User-defined scenarios in ubiquitous environments: creation, execution control and sharing

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

▶ To cite this version:

Matthieu Faure, Luc Fabresse, Marianne Huchard, Christelle Urtado, Sylvain Vauttier. User-defined scenarios in ubiquitous environments: creation, execution control and sharing. 23rd International Conference on Software Engineering and Knowledge Engineering (SEKE 2011), Jul 2011, Miami Beach, Fl, United States. pp.302-307. hal-00618264

HAL Id: hal-00618264
https://hal.science/hal-00618264

Submitted on 4 Jun 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

User-defined Scenarios in Ubiquitous Environments: Creation, Execution Control and Sharing

Matthieu Faure, Luc Fabresse
Ecole des Mines de Douai, Douai, France
{Matthieu.Faure, Luc.Fabresse}@mines-douai.fr

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

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

***Abstract*—Ubiquitous computing provides a dynamic access to different functionalities of networkable electronic devices. Whereas basic services have limited use, predefined complex services cannot encompass every end-user's needs nor be adapted to a set of services that are dynamically discovered in an open environment. Alternatively, users need to be provided with means to express their requirements, choosing precisely which services to compose into a scenario of their own. In service-oriented computing, some systems propose mechanisms to develop tailored components that provide composite services; however they are not adapted to end-users, have limited composition capabilities and/or do not consider several characteristics of ubiquitous environments (such as multiple users and devices).**

This paper presents a novel user-centric system called SaS for mobile personal devices. SaS provides end-users with an easy access to services and a simple GUI to combine them into complex scenarios. A new architectural description language is used to specifically support scenario creation by service composition. Scenario may be shared among users and devices. SaS offers scenario execution control for example to start and stop it but also to query the current state of a scenario. In addition, SaS proposes some mechanisms to maintain scenario availability in case of service/device unavailability. SaS is currently implemented in a proof-of-concept prototype on top of OSGi.

Keywords*—*Ubiquitous computing, service-oriented computing, user-centric system, service composition, scenario creation.

I. 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. As shown in Figure 1, the more complex user requirements can only be satisfied by compositions of multiple services provided by multiple devices. Different service composition means have been studied and proposed [8], [9], [10], [11]. However, they are not designed for end-users without technical knowledge.

Enabling end-users to describe their own *scenarios* is a first improvement and a step towards ambient intelligence [12]. In addition, users should easily manage the created scenarios and have access to them from several control devices (PDA, mobile phone or laptop). Moreover, ubiquitous environments imply that several users might be eager to share scenarios. Scenarios should therefore be exported in the environment to be shared and reused.

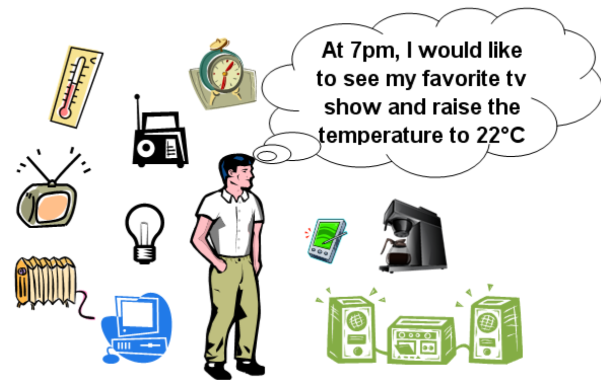

Figure 1. User's main issue

In this paper, we propose the SaS (Scenarios as Services) system specifically designed for end-users to describe, use and share *scenarios* in a high level manner. SaS comprises a new architecture description language (ADL) [13], [14], [15] dedicated to user scenario creation. SaS also integrates a graphical user interface (GUI) based on this ADL. This GUI presents a classified and filtered view of the services available from a set of widely spread devices and provides tools to easily *compose* selected services. SaS integrates scenarios as *regular services*. This enables easy scenario execution control, scenario sharing among users and hierarchical composition of scenarios. The SaS system has been prototyped. Its current implementation is on top of OSGi [16].

The remainder of this paper is structured as follows. Section II introduces the context of this work, presents the requirements for open and distributed environments and discusses the state of the art. Section III presents the first part of our proposition: scenario creation. Section IV is dedicated to scenario execution control and sharing. Section V presents the architecture of the prototype implementation. Finally, section VI evaluates our proposition, concludes and draws some perspectives to this work.

II. User-Centric Systems in Ubiquitous Environments

This section first describes the terminology of ubiquitous environments and especially that of user-centric systems. It then presents the requirements that are mandatory for that kind of systems. It finally compares some of the main state of the art approaches regarding these requirements.

A. Terminology

Ubiquitous systems involve multiple *users* and multiple *devices* that each provide a set of *services*. A device is an electronic object (such as a clock). Devices publish services (such as `Time`). Each service provides one or more *operations* (such as `getTime` or `setTime`). They are called “capabilities” in the SOA norm [3] by the OASIS consortium. End-users use these operations as an access to functionalities of devices.

Devices can interoperate but the overall goal that the system has to achieve always comes from users. Users can be simple consumers or technical experts that command devices, their needs can always be considered as *scenarios* which are combinations of operations. However, we choose to name this combination a *service composition* because services are not stand-alone elements and to stick to the terminology used in SOC.

A SaS system is deployed on a mobile personal device. The system and the device on which it is deployed together define a SaS *platform*. A SaS platform participate in one or more networks which constitute the platform’s environment. The global environment is the union of all the environment of its constituting SaS platforms.

B. Requirements for user-centric systems

User needs always constitute a scenario. User-centric system must therefore enable *scenario creation*. As seen in Figure 1, user scenarios are not always simple service aggregations but can imply conditions, control statements and logical operators. Users should thus be able to *compose services* according to their needs. Most of the users are not technical experts. Scenario creation should therefore be *user-friendly* and adapted to devices. Ubiquitous environments imply multiple users and devices. So, created scenarios should be available into the environment and *shared* among users. Users must be able to easily start and stop created scenarios and check scenario status and execution advancement. Thus, the system should control the *scenario life-cycle*. Moreover, already created scenarios should be easily modified and *recomposed* into other ones. In addition, devices that provide services and/or scenarios can disappear. The system must therefore *maintain* scenario execution and availability in case of device disappearance.

C. State of the art

With the requirements established previously, we can analyze some of the main systems that provide a solution for ubiquitous environments and enable end-users to create scenarios.

- **SLCA** [17] provides developers with the capability to compose web services. A composite service contains proxy components attached to involved web services. SLCA enables hierarchical service composition. In addition, it is an event-based system which adapts to environment changes. In case of service unavailability the composite service replaces it if an appropriate service is found. If not, the composite service removes the proxy component attached to this service.
- **MASML** [6] is a multi-agent system for home automation. Scenarios are defined with an XML syntax and consist of a sequence of service operation invocations. MASML XML documents can embed ECMA scripts [18] to add logic elements. A mobile agent is in charge of scenario execution. It is moving to each appropriate device with the scenario description file to execute it. This enables scenario advancement tracking but not parallel execution.
- **SODAPOP** [19] proposes an innovative approach based on the same observation than us: user needs are scenarios. What is important is the goal to achieve. The main hypothesis is that each service contains informations about its initial conditions and its effects. SODAPOP automatically classifies new services with these informations. Then, it can combine some of them to reach the user’s goal.
- **SASHAA** [20], [21] 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. It is an adaptive system which creates a new component for each new device.

Table I compares these systems with respect to the requirements that we identified in II-B. Symbol √ means that the requirements is fulfilled, - signifies that it is partially accomplished and X represents an absence of solution.

Systems	Scenario Creation	Advanced Service Composition	User firendliness	Scenario sharing	Scenario Lifecycle	Hierarchical Composition	Scenario Maintenance
SASHAA	√	-	√	X	-	X	√
SLCA	√	√	X	X	X	√	-
MASML	√	√	X	X	-	X	√
SODAPOP	√	X	-	X	X	X	X

Table I
SYSTEM COMPARISON WITH OUR REQUIREMENTS

Except for SAASHA, we can notice in Figure I that all these works propose programming tools for developers. They are not directed to end-users. In addition, scenario sharing is never took into consideration. For scenario life-cycle control, SASHAA only enables to start and stop scenarios whereas MASML just allows users to check scenario advancement. Because of this, we decided to propose a new system which best meets all the expectations of user-centric systems for ubiquitous environments.

III. The SaS System: Scenario Creation

A. Overview of SaS

The purpose of SaS which stands for *Scenarios As Services* is threefold:

1) help end-users create scenarios by service composition,
2) monitor scenario execution on a given platform,
3) export scenarios into the environment for future use or sharing.

To do so, several steps are necessary that define a user-centric cycle, as illustrated by Figure 2:

0. The system (placed into a user device) discovers the services available in its neighboring environment.
1. SaS classifies service operations depending on their providers (devices) and services. 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 based on our ADL.
3. The created scenario is translated into a descriptor file. It therefore becomes easily transmissible and can be shared with other platforms and users.
4. Next, SaS analyzes the scenario descriptor. It extracts information about the different services involved and how they are composed.
5. The system creates a composite with the involved services and a generated *manager*. This manager handles the services according to the previously made user choices.

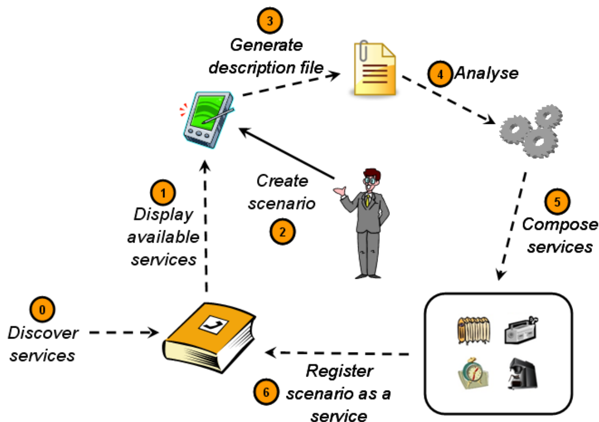

Figure 2. Overview of the proposed SaS scenario creation and reuse cycle

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.

B. Scenario creation

This section describes scenario creation, which is the main part of our system. It consists in three steps: service selection, scenario construction by service composition and scenario export. As described in Section III-A, the first functionality of the SaS system is service discovery. Some protocols already exist that do so (*e.g.* UPnP [22], SLP [23], Jini [24]) along with different extra functionalities. To be as interoperable as possible, our SaS system does not prescribe the use of a particular discovery protocol. Once the available services are discovered, they can be listed and ordered by SaS to enable service selection.

1) Service Selection: Every service proposes one or more operations (for example, the *light* service might 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 do not always matter. For example, to print a document a person generally chooses his favorite printer, accesses the specified service and selects the appropriate operation. Alternatively, if he needs to know what time it is, he directly selects the `getTime` operation, no matter which clock or service provides it.

Service selection in SaS sticks to this requirement. SaS proposes three filtered views to select available operations: by device type (*e.g.* the list of available printers), by service name (*e.g.* the *printService* service) or directly by operation name (*e.g.* 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, distinct devices can propose services with the same name, sometimes with additional operations. SaS groups these services together and displays all the available operations collectively.

2) Service Composition: SaS enables service composition thanks to an ADL and its GUI. Depending on user choices, created scenarios can be then exported into the environment.

a) Presentation of the ADL: In order to help end-users create scenarios that correspond to their needs, we propose a new ADL. It is simple and tailored to scenario creation. Compared to other programming languages for service composition (like BPEL [25]) which are imperative and designed for executable process, our ADL is a high level language, declarative and destined to end-users. With this ADL, one can declare both services and scenarios.

- Service declaration
  We define a service by a device (its provider), a name and an operation list. This list cannot be empty. Operations have a return type (which can be *void*) and can have typed parameters. We represent only the main elements of the grammar in Listing 1.

```
<service> ::= service <device> <service_name> <op_list>

<op_list> ::= ( <operation> ; )*

<operation> ::= operation <operation_name> (
   [<parameter_list>] ) : <return_type>

<parameter_list> ::= <parameter_type> (,<parameter_type>)*

<return_type>::= <type>
<parameter_type> ::= <type>
```

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

- Scenario declaration
  By definition, a scenario has a name and an action list. An action can be:
  - **an operation invocation**: a service operation is invoked, with its parameter values. Users can directly enter parameter values or invoke another service operation to create the desired value (operation composition). SaS checks if parameter types conform to the service definition.
  - **an alternative (*if - else*)**: conditions compare the result of two service operations or the result of a service operation and a value chosen by the user.
  - **a repetition loop**: enables `while` loops iterations while a condition remains satisfied. Alternatively, it is possible to precise how many `times` a series of actions should be invoked.

Listing 2 describes the main elements of a scenario declaration using the BNF notation.

```
<scenario> ::= scenario <scenario_name> <action_list>

<action_list> ::= { <action> + }
<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_cond><action_list> [<else_clause>]
<else_clause> ::= (else <action_list> )*
<cplx_cond> ::= (<condition> (<log_operator><condition>)*)
<condition> ::= <op_invocation> <comp_operator>
                (<op_invocation> | <compare_to_value>)
<repeat> ::= (while <cplx_cond> | <repeat_value> times)
            <action_list>

<log_operator> ::= and|or|not
<comp_operator> ::= < | <= | > | >= | ==
```

Listing 2. Grammar of the scenario declaration using the BNF notation

b) The graphical user interface: This ADL syntax is simple and declarative as we can see in the example on the right of Figure 3. Nevertheless, SaS proposes a more user-friendly option to create scenarios through a graphical representation of the ADL. Users therefore do not manipulate the ADL anymore but compose service operations with basic instructions (based on the *operators* of our ADL): `if, else, while, times, and, or, not,` `<, >,` $\leq$, $\geq$, `==`. For parameters entries, users can select an operation result or choose fixed values and apply an arithmetic operation (such as `+, -, *, /`).

Once the scenario is defined, users can choose to export it into the environment. They also have to specify if the scenario can be redeployed into another SaS platform. Thanks to a transformation process, the scenario is then automatically transcribed into our ADL. Figure 3 (right) shows the scenario transcription with a simplified version of the GUI (left).

3) Composite service creation: After the scenario is created, SaS analyzes its description file to create a composite that manages

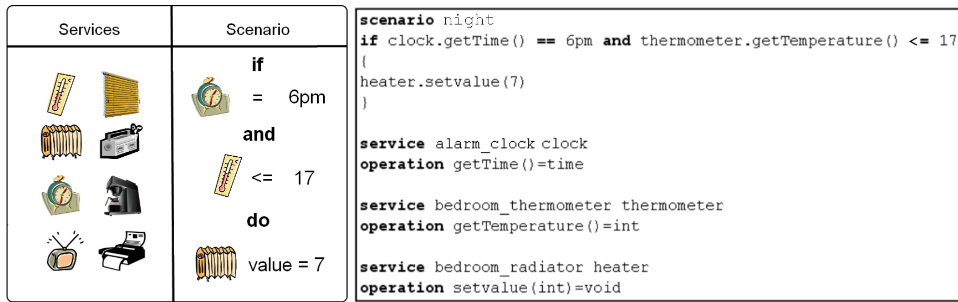

Figure 3. Scenario Transcription: from our ADL

its deployment. This composite includes references to the services chosen in the scenario and a *Scenario Manager*. The manager has two roles: manage the different services and export the scenario as a new service in the system according to users preferences.

Depending on user choices, services instantiated inside the composite are specific to a device or come from any of its available providers[1]. In this last case, if the service provider disappears, SaS dynamically recomposes the composite that implements the scenario to integrate another implementation of the same service (if available). Figure 4 illustrates composite services 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: only one is defined from a specific device (the main door). Others are instantiated from any devices that provide these services.

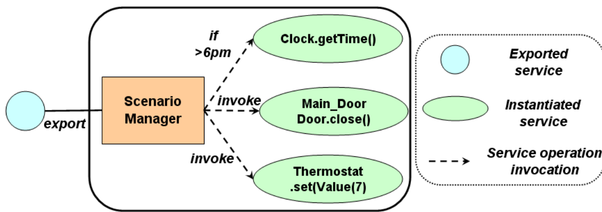

Figure 4. A Composite Service

IV. Scenario execution control and sharing

Once scenarios are created, they can be shared among users. In addition, scenarios are easily manageable and should stay available into the environment in case of service or device unavailability.

A. Scenario sharing

As seen in subsection III-B3 the Scenario Manager registers the scenario as a new service. The scenario can then be used as a service and, as such, composed into a new coarser grained scenario (scenario hierarchical composition). This service has four operations: `start`, `stop`, `getScenarioState` and `getDescriptor`. It 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 gray boxes) thanks to a service operation: `getDescriptor`. This operation provides access to the scenario descriptor file. Users can directly read this file or obtain a visual transcription of the scenario on his GUI.

Figure 5 illustrates scenario sharing. The user of the SaS platform named A creates and exports a scenario. This scenario is registered as a new service. It is then discovered by platforms B and C. The user of platform B recomposes this scenario into a new one, whereas the user of platform C just gets an overview of the scenario on its GUI.

[1]An optimized selection scheme is a perspective.

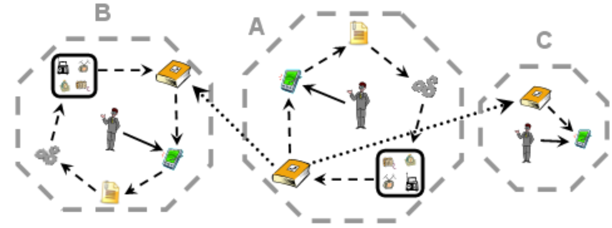

Figure 5. Overview of SaS scenario sharing

B. Scenario execution control

We define scenarios as service compositions. We can see a scenario as an active entity, which evolves, changes from one state to another. Moreover, the execution contains several steps which can fail or succeed. For example, users that discover a scenario might want to know if this scenario is currently in execution. If so, it is important to check which steps have been executed, which succeeded and what are the next steps. This is why, SaS manages scenarios' life-cycles and enables to check scenario execution status.

1) Scenario life-cycle: Scenarios are dynamic. They can be executed, stopped, have a missing service... The state diagram of Figure 6 illustrates the different states of scenario life-cycle which are:

- **Installed,** the scenario has been deployed and registered (and so discovered) as a new service. SaS automatically checks if the different involved services are present.
- **Ready to launch,** all involved services are available. If a service disappears, the scenario goes to the previous state.
- **Running,** the scenario has been launched and is currently executed. The scenario could finish and come back to the previous state or be interrupted.
- **Stopped,** the scenario has been stopped by a user or a service inside the scenario disappeared. The scenario is paused waiting to be restarted or to be executable again by the appearance of an appropriate service.

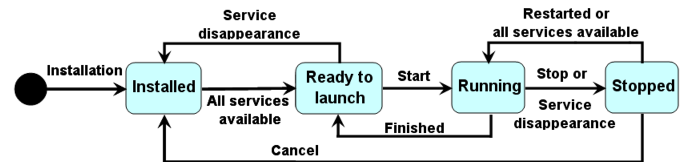

Figure 6. State diagram of scenario life-cycle

2) Scenario running state advancement: Users must know which scenario operations are running and which have already been executed. This is why, SaS registers scenario execution progress. To do so, SaS considers the *Running* state of the scenario life-cycle as

a succession of stages: the operation invocations. These stages are evaluated depending on their types:

- **Functions**, if an operation is invoked to obtain a result, SaS logs this operation as done when we obtain the result. If an error occurs, SaS continues to execute the scenario if possible (*i.e.* if the operation result is not needed) and displays a warning to the user.
- **Procedures**, if the operation does not return a result, SaS logs it as *executed* when the operation is invoked.

In addition, SaS logs the execution times of the different scenario steps. Users can see when the scenario began and how long every operation took. So, SaS enables users to check the current scenario position and control its correct advancement. Users can get these informations thanks to the `getScenarioState` operation.

C. Scenario availability

With scenario export as new services, users can have access to the same scenario on several devices, however, they might want access to it even if the original provider is off. This is why, scenario access should be maintained if the original provider disappears.

To do so, SaS enables scenario redeployment on other platforms. This is possible because SaS differentiates scenarios from available services. When a scenario appears, every SaS platform downloads the scenario description. Thus, users can directly have an overview of the scenario definition and platforms can redeploy the scenario if the original scenario provider disappears.

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 [16], [26] with iPOJO [27]. OSGi is a popular SOC framework that is widely adopted by industry for developers to create *bundles* (Java components). iPOJO is based on OSGi and follows the Service-Oriented Component Model [28]. The main idea is that a component should only contain business logic as in EJB 3.0 [29] (*EJB entities*); SOC mechanisms should seamlessly be handled by the component container as container-managed cross-cutting services.

A. Model

As shown in Figure 7, four components compose SaS. Each of them is packaged as an OSGi bundle because it is safer and easier to update.

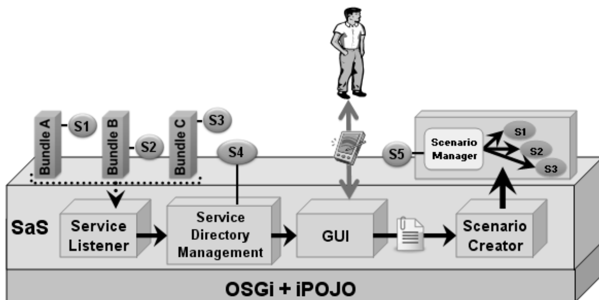

Figure 7. The SaS prototype implemented over OSGi and iPOJO

- **Service Listener**. This bundle obtains, orders and dynamically updates the list of available services from the OSGi context.
- **Service Directory Management (SDM)**. It is the intermediate between the *Service Listener* and the GUI responsible of managing (adding, removing) services and scenarios in the *Service Directory*.
- **GUI**. The GUI is platform and operating system specific (PDA, mobile phone, Android, iOS,...). It provides a graphical representation of our ADL which provides users with the capability 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.

B. Insights into the SaS prototype

This subsection presents the implementation of the main functionalities of the SaS system.

The latest version (4.2) of OSGI now supports distribution (RFC 119 [26]). So, for service discovery, SaS uses the API of distributed OSGi which can be implemented by many discovery protocols. Then, the *Service Listener* retrieves the list of available services from the directory provided by the OSGi framework and sends it to the *SDM*. This latter orders and classifies the service list. The GUI[2] displays available devices, services and operations. It filters the displayed services (*resp.* operations) if a specific device (*resp.* service) is selected. A user creates a scenario through the GUI which stores it in an XML-based description file. This description becomes easily transmissible and promptly interpretable and analyzable since XML is a standard as an exchange format. Using this description, the *Scenario Creator* of SaS automatically (i) generates a scenario manager which (ii) exports and manages the scenario. Finally, the scenario (iii) is redeployed on other SaS platforms according to user preferences. The remainder of this section describes more deeply this three important steps.

1) Scenario Manager generation: Starting from the XML-based description of a scenario, SaS generates a Java class that represents the manager of this scenario (*Scenario Manager*). To do so, the first step consists to parse the XML description file using a *SAX* parser [30]. SAX translates XML elements into a sequence of Java instructions. Then, SaS generates a *Scenario Manager* class with the Javassist [31] library that enables dynamic byte code edition such as creating classes or modifying existing classes. The *Scenario Manager* is generated as a class that implements the *ScenarioManagerInterface* interface. This interface declares four public methods including a `start` method which is automatically filled in the *Scenario Manager* class with the Java instructions resulting from the SAX parsing.

2) Scenario export and execution control: SaS uses the iPOJO API [32] to dynamically create an OSGi composite bundle that packs together the generated *Scenario Manager* and the involved services. This composite bundle is then installed and started into the OSGi platform. Finally, the *Scenario Manager* registers a new service inside the OSGi directory, specifying its capability to execute four public methods (`start, stop, getScenarioState` and `getDescriptor`).

For scenario execution control, the *Scenario Manager* creates a log file every time the `start` service operation is invoked with the invoker platform *id* and the current time. Then, the *Scenario Manager* logs in this file every service invocation success (or failure) with time. With this log file, SaS knows at every moment if the scenario is currently in execution, when it began, who launched it and which steps are already executed. These informations are available through the service operation `getScenarioState`.

3) Scenario automatic deployment: As seen in V-B1, all scenarios implement the same Java interface (*ScenarioManagerInterface*). So, SaS can easily recognize them. Thus, when a scenario is discovered as a new service, the *Service Directory Management* automatically gets the scenario description file thanks to the `getDescriptor` operation provided by the service. The scenario is not deployed again but SaS keeps the XML description file. *SDM* sends the scenario description to the GUI. If the original provider disappears, another SaS platform may redeploy the scenario if its directory contains all the involved services.

[2]which is still under development.

VI. Evaluation and Conclusion

With the SaS system, we propose a newly user-centric system that meets the expectations of ubiquitous environments. First, we provide *scenario creation* by service composition. Users can create *complex scenarios* that correspond to their needs thanks to an appropriate ADL. This ADL is simple, user-oriented and proposes an alternative graphical view to be *accessible for everyone*. SaS exports scenarios as new services into its environment, thus, users can easily *share* their scenarios. Moreover, SaS manages *scenario life-cycle*: it enables users to start and stop scenarios, check scenarios status and scenario execution advancement. Moreover, users can get an overview of a scenario specification (scenario introspection capability) thanks to a descriptor file and *reuse a scenario* as a service being part of a new encapsulating scenario composition (scenario hierarchical composition). SaS also tries to *maintain scenario availability*. Locally, if a service involved into a scenario disappears, SaS tries to replace it into the composite. Globally, when a SaS platform disappears, scenarios exported by this platform are redeployed on other ones to remain available. In conclusion, the SaS system presented in this paper satisfies all the requirements defined is section II-B. A prototype of SaS in Java over OSGi and iPOJO is currently under development.

We have three major perspectives. First, we want to evaluate SaS to show its simplicity for non technical end-users by comparing it with existing tools such as Yahoo Pipes [33], Automator [34] and Scratch [35]. Such tools provide non-technical end-users of the capability to graphically develop small applications by composing elements. Then, we plan to define some recovery strategies to anticipate service loss such as *caching* and *hoarding*. Finally, we want to improve scenario distribution and propagate scenarios into the network.

Acknowledgements

This work is partially 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*, pp. 78–89, 1995.

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

[3] OASIS, "Reference Model for Service Oriented Architecture 1.0," pp. 12 – 13, oct 2006. [Online]. Available: http://docs.oasis-open.org/soa-rm/v1.0/soa-rm.html

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

[5] A. Bottaro, A. Gérodolle, and P. Lalanda, "Pervasive service composition in the home network," in *Proc. of the 21st International Conference on Advanced Networking and Applications*, 2007, pp. 596–603.

[6] C.-L. Wu, C.-F. Liao, and 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)*, vol. 37, no. 2, pp. 193–205, mar 2007.

[7] D. Valtchev and I. Frankov, "Service gateway architecture for a smart home," *Communications Magazine, IEEE*, pp. 126–132, 2002.

[8] M. Bakhouya and J. Gaber, *Agent Systems in Electronic Business*. IGI Publishing, 2007, ch. Service Composition Approaches for Ubiquitous and Pervasive Computing Environments: A Survey, pp. 323–350.

[9] J. Bronsted, K. M. Hansen, and M. Ingstrup, "Service composition issues in pervasive computing," *IEEE Pervasive Computing*, vol. 9, pp. 62–70, 2010.

[10] A. Urbieta, G. Barrutieta, J. Parra, and A. Uribarren, "A survey of dynamic service composition approaches for ambient systems," in *Proceedings of the 2008 Ambi-Sys workshop on Software Organisation and MonITored of Ambient Systems*, ser. SOMITAS '08, 2008, pp. 1–8.

[11] N. Ibrahim and F. Le Mouël, "A Survey on Service Composition Middleware in Pervasive Environments," *International Journal of Computer Science Issues (IJCSI)*, vol. 1, pp. 1–12, 2009. [Online]. Available: http://hal.inria.fr/inria-00414117/en/

[12] E. Aarts and B. de Ruyter, "New research perspectives on Ambient Intelligence," *Journal of Ambient Intelligence and Smart Environments*, vol. 1, pp. 5–14, 2009.

[13] P. Clements, "A survey of architecture description languages," in *Proc. of the 8th international workshop on software specification and design*. IEEE Computer Society, March 1996, pp. 16–25.

[14] S. Vestal, "A Cursory Overview and Comparison of Four Architecture Description Languages," Honeywell, Tech. Rep., February 1993.

[15] P. Mishra and N. Dutt, "Architecture description languages," *IEEE proc. - Computers and Digital Techniques*, vol. 152, no. 3, p. 285, 2005.

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

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

[18] Ecma International, "ECMA-262: ECMAScript Language Specification," December 2009. [Online]. Available: http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-262.pdf

[19] J. Encarnaçao and T. Kirste, "Ambient intelligence: Towards smart appliance ensembles," *From Integrated Publication and Information Systems to Information and Knowledge Environments*, no. December, pp. 261–270, 2005.

[20] 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 International Conference on Software Engineering and Knowledge Engineering (SEKE 2009)*, July 2009.

[21] ——, "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)*, Mars 2010, pp. 35–49.

[22] UPnP Forum, "Understanding UPnP: A White Paper," 2000. [Online]. Available: http://www.upnp.org/download/UPNP_UnderstandingUPNP.doc

[23] C. Bettstetter and C. Renner, "A comparison of service discovery protocols and implementation of the service location protocol," in *Proc. of the 6th EUNICE Open European Summer School: Innovative Internet Applications*. Citeseer, 2000, pp. 13–15.

[24] G. Aschemann, R. Kehr, and A. Zeidler, "A Jini-based Gateway Architecture for Mobile Devices," *In Proc. of the Java-Informations-Tage (JIT99)*, p. 203–212, September 1999.

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

[26] OSGi Alliance, "OSGi Service Platform Enterprise Specification," pp. 15 – 27, march 2010. [Online]. Available: http://www.osgi.org/download/r4v42/r4.enterprise.pdf

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

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

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

[30] S. Means and M. A. Bodie, *Book of SAX: The Simple API for XML*. No Starch Press, 2002.

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

[32] Apache Foundation, "ipojo api," 2010. [Online]. Available: http://felix.apache.org/site/apache-felix-ipojo-api.html

[33] Yahoo, "Rewire the Web." [Online]. Available: http://pipes.yahoo.com/pipes

[34] Apple, "Automator: Your personal Automation Assistant." [Online]. Available: http://www.macosxautomation.com/automator

[35] M. Resnick, J. Maloney, A. Monroy-Hernandez, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, and Y. Kafai, "Scratch: Programming for Everyone," *Communications of the ACM*, vol. 52, no. 11, pp. 60–67, 2009.