, the system structures the different operations with *logical operators* (`and, or, xor`). Scenario templates are derived from this logical structure. By giving appropriate names to operations, SaS can bind operations that have as parameters the result of another one. Finally, the XML description file is created and transmitted to the *Scenario Creator*.

5.2.4. Scenario Manager generation

Inside the composite, the *Scenario Manager* manages the different services as defined by the scenario. First, the scenario is represented as a succession of Java instructions. Then, a new class that integrates these instructions is created.

The Scenario Creator parses the XML description file (to generate the Java instructions) with *SAX* (Simple API for XML) [18]. *SAX* is better than DOM (Document Object Model) in this case because we need to follow the structure of the scenario. Depending on XML tags, SAX immediately translates XML elements into Java instructions. The result is a sequence of Java instructions which is the exact interpretation of the scenario definition.

Then, SaS uses Javassist [19] to generate the *Scenario Manager* class. Javassist is a Java library for editing byte codes. Javassist dynamically creates classes at runtime. It also modifies existing class files. We use it because it is simpler than ASM [20] and provides a source level API to easily generate a class file with attributes and methods.

SaS generates a *Scenario Manager* class implementing the *ScenarioManagerInterface* interface. This interface is packaged into the bundle and declares three public methods: `start, stop` and `getDescriptor`. In OSGi, services need to implement an interface to be exported. Thus, SaS can recognize scenarios (in comparison to other basic services) because they all implement the same interface.

A generated *Scenario Manager* class has one attribute per service involved in its scenario. By dependency injection in these attributes, SaS (thanks to iPOJO) is able to automatically handle service dependencies and instantiate the right service implementation. SaS also generates the three methods declared in the *ScenarioManagerInterface* interface. The `start` method is the method that launches the scenario. The `start` method body is filled with the Java instructions resulting from the SAX parsing. The `stop` method is the method that stops a scenario when it is running. The last method, `getDescriptor`, provides the XML description file.

5.2.5. Scenario deployment

Once the *Scenario Manager* class is created, SaS can create the composite that packs together the elements that implement the scenario. First, it declares the *Scenario Manager* class as a *primitive component* with the iPOJO API [21]. It specifies service dependencies for the *Scenario Manager* and declares the scenario management service it provides. The three methods (`start, stop` and `getDescriptor`) are the operations of this service. Then, the scenario creator declares a new composite composed of the *Scenario Manager* and of services instances. Depending on users' choices, the services are instantiated from a specific device or not. Finally, the composite bundle is installed and started into the OSGi platform. The scenario is automatically registered as a service by the *Scenario Manager* into the OSGi directory.

6. Related Work

Service composition is an important part of Service Oriented Architecture (SOA) and Web Services (WS), are the most popular SOA. BPEL4WS (Business Process Execution Language for Web Services) [17] is the most widespread standard for WS composition. It is complete and enable the specification of highly elaborated service composition; but it is complex and not appropriated for end-users. In addition, WS are numerous and widely used but are not targeted at small devices and are therefore not appropriated due to high resource consumption and bad performances. Moreover, the OSGi platform provides the possibility to deploy, start and stop components on the fly without altering the whole system. This is why we decided to propose therefore an ADL inspired from BPEL (but simpler) to enable service composition in OSGi for end-users.

We can class ubiquitous systems that respond to user needs by providing service composition into three categories. Some systems:

- Analyze users behavior to define his needs and anticipate future actions.

  With MavHome [22], Cook *et al.* enable to specify some activities in the house that the system tries to automate. The system is based on a learning algorithm which analyzes the inhabitant behavior and adapts to user response. With this system, users do not control the system. They can just react to the system actions. Moreover, they need to wait for the system to analyze a significant number of users behavior to have it make a decision (that can be wrong).

- Automatically compose some available services to respond to users concrete objective ([23, 24, 25]).

  This is the approach of Julian Lancia in his thesis [26]. He proposes a system based on ontologies to define applications as sets of logical rules (with SWRL [27]: semantic web rule language) executed automatically by an inference engine. The proposed language enables to define system tasks abstractly by logical rules independently of the available services. The engine connects the rules and the services to compose some of them and execute tasks. This is an interesting system that adapts to environment changes, but it is not destined to end-users (the language is too complex) and does not enable to export the service composition as a new service.

  This approach is better for users to specify their need but does not provide them a direct access to available services. In addition, the system depends on service semantics that might be irrelevant and can misinterpreted.

- Provide a programmable system which adapts to the environment ([28]).

  In a previous work, SAASHA [29, 30], we focused on service composition for home automation. That is why we based our work on UPnP over OSGi. The idea was to adapt the system by creating a new component for each new device. In addition SAASHA proposed the user to compose scenarios with Event - Conditions - Action rules. In this paper users can define more complex scenarios and export them as new services (hierarchical scenario composition).

In conclusion, we could say that (except for SAASHA) all these works propose programming tools for developers. Our approach is destined to end-users, based on a high level, non technical and declarative language. We share with SAASHA the same user orientation, however the language proposed in this paper is a process language specially thought for service composition. This composition model is used to enable users to create new scenarios and share them as services among users. The idea of hierarchical composition is based on scenario reuse and leads to scenario mobility (scenario use in different context and with different devices). Thus, the scenario implementation into components assembled over the existing architecture combines flexibility and dynamicity.

7. Conclusion

In this paper we propose the SaS system. This system allows end-users to easily create, use and share scenarios. A scenario is a composition of services provided by multiple devices. End-users of SaS use a GUI to compose services. Service composition is internally based on a new ADL. Each created scenario can be registered as a regular service and then involved in another scenario. We also described the implementation of a prototype of SaS in Java over OSGi and iPOJO.

We have two major perspectives. On the one hand, we want to evaluate SaS to show its simplicity for non technical end-users. Our idea is to compare SaS with existing tools such as Yahoo Pipes [31], Automator [32], Scratch [33]. Such tools enable non-technical end-users to graphically develop small applications by composing elements. On the other hand, we plan to define some recovery strategies to maintain scenario continuity in case of service disappearance. These strategies will try to anticipate service loss and recover it afterwards.

Acknowledgements

This work is supported by a grant from 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 (1995) 78–89.
URL `http://wiki.daimi.au.dk/pca/\_files/weiser-orig.pdf`

[2] H. Schulzrinne, X. Wu, S. Sidiroglou, S, Ubiquitous computing in home networks, IEEE Communications (2003) 128–135.
URL `http://ieeexplore.ieee.org/xpls/abs\_all.jsp?arnumber=1244933`

[3] OASIS, Reference Model for Service Oriented Architecture 1.0 (oct 2006).
URL `http://docs.oasis-open.org/soa-rm/v1.0/soa-rm.html`

[4] M. P. Papazoglou, Service-Oriented Computing : Concepts , Characteristics and Directions, in: Proceedings of the 4th International Conference on Web Information Systems Engineering, IEEE Computer Society, 2003, pp. 3–12.
[5] A. Bottaro, A. Gérodolle, P. Lalanda, Pervasive service composition in the home network, 2007, pp. 596–603.
URL http://www.computer.org/portal/web/csdl/doi/10.1109/AINA.2007.112
[6] C.-L. Wu, C.-F. Liao, L.-C. Fu, Service-Oriented Smart-Home Architecture Based on OSGi and Mobile-Agent Technology, IEEE Transactions on Systems, Man and Cybernetics, Part C (Applications and Reviews) (2007) 193–205.
URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4106037
[7] D. Valtchev, I. Frankov, Service gateway architecture for a smart home, Communications Magazine, IEEE (2002) 126–132.
URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=995862
[8] E. Aarts, B. de Ruyter, New research perspectives on Ambient Intelligence, Journal of Ambient Intelligence and Smart Environments 1 (2009) 5–14.
[9] OSGi Alliance, OSGi Service Platform Core Specification Release 4 (2005).
URL http://www.osgi.org/download/r4v40/r4.core.pdf
[10] OSGi Alliance, OSGi Service Platform Enterprise Specification (march 2010).
URL http://www.osgi.org/download/r4v42/r4.enterprise.pdf
[11] C. Escoffier, R. Hall, Dynamically adaptable applications with iPOJO service components, Software Composition (2007) 113–128.
URL http://www.springerlink.com/index/9r3u86q510178084.pdf
[12] H. Cervantes, R. Hall, Autonomous adaptation to dynamic availability using a service-oriented component model, IEEE, 2004, pp. 614–623.
URL http://www.computer.org/portal/web/csdl/doi/10.1109/ICSE.2004.1317483
[13] Sun Microsystems, Enterprise javabeans specifications (may 2006).
URL http://java.sun.com/products/ejb/docs.html
[14] UPnP Forum, Understanding UPnP: A White Paper (2000).
URL http://www.upnp.org/download/UPNP_UnderstandingUPNP.doc
[15] C. Bettstetter, C. Renner, A comparison of service discovery protocols and implementation of the service location protocol, in: Proceedings of the 6th EUNICE Open European Summer School: Innovative Internet Applications, Citeseer, 2000, pp. 13–15.
URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.115.4652\&rep=rep1\&type=pdf
[16] G. Aschemann, R. Kehr, A. Zeidler, A Jini-based Gateway Architecture for Mobile Devices, In Proceedings of the Java-Informations-Tage (JIT99) (1999) 203–212.
[17] OASIS, Web services business process execution language version 2.0 (april 2007).
URL http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf
[18] S. Means, M. A. Bodie, Book of SAX: The Simple API for XML, No Starch Press, 2002.
[19] S. Chiba, M. Nishizawa, An Easy-to-Use Toolkit for Efficient Java Bytecode Translators, Proceedings of the 2nd international conference on Generative programming and component engineering (2003) 364–376.
[20] E. Bruneton, R. Lenglet, T. Coupaye, ASM: a code manipulation tool to implement adaptable systems, 2002.
URL http://asm.ow2.org/current/asm-eng.pdf
[21] Apache Foundation, ipojo api (2010).
URL http://felix.apache.org/site/apache-felix-ipojo-api.html
[22] D. Cook, M. Huber, K. Gopalratnam, Learning to control a smart home environment, in: Innovative Applications of Artificial Intelligence, Citeseer, 2003.
URL http://www.eecs.wsu.edu/~holder/courses/cse6362/pubs/Cook03.pdf
[23] T. Heider, T. Kirste, Multimodal appliance cooperation based on explicit goals: concepts & potentials, in: Proceedings of the 2005 joint conference on Smart objects and ambient intelligence: innovative context-aware services: usages and technologies, ACM, 2005, pp. 271–276.
URL http://portal.acm.org/citation.cfm?id=1107614
[24] A. Bottaro, J. Bourcier, C. Escoffier, P. Lalanda, Context-Aware Service Composition in a Home Control Gateway, IEEE International Conference on Pervasive Services (2007) 223–231.
URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4283920
[25] Y. Charif, N. Sabouret, Dynamic Service Composition and Selection through an Agent Interaction Protocol, 2006, pp. 105–108.
URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.76.3353\&rep=rep1\&type=pdf
[26] J. Lancia, Infrastructure orientée service pour le développement d'applications ubiquitaires, Ph.D. thesis, Université Bordeaux 1 (2008).
[27] I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, M. D. Benjamin Grosof, Swrl: A semantic web rule language combining owl and ruleml, W3C (May 2004).
URL http://www.w3.org/Submission/SWRL/
[28] V. Hourdin, J. Tigli, S. Lavirotte, G. Rey, M. Riveill, SLCA, composite services for ubiquitous computing, ACM Press, New York, New York, USA, 2008, pp. 1–8.
URL http://portal.acm.org/citation.cfm?id=1506284
[29] F. Hamoui, M. Huchard, C. Urtado, S. Vauttier, Specification of a component-based domotic system to support user-defined scenarios, in: Proceedings of 21st International Conference on Software Engineering and Knowledge Engineering (SEKE 2009), 2009.
URL http://hal-lirmm.ccsd.cnrs.fr/lirmm-00415722/
[30] F. Hamoui, M. Huchard, C. Urtado, S. Vauttier, Un système d'agents à base de composants pour les environnements domotiques, in: Actes de la 16ème conférence francophone sur les Langages et Modèles à Objets (LMO 2010), 2010, pp. 35–49.
[31] Yahoo, Rewire the Web.
URL http://pipes.yahoo.com/pipes
[32] Apple, Automator: Your personal Automation Assistant.
URL http://www.macosxautomation.com/automator

[33] M. Resnick, J. Maloney, A. Monroy-Hernandez, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, Y. Kafai, Scratch: Programming for Everyone, Communications of the ACM 52 (11) (2009) 60–67.
URL http://web.media.mit.edu/~mres/papers/Scratch-CACM-final.pdf