Management of Scenarized User-centric Service Compositions for Collaborative Pervasive Environments.

Matthieu Faure

Académie de Montpellier
Université Montpellier II
Sciences et **T**echniques du **L**anguedoc

Thèse

pour obtenir le grade de docteur

Spécialité : **Informatique**
Formation Doctorale : **Informatique**
École Doctorale : **Information, Structures, Systèmes**

présentée et soutenue publiquement par

Matthieu Faure

Management of scenarized user-centric service compositions for collaborative pervasive environments

Soutenue le 07 décembre 2012, devant le jury composé de :

Rapporteurs
Jean-Michel Bruel (Prof.) — Université de Toulouse
Lionel Brunie (Prof.) — Institut National des Sciences Appliquées (INSA) de Lyon

Examinateurs
Philippe Collet (MCF-HDR) — Université Nice Sophia Antipolis
Peter Kriens (PDG) — aQute
Joël Quinqueton (Prof.) — Université de Montpellier III

Directrice de thèse
Marianne Huchard (Prof.) — Université de Montpellier II

Co-encadrants de thèse
Luc Fabresse (MCF) — École des Mines de Douai
Christelle Urtado (MCF) — Invitée, École des Mines d'Alès
Sylvain Vauttier (MCF) — École des Mines d'Alès

Osez ! Ce mot renferme toute la politique de notre révolution.
Louis Antoine de Saint-Just

Remerciements

Tout d'abord, je remercie l'Institut Carnot M.I.N.E.S, financeur de cette thèse. Ce financement m'a permis de me consacrer pleinement à mon travail.

Je remercie Messieurs Stéphane Lecoeuche, directeur de l'Unité de Recherche Informatique et Automatique (URIA) de l'École des Mines de Douai, et Yannick Vimont, directeur du Laboratoire de Génie Informatique et d'Ingénierie de Production (LGI2P) de l'École des Mines d'Alès, pour m'avoir successivement si bien accueilli et permis de réaliser cette thèse dans d'idéales conditions.

Je voudrais par ailleurs remercier l'ensemble des membres du jury. Je remercie particulièrement Messieurs Lionel Brunie et Jean-Michel Bruel pour avoir accepté de rapporter ma thèse. Leurs remarques et conseils furent précieux. Je remercie également Messieurs Joël Quinqueton, Philipe Collet et Peter Kriens pour avoir accepté d'examiner mon travail. Ils honorent mon jury de thèse de leur présence.

Je tiens à remercier tout particulièrement et à témoigner toute ma reconnaissance à mon équipe encadrante: Madame Marianne Huchard pour avoir dirigé ce travail de thèse, Madame Christelle Urtado et Messieurs Luc Fabresse et Sylvain Vauttier qui m'ont accompagné dans cette aventure. Je tiens à louer votre complémentarité, votre enthousiasme, votre disponibilité et votre pédagogie. Cette thèse fut une expérience incroyable qui n'aurait pu se faire sans vous.

Je remercie l'ensemble du personnel technique et administratif du Département Informatique et Automatique de l'Ecole des Mines de Douai et du site de Nîmes de l'Ecole des Mines d'Alès. Toujours disponibles et aimables, ils sont garants de la réussite de ces deux institutions. Merci aux enseignants-chercheurs, doctorants, post-doctorants et ingénieurs des deux laboratoires qui m'ont toujours bien accueilli et furent souvent d'une aide précieuse. J'ai pu apprécier leur bonne humeur. A ce titre, je tiens particulièrement à remercier: Lyes, qui m'a introduit aux subtilités du doctorat et m'a entraîné dans des discussions toutes aussi passionnantes qu'éclairantes, Mariano, pour son amitié, son amour du ballon rond et ses expressions argentines qui marquèrent si durablement mon vocabulaire, Guillaume, collègue de travail et ami, ses remarques enrichissent continuellement mon travail, Abdelhak, ami sincère et pronostiqueur imparable.

Par ailleurs, je remercie aussi Laurent, pour son temps et son travail précieux ainsi que tous ceux qui ont contribué, de quelque manière que ce soit, à l'accomplissement de ce travail.

Je voudrais remercier mes amis, toujours présents (malgré la distance parfois). Merci à Niamh, Réjane, Ali, Benoit et tous ceux que je ne cite pas mais que je n'oublie pas...

Je remercie ma famille qui m'a toujours accompagné et encouragé, mes parents et beaux-parents, pour leur éducation et leur amour, mes sœurs et mon frère, à jamais dans mon cœur. Cette thèse est le fruit de votre réussite.

Pour finir, je tiens à remercier Adélaïde pour son soutien quotidien indéfectible et son enthousiasme contagieux à l'égard de mes travaux comme de la vie en général. Elle est la raison principale de mon épanouissement. Merci d'être là, à mes côtés...

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Contents

This chapter introduces the context of the thesis. Progress in nanotechnologies makes that all the everyday devices can embedded a small computer. Thus, we are surroundedg by electronic devices that can assist us in our daily life: this the emergence of *pervasive computing*. In this context, we present the problematics of this thesis and introduce the solutions that underlie our contribution. Finally, we present the thesis outline.

1.1 Context of the Thesis

With the progress in nanotechnology, computers become smaller and more powerful over time. As described by Moore in 1960's, power of microprocessors doubles every 18 months [55]. Computer size changes (and its memory increases) can be easily illustrated. The IBM 350 (first computer with a hard drive), commercialized in 1956 by IBM, was 9 meters long and 15 meters wide [40]. Its storage capacity amounted five megabytes. Nowadays, any smartphone has a capacity of several gigabytes (and fits in one hand). In addition, the improvement of electric batteries makes electronic devices energetically independent and thus *mobile*. Additionally, the development of wireless communication technologies makes computers communicant. Electronic devices become remotely controllable and moreover, can be networked without being manually configured to. They can also *collaborate*.

Miniaturization and reduction of costs make the number of electronic devices increase significantly. In few decades, we evolved from a situation where there was one computer for a group of persons, to one computer by person (advent of the personal computer) and now, a set of computers for each person. Computer miniaturization gives everyday devices (television, light, fridge, etc.) the potential to embed an electronic system. Each electronic device can have its own central processing unit (CPU), its own memory and its own network connectivity. This is the rise of *ubiquitous computing* as defined by Mark Weiser in 1995 [89]. We are surrounded by electronic devices, through wired or wireless networks.

Figure 1.1 illustrates the evolution of computer these last 50 years. This Figure is adapted from [88] by Jean-Baptiste Waldner. We could discuss the pertinence of the dates chosen by the author. However, this figure clearly represents the emergence of ambient intelligence [1]: environments tend to be totally personalized and plenty of electronic devices provide access to a multitude of functionalities that assist us in our daily life.

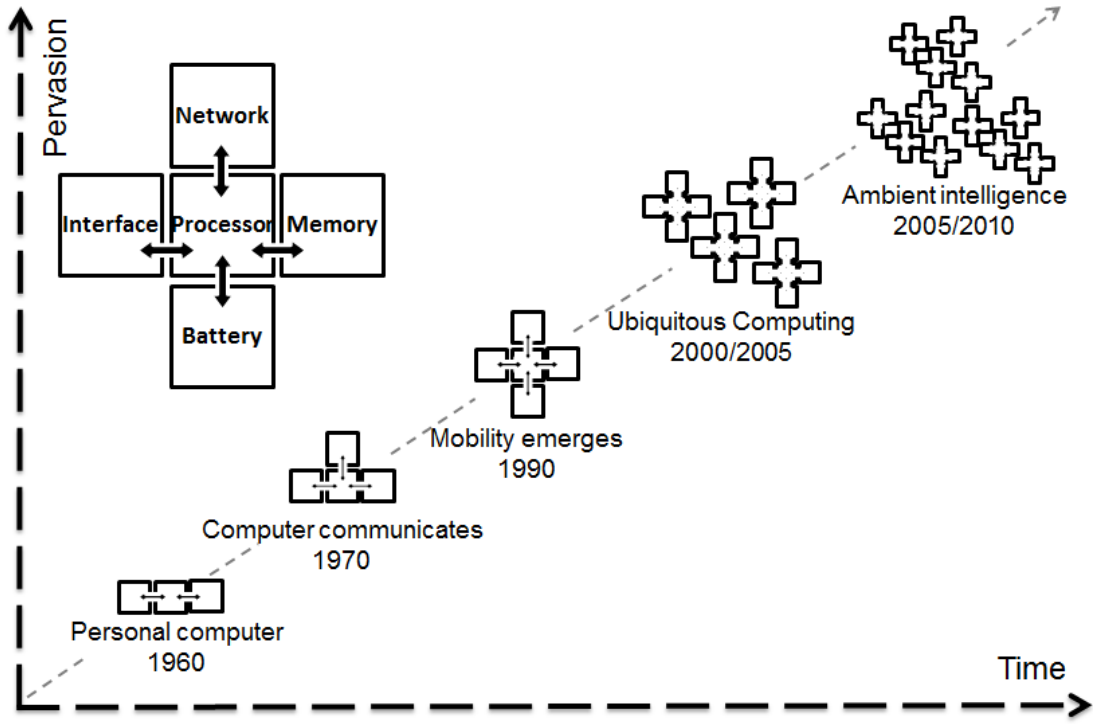

Figure 1.1: Computer evolution by JB Waldner

1.1.1 From Ubiquitous Computing to Pervasive Systems

Since recently, the term *"pervasive"* is often used instead of "ubiquitous". The two words can be understood similarly even if some people highlight the differences [51]. Various interpretations of these two words can be found in the literature. This is why, when the Institute of Electrical

and Electronic Engineers (IEEE) published its first issue of Pervasive Computing [72] in 2002, the editor-in-chief Satyanarayanan Mahadev clarified the terminology to be used: *"This magazine will treat ubiquitous computing and pervasive computing as synonyms - they mean exactly the same thing and will be used interchangeably throughout the magazine"*.

Literally, ubiquitous means *"everywhere"* while pervasive means *"diffused throughout every part of"*. Thus, we understand ubiquitous as "a functionality accessible anywhere, anytime". This is exactly what applications on web-enabled smartphones bring us. We understand pervasive as "a functionality accessible through the devices around me that thus adapts to my environment". Therefore, in this thesis, we will use the pervasive term.

1.1.2 Pervasive Environments

This subsection details the elements that compose a pervasive environment. Figure 1.2 presents a pervasive environment example: a living room, with several electronic devices and three users. We use this example to illustrate the pervasive environment elements.

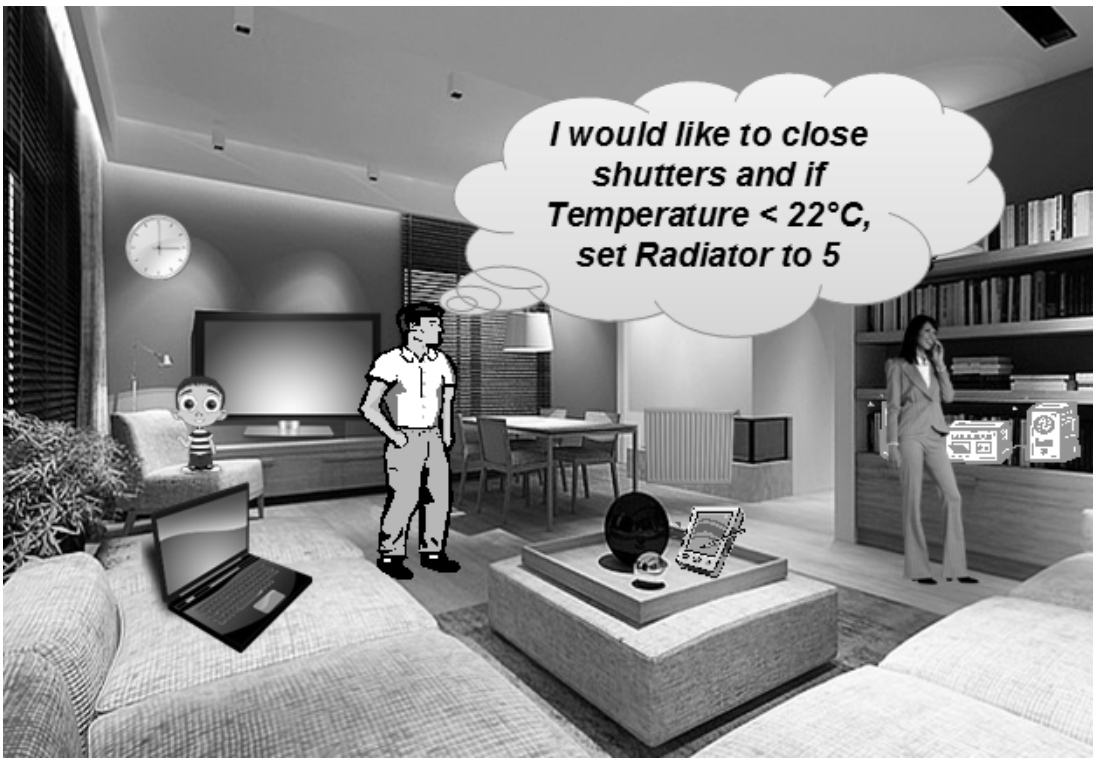

Figure 1.2: User's main issue

1.1.2.1 Services

A pervasive environment is characterized by electronic devices that can interact with their environment and propose various functionalities to users. According to the OASIS organization, *"a service is a mechanism to enable access to one or more functionalities"* [58]. Thus, device functionalities can be handled as services, and thus, Service-Oriented Computing (Soc) [61] is a suitable paradigm to design software for ubiquitous environments. Therefore, Soc eases the use of these electronic devices through their services. In the example of Figure 1.2, we can imagine that the clock proposes a `Clock` service and the TV a `Watch` service and so on.

1.1.2.2 Scenarios

Each device provides its own set of services. Users are often limited to use a single service at a time. However, users usually want to describe *scenarios*. As shown in Figure 1.2, user's need can be satisfied by a scenario that involves a composition of multiple services provided by multiple devices. Enabling end-users to describe their own scenarios would be a first improvement and a step towards ambient intelligence. In addition, users should easily manage the created scenarios and have access to them from several control devices (PDA, mobile phone, etc.).

1.1.2.3 Users

As illustrated by Figure 1.2, several users can share the same pervasive environment. Users can possess a device (*e.g.* a smartphone) or share one (*e.g.* a TV). Moreover, they might want to reach the same goal (*e.g.* parents that want to close all shutters at night) and thus, they must be able to *collaborate*. Additionally, users can have different rights (*e.g.* children might not have access to all devices).

1.2 Challenges in Pervasive Environment including Users' Expectations

The idea of pervasive computing as stated by Weiser in 1995 introduced a new perspective on our daily life. This implies some challenging issues that are detailed in this section.

1.2.1 Benefit of Pervasive Environments

The dimension of pervasive environments is highly problematic because of the various aspects of pervasive environment elements (heterogeneity and volatility of devices, etc.). The challenge is not just to adapt these singularities, but also to benefit from them.

1.2.1.1 Heterogeneity

Devices present in pervasive environments can be of various kinds: simple devices (*e.g.* light or clock), mobile assistant devices (*e.g.* PDA or smartphones), multimedia devices (*e.g.* PC or TV), etc. These devices have different hardware and are based on various operating systems. They can have different communication technologies (*e.g.* wired/wireless), and implement different network protocols (*e.g.* UPnP, Jini, SLP, GSM). Thus, we need to handle this heterogeneity and provide users with an *interoperable* system.

1.2.1.2 Dynamicity

In pervasive environments, devices are volatile. They can promptly appear and disappear. Moreover, unexpected devices can appear, thus giving access to a new functionality that could interest users. Therefore, a pervasive system must be *adaptive* to changes and even, benefit from unexpected environment elements.

1.2.1.3 Representation

To control a pervasive environment, users must have its representation. This representation must adapt to the heterogeneity and the dynamicity of pervasive environments but also to the diversity of environment elements (devices, services, users). In addition, users must have the capacity to adapt this representation as they wish. Typically, surrounding devices can fulfill to different goals (multimedia, work, etc.) or be attached to different locations (home, kitchen, office, etc.).

1.2.2 Meet users' expectations

Users are at the center of pervasive environments. A pervasive system must meet users' expectations. This implies to enable users to control their environment as they wish, and to adapt to their mobility and multiplicity.

1.2.2.1 Users' needs

In a pervasive environment, users have access to devices' functionalities. Soc eases service discovery and access. Thus, it enables users to reach a device functionality. However, it still is impossible for end-users, which have no specific technical knowledge, to fully benefit from the services proposed by their surrounding devices. As seen in Figure 1.2, users' needs can be reached by scenarios, which are service composition. Different service composition mechanisms have been proposed in the literature [6, 12, 83, 41]. However, none of them has been specifically designed for users without technical knowledge. In addition, to easily express their goals by defining scenarios, users must control (start, pause, resume, abort) their creations, and be able to retrieve and reuse them (even in another scenario).

1.2.2.2 Users Mobility

Users are mobile. They can move from a pervasive environment to another. The system must therefore dynamically adapt to its changing environment. Besides, users might have a need that imply functionalities present on different locations and thus, not available simultaneously. Thus, users' mobility is both a constraint on the system (need of adaptability) and a chance. The system can benefit from mobility, while executing scenarios that involve services that never coexist in a same environment.

1.2.2.3 Collaboration

Several users can share the same pervasive environment. They can have common interests. Moreover, a user can possess several devices in a pervasive environment (*e.g.* a smartphone, a tablet and a PC can be owned by the same user and available simultaneously). These devices could have access to functionalities that the other ones do not have (*e.g.* smartphones can send text messages). Thus, they must collaborate by sharing information. Besides, devices can disappear from the environment (they can be turned off or took away). The collaboration must enable devices to supersede each other when possible.

1.3 The Problematics of this Thesis

Pervasive computing is the next step in the evolution of computer science. It promises a bright future where users' expectations are fully realized by pervasive systems. However, if some progresses have been made, especially in home automation [11, 91], we have not yet reached Weiser's vision. Controlling a pervasive environment implies that devices that populate the environment are remotely controllable, even through various protocols. Typically, a shutter that can only be used manually cannot be integrated in a *pervasive system* (*i.e.* a system that tries to enable users to control a pervasive environment). Everyday devices which are remotely controllable become more and more frequent in our environment, but are not yet widely adopted today. However, if users cannot control their environment as they wish, the lack of appropriate devices is not be the only reason. To our knowledge, existing systems fail to meet users' expectations. Currently, pervasive environments are hardly directly usable by non-technical users.

In this thesis, we advocate that a pervasive system must enable users to fully benefit from their environment. A pervasive system must *adapt to the environment* and its particularities (presence of mobile users and devices, use of different protocols). It must provide users with a *representation of their context* and enable them to manage this representation as they wish. Moreover, users must be able to *easily express their needs* by creating scenarios. Such scenarios represent users' goal and thus, must be *customizable* and *easily controllable and reusable*. In addition, the presence of several users implies that users must be able to collaborate (*e.g.* share scenarios). Scenarios execution must be *resilient* to environmental changes and user actions, but also dynamically adapt and benefit from the context.

In this thesis, we raise two main issues:

1. What is a user-centric system in pervasive environments, *i.e.* a system that enables its user to control pervasive environments?
2. How to design and build a user-centric system that enables users to fully benefit of pervasive environments?

1.4 Contribution of this Thesis

In a nutshell, this thesis advocates that pervasive systems should be user-centric and collaboration aware. They should thus fulfill precise functional and non-functional requirements (detailed in Chapter 2.2) such as context management or scenario definition, control and sharing.

The main contributions of this thesis are:

- definition of user-centric systems in pervasive environments and identification of their requirements,
- development of an interoperable approach for context management which is adaptive and provides a customizable and persistent context representation to users,
- definition of a scenario description language which corresponds to users' expectations,
- establishment of scenario control mechanisms for users that include scenario reuse capabilities,
- advanced scheduling of the step-by-step scenario execution that adapts to environmental changes and benefits from users' mobility,
- implementation of sharing mechanisms that enable users to collaborate.

1.5 Thesis Outline

This thesis is split into three parts. It is further organized as follows:

Part I: The first part is dedicated to the study of pervasive systems. It is composed of two chapters:

Chapter 2 introduces the context of pervasive systems. We present a motivating example, detail the attached terminology and establish the list of requirements for user-centric systems in pervasive environments.

Chapter 3 is dedicated to the state of the art. Based on the requirements established in the previous chapter, we analyze and compare some existing pervasive systems that try to enable users to control pervasive environments.

Part II: The second part presents our contribution. It is split into four chapters:

Chapter 4 introduces the contribution and presents a brief overview. Then, we study how our contribution manages its context (a pervasive environment) and enables users to easily compose a set of services into a scenario that represents their needs.

Chapter 5 details scenario management. We discuss how users can control, reuse and share scenarios.

Chapter 6 is dedicated to scenario execution. We present how a scenario description is analyzed to enable its dynamic and mobile execution and we detail the adaptive and fault-tolerance mechanisms of our contribution.

Chapter 7 presents the implementation and validation of our contribution. A prototype has been developed as a proof of feasibility and enables to make experiments to evaluate and validate the contribution. Then, the chapter presents and discusses the results obtained by experimentation regarding the accomplishment of each of the previously identified requirements.

Part III: The third and last part of this thesis concludes. It contains one chapter:

Chapter 8 concludes this work. We summarize our contribution and discuss its limits. Then, we draw some perspectives.

Part I

Pervasive Systems

CHAPTER 2

Context of User-Centric Systems in Pervasive Environments

Contents

The chapter presents the context of our study. A first section introduces some motivating examples that illustrate the context, and establishes a terminology, used all along this thesis. Then, a second section establishes the list of requirements for user-centric systems in pervasive environments. This exhaustive list is used as a reading grid for the state of the art study, and corresponds to the requirements qualities expectations that our contribution tries to satisfy.

2.1 Presentation and Terminology

We present here a simple example to illustrates users' needs in pervasive environments. Then, we pose the terminology used all along this thesis.

2.1.1 Motivating Example

Our example takes place in a smart home [84] (illustrated by Figure 1.2). Every night, parents close all shutters and the main door. Depending on temperature, they adjust the thermostat. Obviously, users' goal here is a need for security and comfort. This goal is achieved by a combination of services provided by different devices. This combination further obeys a simple logical structure. To reach this goal, one of the parents wishes to easily define a scenario on a *control device* (*i.e.* a laptop). Once defined, this scenario might be shared with the other parent

(but maybe not with children) and available on other control devices of the house (smartphones, PCs, etc.) even if the original provider device disappears. This scenario must be easy to control (start, pause, check status) and to parameterize (possibility to define a new thermostat value). Of course, scenario execution should adapt to environmental changes. Typically, the system should maintain scenario execution if a service disappears (the kitchen clock has no more battery) by trying to find an appropriate replacement service (the bedroom clock still works).

2.1.2 Terminology

The terminology posed in this section is relative to pervasive environment elements, briefly introduced in Chapter 1 (such as users, devices, etc.). Then, it clarifies the terms relative to the system that enables to control the pervasive environment (such as scenarios and control devices).

2.1.2.1 Users, devices and services

Pervasive environments involve multiple *users* and multiple *devices* that each provides a set of *services*. A device is a communicating electronic object (such as a clock, a DVD player or a smartphone). Devices publish services (such as `Time`, `MultimediaPlayer`, or `Localization`). Each service provides one or more *operations* (such as `getTime` or `setTime`, `play`, `getLocation`). They are called "capabilities" by the OASIS consortium in the *Service Oriented Architecture* (SOA) norm [58]. Users use these operations to access functionalities of devices. Operations can require entry *parameters*.

2.1.2.2 Service composition and scenarios

Devices can interoperate but the goal the system has to achieve always comes from users, which can be simple consumers or technical experts that command devices. Their needs can always be considered as *scenarios* defined as combinations of operations from different services / devices. Such combination is said to be a *service composition* because services are not stand-alone elements and to stick to the terminology used in *Service Oriented Computing* [61].

2.1.2.3 Control devices

Some devices have a graphical user interface and implement different wireless protocols (wifi, bluetooth, etc.). They thus are suitable to implement control capabilities and personally assist users in pervasive environments. Typically, users must be able to get an overview of surrounding services with their control devices. Additionally, control devices must provide functionalities to define and control (*e.g.* start, pause, etc.) scenarios.

2.1.2.4 Synthesis

Notions previously presented interrelate. Figure 2.1 depicts the class diagram of pervasive environments elements. Devices (which can be simple or control devices) export services. Services expose operations, that can have parameters. A control device is owned by a single user.

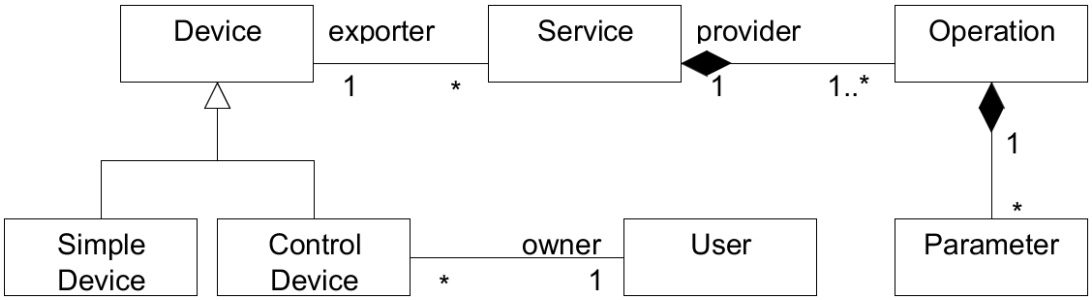

Figure 2.1: Class diagram of pervasive environments elements

2.2 Requirements for User-Centric Systems in Pervasive Environments

Requirements for user-centric systems in pervasive environments can be split into two categories, functional and non-functional requirements [13]. The requirements numbering is kept and used all along the manuscript (especially in the chapter dedicated to the state of the art).

2.2.1 Functional requirements

By functional requirements we refer to the following end-user needs: manage the context, define, use, reuse and share scenarios.

2.2.1.1 R1. Context Management

According to Coutaz *et al.* in [20], a context should be viewed not simply as a complex state but as a process result that can take into account spontaneous interactions between users, services and resources. This implies being continuously aware of the context and enabling users to represent it as they wish.

(a) Context representation. Pervasive systems should manage a representation of their execution context (presence of services / devices, device location, device owner). Users must be able to add to this inner representation their own preferences (such as defining device or service categories) and should be able to relate this representation to the information of the device/service directory.

(b) Context awareness. Pervasive systems must be continuously aware of service and device availability and should be able to relate them to the context representation.

2.2.1.2 R2. Scenario Definition

Users' needs cannot all be satisfied with predefined, thus limited, scenarios. Moreover, predefined scenarios cannot leverage the new kinds of services that can be dynamically discovered in open environments. Scenario definition is therefore a requirement of pervasive systems.

(a) Service composition. Users' needs involve several services, bound together with conditions, logical statements, etc. A scenario definition thus consists of a composition of services, glued together by control structures (conditional statement, repetition, etc.).

(b) Scenario customization. Users should be able to define generic, parameterizable scenarios to foster their reuse in various and variable contexts. As a part of parameterization, pervasive systems should enable users to specify if a service involved in a scenario has to come from a specific device (or not) or to specify when parameter values for service operations have to be provided (at scenario run-time or at design-time).

2.2.1.3 R3. Scenario Execution

Users must be able to control scenario execution. Moreover, scenario execution must be resilient and the system should dynamically adapt scenario execution.

(a) Scenario user control. Pervasive systems should enable users to easily control scenario execution (start, pause, abort) and check for scenario execution advancement.

(b) Scenario execution resilience. Scenario execution must adapt to user actions and environmental changes (such as service disappearance) so that scenario execution becomes resilient.

2.2.1.4 R4. Scenario reuse

Users must be able to reuse their scenarios. This assumes that scenarios description are persistent to be reused in the future by users. Moreover, scenarios must recomposable into new scenarios.

(a) Scenario description availability. Scenarios description should be persistent to enable future use (users should not need to redefine a scenario each time they want to use it). Scenario description availability must also be preserved in a pervasive environment when the control device used for the scenario definition disappears.

(b) Hierarchical composition. Users may need to reuse existing scenarios to create more complex ones.

2.2.1.5 R5. Scenario Sharing

Pervasive systems must provide mechanisms to share scenarios among users. In addition, users should be able to choose what they share (the scenario description or the scenario execution control) depending on the target.

(a) Select what to share. Created scenarios could interest other users. The simplest way of sharing scenarios is by sharing their description. Additionally, systems must enable scenario execution cooperation. Typically, a home-designed scenario should be controllable at the same time on several control devices owned by family members (PDA, laptops, etc.). Therefore, scenario execution should also be sharable.

(b) Select who to share with. Several users can be present in the same environment. Users might want to take advantage of this by sharing their scenarios with others. However, they must select who they want to share the scenario with. Typically, a home-designed scenario should be shared differently with parents or with children (*i.e.* scenario control must be more restrictive).

2.2.2 Non-Functional Requirements

The non-functional requirements come from the pervasive nature of the environment and impact the functional requirements. The first two requirements are related to users, the two last ones are more context specific. We present each non-functional requirement and how it impacts previously identified functional requirements.

2.2.2.1 RA. User Friendliness

Users often are not technical experts. Therefore, system functionalities must be easy to use. For example, information should be presented in an easy to understand way, possible actions should not necessitate difficult parameterization, etc.

User Friendliness & Context management. Users must easily retrieve environment elements thanks to the context representation (R1.a). Moreover, some changes in context representation imply user interventions (such as defining a device location). It is therefore impacted by user friendliness, whereas context awareness (R1.b) should be an automatic process, transparent to users.

User Friendliness & Scenario definition. Service composition (R2.a) and scenario customization (R2.b) are both user actions. So, they must be realizable without many technical knowledge.

User Friendliness & Scenario execution. Users must easily control scenario execution (R3.a), however, scenario runtime resilience (R3.b) is an automatic system process, transparent to users.

User Friendliness & Scenario reuse. Users should be able to easily find a scenario that they previously defined (R4.a). Moreover, users must be able to reuse a scenario into another one (R4.b). Scenario reuse is therefore impacted by user friendliness.

User Friendliness & Scenario sharing. Select who to share (R5.a) with and what to share (R5.b) must be accessible for users. User friendliness thus affects scenario sharing.

2.2.2.2 RB. Collaborativeness

Several users share the same pervasive environment. Users might want to reach the same goal (*e.g.* parents that want to close all shutters at night) and thus, they must be able to *collaborate*.

Collaborativeness & Context management. Collaborativeness impacts context representation (R1.a): the system must represent surrounding users to enable collaboration. Moreover, users are mobile and can appear or disappear from the environment. Presence of users must be detected. Collaborativeness thus impacts context awareness (R1.b).

Collaborativeness & Scenario definition. Service composition (R2.a) does not need to take into consideration the collaborative aspect of pervasive environments as composition is only targeted at defining an executable sequence of activities. Nonetheless, users may be interested in defining parameterizable scenarios (R2.b) that adapt to different users.

Collaborativeness & Scenario execution. If scenario execution control (R3.a) has been shared, it is affected by collaborativeness. However, scenario execution resilience (R3.b) does not depend on which users control the scenario execution. Thus, collaborativeness does not have any impact on it.

Collaborativeness & Scenario reuse. To maintain scenario description availability (R4.a), users must collaborate. A user may ask to another one to take scenario availability responsibility. However, enabling scenario hierarchical composition (R4.b) is a functional requirement that does not depend on collaborativeness.

Collaborativeness & Scenario sharing. Collaborativeness impacts both the possibility to select who to share a scenario with (R5.a) and what to share: scenario description and / or execution (R5.b).

2.2.2.3 RC. Adaptability

Pervasive environments change (services appear, disappear, etc.). Needs may evolve. They can be different depending on the user, on the location, etc. Pervasive systems should thus be adaptive.

Adaptability & Context management. Context representation (R1.a) must adapt to the different elements of the context (locations, users, etc.). The pervasive system should constantly be aware (R1.b) of the evolution in its environment (dynamic appearance of services without prior knowledge).

Adaptability & Scenario definition. Service composition (R2.a) is a static process. However, pervasive systems must enable users to define adaptable scenarios, which are parameterizable at runtime (R2.a).

Adaptability & Scenario execution. User scenario control (R3.a) is not impacted by adaptability, whereas, scenario execution resilience (R3.b) is a response to environmental changes.

Adaptability & Scenario reuse. Similarly, maintaining scenario description availability (R4.a) depends on environmental changes and so, adaptability affects it. However, specifying scenarios as recomposable entities (R4.b) is not impacted by adaptability.

Adaptability & Scenario sharing. Finally, select who to share with (R5.a) or what to share (R5.b) must adapt to the shared scenario availability. For instance, when a shared scenario is uninstalls, the pervasive system must adapts scenario sharing.

2.2.2.4 RD. Mobility

In pervasive environments, users, devices and services are mobile. It implies to adapt to the mobility (*e.g.* service volatility) but also to benefit from the mobility such as executing scenarios that involve services that never coexist in a same environment.

Mobility & Context management. Mobility impacts context awareness (R1.b), due to the appearance and disappearance of mobile services, and context representation (R1.a), due to the need to represent different locations where users and devices might be.

Mobility & Scenario definition. Users might need to define scenarios on multiple locations. Moreover, users must define customizable scenarios that adapt to different environments encountered. Service composition (R2.a) and scenario customization (R2.b) thus depend on mobility.

Mobility & Scenario execution. Mobility does not impact user scenario control (R3.a). However, handling mobility in scenario execution resilience (R3.b) is primordial (typically: enabling scenario execution continuity in different locations).

Mobility & Scenario reuse. Preserving scenario description availability (R4.a) implies that the scenario description moves from a system to another one. Whereas, hierarchical composition (R4.b) is not affected by this non-functional requirement.

Mobility & Scenario sharing. Users must be able to select other users to share scenarios with, even if these users are mobile and not always available. Moreover, sharing a scenario (its description or its execution) has to deal with scenario provider mobility.

2.2.3 Synthesis

In this section, we have detailed the list of requirements for a system that enables users (without technical knowledge) to control a pervasive environment. These requirements are classified in two categories: functional and non-functional. Figure 2.2 depicts the requirements hierarchy. Table 2.1 synthesizes the impact of the non-functional requirements over the functional ones. ◇ symbolizes an absence of impact, whereas ◆ points out an impact.

Thanks to this study, we can analyze and compare existing systems. Established requirements serve as a reading grid in next chapter, dedicated to the state of the art.

Non functional requirements	**Functional requirements**									
	R1. Context management		R2. Scenario definition		R3. Scenario execution		R4. Scenario reuse		R5. Scenario sharing	
	(a)	(b)	(a)	(b)	(a)	(b)	(a)	(b)	(a)	(b)
RA. User friendliness	◆	◇	◆	◆	◆	◇	◆	◆	◆	◆
RB. Collaborativeness	◆	◆	◇	◆	◆	◇	◆	◇	◆	◆
RC. Adaptability	◆	◆	◇	◆	◇	◆	◆	◇	◆	◆
RD. Mobility	◆	◆	◆	◆	◇	◆	◆	◇	◆	◆

Table 2.1: Impact of non-functional requirements on functional requirements

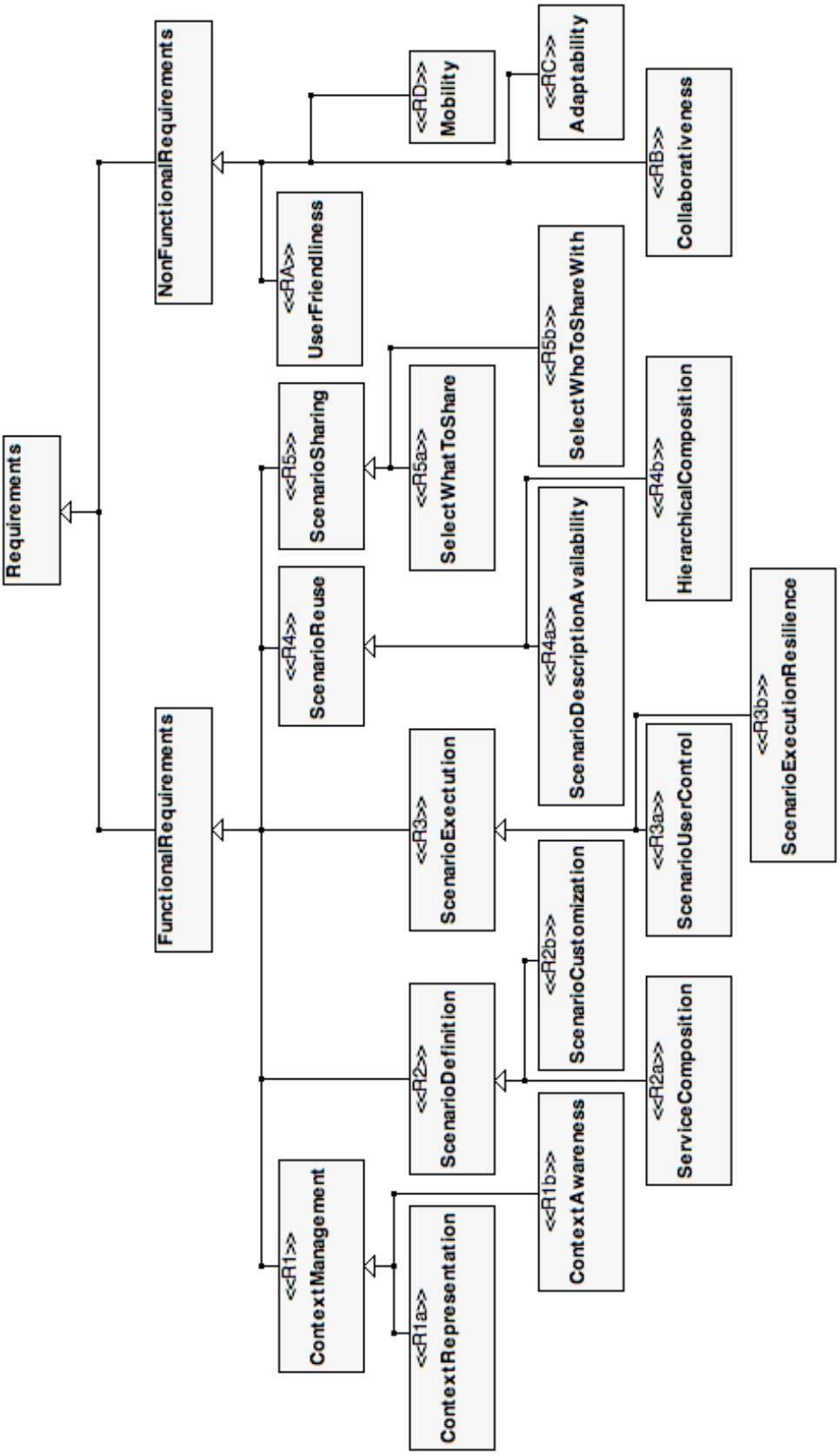

Figure 2.2: Requirements hierarchy

Chapter 3

State of the Art

Contents

In the previous chapter, we presented the essential requirements for systems to enable users to easily control pervasive environments. These requirements are twofold: functional and nun-functional. Now that the requirements of pervasive systems are established, we study how existing approaches answer to the problematics, what are the benefits of an approach and what still misses or is not completely achieved. To do so, this chapter is dedicated to the analysis of some existing pervasive systems. We briefly introduce each system and then, analyze it based on the prior requirements. This study is divided into sections dedicated to each functional requirement and one section that explains how each system fulfills the non-functional requirements that impact the functional ones. For each functional requirement, technical approaches and concepts that try to respond to the requirement are introduced. Then we analyze how each system tries to achieve the requirement. A last section is dedicated to system comparison and summarizes this study.

3.1 Presentation of Studied Pervasive Systems

In this section we briefly present different pervasive systems. These systems are representative of the different approaches that enable the control of a pervasive system. They are recent academic works and have been implemented in an operational prototype to prove their feasibility and evaluate their contribution. This section introduces each system and presents its main characteristics (origin, authors, technical choices, etc.). The systems are alphabetically presented.

3.1.1 Anamika

Anamika [15] is an implementation of the reactive service composition architecture for pervasive environments proposed in 2002. This is a distributed, de-centralized and fault-tolerance design architecture that enables to execute a request (*i.e.* a scenario) corresponding to a user need. Anamika comprises two different techniques to dynamically select service brokers responsible for scenario execution. Moreover, it implements fault-tolerance mechanisms to adapt to environmental changes.

3.1.2 DigiHome

DigiHome [67] is a system dedicated to control home appliances. Its main goal is to facilitate high-level and event programming of *Sensor Wireless Networks*. It is based on a service component architecture (SCA), implemented on FraSCAti [75] and incorporates the REpresentational

State Transfer (REST) architecture. Moreover, it is an event-based system which implements *Complex Event Processing* (CEP).

3.1.3 MASML

MASML [91] is a multi-agent system for home automation by Wu, Liao and Fu designed in 2007. Scenarios are defined with an XML syntax and consist of a sequence of service operation invocations. MASML XML documents can embed ECMA scripts [23] to add logic elements. A mobile agent is in charge of scenario execution. This agent migrates to each appropriate device carrying the scenario description file to execute it.

3.1.4 PHS

Personal Home Server (PHS) [56] is a software infrastructure for home computing environments. Its main purpose is to reduce the complexity of embedded systems (*e.g.* TV, phone). To do so, a PHS software, destined to the control devices, discovers surrounding services and enables users to invoke them. In fact, devices that propose services spontaneously, also propose to users a HTML control page to interact with available services. Such service presentations adapt to user preferences (specified on the user device).

3.1.5 SAASHA

SAASHA [35, 34], has been designed in 2010. It focuses on ubiquitous systems for home automation. It enables end-users to create scenarios with Event - Conditions - Action (ECA) rules. SAASHA combines the agent and component paradigms. Agents are responsible for context management, scenario creation and execution. A graphical user interface also provides easy access to SAASHA mechanisms. Agents dynamically generate new components that enable the control of surrounding equipments.

3.1.6 SODAPOP

SODAPOP [26, 36] (acronym of Self-Organizing Data-flow Architectures suPporting Ontology-based problem decomPosition) proposes an innovative goal-based approach. It has been designed in 2005. For the authors, predefined scenarios cannot encompass every situations and learning systems are not efficient. Therefore, pervasive system must dynamically generate scenarios that try to reach the user's goal. The main hypothesis is that each service contains informations about its initial conditions and its effects. SODAPOP automatically classifies new services with these informations. Thanks to intention analysis [46], SODAPOP translates user interactions and context information into concrete goals. Then, it tries to compose services with Artificial Intelligence (AI) planning to map users' goals.

3.1.7 WComp

WComp [80, 17] is a model for the design of pervasive computing applications designed in 2008. It gives developers the capability to compose services. WComp is based on Web Services [57] which obey a standard and can be implemented in any language. A composite service contains proxy components attached to involved Web Services. WComp enables hierarchical service composition. In addition, it is an event-based system which adapts to environmental 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.

3.2 Service Platforms: an Approach to Context Management (R1)

Users in pervasive environments must access to device functionalities. Moreover, they need a representation of the different elements present in the environment, *i.e.* the *context*. The first requirement of a pervasive system thus is the *context management*. It demands to represent the context (R1.a) (*i.e.* enable users to get an overview of functionalities) and to enable users to manage this representation. Moreover, context management also involves to be continuously aware (R1.b) of environmental changes and to relate this according to the context representation.

The Service Oriented Architecture (SOA) paradigm [58, 63] enables to handle device functionalities as services. According to the OASIS organization, *"a service is a mechanism to enable access to one or more capabilities"* [58]. Therefore, *service oriented platforms*, specially in home automation [11, 91, 84], facilitate the use of these electronic devices through their services and are an approach to handle context management.

In this section we introduce SOA and present several existing systems based on this paradigm. These systems are widely adopted industrial standards. Finally, we analyze how the studied pervasive systems fulfill the context management requirement.

3.2.1 Concepts and Approaches

The Service Oriented Architecture paradigm [63] is based on three main entities (as illustrated by Figure 3.1):

- The *service provider*. This is the entity that furnishes the services. It is responsible for publishing the service contract and for service execution.

- The *service broker*. This entity handles the service directory. It enables service providers to register their services and service consumers to retrieve a desired service.

- The service consumer. It can search a service thanks to the service broker. Thus, if the consumer finds a desired service, it can directly invoke the service instance through the service provider.

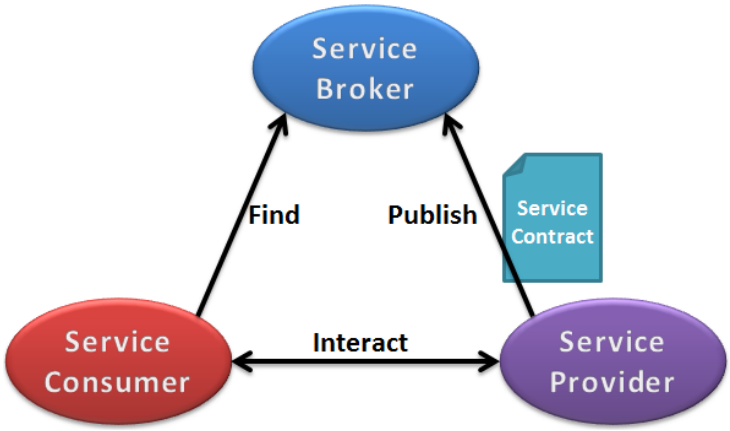

Figure 3.1: Service Oriented Architecture triangle

Some papers already analyzed the different service oriented architectures, specially [66, 60, 8, 93, 24]. We can distinguish two sorts of approaches: centralized or decentralized. Following sub-sections analyze these different approaches.

3.2.1.1 Centralized Approaches

Centralized systems have a main entity, which plays the role of the service broker defined in the service oriented architecture triangle. Their main advantage is that a centralized architecture is easily deployable and maintainable. However, it creates a single point of failure. In this section, we analyze two centralized systems based on services: Jini [5] and Salutation [69] which are two standards, representative of existing systems, and widely adopted.

Jini. Jini [5] is a network architecture, originally designed by Sun Microsystems and now developed and maintained by the Apache foundation under the *River* project. It is based on a central entity: the *Lookup Server* (LUS). The process to publish, discover and invoke services is always the same:

- Step 1. Find a LUS (discovery). Return a RMI [22] proxy for the LUS.
- Step 2. Publish services (join).
- Step 3. Find a service: request the LUS (lookup).
- Step 4. Obtain the proxy and invoke the service.

Jini is implemented in Java and makes it OS and platform independent but requires the presence of a Java virtual machine. Moreover, Jini is independent of the network transport layer. Jini introduces the possibility of code mobility which makes it protocol independent. Applications based on Jini thus are highly interoperable (Jini enables to remotely deploy a driver, for example). However, code mobility can raise security problems (*i.e.* local execution of foreign code).

Salutation. The Salutation architecture [69] is developed by an industrial consortium: the Salutation Consortium which members include IBM, Canon, Epson, etc. Its objective is to enable service discovery and utilization among a broad set of devices.

The Salutation architecture is based on the *Salutation Managers* (SLM) that have the role of service broker in the SOA triangle. The other major component of Salutation is the *Transport Manager* (TM). A SLM is deployed on a TM which provides a reliable communication channel. Several TMs that implement different networks protocol can be present. They use a transport-independent interface (SLM-TI) to communicate with SLMs. This enables the Salutation architecture to physically reach different networks simultaneously. This is transparent for the users, that use a specific interface (SLM-API) to register, discover and invoke services. Several SLM can communicate with the *Salutation Manager Protocol*. This enhances collaboration but service registries are not shared (service discovery is thus restricted to a single SLM). Figure 3.2 illustrates the Salutation architecture.

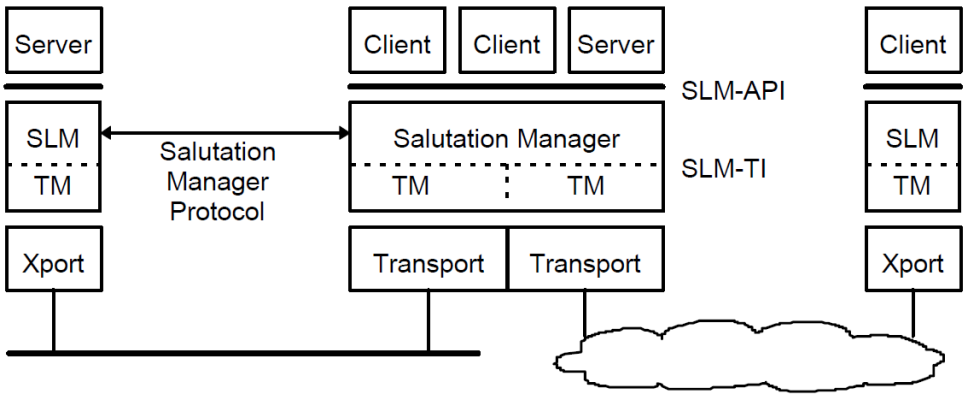

Figure 3.2: Salutation architecture extracted from [70]

3.2.1.2 Decentralized Approaches

In a decentralized architecture, devices can directly discover and interact themselves without a centralized entity that maintains the service directory. Entities in a decentralized architecture can embed their own service directory. In this section, we study three majors standards: Web Services [62], UPnP [82] and SLP [86].

Web Services. Web Services are a set of communication protocols defined by the World Wide Web Consortium (W3C), which is the main international standards organization for the World Wide Web. According to Papazoglou [62]:

> "A Web Service is a platform-independent, loosely coupled, self-contained, programmable Web-enabled application that can be described, published, discovered, coordinated, and configured using Extensible Markup Language (XML) artifacts for the purpose of developing distributed interoperable applications."

Web Services [57] are based on standards such as XML and HTTP. They are platforms and language independent. Web Services are described using the *Web Service Description Language* (WSDL). They are registered inside *Universal Description, Discovery and Integration* (UDDI) registries. Web Services exchange messages through the *Simple Object Access Protocol* (SOAP).

Web Services are largely adopted, particulary in Business to Business activities. However, they are not dedicated (and thus adapted) to volatile environments, where devices can appear and disappear frequently without it being known in advance. In addition, Web Services are not adapted for small devices that are mobile and / or that cannot implement the Internet Protocol transport layer [42].

UPnP. The origin of UPnP [82] is industrial. It is a Microsoft's original idea, then picked up by the UPnP Forum (with Sun, IBM,...). The first version of the UPnP *Device Architecture* appears in June 2000. The motivation of UPnP was to design a set of OS independent protocols, limited to sending data over the network. Moreover, another UPnP goal is to standardize current everyday-devices. It is based on industrial standards (XML, User Datagram Protocol (UDP), Transmission Control Protocol (TCP), IP, SOAP, HTTP). UPnP creates an open architecture network to discover and control services, based on four protocols:

- Simple Service Discovery Protocol (SSDP), for service discovery.
- Generic Event Notification Architecture (GENA), to handle events such as variable state changes or service appearance.
- Service Control Protocol Description (SCPD/DDD), for device (DDD) and service (SCPD) description.
- Simple Object Access Protocol (SOAP), to control devices.

The advantages of UPnP are various. UPnP does not send executable (only data) which limits its security problems. It provides a standardized device format. Additionally, it is programming language and OS independent. It enables to discover, describe and control services. Moreover, it handles events with the possibility to subscribe to variable changes. UPnP it is a *de facto* standard, present in numerous industrial devices.

However, UPnP is not adapted for small devices (because it has an imposing stack). Because there is no registry, UPnP uses heartbeat and active discovery which can cause a heavy network traffic. UPnP is also limited to the use of a single protocol (HTTP over UDP over IP). In addition, it does not define any security mechanism. Moreover, its search function is limited. There is no possibility to add attribute filter such as in the case of Lightweight Directory Access Protocol (LDAP) [29].

Service Location Protocol (SLP). SLP [86] is a standard developed by the Internet Engineering Task Force (IETF), an organization that develops and promotes Internet standards. SLP comprises three types of entities: service agents (SAs), user agents (UAs), and directory agents (DAs). The presence of DA is optional; it is useful for large networks to reduce traffic. SLP can therefore work with or without a service directory.

SLP has the advantage to be flexible, programming language and OS independent. Moreover, it is LDAP compatible (DAs can even work with LDAP directories) and there is the possibility to configure security features. SLP works with TCP and UDP and thus, is restricted to the IP transport protocol.

3.2.1.3 Synthesis

There is no set of protocols that is widely adopted and dominates the market. Each has its advantages and disadvantages. Pervasive systems must be interoperable and therefore cannot be based on a single standard (*e.g.* the exclusive use of Web Services does not encompass small devices that implement a lighter protocol such as UPnP). The solution is certainly to adopt a generic approach that can be bridged with different technologies.

3.2.2 Evaluation of Pervasive Systems

In the previous subsection, we detailed the different concepts and approaches to fulfill the context management requirement (R1). This subsection is dedicated to the analysis of the studied pervasive systems regarding this requirement.

Anamika is restricted to Bluetooth. However, service discovery is flexible (service matchmaking uses semantics and is not limited to unique service identifier). Moreover, each Anamika platform contains surrounding services description and can be requested by other platforms to share service access (R1.b). Service discovery is therefore distributed. In addition, they cannot represent the context (R1.a) as they wish (by annotating service description).

DigiHome is interoperable and can implement various service discovery protocol (R1.a) such as UPnP, Zigbee [45], etc. It separates services in two sorts: sensors and actuators. Moreover, it manages service quality levels. DigiHome objects, placed into user mobile devices, embedded user configuration files (*e.g.* users' preferences). However, this is not a real context representation (R1.a).

MASML handles service discovery which is a part of context awareness (R1.b) thanks to the use of Web Services. Moreover, MASML platforms can share access to local services. This means that a MASML platform can be deployed on a device that contains a local service (*e.g.* a platform can be deployed in a TV). If this service is not designed to be accessible remotely (*i.e.* does not implement a network protocol), the MASML platform can inform other platforms of the existence of this service and thus, be responsible of its remote invocation. This is done through SOAP messages. However, MASML users cannot represent their context (R1.a).

PHS is mainly based on UPnP. It can also integrate bridges that transform Web Services or Jini services into SOAP (used in UPnP) messages. PHS is thus interoperable and enables context awareness (R1.b). Users can embed service control preferences but cannot represent their context (R1.a) such as locations or surrounding users.

SAASHA uses UPnP to discover and invoke services. In addition, SAASHA represents the context (R1.a): the representation of device localization, device type and device owner enables users to have an overview of their context. This context representation (R1.b) is constantly updated thanks to the use of UPnP.

SODAPOP contains a context representation (R1.a), with service effects (such as impact on the environment) but this is not adapted to users (*e.g.* users and locations are not represented). Moreover, it does not enable users to customize the representation. In addition, SODAPOP discovers surrounding services (R1.b) but they must provide a specific description of their capabilities.

WComp uses Web Services to communicate. It partially treats context management (R1) because context awareness is handled but users cannot represent their context.

3.2.2.1 Synthesis

All studied systems implement a discovery protocol and therefore manage context awareness. However, to enhance interoperability, pervasive systems cannot be restricted to the use of a single protocol.

Moreover, only SAASHA provides users with a real representation of the context (device type, locations, etc.) and enables users to customize this representation. SODAPOP contains information about services and their effect, which is the beginning of a context representation but it does not consider the information necessary for users such as device locations or surrounding users. However, this representation is limited to devices that provide a UPnP service. A pervasive system must also enable users to represent surrounding users. Table 3.1 synthesizes how studied pervasive systems fulfill the context management requirement.

Pervasive Systems	Requirements	
	R1.a Context Representation	R1.b Context Awareness
Anamika	◇	◆
DigiHome	◇	◆
MASML	◇	◆
PHS	◇	◆
SAASHA	◆	◆
SODAPOP	◆	◆
WComp	◇	◆

Table 3.1: System comparison with the context management requirement

3.3 Service Composition: First Response to Scenario Definition (R2)

Service oriented platforms enable users to have access to device functionalities. However, it is still impossible for end-users, that have no specific technical knowledge, to fully benefit from the services proposed by their surrounding devices. Indeed, each device provides its own set of services and users are often limited to use a single service at a time. Alternatively, users usually want to describe *scenarios* which involve multiple services from multiple devices. The more complex user requirements can be satisfied by a *scenario* that involves a composition of multiple services provided by multiple devices.

A pervasive system must therefore enable users to define scenarios. Service composition is a first response to scenario definition (R2.a). Additionally, scenarios must be easily customizable (*e.g.* select a service from a specific device, add meta-informations), parameterizable (*e.g.* some parameter values must be defined at scenario execution and not scenario definition) and adapted to users' needs (R2.b).

In this section, we first introduce the concepts used service composition. Then we present the two main approaches: manual and automatic composition, through different languages and systems that compose services. Finally, we examine how selected pervasive systems enable scenario definition.

3.3.1 Concepts and Approaches

Service Oriented Architecture provides a flexible and adaptable paradigm to program softwares. However, the use of a basic service is limited and service composition encompasses more user needs. Service composition originally comes from workflow definition that can be compared to Petri nets [85] or state machines [74]. Nowadays, we can distinguish two mains approaches to compose services: manual or automatic.

3.3.1.1 Manual Composition

In manual composition, developers (or users) themselves select services that have to be combined. They can be helped with semantics but they keep control of service composition.

ECA Rules. Event-Condition-Action rules [31, 32] are a simple mechanism to define basic compositions. They are also used in active databases. ECA rules are composed of three parts:

- The *event* part. Its satisfaction is the composition starting point.
- The *condition* part. It is a logical test that can combine several conditions.
- The *action* part. It is composed of service invocations. They are executed if the condition remains satisfied.

ECA rules' default is linked to its main advantage: its simplicity. End-users can easily define and understand composition. However, such compositions are limited (*i.e.* parallelism is not considered, it is not possible to define hierarchical conditions, etc.).

BPEL. BPEL [44] (Business Process Execution Language) is a standard maintained by the OASIS consortium. Its origins can be attributed to the fusion of two precursors languages: WSFL [49], developed by IBM and XLANG [79], designed by Microsoft. BPEL uses an XML syntax to model a business process by composing Web Services. Such composition can be proposed as a new Web Service. BPEL is based on W3C norms such as WSDL [18] to describe Web Services, XMLSchema [87] to define data structures and XPath [19] to parse XML documents. BPEL enables to use variables and to define control structures (*i.e.* sequence/parallel, if-then-else, while loops, events).

BPEL does not have a standard graphical representation: it was not relevant for the Organization for the Advancement of Structured Information Standards (OASIS) committee[1]. This is why, the Business Process Management Initiative (BPMI) created *Business Process Model and Notation* (BPMN) [90] as a graphical front-end to define the BPEL processes. BPMN is now adopted and maintained by the Object Management Group (since 2005). BPMN specification includes a partial mapping from BPMN to BPEL. However, differences exist between these two business process modeling languages (especially with BPMN 2.0).

3.3.1.2 Automatic Composition

In automatic composition, services provide their requirements and functionalities and a composition engine is responsible for service composition depending on the goal to reach. The advances in semantic Web Services make automatic composition engines become more and more efficient. However, the absence of user control can be seen as a disadvantage and cannot ensure to achieve the desired result.

[1] some non-officials exist that do so such as the Eclipse plugin *BPEL Designer*, available at http://www.eclipse.org/bpel/

Semantic Based Engine. We can mention systems that automatically compose services based on ontologies such as SHOP2 [76]. Main ontology languages are OWL-S [50], based on DAML-S [2], and Semantic Web Rule Language (SWRL) [37], that combines OWL and RuleML [10].

Automated planning and scheduling. Automated planning considers a goal to achieve, with a set of constraints, and tries to automatically select a sequence of actions. It is based on a specification language that not only briefly define actions but also represents the action effects, preconditions, etc. This information enables to reason automatically an plan a certain sequence of actions to reach a goal. One of the most used planning language is Planning Domain Definition Language (PDDL) [30, 52]. Listing 3.1 illustrates a service description in PDDL. This service enables to close a shutter, indicated with the variable `?x`. This service has a precondition (the selected shutter must be opened to close it) and effects (the shutter is closed and luminosity is therefore low). This example illustrates how automated planning can consider effects to reach a goal. Typically, a scenario that aims to decrease luminosity automatically considers this service as interesting.

```
(:action  closeShutter
   :parameters (?x - int)
   :precondition: open(?x)
   :effects: (and (open(?x)) (luminosity() = low))
)
```

Listing 3.1: Service closeShutter declared in PDDL

3.3.1.3 Synthesis

Automatic compositions are promising. However, they are too dependent on service descriptions and we cannot make the assumption that all services present in the environment correspond to a certain expectation. Moreover, it could be adapted for a simple goal (*e.g.* increase luminosity) but it becomes limited for more complex scenarios. In addition, users might be interested to customize their scenarios (not just compose services), for example adding meta-informations, defining parameter values at runtime, etc. Alternatively, manual compositions seem more adapted for users to define a goal in a pervasive environment. However, ECA is too simple to enable to define all necessary needs whereas BPEL syntax is too rich and complicated for end-users. Graphical user interfaces exist to compose with BPEL but they refer to concepts (such as exception handling) which increase language complexity and which are not necessary for non-technical end-users.

3.3.2 Evaluation of Pervasive Systems

In the previous subsection, we presented different languages to compose services. They are a first response to scenario definition (R2). In this subsection, we analyze and evaluate studied pervasive systems regarding this requirement.

Anamika enables to compose services with DAML-S (R2.a). Such composition is based on semantics but not usable for end-users. Additionally, users cannot customize their scenarios (R2.b).

In DigiHome, scenarios are created (R2.a) as specific configuration rules for the *Complex Event Processing* (CEP). Such capacity does not enable users to define their own scenarios neither to customize them (R2.b). In fact, DigiHome is more destined to programmers and eases high-level, event programming of Wireless Sensors Networks.

MASML users use MASML language to define scenarios (R2.a). Scenarios are a sequence of service invocations. However, MASML scenarios can integrate ECMA scripts. This enables to embed logical elements in the sequence invocation, and thus, improves the possibilities to define more complex scenarios. However, MASML users cannot customize (R2.b) their runtime behavior as there is no possible scenario run-time modification.

PHS does not consider scenarios and thus, does not enable users to compose services (R2.a). Control devices in PHS can embed service control preferences. Such mechanism enables to filter surrounding services and to dynamically generate a HTML control page. However, PHS users can just customize how services are presented and cannot customize scenarios (R2.b).

SAASHA enables users to compose services (R2.a) with ECA rules. This is simple for users but offers limited possibilities. Moreover, SAASHA's users can customize scenarios (R2.b). From example by invoking a specific service from any device. However, users cannot define parameterized scenarios.

SODAPOP tries to automatically compose services (R2.a) to define a scenario that achieves users' goal. This is possible thanks to automated planning. However, it is dependent on the meta-informations that services expose. Moreover, this mechanism does not enable users to define or customize their own scenarios (R2.b).

In WComp, service composition (R2.a) is possible thanks to the Service Lightweight Component Architecture (SLCA) [38]. SLCA is a component architecture that enables event-based Web service composition. Service composition with SLCA is adaptable and reconfigurable and enables to define complex scenarios (R2.a). However, it is not adapted to end-users and does not enable scenario customization (R2.b).

3.3.2.1 Synthesis

SAASHA brings some interesting concepts to scenario definition such as executing a service independently from the provider device (with the keyword `any`), or invoking all instances of a selected service in the environment (with the keyword `all`). This is a first step to enable scenario customization (R2.b). Other mechanisms (such as scenario parameterization) must be taken into consideration to completely achieve this requirement. Table 3.2 summarizes the fulfillment of the scenario definition requirement by the studied pervasive systems.

Pervasive Systems	Requirements	
	R2.a Service Composition	R2.b Scenario Customization
Anamika	◆	◇
DigiHome	◆	◇
MASML	◆	◇
PHS	◇	◇
SAASHA	◆	◆
SODAPOP	◆	◇
WComp	◆	◇

Table 3.2: System comparison with the scenario definition requirement

3.4 Distributed Execution and Recovery Strategies: Complementary Approaches for Scenario Execution (R3)

Once a scenario has been defined, a pervasive system must manage its execution. This is our third functional requirement (R3). This implies to enable users to easily control scenarios execution (R3.a), *i.e.* to start, stop and check scenario execution. Moreover, scenario execution must be resilient and adapt to environmental changes (R3.b) such as service disappearance.

In this section we present two complementary manners to execute a scenario (*i.e.* a service composition) in a distributed environment. Then, we introduce some recovery strategies that enhance scenario execution resilience. Finally, we analyze how studied pervasive systems manage scenario execution.

3.4.1 Concepts and Approaches

3.4.1.1 Distributed Execution

Service composition can be distributively executed by two complementary manners: *orchestration* or *choreography* [68, 64]. Choreography focuses on collaboration between entities in order to reach a goal. Orchestration defines how a central or master element controls all aspects of the process.

Choreography. A choreography defines the sequence and the nature of message exchanged between the partners. This is a collaboration between all entities to reach the same goal (here, the execution of a scenario). Each partner knows its role in the scenario and can detail it. This is a more global view of the execution, detailed from an external point of view. Figure 3.3 illustrates a service choreography example. It is composed of a sequence of message exchanges between four services. Service 1 sends simultaneously a message to service 2 and another to

service 4. Each of them in turn sends and receives messages with the other partner (Service 3). Finally, service 2 answers to service 1 (message 6) and service 4 also responds to service 1 (message 7).

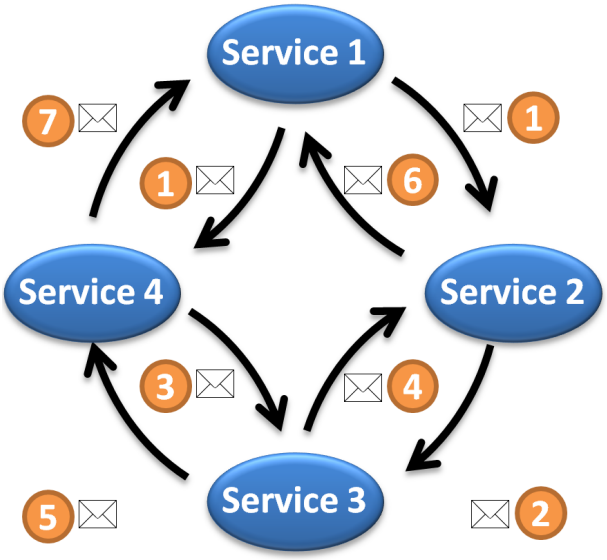

Figure 3.3: Service Choreography Example

Orchestration. Orchestration describes an internal process of an entity. The different steps that the entity must execute are detailed. It can be seen as an executable process. Figure 3.4 exemplifies service orchestration (based on the service choreography example of Figure 3.3). We can see that steps 1 and step 2 must be executed in parallel, corresponding to the exchange message order defined in Figure 3.3. Then, service 1 waits for a response of service 2 (step 3) and for another response of service 4 (step 4). The process is defined from the internal point of view.

3.4.1.2 Recovery Strategies

These strategies are based on the work of Mikic-Rakic and Medvidovic [54] who classified the most commonly used techniques for supporting disconnected operations. There are two sorts of strategies: the anticipation strategies and the repair strategies.

Anticipation strategies. Anticipation strategies consider the loss of a service before it occurs and thus, provide mechanisms to annihilate this loss.

- Caching

  Caching consists in storing locally some data that have been already retrieved. This obviates the service disappearance using the result of a previous service invocation if it is no

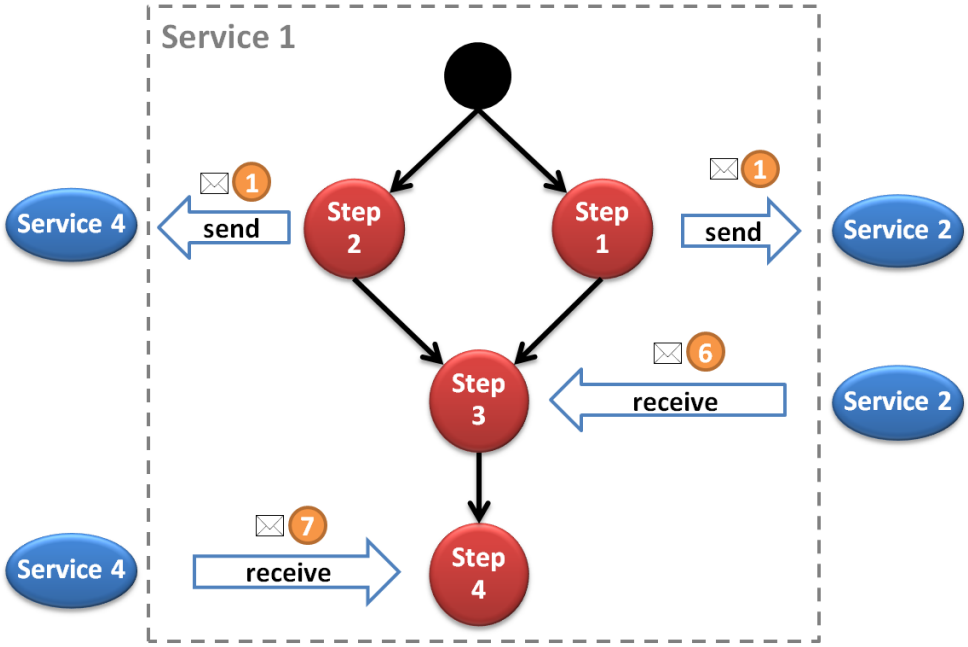

Figure 3.4: Service Orchestration Example

longer available. Because of the size of the cache, it is adapted to services that provide few information (for example a thermometer) and is useful for a service that is called many times.

- Hoarding

  This strategy anticipates the disconnection and prefetches the needed data. If a service is episodically present, it is useful to use this strategy on it.

- Replication

  If possible, a local copy of the service is done. In this strategy, the copy should be synchronized at every change of the original.

Repair strategies. Even if anticipation strategies are defined, they do not fix the problem. They just try to maintain the scenario, waiting for a solution. That is why repair strategies have been defined.

- Find the same

  The same service can be provided by another device. For example the *getTime* service can be offered by two different clocks. It is always the first strategy to apply as the simplest and the best.

- Find an equivalent

The equivalent can be the same service with different properties (for example a printer service that corresponds to a less rapid printer). It also can be an association of other services. The system always proposes to users the matching solutions that it finds and lets them choose their preferred one. If no equivalent, has been found the system allows the user to specify one manually.

- Queuing

  Queuing is not a fixing solution but enables to not lost requests. The idea is to store the requests waiting for the service return. Of course, this strategy is only effective if the service result is not needed immediately.

3.4.1.3 Synthesis

Orchestration is simple to elaborate. A single entity in charge of scenario execution enables to easily control the execution (start, stop, etc.). Indeed, the responsible entity can easily provide a checking mechanism of scenario execution advancement. However, an execution that depends on a single entity augments the risk of failure. Choreography is more complicated and sophisticated to develop but it can help reach a better performance. In addition, such execution better adapts than orchestration in a volatile environment. However, with several entities responsible for scenario execution, it becomes less evident to dynamically change the course of execution (*e.g.* stop and resume the execution). Moreover, with choreography, entities have a limited view of both the environment and scenario execution. Thus, it is more complicated to easily check scenario execution advancement. Therefore, to be able to easily control scenario execution (and check its status), execution should be managed by a single entity (that can be defined dynamically). Of course, execution responsibility must be able to be dynamically given to another. In addition, a part of the scenario must be possibly delegated to another entity.

Because of mobility and use of wireless networks some services may disappear, even for a few seconds. Therefore, pervasive systems have to handle these interruptions, even more if the disappeared service is inside a scenario being use. This mechanism should take into consideration the fact that services are composed in a scenario. They thus have interactions, can be invoked several times, have a different role, etc. Therefore, it could be interesting to implement different recovery strategies, based on the work of of Mikic-Rakic and Medvidovic [54], and to apply them depending on the service role. Typically, the *caching* strategy could be useful for a service invoked many times in a scenario.

3.4.2 Evaluation of Pervasive Systems

In the previous subsection, we summarized the different representative techniques to execute a scenario. In this subsection, we compare the pervasive systems based on the scenario control requirement (R3) fulfillment.

Anamika only enables to execute a request which is similar to start a scenario (R3.a). There is no possibilities to pause nor resume the scenario. In Anamika, scenario execution is orchestrated and several platforms can execute a part of the composition. Anamika uses a service broker for service invocation. This enhances service matchmaking. The selection of a service broker obeys to two different strategies that enable to dynamically select a broker or to distribute the execution between several brokers. Moreover, it comprises a fault-tolerance mechanism that enhances scenario execution resilience (R3.b). However, this mechanism does not take into account the possibility that the device which provides the scenario disappears.

DigiHome has a centralized architecture based on the *DigiHome core*. DigiHome objects can communicate together but all decisions are taken into the *DigiHome core*. It collects data from the DigiHome objects (that embed sensors) and decides to execute a corresponding scenario depending on the events. Such mechanism does not enable scenario users control (R3.a) but adapts to environmental changes (R3.b).

MASML's mobile agents can launch the scenario and check its execution (R3.a). However, it is impossible to pause and resume scenario execution. Moreover, MASML tries to maintain scenario execution in case of service disappearance (R3.b) by replacing the disappeared service.

SAASHA has a distributed architecture where the responsibility of scenario execution is delegated to multiple platforms. Moreover, it enables users to control scenario execution (R3.a) with start, pause and resume functionalities. However, it is impossible to check its execution. Additionally, SAASHA tries to preserve scenario execution (R3.b) from a service's disappearance by replacing the service. But, it does not anticipate the loss of a service.

With SODAPOP, users can control scenario execution (R3.a) by expressing a new goal (such as *watch TV* or *stop watching TV*). This does not enable advanced control mechanisms (*i.e.* pause, check execution). Moreover, execution is completely centralized and orchestrated. In addition, scenario execution resilience is not maintained in case of service disappearance (R3.b).

PHS does not consider scenarios. Users can easily invoke surrounding services that propose an HTML control GUI (such services can be complexes services similar to scenarios). However, they cannot define nor execute their own scenarios (R3.a). There is therefore no scenario execution resilience (R3.b).

Scenarios in WComp are composite services and can just be invoked (*i.e.* started). WComp does not provide to users other scenario control mechanisms such as pause, resume or check the execution (R3.a). WComp tries to maintain service composites in case of service disappearance (R3.b). This mechanism implements simple service replacement. Indeed, there are no anticipation strategies for service loss.

3.4.2.1 Synthesis

As a synthesis, it appears that none of the systems provide an advanced user control mechanism. It could be interesting to provide to users similar controls as SAASHA exposes (start, pause, resume, abort) with a possibility to check scenario execution advancement as MASML

Pervasive Systems	Requirements	
	R3.a User Control	R3.b Scenario Execution Resilience
Anamika	◆	◆
DigiHome	◇	◆
MASML	◆	◆
PHS	◇	◇
SAASHA	◆	◆
SODAPOP	◆	◇
WComp	◆	◆

Table 3.3: System comparison with the scenario execution requirement

does. In addition, most of the pervasive systems try to replace a disappearing service. However, enhancing scenario execution resilience also implies to anticipate service loss. A pervasive system must therefore propose fault anticipation strategies as exposed in Section 3.4.1.2. Table 3.3 summarizes the comparison between studied systems and the scenario execution requirement.

3.5 Service Component Platforms: an Architecture for the Reuse Paradigm (R4)

In prior sections, we detailed existing mechanisms of pervasive systems to manage the context, define and control a scenario. The scenario can thus be seen as an executable entity. However, users must be able to reuse (R4) a scenario. Users need to retrieve the scenario if they want to reuse it in the future. This implies to maintain scenario description availability (R4.a). Moreover, users might want to recompose a scenario into another one (R4.b).

In this section, we introduce Component-based Software Engineering (CBSE) [47] field and how it can partially answer to the scenario reuse requirement. To do so, we particularly analyze some component-based platforms that promote the use of services. Then, we study how selected pervasive systems try to enable scenario reuse.

3.5.1 Concepts and Approaches

The need for reuse in software engineering is linked to Component-based software engineering [78]. This paradigm promotes the decomposition of applications into modules called *components*. This improves, inter alia, adaptability, late-binding and reuse.

With the need of adaptability and interoperability, new platforms based on components that follow the service oriented architecture emerge. We called them *Service Component Platforms*.

3.5.1.1 Service Component Architecture

The Service Component Architecture [73] (SCA) is a set of specifications that provides a model for composing applications based on the Service Oriented Architecture. It is created by industrial partners (IBM, Oracle, etc.) and now maintained by the OASIS organization.

SCA applications are based on components that provide and / or require services. Components can be reused to create composite components. Figure 3.5 illustrates the service component architecture with a basic example. Two primitive components (Account Service and and AccountData Service) are composed into a new one. This component provides a service and requires a reference (both wired to the Account Service component).

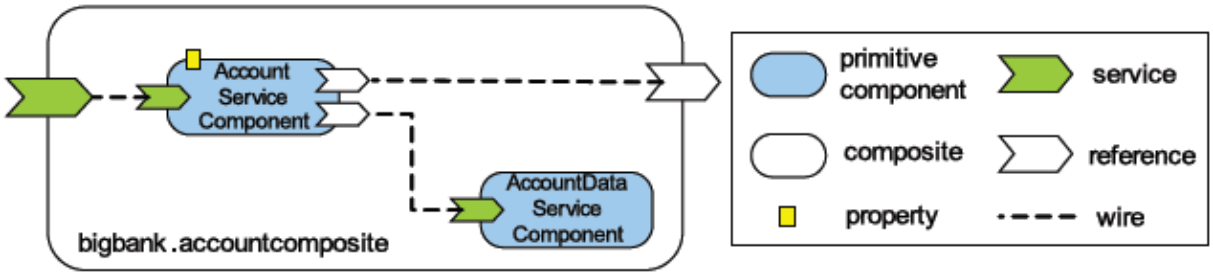

Figure 3.5: Service Component Architecture example

3.5.1.2 OSGi

OSGi [59] is a service oriented platform specification defined by the OSGi Alliance (previously known under the name *Open Services Gateway initiative*). This organization is created in 1999 (under the initiative of Ericsson, IBM, Motorola and Sun Microsystems) and the first OSGi specification release appeared in May 2000.

> "OSGi technology is Universal Middleware. OSGi technology provides a service-oriented, component-based environment for developers and offers standardized ways to manage the software lifecycle."
>
> The OSGi Alliance[2]

The platform enables the deployment of components that provide and use services. Components can be dynamically (and remotely) deployed without requiring a reboot. The platform is responsible for component life-cycle and the service registry. Component management enables to modularize applications and thus easily reuse part of it. Moreover, some frameworks based on OSGi, such as iPOJO [27], enable to easily recompose an instance component into a new component. This composition is declared into a XML file. This enables to maintain the composition's description and so, to reuse it (redeploy it) in the future.

[2] http://www.osgi.org/

Figure 3.6 illustrates how iPOJO enables to recompose existing components (Checker and English Dictionary) to create a new one (Composition 1). This composite exports the Checker service provided by the Checker component, that can be used by a GUI component. Listing 3.2 shows the composite description.

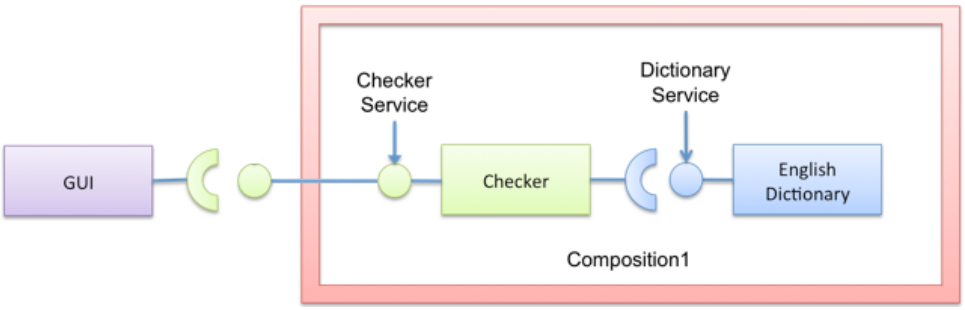

Figure 3.6: iPOJO composition architecture

```
<ipojo>
   <composite name="composition1">
      <instance component="spell.english.EnglishDictionary"/>
      <instance component="spell.checker.SpellCheck"/>
      <provides action="export" specification="spell.services.SpellChecker"/>
   </composite>

   <instance component="composition1"/>
</ipojo>
```

Listing 3.2: Composite declaration with iPOJO

3.5.1.3 FraSCAti

FraSCAti [75] is a platform that enables to design and execute SCA-based applications [73]. It is developed by the OW2 Consortium (founded by INRIA, Bull, and France Télécom in 2007). All FraSCAti functionalities and mechanisms are themselves implemented as SCA components. FraSCAti enables to develop distributed and interoperable applications (with a large support of various protocols). Moreover, FraSCAti provides runtime adaptation, *i.e.* applications can be dynamically modified to add new services or to remove existing ones.

FraSCAti applications are developed as SCA components. They can therefore be easily recomposed into new ones. Figure 3.7 illustrates the FraSCAti platform architecture based on the Figure 3.5 example. We can see how FraSCAti components act on a SCA component. The *component factory* creates the components, the *wiring & binding factory* creates wires to link composite ports and the *assembly factory* assembles components to create a composite.

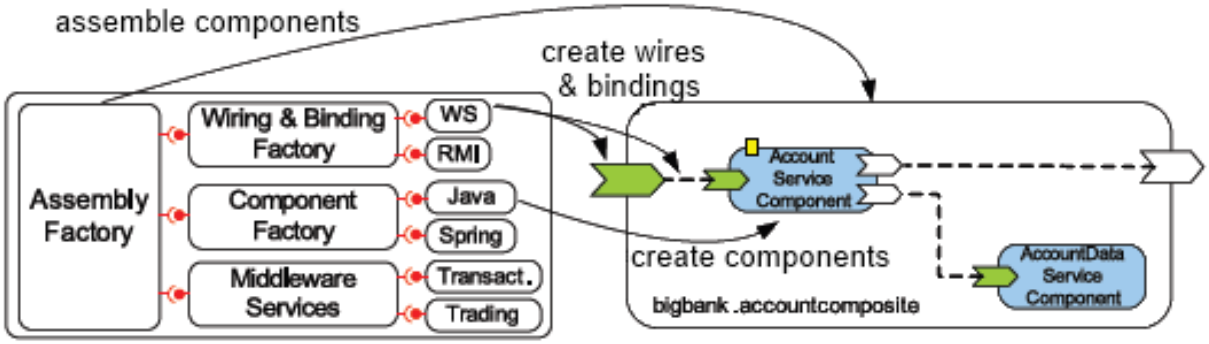

Figure 3.7: FraSCAti platform architecture

3.5.1.4 Synthesis

Service component platforms benefit from the SOA's advantages (such as adaptability, dynamicity and interoperability) and from components' faculty to be composed (*i.e.* reused) to create a higher granularity composite. Therefore, it could be interesting to define a component that manages scenario execution. This component should have a composite description such as in iPOJO. It could thus be easily redeployed (*e.g.* reuse the scenario in the future on another platform). In addition, the components lifecycle management in OSGi enables to dynamically install or uninstall a component without altering the other components. Moreover, similarly to FraSCAti and OSGi, the component can expose its functionalities as services and can therefore be hierarchically composed.

3.5.2 Evaluation of Pervasive Systems

In the previous subsection, we detailed how service component platforms enable to reuse a composite. In this subsection, we analyze the studied pervasive systems based on their scenario reuse requirement (R4) fulfillment.

Anamika is not based on a service component platform but. It nonetheless stores scenario description (R4.a) in XML files for reuse. However, Anamika does not enable scenario hierarchical composition (R4.b).

DigiHome is based on FraSCAti, which makes its architecture adaptable. However, scenarios are considered as neither SCA components nor as services which can hierarchically be recomposed (R4.b). DigiHome considers scenarios as preference rules, registered in persistent files. They can therefore be reused (R4.a).

MASML memorizes scenarios in XML documents which makes scenario descriptions persistent (R4.a). MASML scenarios, however, cannot be recomposed into other ones (R4.b).

PHS does not consider scenarios so scenario description availability is irrelevant (R4.a). Moreover, users cannot define scenarios, they cannot therefore hierarchically compose them (R4.b).

Pervasive Systems	Requirements	
	R4.a Scenario description availability	R4.b Hierarchical composition
Anamika	◆	◇
DigiHome	◆	◇
MASML	◆	◇
PHS	◇	◇
SAASHA	◆	◇
SODAPOP	◇	◇
WComp	◆	◆

Table 3.4: System comparison with the scenario reuse requirement

SAASHA attaches scenario to user profile. Thus, scenario description is registered and persistent. Moreover, user profile is duplicated on several platforms of the environment. Therefore, if the original device where a scenario has been defined disappears, the scenario remains available in the environment (R4.a). However, SAASHA does not consider to invoke a scenario into another one. Thus, hierarchical composition of scenarios is impossible (R4.b).

Scenarios in SODAPOP are an automatic composition to reach an ephemeral goal. Therefore, scenario definitions are not persistent and thus, scenarios cannot be reused in the future (R4.a). Moreover, scenarios in SODAPOP are dynamically executed but not deployed in the environment for reuse. Therefore, scenarios cannot be hierarchically composed with SODAPOP (R4.b).

In WComp, scenarios are service composites. They can therefore be recomposed (R4.b) in other ones. Moreover, a composite service provides an interface to retrieve its internal composition which is the scenario description. However, if the device where the service composite is deployed disappears, the scenario description is not maintained in the environment.

3.5.2.1 Synthesis

This study demonstrates that having a scenario description file (such as for MASML, SAASHA, Anamika and DigiHome) is a response to scenario description availability requirement (R4.a). Thus, users can easily retrieve a scenario previously created. Moreover, this description must be duplicated (as for SAASHA) to remain available into the environment. To enable hierarchical composition (R4.b), WComp adopts an interesting approach, the *composite service*. A scenario deployed as a service is thus recomposable into a new one. Table 3.4 summarizes this part of the related work study.

3.6 Access rights and Remote Invocation: Solutions for Selective Sharing (R5)

Pervasive environments can be populated by several users. These users are mobile, and thus, can come and go without being able to predict their presence (or absence). Though, users must be able to share scenarios (R5) among users. For example, in a family, if one of the parents define a scenario, the other one must be able to control it. However, users might be interested to select who to share with (R5.a). Typically, in the same example, children should have a restricted access to the scenario. Moreover, users must be able to select what to share, the scenario description or its execution (R5.b).

In this section we first introduce the access rights mechanism, to select who to share with. Then, we present some remote invocation methods, which enable to access to a distant resource.

3.6.1 Concepts and Approaches

Sharing a resource is an old problem treated by computer science. Defining some permissions to selected users enables to control who has access to resources and how. This process can be stated to access rights definition. Moreover, pervasive systems must enable to reach a distant resource (*e.g.* a scenario defined on a device must be controllable from another one). This aspect is possible thanks to *remote invocation*.

3.6.1.1 Access Rights

Defining access rights in computer science began with multi-user systems (such as operating systems). The basic solution is to assign to a user (or a group of users) permissions (*e.g.* Read, Write, Execute, and Delete for a filesystem). *Role-Based Access Control* (RBAC) [28] introduces the concept of roles to enhance and simplify the management of permissions. Sandhu et al. define the roles in [71]:

> *"Roles are closely related to the concept of user groups in access control. However, a role brings together a set of users on one side and a set of permissions on the other, whereas user groups are typically defined as a set of users only."*

RBAC has been formalized in 1995 by David Ferraiolo and Rick Kuhn. Since that time, it has become the main model for defining access rights.

3.6.1.2 Remote Invocation

The development of networks in the 1980s introduced the need to invoke a process (typically a method) in another computer on a shared network. Bruce Jay Nelson introduced the term *Remote Procedure Call* [9] (RPC) in 1982 during his PHD. Some notable implementations are the *Open Network Computing Remote Procedure Call* (ONC RPC) defined by Sun (previously

named *Sun RPC* [53]) and the Java remote method invocation (Java RMI) [22]. Systems presented in Section 3.2 such as Jini or UPnP, that provide discovery protocols, also enable remote invocation.

3.6.1.3 Synthesis

RBAC integrated in a pervasive system would enable to select users to share scenarios with. Moreover, both scenario description and execution must be provided by remote invocation. Therefore, users can easily select users they want to be able to access their scenarios and assign them permissions, *e.g.* access either to scenario description or scenario execution, or both.

3.6.2 Evaluation of Pervasive Systems

In the previous subsection, we presented two different approaches for selective sharing. In this subsection, we evaluate the studied pervasive systems regarding the scenario sharing requirement (R5).

Anamika can distribute scenario execution and thus, transmit a part of the scenario description to another platform but it cannot be assimilated to a sharing mechanism. Indeed, users cannot select other users to share scenarios with (R5.a). Moreover, neither scenario description nor execution can be shared in the environment (R5.b). Therefore, we can consider that Anamika does not consider scenario sharing.

Scenarios in DigiHome are preference rules attached to a single device. DigiHome considers that several devices can be present and therefore tries to avoid conflicts between scenarios. However, there is no mechanism to share neither a scenario description file nor its execution (R5.b). Moreover, users cannot select surrounding users to share a scenario with (R5.a).

MASML scenarios can migrate from a device to another but to invoke services present on the latter device, not to be controllable with this device. Moreover, MASML does not propose any sharing mechanism.

PHS does not consider scenarios and then, does not enable users to select who to share scenarios with (R5.a). It does not integrate the possibility to choose what to share either (R5.b).

SAASHA enables to define users and their attached role (R5.a). Roles enable to define device access (given users might have a limited access to certain device services). Therefore, a user that connects to a SAASHA platform must login and gains access to devices that his/her role allows him/her to. Such configuration is shared among all SAASHA platforms present in the environment. Moreover, a scenario defined by a user is attached to his/her configuration and so, also shared. A user can thus access a scenario he/she defined on another platform. However, this is not a configurable mechanism as scenarios are shared like this, with both description and control. Moreover, it is impossible to share a scenario with other users (R5.b). SAASHA users must know the user's login and password to connect to a platform and have access to the scenario.

SODAPOP tries to execute a user request (such as watch TV). Therefore, it does not consider

Pervasive Systems	Requirements	
	R5.a Select what to share	R5.b Select who to share with
Anamika	◇	◇
DigiHome	◇	◇
MASML	◇	◇
PHS	◇	◇
SAASHA	◆	◆
SODAPOP	◇	◇
WComp	◇	◆

Table 3.5: System comparison with the scenario sharing requirement

surrounding users and thus, does not enable to select surrounding users to share a scenario with (R5.a). Moreover, scenarios are automatically composed to be dynamically executed when the user does the request. Scenarios in SODAPOP are not considered as sharable entities, neither their description nor their execution (R5.b).

WComp considers scenario as a composite service. This composite provides two interfaces: one to retrieve the composite service's internal description, and another to dynamically interact with the composite service. The composite does not check the requester identity, thus, it is impossible to select who to share with (R5.a). However, surrounding users can access to the composite service description (R5.b). In addition, WComp users cannot control scenario execution. Therefore, WComp does not enable scenario execution sharing.

3.6.2.1 Synthesis

Having access to a scenario from another device is possible with SAASHA. This is possible thanks to user preferences (that contain users' scenarios) which are shared in the environment. WComp deploys the scenario as a composite service in the environment. It is thus shared with everyone. It could be interesting to create categories (*e.g.* a user category such as Jimmy's devices) and attach permissions to this category to enable users to share their scenarios with other ones. Table 3.5 summarizes how studied pervasive systems treat the scenario sharing requirement (R5).

3.7 Non-Functional Requirements (RA-D)

In the previous sections we studied concepts and approaches to fulfill the functional requirements. Then we analyzed how studied pervasive systems try to respond to each requirement. This section is dedicated to the non-functional requirements (defined in Section 2.2.2), that affect functional requirements. In next subsections, we present each non-functional requirements

and analyze mechanisms proposed by pervasive systems for this requirement.

3.7.1 User Friendliness (RA)

3.7.1.1 Presentation

Users in pervasive environments are not necessarily technical experts. Nevertheless, they have a goal. Therefore, pervasive systems must help them to easily reach this goal. It implies to represent surrounding users, enable to easily define scenarios, provides users with means to control, enable to easily reuse a scenario in the future or from another scenario and to share scenarios. Moreover, a dedicated graphical user interface highly improves user friendliness.

3.7.1.2 Pervasive Systems Analysis

Anamika does not enable users to easily represent their context such as locations. Moreover, scenario definition is possible thanks to DAML-S which is not accessible for end-users. Additionnaly, there is no mechanism to easily recompose a scenario or share it. However, Anamika provides a graphical user interface to easily start already defined scenarios.

DigiHome is dedicated to developers. Therefore, it does not consider any mechanism to facilitate users access to functionality. Users cannot easily represent their context. Scenarios in DigiHome are configuration rules for Complex Event processing. These are too complex for users. Moreover, scenarios can be started but there is no other execution control mechanism such as pause or resume.

MASML enables scenario definition with ECMA scripts that are not adapted to users without technical knowledge. It does not provide user control of scenario execution. Scenarios are registered in XML files and are thus easily reusable. However, it is impossible to recompose them. Finally, no mechanism dedicated to scenario sharing is provided.

PHS is based on services that propose a HTML control page. Users can therefore easily retrieve surrounding functionalities and invoke them. However, control of pervasive environments is limited with this single mechanism. Users can neither represent their context, nor define, reuse and share scenarios.

SAASHA is dedicated to users and thus, proposes a dedicated graphical user interface. This interface is specially designed to represent the context, define and control scenarios execution in a user-friendly way. SAASHA enables users to easily control a pervasive environment. However, there are limitations to scenario definition (due to the use of ECA rules), scenario reuse and sharing.

SODAPOP functionalities are almost entirely automatic processes, so users can only interact with the system by expressing their needs (which is easy but limited). Typically, a user can easily define a goal to achieve but there is no possibility to customize it. Moreover, users can control scenario execution by choosing actions (such as watching TV or stop watching TV). However, it is impossible to check scenario execution advancement. Additionally, users with SODAPOP cannot reuse or share a scenario.

Systems	RA User Friendliness
Anamika	◆
DigiHome	◇
MASML	◇
PHS	◆
SAASHA	◆
SODAPOP	◆
WComp	◇

Table 3.6: System comparison with the user friendliness requirement

WComp is not user-friendly since defining a scenario is only possible thanks to SLCA, which is powerful but not adapted to end-users. Moreover, there is no mechanism for users to control scenario execution. Scenarios in WComp are composite services. They can therefore only be hierarchically recomposed with SLCA (which is not user-friendly). Similarly, scenarios are accessible (and thus sharable) through interfaces dedicated to developers only.

3.7.1.3 Synthesis

SASHAA proposes some easy mechanisms to represent the context and define scenarios. However, they are limited (*i.e.* use of ECA rules to define scenarios cannot encompass all user needs). It could be interesting to propose a simple language that enables to represent the context (users, locations, etc.) and to define scenarios (by composing services). Moreover, a dedicated user interface to provide a better access to the pervasive system mechanisms seems mandatory. Table 3.6 summarizes how studied pervasive systems treat the user friendliness requirement.

3.7.2 Collaborativeness (RB)

3.7.2.1 Presentation

Several users can share the same pervasive environments. Pervasive systems must thus handle *collaborativeness*. They must therefore consider surrounding users in the context representation, enable to customize a scenario depending on a user, handle several users simultaneously accessing a given scenario and provide sharing mechanisms.

3.7.2.2 Pervasive Systems Analysis

Anamika enables several platforms to collaborate by sharing surrounding services. This is implemented in Anamika's advanced service discovery system. However, Anamika does not provide any mechanism to represent surrounding users nor share scenarios.

DigiHome comprises a user manager which enables to determine different roles for users (*e.g.* rights to invoke a service operation such as activation of the sprinklers). Moreover, DigiHome considers that several platforms (called DigiHome objects) can communicate together through different protocols with the REST interfaces. However, this is not really collaboration considering that platforms only provide their local services to the environment. Users cannot have a representation of surrounding users nor share scenarios.

MASML enables platforms to share their local services. Platforms therefore collaborate to obtain a better representation of their environment and access services that cannot be directly reached. However, MASML users cannot represent surrounding users nor share a scenario with them.

A PHS system is deployed on a mobile device and embeds user preferences. PHS considers that several users can share the same environment and that they must have a personal way to control the pervasive environment. However, this mechanism does not enable collaboration between users as it cannot represent surrounding users nor share scenarios.

SAASHA enables to represent surrounding users and to attach them corresponding devices. Moreover, SAASHA handles the multi-user aspect by providing access right policies. Some users have a limited access to some services (*e.g.* a child cannot invoke the security system service). Moreover, different access rights (on the same device) enable to define priority scenarios. However, it still is impossible to select surrounding users to share scenarios with.

SODAPOP does neither represent surrounding users, nor provide multi-user access to scenarios. Additionnaly, SODAPOP platforms are not collaborative (they do not share service access).

WComp does not provide mechanisms to handle multi-user support. Typically, it is impossible to represent surrounding users. Moreover, scenarios can be redeployed in the environment but it is not possible to select who to share with.

3.7.2.3 Synthesis

The idea of SAASHA to define surrounding users and to attach them a device is interesting. However, this representation depends on the presence of devices. It could be interesting to make this presentation persistent. Moreover, systems (such as MASML and Anamika), that share local services, enhance collaborativeness. Table 3.7 summarizes the analysis of the collaborativeness requirement fulfillment by the studied pervasive systems.

3.7.3 Adaptability (RC)

3.7.3.1 Presentation

Pervasive environments are volatile, they can quickly change. Users and / or devices can appear and disappear without prior knowledge. Context management should therefore adapt to context evolution and enable to discover services without prior knowledge. Users must be able to define adaptable (*i.e.* customizable, parameterizable) scenarios. Scenario execution should adapt to

Systems	RB Collaborativeness
Anamika	◆
DigiHome	◆
MASML	◆
PHS	◇
SAASHA	◆
SODAPOP	◇
WComp	◇

Table 3.7: System comparison with the collaborativeness requirement

available devices (select appropriate services to execute the scenario) and to environmental changes (such as service disappearance). Scenario description availability must be resilient, even if the provider device disappears. Finally, scenario sharing needs to adapt to changes of the context.

3.7.3.2 Pervasive Systems Analysis

Anamika proposes an advanced service discovery mechanism that enables to adapt to different and scalable environments. Moreover, scenario execution comprises dedicated service brokers that enable to adapt to available services. In addition, a fault-tolerance mechanism adapts scenario execution to service disappearance. However, Anamika does not consider scenario description resilience if the scenario provider disappears. Additionally, Anamika does not adapt to users' mobility: scenario execution is not possible in several places.

DigiHome is based on FraSCAti and, therefore, can implement various network protocols to adapt to different environments. Moreover, DigiHome platforms propose their services through standard protocols and are therefore not restricted to DigiHome platforms. In addition, DigiHome platforms embed a reconfiguration engine that adapts the internal architecture depending on available services. However, scenario execution does not adapt to the scenario provider device disappearance. Additionally, scenario description availability is never maintained if the scenario provider device disappears.

MASML uses Web Services to discover services without prior knowledge. Moreover, MASML tries to preserve scenario execution if an involved service disappears. However, users cannot define parameterizable scenarios that enable to adapt to users' needs. Users can retrieve a scenario description for reuse. However, MASML does not preserve scenario description availability if the provider device disappears.

PHS can discover services deployed in UPnP, Web Services, Jini and even X10. It is therefore interoperable and adapts to surrounding services. Moreover, PHS enables users to define preferences. The system therefore proposes to users an adaptable control interface. However, PHS does not consider scenarios and thus, adaptability of scenario execution, reuse and share

Systems	RC Adaptability
Anamika	◆
DigiHome	◆
MASML	◆
PHS	◆
SAASHA	◆
SODAPOP	◇
WComp	◆

Table 3.8: System comparison with the adaptability requirement

is irrelevant.

In SAASHA, context awareness is not really adaptive (limited to UPnP devices), however appearance and disappearance of unknown devices is handled. SAASHA enables to define scenarios with service invocations independently from the provider device (which enhances adaptability). Moreover, SAASHA tries to maintain scenario execution in case of service disappearance by selecting a similar device (device of the same type). In addition, SAASHA makes scenario descriptions persistent even if the provider device disappears. Therefore, we can conclude that SAASHA almost handles adaptability.

SODAPOP dynamically discovers services. However, these services must provide a description of their capabilities. Instead, they cannot be considered for integrating a scenario. Moreover, scenario execution does not dynamically adapt to environmental changes. In addition, scenario description availability is not resilient and scenario sharing is not possible. SODAPOP is thus not adaptable.

WComp enables context awareness with no prior knowledge. Moreover, it handles scenario execution adaptability by recomposing the composite service depending on available services. However, it is impossible to define adaptable scenarios (runtime scenario customization). Additionally, defining adaptable scenarios is not possible.

3.7.3.3 Synthesis

All the pervasive systems studied try to adapt their context representation and scenario execution to context changes (except for PHS). Some of them preserve scenario description availability or enable to define adaptable scenarios. For example, SAASHA and Anamika propose to invoke services independently from the provider device. However, there are some lacks such as preserving scenario sharing in case of device disappearance. Table 3.8 summarizes our systems comparison based on the adaptability requirement.

3.7.4 Mobility (RD)

3.7.4.1 Presentation

Pervasive environments are populated by users and devices. These two entities are mobile. Context management thus require representation of device and user mobility. Users must be able to define mobile scenarios (with services present on different locations, *i.e.* not simultaneously available). Therefore, scenarios have to be executable in different times at multiple locations. Moreover, scenario description must be mobile between surrounding platforms and scenario sharing requires to handle user and device mobility.

3.7.4.2 Pervasive Systems Analysis

Anamika does not enable users neither to represent several places nor to memorize services encountered while users move. Moreover, a scenario cannot be executed if all services are not available simultaneously. Mobile scenario execution is therefore not possible. In addition, scenario description is not mobile and users cannot share scenarios with mobile users.

DigiHome platforms can be embedded in mobile devices and therefore follow user moves. However, it is not possible to represent different locations. Moreover, definition or execution of mobile scenarios is not considered by DigiHome.

MASML agents are mobile. They can migrate from a device to another. MASML agents have a service database and can thus retrieve and execute a service remotely (*e.g.* in another room). But, these services are simultaneously available. MASML does not enable to define a scenario with services that are not simultaneously available. Thus, mobile execution of a scenario (discovering and invoking new services on the fly) is not possible. Moreover, users cannot represent several locations, nor share a scenario with mobile users.

A PHS system is embedded in a user's control device. It is therefore mobile and adapts to different locations. However, users cannot compose services in different locations or execute a mobile scenario.

Scenarios in SAASHA are mobile. They can migrate from a platform to another. However, this process only serves to maintain the scenario description in the environment. SAASHA users cannot compose services that are not simultaneously available. Moreover, scenario execution in different times at multiple locations is not considered.

SODAPOP users can express a goal at a certain moment. The system tries to achieve this goal with available services. SODAPOP does not consider that users are mobile and can encounter services in different locations. Thus, SODAPOP service compositions are not mobile. Moreover, users cannot represent the different environments where they can evolve. Therefore, SODAPOP does not fulfill the mobility requirement.

WComp handles service mobility (service appearance or disappearance). However, it is not possible to define mobile scenarios (with services available on multiple locations). Additionally, scenario execution is not expected to be realized on multiple locations. Scenario sharing is not possible with mobile users.

Systems	RD Mobility
Anamika	◇
DigiHome	◆
MASML	◆
PHS	◆
SAASHA	◇
SODAPOP	◇
WComp	◆

Table 3.9: System comparison with the mobility requirement

3.7.4.3 Synthesis

Several systems can be embedded in mobile devices (such as DigiHome or WComp), but none of them considers that the user can successively be in different locations that must be represented in the system. Moreover, none of the studied pervasive systems enables to define a scenario than can be executed in multiple places (with services that are not simultaneously available). A pervasive system must therefore provide users with a better context representation, enable users to define a scenario with services on different locations and handle mobile scenario execution. Table 3.9 summarizes this study.

3.8 Synthesis and Conclusion

In this section, we analyzed various systems to build applications for pervasive environments. None of these studied systems fulfills all the requirements we defined in Section 2.2. Most of them enable to discover services, compose services define a scenario and handle scenario execution, which can be seen as the minimum for pervasive environment control. Moreover, some functional requirements (such as scenario sharing or hierarchical scenario composition) are almost completely ignored. Table 3.10 summarizes this state of the art study.

The rest of this thesis describes our SaS system which best meets all the expectations of user-centric systems for pervasive environments. Our contribution is divided into three parts to be better presented. The first part introduces a brief overview of the SaS system, and presents how SaS handles context management and enables scenario definition. The second part details scenario installation, and how users can easily control, reuse and share scenarios. The third part is dedicated to mechanisms that manage scenario execution.

Systems	Functional requirements									
	R1. Context management		R2. Scenario definition		R3. Scenario execution		R4. Scenario reuse		R5. Scenario sharing	
	(a)	(b)	(a)	(b)	(a)	(b)	(a)	(b)	(a)	(b)
Anamika	◇	◆	◆	◇	◆	◆	◆	◇	◇	◇
DigiHome	◇	◆	◆	◇	◆	◆	◆	◇	◇	◇
MASML	◇	◆	◆	◇	◆	◆	◆	◇	◇	◇
PHS	◇	◆	◇	◇	◇	◇	◇	◇	◇	◇
SAASHA	◆	◆	◆	◆	◆	◆	◆	◇	◆	◆
SODAPOP	◆	◆	◆	◇	◆	◇	◇	◇	◇	◇
WComp	◇	◆	◆	◇	◆	◆	◆	◆	◇	◆

Systems	Non-Functional requirements			
	RA. User Friendliness	RB. Collaborativeness	RC. Adaptability	RD. Mobility
Anamika	◆	◆	◆	◇
DigiHome	◇	◆	◆	◆
MASML	◇	◆	◆	◆
PHS	◆	◇	◆	◆
SAASHA	◆	◆	◆	◇
SODAPOP	◆	◇	◆	◇
WComp	◇	◇	◆	◆

Table 3.10: System comparison with our requirements

Part II

Contribution

Chapter 4

From Services to Scenarios

Contents

In this chapter, we introduce our contribution, a pervasive system which responds to the requirements previously established in Section 2.2. As we explain in the previous chapter, the goal of a pervasive system is to enable users to control pervasive environments. To do so, users must express their needs, and we consider scenarios as the representation of users' needs. Therefore, scenarios are a central entity of a pervasive system. Moreover, services populate pervasive environments and are the interface between users and devices' functionalities. They can be invoked or composed and constitute the main constituents of scenarios. Deploying scenarios as services enables to make scenarios accessible in the environment as if they were simple services. This mechanism is an originality of our system named SaS which stands for *Scenarios as Services*.

This chapter is organized as followed. A first section introduces SaS and its main functionalities: definition, control and sharing of scenarios. This is a brief overview to see how SaS

integrates in a pervasive environment. Then, we deeper explore SaS's mechanisms. Section 4.2 presents how SaS manages the context. It details how SaS is aware of the system's surrounding context and how it provides users with means to describe it. Then, Section 4.3 presents how users can define scenarios with SaS thanks to a dedicated scenario description language named SaS-SDL.

4.1 Overview of SaS

The purpose of SaS, which stands for *Scenarios as Services*, is to fulfill all the requirements presented in Section 2.2. This section presents how SaS is set in its environment, and a brief overview of SaS's main functionalities.

4.1.1 SaS Software in its Environment

Figure 4.1 shows the class diagram of SaS (in grey) and how it extends the class diagram of pervasive environments (in white, described in Figure 2.1). Pervasive environments involve electronic devices. As detailed in Section 2.1.2, we define two types of devices: simple devices (*e.g.* radiator, light) and control devices (*e.g.* laptop, PDA) which have an advanced user interface (*e.g.* touch screen), and can be considered as personal and mobile.

SaS is a pervasive system that enables users to control a pervasive environment. The SaS *system* (which contains all SaS mechanisms) is a software deployable on an electronic device that may benefit of the device capabilities such as a network access. The SaS system can therefore only be deployed on a control device to constitute a SaS *platform*.

Every SaS platform has a unique identifier and enables its user to specify his/her name (one platform = one user name)[1]. Such information is available for surrounding platforms. A SaS platform participates in one or more networks which constitute the platform's *environment*. This environment is represented on the SaS platform by the SaS *context directory*. It comprises the service directory, the platform directory and the scenario directory. These directories enable users to memorize encountered services and platforms and the created scenarios. Platforms, services and scenarios can be grouped (into their own directory) with name categories.

4.1.2 Scenario Creation and Deployment

The SaS system is deployed on a user's control device, and enables, inter alia, to create and deploy scenarios. Figure 4.2 illustrates the necessary steps from the user perspective to do so. This figure does not illustrate scenario execution and sharing, which are described in the next subsections.

[1] As a perspective, we plan to enable multiple user accounts on a single platform.

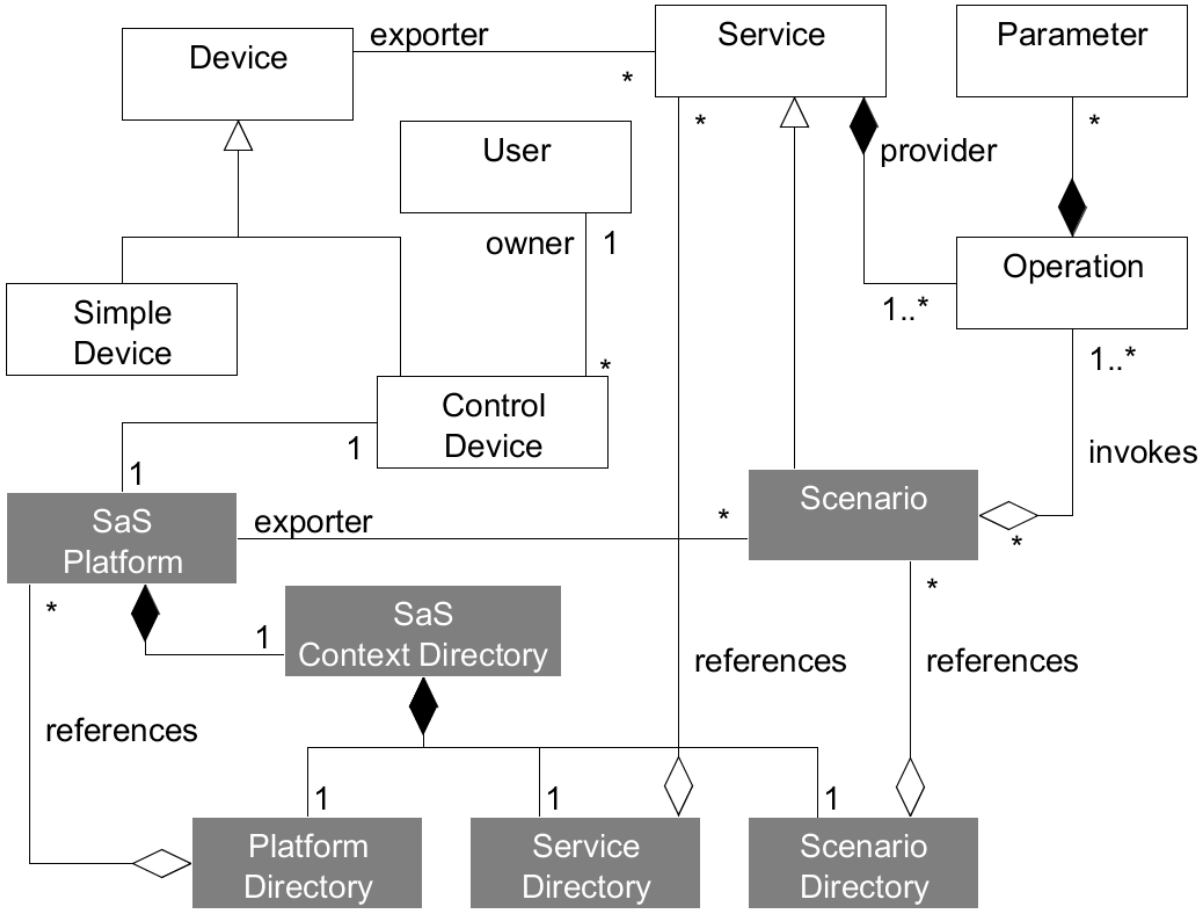

Figure 4.1: Class diagram of SaS in pervasive environments

- Context Awareness. The SaS platform first needs to be aware about its environment. This implies to discover the surrounding services, and represent the context. Context Awareness is detailed in Section 4.2.
  - *Service discovery:* SaS discovers the services available in its environment. Service descriptions that come from various service discovery protocols are homogenized and services are automatically redeclared with our scenario description language (SaS-SDL).
  - *Context representation:* SaS extracts from the discovered services, the different environment elements (devices, SaS platforms) and displays the context representation to the user.
- Scenario creation. With the list of available services, users can create a scenario.
  - *Scenario definition:* Users can define scenarios thanks to SaS-SDL by composing services.Section 4.3 is dedicated to the scenario definition.

  - *Scenario memorization:* A SaS-SDL scenario definition is memorized into a scenario description file for future use. It therefore becomes easily reusable by and transmissible to other platforms and users. Section 5.1.1 presents the scenario description memorization.

- Scenario deployment. Once the scenario has been created, the user can decide to deploy it. The scenario thus becomes executable.

  - *Scenario analysis:* SaS then analyzes the scenario description file. It extracts information about the different services involved and how they are composed. Section 6.2 details the scenario analysis.
  - *Scenario execution scheduling:* From this analysis, SaS computes the scenario execution algorithm. A scenario *orchestrator* is dynamically generated. It is responsible for scenario execution control availability and will enforce the scenario execution life cycle. Section 6.1 presents how SaS schedules the scenario execution.
  - *Scenario registration:* Finally, the orchestrator deploys the scenario as a new service. The newly created scenario then becomes accessible as a service and can even be hierarchically composed into a new scenario. Section 5.1.3 details the SaS's mechanisms to register a scenario as a service.

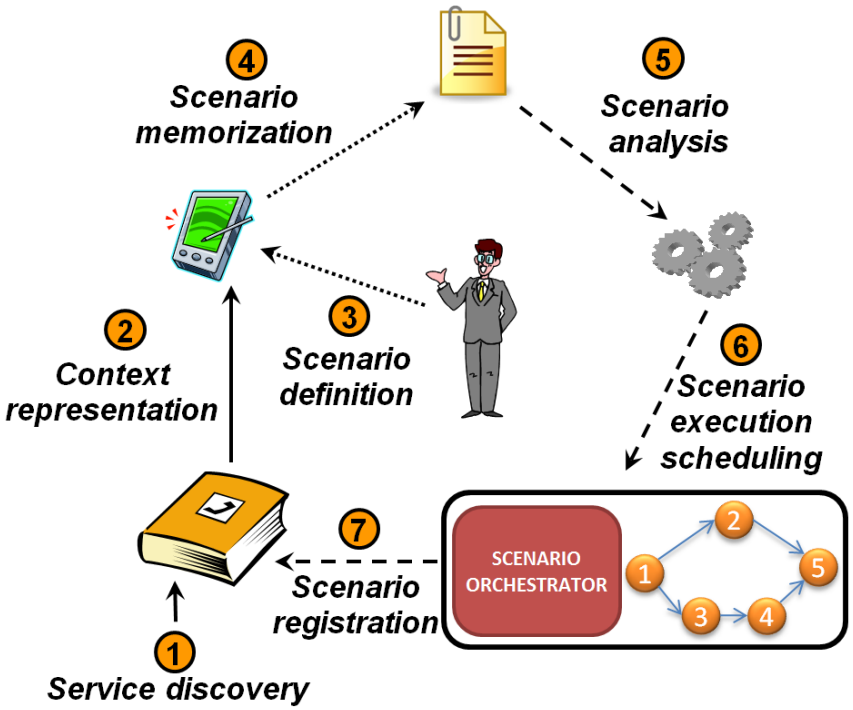

Figure 4.2: Overview of the proposed SaS scenario creation and deployment cycle

4.1.3 Scenario Execution

SaS enables users to easily control scenario execution. It provides functionalities to start, pause, abort or check the status of a scenario execution. In addition, the scenario orchestrator dynamically generated comprises a scenario execution algorithm. Thus, scenario execution is split into parts that can run in different locations at different moments. Moreover, SaS enhances scenario execution resilience by providing some fault-tolerance mechanisms. Therefore, the scenario orchestrator enables scenario execution to adapt to environmental changes. Section 6.2.2 is dedicated to the scenario execution.

4.1.4 Scenario Sharing

SaS enables to share scenarios among SaS platforms (and thus with other users). However, users might not all have identical access rights. SaS proposes several sharing modes adapted to different situations. Section 5.2 details the SaS's scenario sharing mechanisms.

4.2 Context Management

As detailed in Section 2.2, context management is the first requirement of pervasive systems. A pervasive system must be aware about its context. This implies to discover surrounding services. However, several protocol exist that do so (*cf.* Section 3.2) and a pervasive system cannot be restricted to a single one. Moreover, the context is also characterized by the presence of devices, platforms, users. SaS must extract this information from the service discovery and propose to users a context representation. This context representation must also be pervasive. This enables users to keep a representation of various services or users that they can encounter in different locations and that are not always available. Additionally, services can be attached to a location (*e.g.* kitchen lights) or a specific use (*e.g.* multimedia). Users must be able to personalize their environment representation as they wish.

4.2.1 Context Awareness

Context awareness implies to discover services and to continuously listen for environmental changes. The result should be an homogenized service directory. This is a twofold automatic process handled by SaS. First, SaS discovers available services. Then, SaS declares discovered services with SaS-SDL in SaS's service directory. This is a transformation process from a service instance discovered to a description written in SaS-SDL. Users then query this directory to select services and build their scenarios.

4.2.1.1 Service Discovery

In pervasive environments, devices provide functionalities through services. These services can be discovered thanks to service discovery protocols (SDP). Some protocols already exist that do so (*e.g.* Simple Service Discovery Protocol (SSDP) [82] implemented in UPnP, SLP [8], Jini [5]) along with extra functionalities. To be as interoperable as possible, SaS does not prescribe the use of a particular SDP. SaS uses an abstract API that can be easily mapped into any chosen concrete protocol implementation. Thus, SaS is not restricted to a single SDP. Moreover, it can evolve and adapt to any future SDP.

Context awareness is not restricted to the discovery of services at a certain moment. SaS is also continuously listening for service events (appearance, disappearance and modification). This makes it possible to maintain updated a directory of available services.

4.2.1.2 From Service Instances to an Homogenized Service Directory

Each SDP uses a service description syntax which can be specific. Typically, Web Services Dynamic Discovery (WS-Discovery) enables the discovery of Web Services described with Web Services Description Language (WSDL), SSDP discovers UPnP services, etc. To not be restricted to a single discovery protocol, we define a *generic service syntax*. Thus, service instances discovered with SDPs are automatically translated in our syntax to be dynamically published in an homogenized service directory. Therefore, the service description syntax in SaS-SDL can be seen as a protocol-independent language for service description.

Service description Syntax. Service description syntax in SaS-SDL is generic. It contains elements that we think important for users to know about the service. Figure 4.3 depicts the class diagram of the SaS service declaration syntax. Namely, services are provided by a *device* (*e.g.* the `SwitchPower` service is exported by the `Kitchen_Lights` device). They export one or several *operations* (*e.g.* `SetTarget` or `GetTarget`). Operations have a return type (which can be *void*) and can have typed parameters. Additionally, services can have *properties* (*e.g.* `deviceType = BinaryLight`).

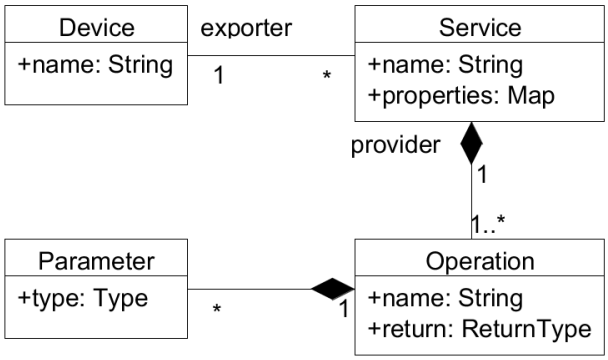

Figure 4.3: Class diagram of SaS service description syntax

Listing 4.1 presents the service description grammar of SaS-SDL. Listing 4.2 is a `Light` service description example. We can see that the service description is easily readable. Moreover, SaS-SDL comprises few object types (*i.e.* String, Number, Boolean, Date and Object), once again to facilitate the user service description comprehension. Typically, an end-user may not make the difference between different numbers type such as integer, float, double, etc. Identifiers in SaS-SDL are similar to various programming languages such as Java. They must be composed of letters, numbers, the underscore _ and the dollar sign $. Identifiers may only begin with a letter, the underscore or a dollar sign.

```
<service> ::= service <service_name> : <device> [<service_prop>] <op_list>

<device> ::= device <device_name> ;
<service_prop> ::= ( property <attr> : <value> ; )+
<op_list> ::= ( <operation> ; )+
<operation> ::= operation <operation_name>([<param_list>]) : <return_type>
<param_list> ::= <parameter_type> (,<parameter_type>)*

<service_name> ::= <identifier>
<device_name> ::= <identifier>
<operation_name> ::= <identifier>

<attr> ::= String
<value> ::= String
<return_type> ::= <type> | void
<parameter_type> ::= <type>
<type> ::= String | Number | Boolean | Date | List <type> | Object
<identifier> ::= a..z, $, _ (a..z, $, _, 0..9, unicode character over 00C0)*
```

Listing 4.1: Service description in SaS-SDL with the Backus–Naur Form (BNF)

```
service SwitchPower :
device Kitchen_Lights;
property deviceType : BinaryLight;
property protocol : UPnP;
operation SetTarget(Boolean) : void;
operation GetTarget() : Boolean;
```

Listing 4.2: Service description example in SaS-SDL

Service transcription. SaS retrieves necessary information about the service (*i.e.* its operations, its properties, etc.) using the implementation language introspection capability. Then, it automatically declares them in SaS-SDL. This is a model transformation where SaS tries to match elements of a service description syntax to SaS-SDL service description syntax elements. We illustrate this mechanism with two service description language: UPnP [82] and WSDL [62].

Figure 4.4 depicts the simplified class diagram (without class attributes) of the UPnP service representation. It is quite similar to the SaS-SDL service description syntax (illustrated by Figure 4.3), and thus easily transcribable into SaS-SDL. SaS operations are named actions, SaS

parameters are named arguments. Moreover, UPnP comprises state variables with a listener mechanism that SaS does not take in consideration because it is irrelevant in most of the SDPs.

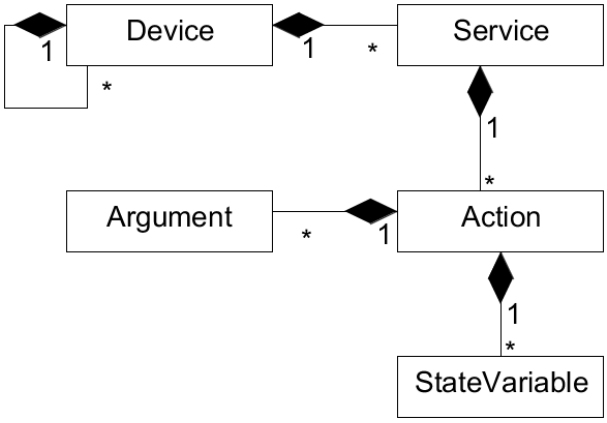

Figure 4.4: Class diagram of UPnP service description syntax

Listing 4.3 shows the description in UPnP of the service example from Listing 4.2. We can notice that SaS-SDL is shorter and more readable for users. We retrieve the device name on line 3, the service name on line 7 (*serviceType*), the operations name on lines 15 and 25. UPnP uses URLs to indicate the addresses of service descriptions XML files. Additionally, UPnP can propose a link to an HTML control page. The use of XML and of several files, linked with URLs make the service description difficult to read for the user.

```
1  <device>
2     <deviceType>urn:schemas-upnp-org:device:BinaryLight:1</deviceType>
3     <friendlyName>Kitchen Lights</friendlyName>
4     ...
5     <serviceList>
6        <service>
7        <serviceType>urn:schemas-upnp-org:service:SwitchPower:1</serviceType>
8        <serviceId>urn:upnp-org:serviceId:SwitchPower:1</serviceId>
9        <SCPDURL>/SwitchPower1.xml</SCPDURL>
10       <controlURL>/SwitchPower/Control</controlURL>
11       <eventSubURL>/SwitchPower/Event</eventSubURL>
12       ...
13       <actionList>
14       <action>
15         <name>SetTarget</name>
16         <argumentList>
17          <argument>
18            <name>NewTargetValue</name>
19            <relatedStateVariable>Target</relatedStateVariable>
20            <direction>in</direction>
21          </argument>
22         </argumentList>
23       </action>
24       <action>
25         <name>GetTarget</name>
26         <argumentList>
27          <argument>
```

```
28            <name>RetTargetValue</name>
29            <relatedStateVariable>Target</relatedStateVariable>
30            <direction>out</direction>
31          </argument>
32        </argumentList>
33      </action>
34      </actionList>
35      <serviceStateTable>
36         <stateVariable sendEvents="no">
37            <name>Target</name>
38            <dataType>boolean</dataType>
39            <defaultValue>0</defaultValue>
40         </stateVariable>
41      </serviceStateTable>
42      ...
43      </service>
44   </serviceList>
45 </device>
```

Listing 4.3: Service declaration example in UPnP

In WSDL, a service description comprises two sections: the abstract section that specifies the service syntax (with its operation, parameters, etc.) and the concrete section which describes how to reach the service instance: which protocol to use (*e.g.* SOAP, HTTP, etc.) and which address provides the service instance (URL or IP). Figure 4.5 depicts the class diagram of the abstract section of WSDL. We can see that the service is attached to an interface, that provides operations with input and output parameters which are typed. Operations can have several output parameters, which is not the case in SaS-SDL. SaS treats the multiple output parameters by creating a list which that the operation in SaS-SDL returns. Moreover, WSDL does not contain information about the provider device. We can just know (thanks to the concrete section) its url or its IP address. Thus, without any other information, SaS considers the device address as the device name.

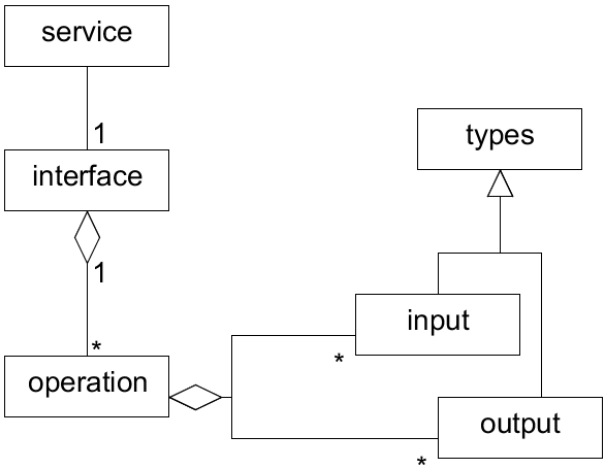

Figure 4.5: Class diagram of WSDL service description syntax

Figure 4.5 depicts the representation of a WSDL document. It contains the abstract section (types and interface) and the concrete section (binding and service). Listing 4.4 illustrates a service description example in WSDL. It is based on the service example from Listing 4.2. We retrieve the abstract section from line 3 to line 19 and the concrete section (binding and service) from line 21 to line 33. The binding element (in the concrete section) details the protocol to reach the service, namely SOAP here. The endpoint element (inside the service element) references the binding element and provides the URL address to invoke the service operations.

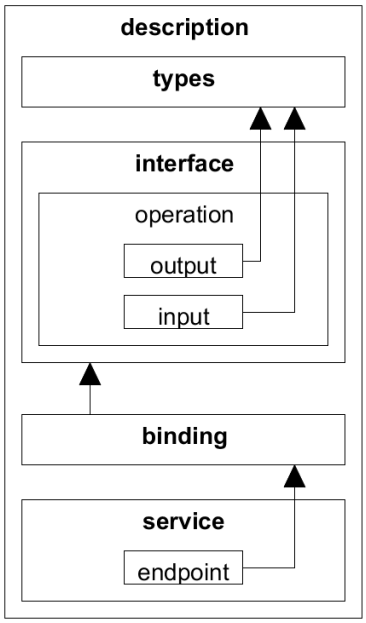

Figure 4.6: WSDL document representation

```
1  <description
2  ...
3     <types>
4        <xs:schema
5        ...
6           <xs:element name="status" type="xs:boolean"/>
7        </xs:schema>
8     </types>
9
10    <interface  name = "SwitchPower" >
11       <operation name="SetTarget"
12                  pattern="http://www.w3.org/ns/wsdl/in-only">
13          <input messageLabel="In" element="tns:status" />
14       </operation>
15       <operation name="GetTarget"
16                  pattern="http://www.w3.org/ns/wsdl/out-only">
17          <output messageLabel="Out" element="tns:status"/>
18       </operation>
19    </interface>
20
21    <binding name="SwitchPowerSOAPBinding"
```

```
22              interface="tns:ISwitchPower"
23              type="http://www.w3.org/ns/wsdl/soap"
24              wsoap:protocol="http://www.w3.org/2003/05/soap/bindings/HTTP/">
25        <operation ref="tns:SetTarget" />
26        <operation ref="tns:GetTarget" />
27     </binding>
28
29     <service name="SwitchPowerService" interface="tns:ISwitchPower">
30        <endpoint name="SwitchPowerEndpoint"
31                  binding="tns:SwitchPowerSOAPBinding"
32                  address ="www.service.com/SwitchPowerService"/>
33     </service>
34
35  </description>
```

Listing 4.4: Service declaration example in WSDL

Service Registration. Service discovery protocols enable to discover services and use various service description syntax. SaS can implement various SDPs and thus, transcript service instances discovered with SaS-SDL. When a discovered service has been transcripted, it can be memorized inside the service directory. We thus obtain an homogenized service directory.

Besides, when SaS executes a scenario, it needs to invoke services (selected by users in the scenario description). SaS uses the service directory to find an available service that match with the one selected by users. To invoke a service, SaS needs to know which protocol to use to reach the service instance. Thus, when SaS registers a service description inside the service directory, SaS adds to the service description a service property that indicates the protocol used for service discovery. Listing 4.2 illustrates a service description example of a service discovered with UPnP. Thus, SaS adds on line 4, as a service property, the protocol used.

4.2.2 Context Representation

Context awareness enables to maintain an updated service directory. This directory is a first representation of the user environment. Besides, a pervasive environment contains devices, that exports services. These devices can be SaS platforms, owned by users. SaS must represent this information. In addition, the context representation must adapt to users mobility and enable representation of environments elements even if they are not available. Moreover, a pervasive system must enable users to customize the environment representation as they wish.

4.2.2.1 SaS Context Directory

The SaS context directory is the representation of the user context. It enables users to detect what is available in their environment. Moreover, it must adapt to users expectations and users mobility.

Services and devices representation. A SaS platform discovers surrounding services and maintain updated a service directory. SaS retrieves from the list of available services the set of

devices present in the environment. Therefore, service directory enables the representation of the surrounding services and devices.

Platforms and users representation. Several SaS platforms can also populate the environment. To integrate the context representation, SaS platforms export an identification service into the environment. Such service provides platforms with an identifier and an owner name. This enables surrounding platforms to be aware about the platform (and therefore the user) presence. Listing 4.5 presents the platform description syntax in SaS-SDL. Listing 4.6 illustrates the platform description with an example.

```
<platform> ::= platform <platform_id> : <device>  <user> [<platform_prop>]
<user> ::= user <user_name>;
<platform_id> ::= Number
<user_name> ::= <identifier>
<platform_prop> ::= ( property <attr> : <value> ; )+
```

Listing 4.5: Platform description syntax with SaS-SDL

```
platform 17890804 :
device AndroPhone;
user Max;
```

Listing 4.6: Platform description example

4.2.2.2 Users Personalization of the Context Representation

Users may see their environment as a set of functionalities, that can be specific to a location (*e.g.* kitchen) or to a specific usage (*e.g.* multimedia). Similarly, SaS platforms discovered in the environment can be owned by different users (*e.g.* dad's platforms) or attached to a specific location (*e.g.* home). To enable users to better represent their environment, SaS allows them to group services and platforms previously memorized into named categories. These *categories* are like *tags* because a service (*resp.* a platform) can be included into several distinct categories. Examples of categories are locations (*e.g.* all services available at home) or users (*e.g.* all platforms owned by kids). Categorization also eases directory browsing, and diminishes the amount of information presented to users.

To enable categorization, SaS adds to the service description (inside the service directory) the categories to which the service belongs. Listing 4.7 represents the context directory syntax in SaS-SDL. As illustrated by Figure 4.1, the context directory contains three directories dedicated to services, platforms and scenarios. We can see that the directories contain a list of their corresponding elements that can be associated with categories. We illustrate service and platform directories in this subsection. Scenario directory is presented and discussed in Section 5.1.1.1.

Listings 4.8 and 4.9 illustrate the service and platform directories with examples. The service directory examples contains two services. The SwitchPower service is provided by the

Kitchen_Lights device, discovered with UPnP and categorized in the *home* category. The Player service is provided by the Lounge_PC device. Its protocol is set to local, which means that the SaS platform is deployed on the Lounge_PC and locally discovers the Player service. The service is categorized in the *home* and *multimedia* categories. The platform directory contains two platforms. The first platform is owned by Max and categorized in the *Max's_platforms* category. The second platform is also owned by Max and categorized in the *Max's_platforms* and *home* categories.

```
<sas_platform> ::= platform <platform_id> <device_name> <user_name>
                   <service_dir> <platform_dir>

<service_dir>  ::= service_directory  { (<service>  <category>*)* }
<platform_dir> ::= platform_directory { (<platform> <category>*)* }
<scenario_dir> ::= scenario_directory { (<scenario> <category>*)* }
<category> ::= category <category_name> ;
```

Listing 4.7: Context representation with SaS-SDL

```
service_directory {
   service SwitchPower :
      device Kitchen_Lights;
      property deviceType : BinaryLight;
      property protocol : UPnP;
      operation SetTarget(Boolean) : void;
      operation GetTarget() : Boolean;
      category home;

   service Player :
      device Lounge_PC;
      property deviceType : Computer;
      property protocol : Local;
      operation play(Object) : void;
      category home;
      category multimedia;
}
```

Listing 4.8: Service directory example

```
platform_directory {
   platform 17890804 :
      device AndroPhone;
      user Max;
      category  Max's_platforms;

   platform 18710318 :
      device Lounge_PC;
      user Max;
      category Max's_platforms;
      category home;
}
```

Listing 4.9: Platform directory example

4.2.2.3 Context Representation Persistence

The SaS context directory contains three directories: the *service directory* (which contains service descriptions with devices associated), the *platform directory* (with the owner name) and the *scenario directory*. Besides, users are mobile. Surrounding services and platforms are not always available whereas users can need this information at every moment. Typically, users might want to define scenarios with services that might not be simultaneously available. Context representation is therefore persistent. Thus, a service (*resp.* a platform) which is not available in the environment is still memorized in the service (*resp.* platform) directory. By this means, scenarios can be defined using services that temporarily miss (thanks to the service directory). Platform directory can also be used to collectively share scenarios (which can be equivalent to providing grouped access rights, *see* Section 5.2.1 for details).

The *available* service category. To enable service persistence and distinguish the services currently available from the services previously memorized, SaS uses the ability to manipulate categories in the service directory. Thus, if a service is available in the environment, SaS adds to its service description (inside the service directory) the category named *available*. If the service disappears, SaS just removes the category entry from the service description. The service description thus still remains in the service directory.

Last time presence and cleaning mechanism. A service (*resp.* platform) discovered but not available anymore is thus still memorized in the service (*resp.* platform) directory. To inform the user about last time service (*resp.* platform) presence, SaS provides for each service (*resp.* platform) its last seen date. This is added as a property of the service (*resp.* platform) into the service (*resp.* platform) description.

Moreover, some services (*resp.* platforms) description not available anymore may become obsolete. Thus, SaS enables users to clean the service (*resp.* platform) directory of descriptions they do not want anymore.

4.3 Scenario Definition

Scenarios are the representation of users' needs for pervasive systems. Users must therefore easily define scenarios that correspond to their expectations. In this section, we present the part of SaS-SDL dedicated to the scenario description language. The syntax of SaS-SDL provides the elements that are strictly necessary to create scenarios. It is simpler than most existing programming languages for service composition (such as BPEL [44]) which contain complex mechanisms. Such mechanisms (*e.g.* asynchronous invocations, exception handling, etc.) are transparently handled by SaS when necessary. Thus, SaS-SDL can be seen limited for certain aspects compared to BPEL, but we advocate that these mechanisms are not necessary for end-users who want to express their needs in a pervasive environment. SaS-SDL scenarios are therefore easy to read (see Listing 4.10 for an example). Moreover, SaS-SDL contains dynamic

configuration mechanisms for users to be able to customize the scenario they are going to execute.

In this section, we first present mechanisms to compose services (requirement R2.a) and then, we detail the possibilities for users to customize scenarios (requirement R2.b). We introduce each of the subsections by a scenario example to illustrate SaS-SDL mechanisms. Then, we present the full SaS-SDL scenario description syntax.

4.3.1 Service Composition

With SaS, defining scenarios is quite similar to designing basic workflows with a control-flow perspective as described in [85]. To define a scenario, SaS-SDL proposes only essential service composition mechanisms, detailed in this subsection.

Listing 4.10 illustrates SaS-SDL with a scenario example. This scenario, called NightScenario, is a comfort scenario in a home-automation context. The scenario tries to close main door and shutter, and adjust luminosity. Then, it adjusts the thermostat to maintain a certain temperature. Finally, it uses the player of the personal computer to listen the news from the radio at 8pm.

```
scenario NightScenario [
   MainDoor.DoorService.close();
   [ parallel:
      [
         Shutter.ShutterService.close();
         while (LuminosityCaptor.LuminosityService.getValue() > 500)
         [
            RoomLight.DimmableLightService.decrease();
         ]
      ]
      if (LivingRoomThermomether.ThermometerService.getTemperature() <= 18
            and HouseThermostat.ThermostatService.getValue() < 5 )
      [
         HouseThermostat.ThermostatService.setValue(5);
      ]
      else
      [
         PC.AdjustTemperatureScenario.start(18);
      ]
   ]
   at (8pm)
   [
      PC.Player.play(WebRadio.RadioService.getChannel(InfoChannel));
   ]
]
```

Listing 4.10: Scenario declaration example

4.3.1.1 Sequential or parallel execution

Actions may either be executed sequentially – in a defined order – or in parallel – if their executions do not have any side effects on one another. To specify so, SaS provides the *parallel*

keyword, to indicate that action execution order does not matter. In parallel mode, the first available action is executed. By default, actions in an action block are executed sequentially. Line 3 of Listing 4.10 illustrates an action block defined as parallel. Actions involved are executed without invocation order.

4.3.1.2 Operation composition

Service operations might have formal parameters. Users can either directly fix actual parameter (*e.g.* line 14 of Listing 4.10) or invoke another service operation to create the desired value (*e.g.* line 23 of Listing 4.10) thus composing operations. SaS checks if parameter types conform to the service definition.

4.3.1.3 Conditional statement

Users might define alternative execution flows (if-then-else or conditional statements), that are conditions, with their consequences and alternatives. Lines 11 to 19 of Listing 4.10 illustrate a conditional statement (with condition on lines 11 and 12, consequences on line 14 and alternatives on line 18).

4.3.1.4 Repetition loop

Users might want to repeat the execution of an action several times. SaS-SDL proposes two ways to do so: the `while` loops is to execute an action while a condition remains satisfied. The `times` keyword can be used alternatively to precise how many times an action should be invoked. Lines 6 to 9 of Listing 4.10 illustrate a while loop.

4.3.1.5 Event

Users must be able to define event actions. Events can be time-event (*e.g. at 7pm*) or a basic condition (*e.g. when any.Thermometer.getTemperature() > 17*). Consequences of the event are immediately invoked (if possible) when the event happens. Lines 21 to 24 of Listing 4.10 illustrate a time event (condition is specified at line 21 and its dedicated action block is on line 23).

4.3.2 Scenario Customization

Scenarios are not just service compositions: they must represent user needs as much as possible and thus be *customizable*. Such possibilities are illustrated by Listing 4.11, which adds to the example from Listing 4.10 some customizations.

```
1  scenario NightScenario (description:close the main door and shutters,adjust
2                     luminosity, change thermostat value and listen to radio;
3                          creator: Antoine;
4                          timeout: 4 hours) [
5     MainDoor.DoorService.close();
6     [ parallel:
7        [
8           all.ShutterService.close();
9           while (LuminosityCaptor.LuminosityService.getValue() > ?) [
10             RoomLight.DimmableLightService.decrease();
11          ]
12       ]
13       if ( LivingRoomThermomether.ThermometerService.getTemperature() <= 18
14            and HouseThermostat.ThermostatService.getValue() < 5 ) [
15          HouseThermostat.ThermostatService.setValue(5);
16       ]
17       else [
18          (creator=matt) any.AdjustTemperatureScenario.start(18);
19       ]
20    ]
21    at (8pm) [
22       PC.Player.play(WebRadio.RadioService.getChannel(InfoChannel));
23    ]
24 ]
```

Listing 4.11: Scenario customization example

4.3.2.1 Dynamic Service Selection

User scenarios are operation combinations. Operations define the concrete ways to invoke services as syntactical signatures. Users evolve in different environments where same service can be provided by different devices (*e.g.* a *Clock* service). Service selection cannot be restricted to a single and specific device, it must but dynamic.

Generic service use. The identity of a service or of its provider device do not always matter. For example, to print a document, the chosen printer generally matters: a person chooses his favorite printer, accesses the specified service and selects the appropriate operation. On the contrary when a user needs to know what time it is, he directly selects the `getTime` operation, no matter which clock provides it. To specify so, SaS-SDL has a specific keyword, *any*, that can replace the provider device or service name. The ability to not specify a device's or a service's name leads to a dynamic and automatic selection of the device or the service that is going to be effectively used. On line 18 of Listing 4.11, the *AdjustTemperatureScenario* service can now be provided by any device (thanks to special word *any*). SaS will therefore select an appropriate device at scenario runtime.

Multi service use. Users must also be able to select all devices that provide a service. SaS enables users to do so thanks to the *all* keyword. Therefore, all available devices that propose the appropriate service are selected for invocation. Line 8 of Listing 4.11 illustrates the multi-service use. All devices that propose the *Shutter* service will be closed.

4.3.2.2 Dynamic Parameterized Execution

Users usually provide parameter values for operation calls at scenario design-time. However, some parameter values should be defined at scenario runtime. SaS enables this thanks to a *joker*, symbolized by the `?` symbol in SaS-SDL. Users thus choose to either specify a value for each parameter or set them at `?` which can be interpreted as “to be specified at runtime". Scenarios therefore become parameterizable entities (like most services). Line 9 of Listing 4.11 illustrates the use of joker for dynamic parameterized execution (with `?`). The value of the parameter of the `getValue` operation must now be specified at scenario execution time by a user (who starts the scenario).

4.3.2.3 Service Filter

SaS enables users to add a preference filter for each invoked service in an operation invocation. It is comparable to the LDAP filter [39], and is composed of the attribute name and its desired value (*e.g.* for a print service, *color:true*). Filters can be combined with binary operators to define more complex selection criteria. Line 18 of Listing 4.11 illustrates a filter example. The service `AdjustTemperatureScenario` must provide the property (creator=Matt). This signifies that the scenario made available by this service, must have been created by an user called Matt.

4.3.2.4 Scenario Documentation

Users have to name their scenarios. This is a first information that illustrates the scenario goal (*e.g. AdjustTemperatureScenario*). In addition, SaS enables users to provide scenario metadata to document them. It could be a textual description (*e.g. adjust the radiator value to maintain a desired temperature*) or extra functional characteristics (*e.g.* creator: Antoine). On line 1 of Listing 4.11, a brief textual description has been added and on line 3 the creator name is appended.

4.3.2.5 Scenario Timeout

A scenario being executed could become obsolete after it has run a certain time. Users can thus define a timeout for the scenario. A scenario launched will be interrupted after the time specified by users. On line 4 of Listing 4.11, a scenario timeout has been defined. Scenario will automatically abort (if not already finished or stopped) after four hours.

4.3.3 Scenario Description Syntax

Listing 4.12 presents the grammar of SaS-SDL for scenario declaration using the BNF notation. Elements of this syntax are described in Section 4.3.1. Figure 4.7 presents another view (class diagram) of the SaS-SDL scenario syntax.

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

<action_block> ::= [ [<parallel_exec>] (<action> | <action_block>)+ ]
<action> ::=  <service_exec> | <conditional_statement> | <loop> | <event>

<service_exec> ::= <op_invocation> ;
<op_invocation> ::= [<filter>] <device>.<service_name>.<operation_name>
                    ([<parameter_list>])
<parameter_list> ::= (<op_invocation> | <parameter_value>)
                     (, (<op_invocation> | <parameter_value>) )*
<conditional_statement> ::= if<condition><action_block>[<else_clause>]
<else_clause> ::= else <action_block>
<loop> ::= (while <condition> | <repeat_value> times) <action_block>
<event> ::= (when <condition> | at ( <time_event> )) <action_block>

<condition> ::=  ( [not] ( <unary_condition> | <cplx_condition> )  )
<unary_condition> ::= <op_invocation> <comp_operator> (<op_invocation> | <value>)
<cplx_condition> ::=  ( <condition> ( <bin_operator> <condition> )* )

<scenario_prop> ::= ( <attr> : <value> (; <attr> : <value>)* )
<filter> ::= ( <filtercomp> )
<filtercomp> ::= <binary_filter> | <item>
<binary_filter> ::= <binary_operator> <filterlist>
<filterlist> ::= <filter> | <filter> <filterlist>
<item> ::= <attr> <comp_operator> <value>
<bin_operator> ::= and | or
<comp_operator> ::= < | <= | > | >= | == | !=
<parameter_value> ::= <value> | ?
<device> ::= <identifier> | any | all
<service_name> ::= <identifier> | any
<parallel_exec> ::= parallel:
<identifier> ::= a..z, $, _ (a..z, $, _, 0..9, unicode character over 00C0)*
```

Listing 4.12: Grammar of the scenario declaration using the BNF notation

Figure 4.8 represents the instance diagram of the scenario example described by Listing 4.10. To simplify the diagram, we do not represent the operation invocations as objects, but we set them as attributes of the service execution object and the condition object. We annotate relations for the conditional statement object and the while loop object to help reading. To ease the reading, we specify some color rules, scenario is in orange, action blocks are in blue, actions are in red, complex condition is in green and unary conditions are in gray.

4.4 Synthesis and Conclusion

In this chapter, we presented how SaS enables users to pass from a set of available services to a scenario. This chapter responds to the two first requirements of pervasive systems: context management (R1) and scenario definition (R2).

4.4.1 Context Management with SaS

We have seen in Section 4.2 that SaS can implement various service discovery protocols (SDP). It enables to be aware about the environment elements. Moreover, SaS provides to users a con-

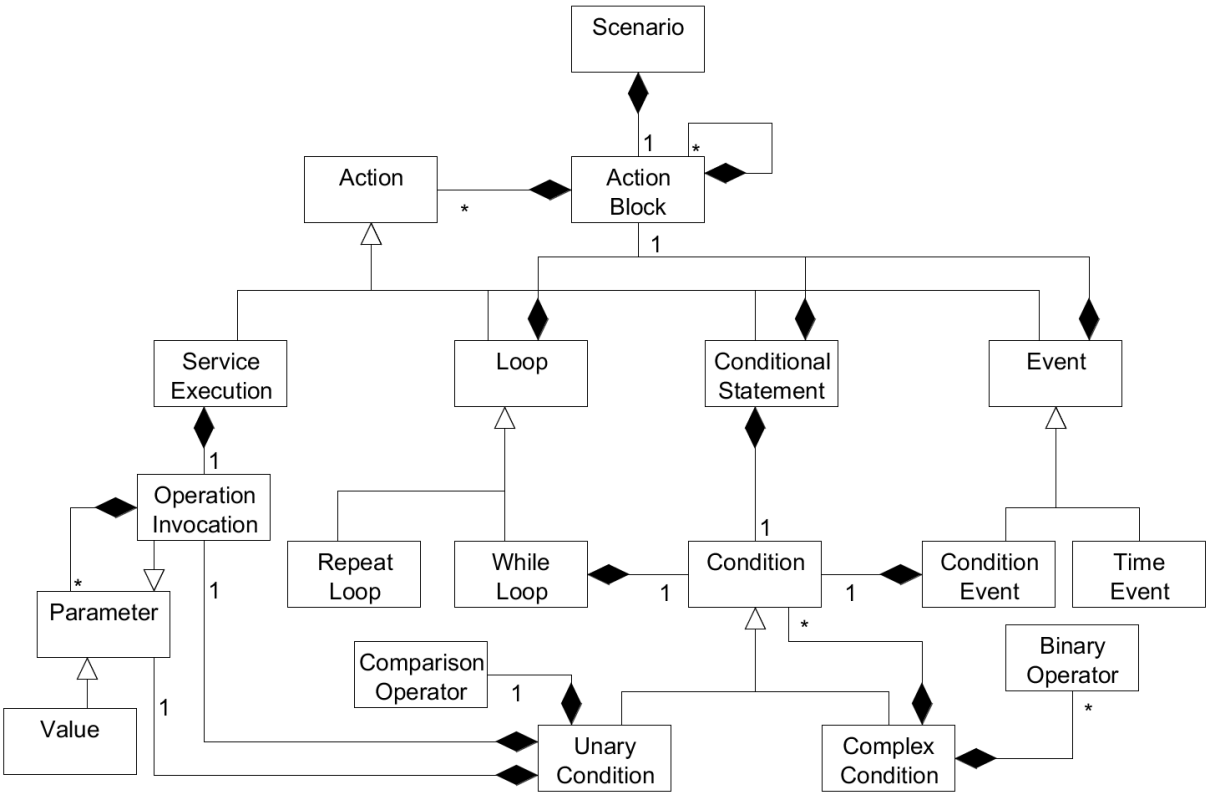

Figure 4.7: Class diagram of the SaS-SDL scenario syntax

text representation composed of a service directory (that also enables to retrieve surrounding devices) and a platform directory (that also enables to retrieve surrounding users). Such representation is customizable and persistent. The context management requirement is thus fulfilled by SaS. It has advantages and some limits that we detail in this subsection.

Advantages of SaS context management:

+ SaS uses a generic language for service description, that enables to be not restricted to a single SDP. This enhances interoperability.

+ SaS is continuously listening for context events and thus adapts to environmental changes.

+ SaS enables users to customize the context representation as they wish thanks to categories.

+ The SaS context representation is persistent, which thus adapts to users mobility. This enables users to retrieve and use services in scenarios even if they are not available simultaneously. Moreover, the platform directory enables users to select surrounding platforms for sharing (this process is explained in Section 5.2).

Limits of SaS context management:

- Some SDPs can propose extra functionalities. Typically, UPnP enables to listen for value changes of some devices variables. Such functionalities are not implemented by all service discovery protocol. Thus, the SaS generic approach limits to the discovery of service (with their operations) and their provider devices.

- SaS does not prescribe a device type model. The nature of devices, specially in home-automation, may be listed (typically: light, radiator, fan, etc.). With such mechanism, a pervasive system could propose to users a better representation of the surrounding devices, classified by types. This mechanism is difficult to implement with various SDPs, because it implies that they semantically agree on the device type.

4.4.2 Scenario Definition with SaS

SaS comprises a scenario description language, namely SaS-SDL, that enables to define scenarios by composing services such as basic workflows (*cf.* Section 4.3.1). This language only contains necessary elements for users to express their goals. Moreover, SaS-SDL also enables users to customize their scenarios to make them parameterizable, more dynamic and representative of their needs (*cf.* Section 4.3.2). These two mechanisms have advantages and limits, detailed here.

Advantages of scenario definition with SaS:

+ SaS-SDL comprises various composition concepts (such as parallel or sequential execution) that enables to define scenarios which correspond to users' needs.

+ SaS-SDL only contains necessary elements for users to express their goals. Thus, SaS-SDL is simpler than existing composition languages (such as BPEL).

+ SaS-SDL contains mechanisms to create parameterizable and dynamic scenarios. Scenarios thus dynamically adapt to users needs and to the environment.

Limits of scenario definition with SaS:

- A composition language enables to define scenarios however, it is still too complex for users without any technical knowledge. Scenario descriptions with SaS-SDL are easily readable (*cf.* Listing 4.10), however, it may be difficult for users to express their needs if they begin to be complex (conditions imbricated, etc.).

- Some complex mechanisms (that we specified as not necessary defining scenarios such as exception handling) are not provided by SaS-SDL. They are handled by the system transparently for the user. Thus, for an advanced user, SaS-SDL can be seen as limited compared to other existing languages by some aspects.

4.4.3 Requirements Fulfillment

Context management requirement (R1) implies context representation (R1.a) and context awareness (R1.b). The scenario definition requirement (R2) has two sub-requirements: service composition (R2.a) and scenario customization (R2.b). This chapter has detailed how SaS fulfills these four functional sub-requirements. Table 4.1 synthesizes the SaS' functionalities that answer to the sub-requirements.

Functional Requirements	**Functionalities Associated**
R1.a Context representation	SaS provides users with a persistent and personalizable context directory that contains services, devices, platforms, users and scenarios.
R1.b Context awareness	SaS proposes an interoperable approach that enables to use various service discovery protocols thanks to a model transformation.
R2.a Service composition	SaS comprises a scenario description language (SaS-SDL) that enables service composition and contains only necessary elements for users.
R2.b Scenario customization	The SaS-SDL enables to customize scenarios in order to better correspond to users' needs.

Table 4.1: Fulfillment of requirements detailed in Chapter 4

In this chapter, we have seen, inter alia, how users can define scenarios. Next chapter presents how SaS enables users to manage scenarios (*i.e.* control, reuse and share).

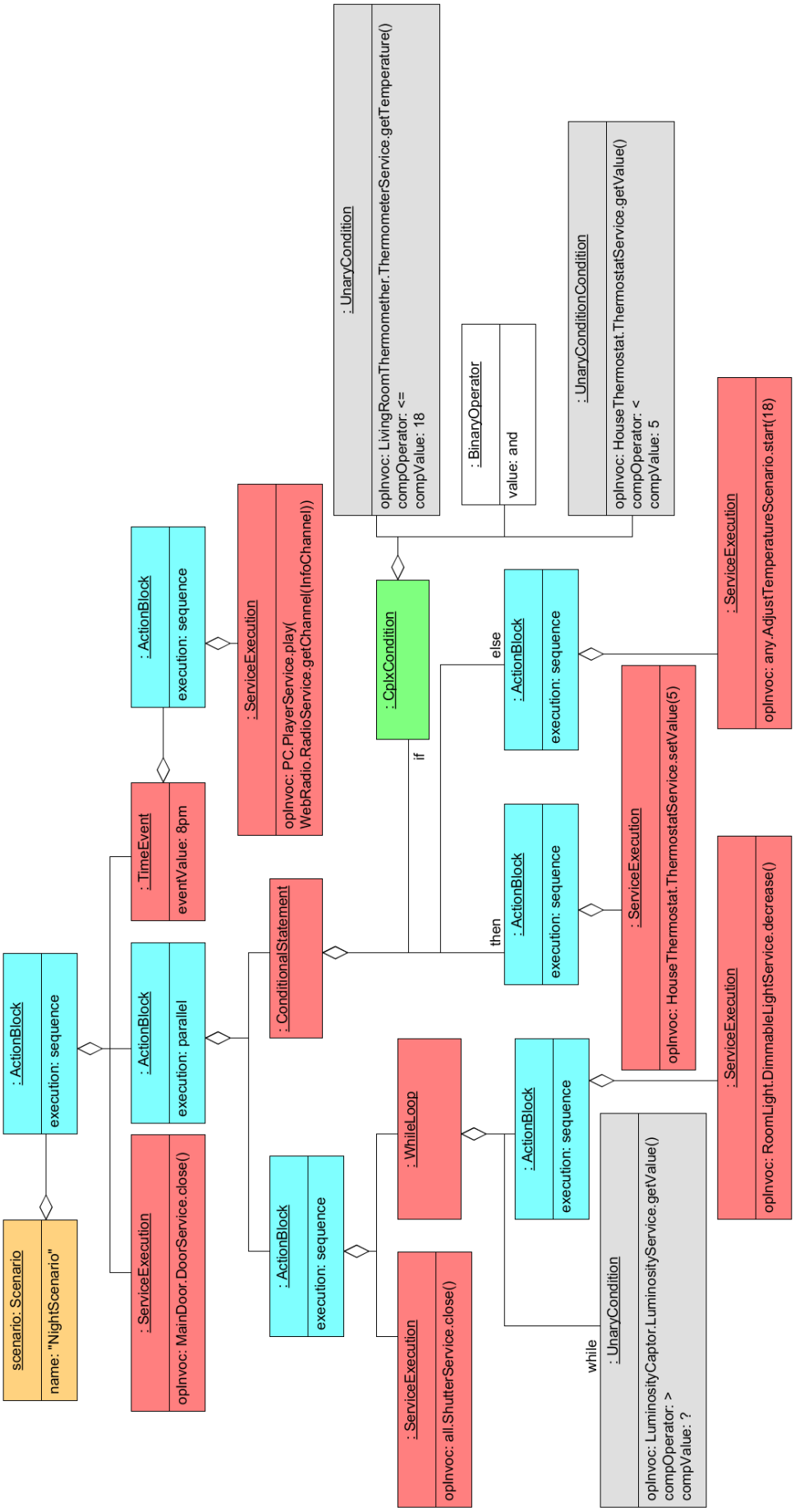

Figure 4.8: Instance diagram of the scenario example

Chapter 5

Scenario Management: Control, Reuse and Share

Contents

In the previous chapter, we presented how users obtain a representation of their context, can manage it and are able to define a scenario that corresponds to their needs. Now that we have seen how SaS enables to define a scenario, we present in this chapter the different mechanisms to manage a scenario. This management corresponds to three functional requirements of pervasive systems and implies scenario control, reuse and share.

Section 5.1 details how SaS handles scenario life-cycle. This functionality considers scenario reuse and enables users to control scenario execution. Section 5.2 is dedicated to the platform collaboration. We detail how SaS enables users to share scenarios and services among them. Finally, Section 5.3 synthesizes this chapter, concludes and draws some perspectives.

5.1 Scenario Life-Cycle

As detailed in Section 2.2, a scenario created by a user must be easily controllable and reusable. It first implies to *memorize* the created scenario. When memorized, a scenario can be *installed* by its owner. This provokes scenario deployment to become available for execution. This installation is locally done on the platform. The internals of scenario when it is in the *Installed* state are discussed in Section 5.1.2.2. A scenario could become obsolete or useless and so, scenario owner can decide to *uninstall* its scenario at every moment. Such scenario therefore becomes unavailable for execution (and can be reinstalled) and can be *deleted* permanently. Figure 5.1 depicts the state diagram of scenario life-cycle.

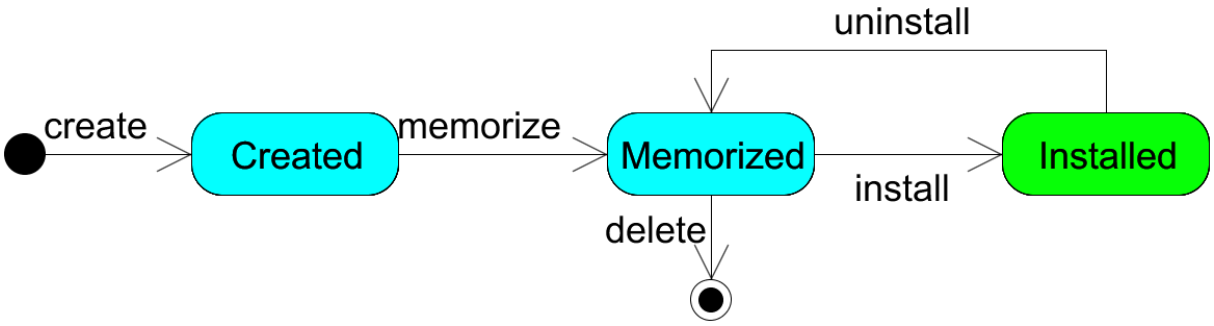

Figure 5.1: State diagram of scenario life-cycle

In this Section, we first introduce how SaS memorizes scenarios, which enable users to reuse their creations in the future. Then, we present the *scenario orchestrator*, which enables users to control scenario execution and enforces scenario execution life-cycle. Finally, we detail the SaS' mechanism that registers scenario as services, which enables, inter alia, scenario hierarchical composition.

5.1.1 Scenario Description Resilience

To enable users to reuse a scenario in the future, SaS makes scenario description persistent. This enables to easily retrieve the scenario and to modify it.

5.1.1.1 Scenario Memorization

The scenario memorization makes scenario description persistent. This implies to memorize the scenario into a text file, and register this file inside a directory.

The scenario description file. To make scenario description persistent, SaS memorizes the scenario declaration in SaS-SDL into a text file (as illustrated by step 3 in Figure 4.2): the *scenario description file*. The scenario description file is named with the scenario name. Typically,

the scenario example from Listing 4.11 is memorized in a text file, named `NightScenario.sas`. Scenario text files are stored inside the SaS platform.

The scenario directory. To list the scenario description files stored, SaS contains a *scenario directory*. This directory is part of the the SaS context directory as illustrated by Figure 4.1 and Listing 4.7. To memorize scenarios into the scenario directory, SaS creates the scenario description. Listing 5.1 illustrates the scenario description syntax with SaS-SDL. Listing 5.2 illustrates a scenario description example. This is the scenario example from Listing 4.11.

```
<scenario> ::= scenario scenarioName : <property>*
<property> ::= property <attr> : <value>:
<attr> ::= String
<value> ::= String
```

Listing 5.1: Scenario description syntax with SaS-SDL

```
scenario NightScenario :
property creator : Antoine;
property description : close the main door and shutters, adjust luminosity,
                       change thermostat value and listen to radio;
property timeout : 4 hours;
```

Listing 5.2: Scenario description example

Similarly as the service and the platform directories (*cf.* Section 4.2.2.1), SaS enables users to categorize their scenarios. Users therefore have an easy access to already memorized scenarios. Listing 4.7 represents the context directory syntax in SaS-SDL with the scenario directory syntax. Listing 5.3 illustrates a scenario directory example that contains two scenarios both categorized in the home category.

```
scenario_directory {
   scenario NightScenario :
      property creator : Antoine;
      property description : close the main door and shutters, adjust luminosity,
                             change thermostat value and listen to radio;
      property timeout : 4 hours;
      category home;

   scenario AdjustTemperatureScenario :
      property creator : Louise;
      property description : adjust the temperature value;
      property timeout : 8 hours;
      category home;
}
```

Listing 5.3: Scenario directory example

Users can then install their scenarios while the descriptions remain available (*i.e.* while the description file is not deleted).

5.1.1.2 Scenario Future Use

As seen in the previous subsection, scenario description persistence enables users to install scenarios as such in the future. However, users' needs can change which might lead them to modify their creations. SaS therefore enables to modify scenario descriptions. To do so, SaS checks if the scenario is already installed. If so, SaS uninstalls the scenario (based on the old description) and installs the new one. The new scenario automatically replaces the new one inside the scenario directory.

5.1.2 Scenario Orchestrator

Scenario execution is a complex process that implies to adapt to users' action and to context (presence / absence of certain services, environmental changes, etc.). Therefore, SaS dynamically generates a *scenario orchestrator* for each deployed scenario. The orchestrator provides to users the scenario execution control operations. Moreover, it enforces the scenario execution life-cycle.

5.1.2.1 Scenario Execution Control Commands

An installed scenario is executable. Users should therefore be able to start it. Additionally, users might want to pause or abort a scenario in execution. For example, a *listen music* scenario, that sends music from the PC to the stereo system, must be easy to pause (and also to resume). Therefore, SaS provides users with execution control commands: *start*, *pause*, *abort* and *resume*.

5.1.2.2 Installed Scenario Life-Cycle

The possible user operations are available to users as defined in SaS' installed scenario life-cycle depicted by Figure 5.2. This represents the state diagram of the installed state of Figure 5.1. A scenario can be in one of these states:

- Deployed. When the scenario is installed it becomes ready to be executed. It can also be uninstalled.
- Running. The scenario has been launched and is currently being executed. The scenario can finish normally. Afterwise, it comes back to the *deployed* state. Otherwise, it can be paused or aborted and goes to the *paused* state. A scenario aborted automatically passes through the paused state before it returns to the deployed state.
- Paused. Scenario execution has been interrupted by a user. It is paused waiting to be resumed (and return to the *running* state) or to be aborted.

The transition *when finished* (in italic on Figure 5.2) is the only one which is automatic and not controlled by users. The internals of scenario execution when it is in the running state are discussed in Section 6.2.2.

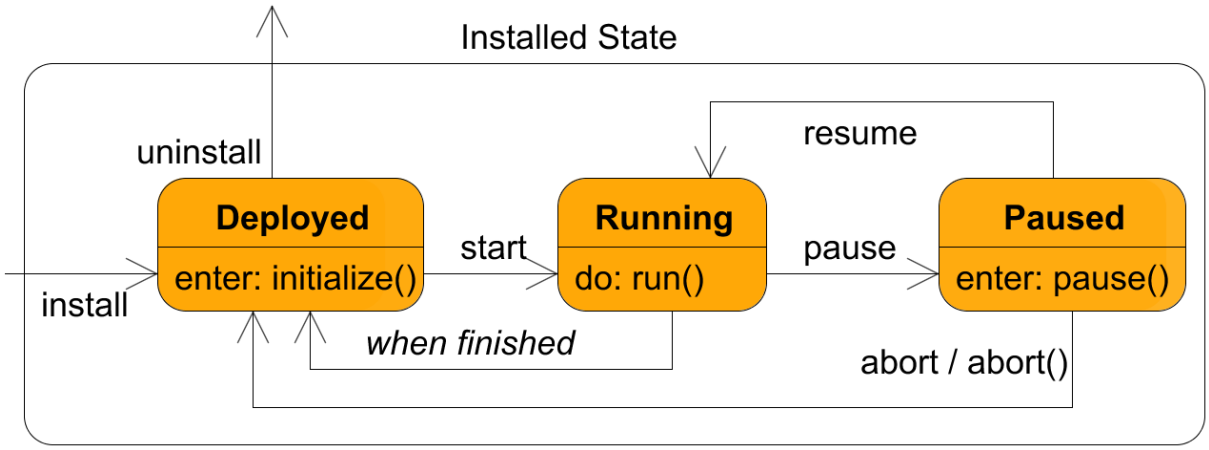

Figure 5.2: State diagram of installed scenario life-cycle

5.1.3 Scenario Registered as Service

As seen in the previous subsection, when SaS installs a scenario (to make it executable), SaS generates a scenario orchestrator that provides users with scenario execution control commands. These control commands are similar to service operations which are invocable and can have entry parameters. Moreover, users must be able to reuse scenario into other ones. Scenarios have therefore to be hierarchically recomposable. Besides, SaS considers service as an atomic entity which is composable. This is why, SaS considers *scenarios as new services*. A scenario can then be used simply as a service and, as such, composed into a new coarser grained scenario (*scenario hierarchical composition*).

Listing 5.4 illustrates the service description of the scenario example described in Listing 4.11. The SaS platform where the scenario is deployed is named *MyPlatform*.

```
1  service MyPlatform NightScenario
2  property description : close the main door and shutters, adjust luminosity,
3                         change thermostat value and listen to radio;
4  property timeout: 4 hours;
5  operation start(Object...) : Boolean;
6  operation pause() : Boolean;
7  operation resume() : Boolean;
8  operation abort() : Boolean;
9  operation getScenarioState() : File;
10 operation getDescriptor() : File;
```

Listing 5.4: A scenario registered as a service example

The service exported by the scenario orchestrator is automatically discovered by the platform (where it is locally provided) and thus, memorized into the service directory with the list of available services.

5.1.3.1 Scenario Properties

The scenario orchestrator is responsible for scenario execution control. To do so, scenario orchestrator locally provides a new service that enables the platform owner to control scenario execution thanks to the control commands (*e.g.* start, pause, etc.). The service exported by the scenario orchestrator has the same name as the scenario's to be easily recognizable as illustrated by line 1 of Listing 5.4.

Users can detail some scenario information at scenario definition (*cf.* Section 4.3.2), such as textual description, scenario timeout and multi-execution. Such scenario information is embedded in the scenario description file and must be provided to users simultaneously as the control commands. Besides, a service can embed some properties. Scenario information are therefore transcribed into the service properties. This is illustrated from line 2 to line 4 of Listing 5.4.

5.1.3.2 Scenario Operations

We have seen in Section 5.1.2.1 that SaS enables several actions to control scenarios execution. To provide these functionalities, SaS thus registers scenarios as services with operations corresponding to users action control. We retrieve these operations illustrated from line 5 to line 10 of Listing 5.4.

Scenario control commands. Figure 5.1 shows that four operations are dedicated to control a scenario execution: `start, pause, abort` and `resume`. The `start` operation is not the same for every scenario. It might have parameters or not depending on the presence of `?` values for the parameters of its constituting operations (cf. Section 4.3.2.2). SaS matches parameter values entered by users to the corresponding operations by order of appearance in the scenario. Control operations return a boolean which reports the success (or failure) of the operation invocation.

Scenario extra operations. Scenario execution is a statefull process. Depending on users' control, scenarios can be deployed, started, paused or aborted. Moreover, scenario execution might involve several processes that are not executed atomically. Some feedback is necessary. SaS provides therefore users with a mechanism to check scenario execution advancement. Users can thus check if the scenario is running and what scenario parts have been already executed. This is done thanks to the `getScenarioState` operation. This operation provides the log file that reports scenario starting time, user actions and parts of the scenario already executed.

A scenario has its description memorized into a scenario description file. The user can retrieve the scenario description file thanks to the scenario directory. However, users may want to have a direct access to see what the scenario registered as a service is composed of. Moreover, scenarios can be shared (detailed in Section 5.2) among users. However, the scenario directory remains only accessible locally. Other users may need to retrieve the scenario description file. Thus, the `getDescriptor` operation provides an access to the scenario description file.

5.1.3.3 Hierarchical Composition

As a service, a scenario is therefore hierarchical recomposable. To do so, users simply need to invoke a scenario operation (*e.g.* start to launch the scenario) into their composite scenarios. Additionally, if the *start* operation contains parameters (due to the presence of jokers inside the scenario exported as service), users should enter values for them. Otherwise, these parameters become parameters of the scenario currently defined. Line 17 of scenario example (Listing 4.11) illustrates scenario hierarchical composition. The AdjustTemperatureScenario service provides the start operation that requires a single parameter.

5.2 Platforms Collaboration

Users must be able to share scenarios among surrounding users and devices. As detailed in Section 2.2, this implies to select who to share with and define what to share. SaS therefore provides users with different sharing modes to share their creations. Moreover, pervasive systems must enhance collaborativeness. Platforms can therefore share service access. This makes it possible for platforms to have access to services that are not directly physically reachable.

5.2.1 Scenario Sharing Modes

Users that want to share a scenario must be able to select who to share with. SaS thus enables to select some surrounding platforms for sharing. Besides, users also need to select what they want to share. SaS therefore provides several scenario *sharing modes*. Several users can thus be able to execute the same scenario. SaS must therefore enable users to keep control of the scenario.

5.2.1.1 Platforms Selection For Sharing

A pervasive environment is characterized, inter alia, by the presence of multiple users. A way to be aware of the presence of users is to look at the owner of available platforms. Each SaS platform exports a service in the environment that enables surrounding platforms to detect its presence. This service provides, inter alia, the platform owner's name. Thus, SaS (installed on a platform) detects surrounding platforms and provides to users a representation of the available platforms (and their owner). Users can thus select some available platforms to share scenarios with.

Additionally, users can customize the context representation thanks to categories (*cf.* Section 4.2.2). Users can use this mechanism and shares a scenario with a platform category. Thus, each platform of this category, is selected to share the scenario with.

As a SaS platform is associated to a unique user, sharing scenarios with selected SaS systems is equivalent to defining access rights. SaS platforms are permanently indexed into a *platform directory* (*cf.* Section 4.2.2.1). This makes it possible to share scenarios with a platform even

if the platform is temporarily unavailable (because of a failure or of mobility). However, users must decide who to share their scenarios with. This is why, SaS enables users to share scenarios according to different modes.

5.2.1.2 Scenario Sharing Access Modes

Users must be able to select what they want to share. There are three functionalities interesting for users: (a) the scenario properties (*e.g.* scenario creator, brief textual description, etc.), (b) the scenario description file (to see what the scenario is composed of) and (c) the remote execution control. SaS provides three scenario sharing access modes to select *what* to share:

- **descriptive**. Only the scenario description file is shared with selected users. These users can see what the scenario is composed of and decide to redeploy it locally (on their own platforms) in order to be able to execute a local copy of it.
- **collaborative**. The scenario execution is shared but not its description: the scenario is remotely controllable by selected users. Users can control scenario execution and check its advancement. However, they have not access to the scenario description and thus, cannot redeploy it locally. If the scenario creator has written a scenario textual description, this information is provided as a property of the service (*cf.* Section 5.1.3) and is thus accessible for selected users in collaborative mode.
- **copied**. Selected users have access to both the scenario description and scenario execution.

Table 5.1 synthesizes the comparison between the different scenario sharing modes. We can see the default mode that illustrates that by default, a SaS platform does not share scenarios with surrounding platforms. Users must select surrounding users to share scenarios with and attribute them an access right. Moreover, this table illustrates that users selected for sharing have access to scenario properties (and thus a brief textual description) independently on the sharing mode. Thus, they can see the purpose of the scenario.

Scenario shared part	**Sharing mode**			
	Default	Descriptive	Collaborative	Copied
Scenario properties	×	✓	✓	✓
Description file	×	✓	×	✓
Execution control	×	×	✓	✓

Table 5.1: Scenario sharing modes comparison

Figure 5.3 illustrates these different scenario sharing modes. Each octagon represents a SaS platform, with its owner at the center and the creation and deployment cycle (*cf.* Section 4.1) that the scenario has on each platform. To illustrate the different sharing modes, we symbolized

the scenario description file by a text file icon and the scenario execution control by a remote control. The central user (Matt) creates a scenario (which is registered as a new service in its directory). He shares the scenario description file with Janis. Janis can therefore locally redeploy the scenario. James has access to the scenario execution control (symbolized by a remote control). This is the collaborating mode. James can thus execute the scenario or recompose it into a new one. Kurt illustrates the copied sharing mode: he has access to both the scenario description file and the execution control. He can therefore control scenario execution remotely or decide to redeploy the scenario (on its own platform).

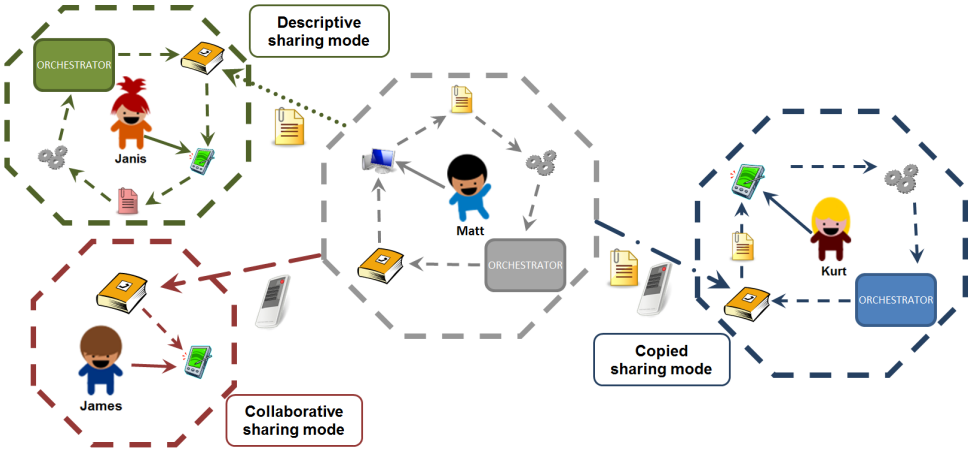

Figure 5.3: Overview of SaS scenario sharing modes

5.2.1.3 Scenario Execution Conflict Management Modes

On collaborative and copied modes, a platform (*e.g.* James' platform of example from Figure 5.3) can remotely control scenario execution (*e.g.* installed on Matt's platform). The scenario creator (*e.g.* Matt) must be able to keep the control of the execution, even if he/she shares scenario execution control. SaS therefore provides two extra modes: *veto* and *free* modes. The scenario creator selects a mode for each user with whom scenario execution has been shared. In veto mode, the scenario creator (*e.g.* Matt) is warned when another user (*e.g.* James) wants to remotely control scenario execution (*e.g.* installed on Matt's platform). The scenario creator can therefore choose to authorize the user action or not. In free mode, selected users can interact with the scenario without asking for the scenario's creator approval.

5.2.2 The Collaborate Service

In Section 4.2.2.1 (dedicated to context representation), we explained that platforms export an identification service into the environment. Such service provides platforms with an identifier and an owner name and enables surrounding platforms to be aware about the platform presence. SaS uses this service for sharing scenarios and services. This is the *Collaborate* service.

5.2.2.1 Collaborate Service Description

Listing 5.5 presents the Collaborate service description. It contains three properties, corresponding to the information necessary for context representation for surrounding platforms. The Collaborate service provides five operations. Four operations are dedicated to sharing scenarios and services and one operation enables platform substitution (*cf.* Section 5.2.4).

```
service Collaborate :
device MyPlatform;
property platformId : 17890804;
property device : AndroPhone;
property user : max;
operation shareScenario(ServiceDescription) : void;
operation shareService(List<ServiceDescription>) : void;
operation invokeScenarioOperation(OperationInvocation) : Object;
operation invokeServiceOperation(OperationInvocation) : Object;
operation substitute(String) : Boolean;
```

Listing 5.5: The Collaborate service description

5.2.2.2 Scenario Sharing

Thanks to the *Collaborate* service, users can share their scenarios. To do so, they use the `shareScenario(ServiceDescription)` operation of the *Collaborative* services provided by other platforms. The value of the parameter `ServiceDescription` depends on the shared scenario and the sharing mode. The user sends the service description corresponding to the scenario that he/she wants to share.

Scenario as service description sharing. A scenario deployed as a service proposes two types of operation: five execution control operations (`start`, `pause`, `resume`, `abort` and `getScenarioState`) and one operation to get the scenario description file (`getDescriptor`). Moreover, a scenario as a service embeds properties (*e.g.* the scenario owner name, a brief textual description, etc.). Listing 5.4) illustrates an example of a scenario as service description. Depending on the sharing mode, the user that shares a scenario sends a service description with the five execution control operations, the get scenario description file operation or both.

We can illustrate the use of the *Collaborate* service thanks to Figure 5.3. The four users provide the *Collaborate* service into the environment. Thus, they are all aware about each other. The user Matt wants to share a scenario he created. To do so, he invokes the operation named `shareScenario(ServiceDescription)` provided by the *Collaborative* service exported by other platforms. The value of the parameter ServiceDescription depends on the shared scenario and the sharing mode. In descriptive sharing mode (with Janis), the user Matt sends a scenario description that contains only one operation (`getDescriptor`). In collaborative sharing mode (with James), the user Matt sends a scenario description that contains the five execution control operations (`start`, `pause`, `resume`, `abort` and `getScenarioState`). In copied sharing mode (with Kurt), the user Matt sends a scenario description with all the operations.

Remote scenario operation invocation. Users that have been selected for sharing can thus invoke scenario operations (that they have access). To do so, they use the operation named `invokeScenarioOperation(OperationInvocation)` from the *Collaborate* service provided by the platform of the scenario provider.

Typically, on the example from Figure 5.3, Janis can get the scenario description file. To do so, she invokes the `invokeScenarioOperation` operation from the *Collaborate* service provided by Matt's platform. The parameter value of the operation invocation has to be `MyPlatform.NightScenario.getDescriptor()`. The Matt's platform, that receives the request, checks the access rights of the requester. If the requester has the right to invoke the operation on the scenario requested, the Matt's platform executes the request. Instead, the Matt's platform responds an error message saying that the requester does not have rights to invoke the requested operation on this scenario.

Scenario sharing example. Figure 5.4 depicts the sequence diagram of a sharing scenario example based on the example from Figure 5.3. First, the two platforms (MP and JP) export the *Collaborate* service into the environment. Each platform discovers the service provided by the other one and updates its platform directory. Matt wants to share a scenario (NightScenario) with Janis on descriptive mode. Thus, MP uses the `shareScenario` from the Collaborate service provided by JP. The service description s1 is illustrated in the diagram. It contains a single operation: `getDescriptor`. JP receives the scenario as service description and thus, updates its scenario directory. Janis wants to get the scenario descriptor file. Thus, JP invokes the `invokeScenario` operation from the Collaborate service provided by MP. MP receives the request with the operation invocation o1. It checks if the requester (JP) has access to this operation for the selected scenario. If this is the case, MP invokes the scenario operation and returns the result (here, the scenario description file).

5.2.2.3 Sharing Services

Devices do not implement all protocols and cannot physically have access to services not present in their close environment (*e.g.* Bluetooth protocol has a scope of 50 meters). To enhance collaboration between platforms, platforms must be able to share services.

5.2.2.4 Sharing Service Descriptions

To share services, SaS platforms first need to share the descriptions of the services they want to share. To do so, they use the `shareService("List<ServiceDescription>")` operation from the *Collaborate* services provided by surrounding platforms. It is similar to the `shareService` operation except that there is no different service sharing modes and thus, SaS shares the whole service description (*i.e.* all the service operations).

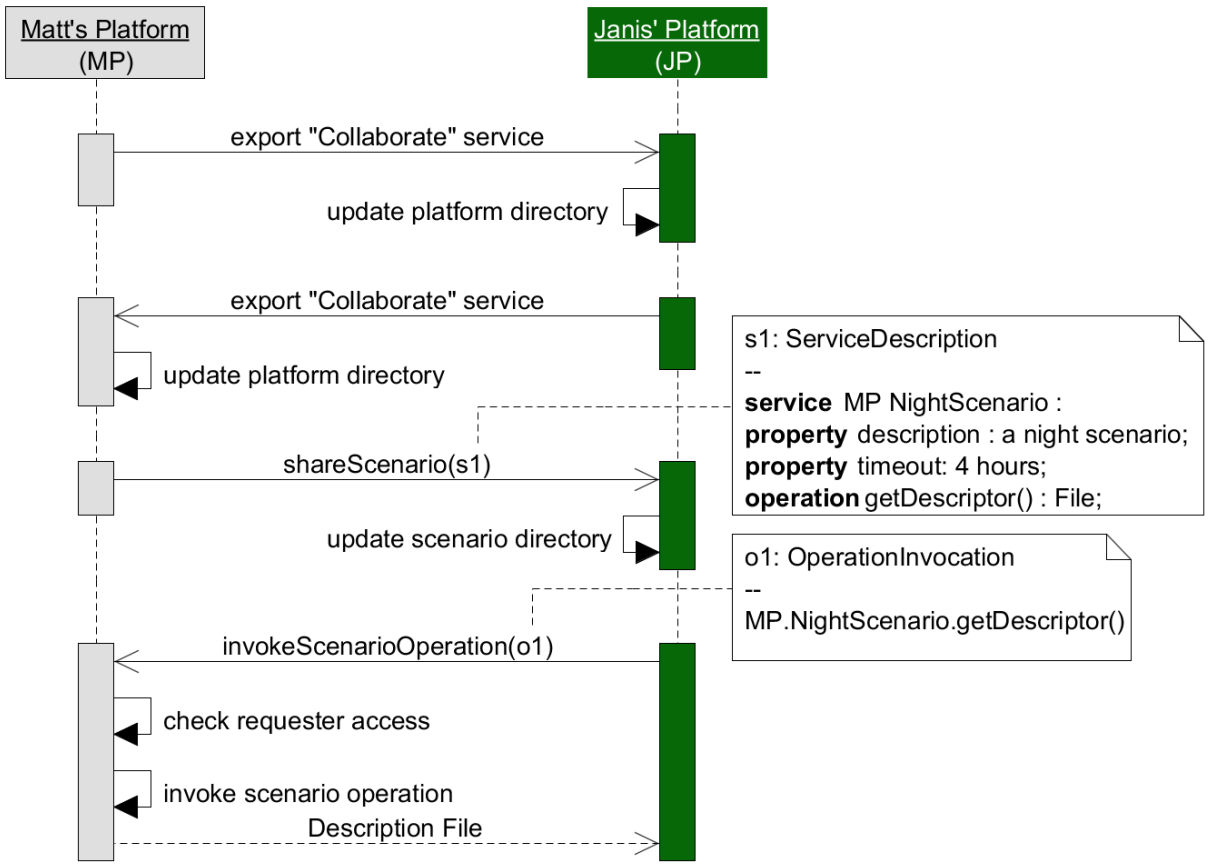

Figure 5.4: Sequence diagram of sharing scenario xample

5.2.2.5 Sharing Service Access

Sharing service descriptions is not enough. Users that are interested in a shared service by another platform need to access to this service. Therefore, platforms that share service descriptions also share access to the same services. SaS makes it possible through the operation named `invokeServiceOperation(OperationInvocation)` of the *Collaborate* service. It is similar to the `invokeScenarioOperation` but for a service operation. The provider platform checks if the requested service is shared with the requester.

5.2.2.6 Shared Services Availability

Sharing a service not available is irrelevant. Therefore, SaS matches service described as sharable and services available. The union is the list of services that are actually shared through the `Collaborate` service. Of course, SaS dynamically updates this union (depending on changes from the user or the context) and thus, invokes the `shareService` operation provided by surrounding platform when necessary. Surrounding platforms are thereby informed of appearance or disappearance of a shared service.

5.2.3 Integration to the Scenario and Service Directories

SaS integrates the scenario and service sharing mechanisms to the scenario and service directories presented in Section 4.2.2.1. Sharing scenarios and services become part of the directories. Thus, they are persistent, easily retrievable and modifiable by the platform owner.

Listing 5.6 complete the context representation detailed in Listing 4.7. We add <service_sharing> and <scenario_sharing> attributes to the directories. Thus, a service can be shared with a platform (thanks to the platform identifier) or a platform category. Similarly, a scenario can be shared with a platform or a platform category. Scenario sharing implies to specify a scenario sharing mode to platforms selected for sharing.

Platforms have an identifier and an owner name. Identifier is unique, contrary to the owner name, so it is used for sharing identification. Alternatively, users can share a service with a platform category. In this case, all platforms registered with this category are selected for sharing. SaS distinguishes a platform identifier from a category because a category name cannot begin by a number (and platform identifiers are numbers).

```
<service_dir>  ::= service_directory {(<service><category>*<service_sharing>*)*}
<scenario_dir> ::= scenario_directory{(<scenario><category>*<scenario_sharing>*)*}

<category> ::= category <category_name> ;
<service_sharing> ::= share (<platform_id> | <category>);
<scenario_sharing> ::= share (<platform_id> | <category>) : <sharing_mode>;
<access_right> ::= descriptive | collaborative | copied
```

Listing 5.6: Scenario and service directories with sharing

Listing 5.7 illustrates a service directory example with a single service description, shared with the platform that has for identifier 17890804.

```
service_directory {
   service SwitchPower :
      device Kitchen_Lights;
      property deviceType : BinaryLight;
      property protocol : UPnP;
      operation SetTarget(Boolean) : void;
      operation GetTarget() : Boolean;
      category home;
      share 17890804;
}
```

Listing 5.7: Service sharing directory example

Listing 5.8 illustrates a scenario directory example, based on Figure 5.3 example, with a single scenario description. This scenario is shared with three platform categories: with the platform Janis_platform category in descriptive sharing mode, with the platform James_platform category in collaborative mode and with the platform Kurt_platform category in copied mode. In this example, the platform owner must have created the three platform categories previously and attached platform identifiers to them.

```
scenario_directory {
   scenario NightScenario :
      property creator : Antoine;
      property description : close the main door and shutters, adjust luminosity,
                             change thermostat value and listen to radio;
      property timeout : 4 hours;
      category home;
      share Janis_platform : descriptive;
      share James_platform : collaborative;
      share Kurt_platform : copied;
}
```

Listing 5.8: Scenario sharing directory example

5.2.4 Platform Substitution

Platforms are mediators between services and users. They enable users to define scenarios and are responsible for their execution. Platforms share their service directory and their ability to execute services. SaS therefore handles platform substitution and re-organise the system in order to maintain its functioning, and the best quality of service from its users' point of view.

5.2.4.1 The Substitute Operation

To support mobility and collaborative usages, SaS enables to share scenarios between platforms. When the original scenario providing platform disappears, platform owner may want that the scenario remains available in the environment. For example, a user creates a morning scenario on his / her laptop with the alarm clock set to 7 am. Later, the user changes his / her mind and wants to change the alarm to 7:30 am. If the laptop is off, the user needs to have access to the scenario on another active device (a mobile phone or a PDA). Additionally, a platform can be responsible for scenario execution. When this platform is shut-downed, platform owner may want to maintain scenario execution.

To do so, the *Collaborative* service provides the *substitute(String)* operation. The operation parameter corresponds to the scenario the user wants to maintain in the environment. Thus, the scenario owner can ask to another platform to maintain scenario availability and execution. The requested platform owner decides to take the scenario responsibility or not. The operation thus returns a Boolean that symbolizes the choice of the requested platform owner. Users can invoke this operation if they previously shared with the selected platform in copied mode the scenario that they want to maintain.

5.2.4.2 The Substitution

If the requested platform owner accepts, it gets the scenario description file (if not already did) and its execution log file (thanks to the *Collaborate* service). SaS platform log scenario execution advancement. Thus, the requested platform can easily know if the scenario was only installed or currently in execution. In both case, it locally installs the scenario. If the

scenario was running, the platform retrieves the execution advancement and continues scenario execution.

Figure 5.5 depicts the sequence diagram of a platform substitution example based on the example from Figure 5.3. Matt wants to maintain the *NightScenario* scenario in the environment. It previously shared this scenario in copied mode with the Kurt's platform (KP). MP invokes the substitute operation from the Collaborate service provided by KP. The KP accepts to take the scenario responsibility. Thus, KP retrieves the scenario description file and locally install it. Then, KP retrieves the scenario execution log file. KP checks the scenario execution advancement. If the scenario is running, KP continues scenario execution from where it was paused (not illustrated in the diagram).

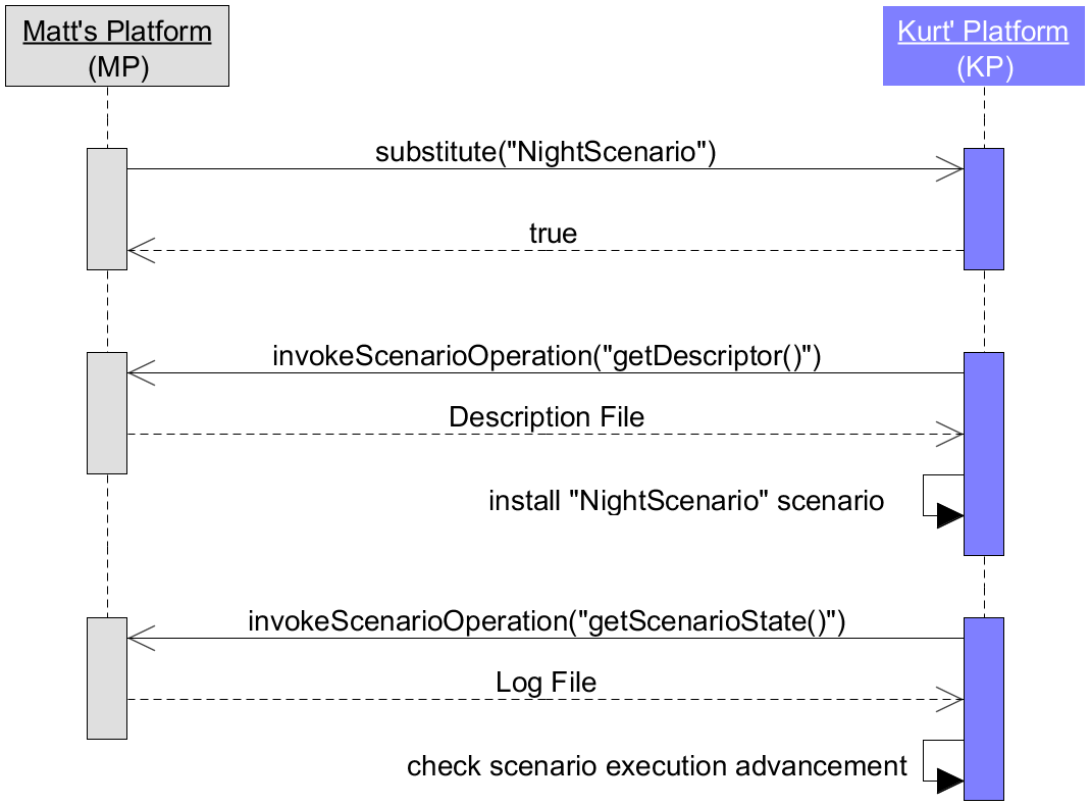

Figure 5.5: Sequence diagram of platform substitution example

5.3 Synthesis and Conclusion

In this chapter, we detailed how SaS manages scenario life-cycle and provides collaboration mechanisms.

5.3.1 Scenario life-cycle

SaS manages scenario life-cycle from their creation to their removal (*cf.* Section 5.1). Scenarios are memorized into description files. Moreover, SaS registers the scenario description into the scenario directory. This enables to make scenario description persistent. Moreover, a scenario orchestrator (dynamically generated per each scenario) registers a service to have access and control scenarios execution. As a service, scenario can be hierarchically composed. Additionally, SaS enables users to start, pause, resume and abort a scenario, but also to check its execution advancement and retrieve its description file.

Advantages of scenario life-cycle management by SaS:

+ SaS comprises a persistent scenario directory that enables users to easily manage their creations. Users can categorize scenarios which eases scenario directory browsing. Thus, users can easily reuse scenarios in the future.

+ SaS provides control commands for scenario execution that are easy to use (start, pause, resume and abort). SaS defines a scenario life-cycle that handles dynamic scenario status evolution.

+ SaS dynamically creates scenario orchestrators dedicated to the management of a scenario. A scenario orchestrator enforces, inter alia, the scenario execution life-cycle.

+ Scenario orchestrators provide to users a service for controlling scenario execution. Considered as services, scenarios can be thus used as such and hierarchically composed.

Limits and perspectives of scenario life-cycle management by SaS:

- The control commands for scenario execution do not enable users to execute a certain part of the scenario. When the scenario is installed, the execution always starts by the beginning. To execute a certain scenario part, users can modify the scenario to create a new one corresponding to their needs.

- As a single scenario orchestrator is generated for an installed scenario, only one scenario instance can run simultaneously. To solve this issue, users can duplicate a scenario description file and rename it to install a new scenario similar to the first one.

5.3.2 Platform Collaboration

SaS provides users with scenario sharing modes. Thus, users can specify for scenarios they created access rights to surrounding users. Moreover, platforms can also collaborate by sharing service access to surrounding platforms that may not have directly access to these services.

Advantages of SaS scenario sharing:

+ SaS comprises an access rights policy that enables users to adapt scenario sharing depending on surrounding users.

+ Thanks to the use of the platform directory, scenario sharing is easy (selection of platform categories) and adapts to users mobility. A scenario can be shared with a platform, even if it is not currently available. When the selected platform appears in the environment, it automatically receives the scenario shared without any other intervention from users.

+ Thanks to the scenario execution conflict management modes, SaS enables users to approve or not (and thus keep the control) a surrounding user request for scenario execution control.

+ Service sharing mechanism enables platform to collaborate. Thus, services only accessible by one platform can be integrated into the whole environment for surrounding platforms.

+ SaS enables to maintain scenario availability and execution when the original platform provider is shut-downed. This is the *platform substitution*.

Limits and perspectives of SaS scenario sharing:

- Scenario sharing implies a manual action. Typically, a new platform never seen before cannot be automatically selected for scenario sharing. In perspectives, we can consider that platforms export semantic informations and consider advanced recognition techniques that enable platforms automatic selection.

- SaS does not enable to share a specific part of a scenario. It could be interesting for platforms to choose a part of the scenario that they know they can execute and thus, improve collaboration.

- Platforms do not share services currently not available. It could be interesting, in perspectives, to enable platforms to share service descriptions that they know they can achieve in the future. Therefore, they could participate in a large collaborative scenario. Typically, in robotic science, a robot can announce that it can access to a service somewhere else, and thus can be given the order to invoke this service.

- Platform substitution requires that the platform owner asks for substitution before he/she closes the platform. Moreover, the requested platform owner must manually answer to the request. It could be interesting to develop automatic substitutions mechanisms.

5.3.3 Requirements Fulfillment

Scenario management involves scenario control, reuse and share. Scenario control (R3) implies, inter alia, to enable users to easily control scenario execution (R3.a). Scenario reuse requirement (R4) implies to maintain scenario description availability (requirement R4.a) and enable scenarios hierarchical composition (requirement R4.b). Moreover, platform collaboration enables users to share scenarios (R5). Besides, users must be able to select with who they want to share scenario with (sub-requirement R5.a) and what they want to share (sub-requirement R5.b). This chapter details how our contribution fulfills these five functional sub-requirements. Table 5.2 synthesizes the SaS' functionalities that answer to the sub-requirements.

Functional Requirements	**Functionalities Associated**
R3.a Scenario user control	The scenario orchestrator provides users with scenario execution control commands.
R4.a Scenario description availability	Scenarios are memorized into scenario description files and stored into the scenario directory.
R4.b Hierarchical composition	Scenarios are considered as services and thus, hierarchically composable.
R5.a Select what to share	SaS provides scenario sharing modes that enable to select what to share.
R5.b Select who to share with	Platforms export a *Collaborate* service, that enable users to do a selected collaboration.

Table 5.2: Fulfillment of requirements detailed in Chapter 5

In this Chapter, we have seen that users can control scenario execution, and SaS handles scenario life-cycle that dynamically adapts to users request. Next chapter is dedicated to SaS' mechanisms that manage scenario execution resilience.

Chapter 6

Scenario Step-by-Step Execution

Contents

In the previous chapter, we detailed how SaS manages scenario deployment (through a scenario orchestrator) and enables users to easily control (sub-requirement R3.a), reuse (requirement R4) and share scenarios (requirement R5). In this chapter, we focus on scenario execution resilience (sub-requirement R3.b).

With SaS, users have a service directory (*cf.* Section 4.2.2.1) and can thus define a scenario referring to services that are not simultaneously available. To better handle environmental changes (such as service disappearance), and to enable mobile scenario execution, SaS executes scenarios step-by-step. This mechanism enhances scenario execution resilience and mobility as it makes it possible to execute a scenario even if all the involved services are not simultaneously available.

Scenario execution needs to adapt to both context and available services. SaS contains a dynamic and adaptive service invocation mechanism through a *service broker*.

Moreover, even if the scenario execution is adaptive, errors (*e.g.* service disappearance, wrong service invocation, etc.) may occur. SaS thus proposes *recovery strategies* to anticipate and recover from these errors.

Section 6.1 introduces scenario execution scheduling and defines the scenario execution graph. Section 6.2 presents how SaS analyzes scenario description file to extract steps and manage scenario execution life-cycle. Section 6.3 is dedicated to the dynamic service invocation and details SaS's scenario recovery strategies that aim to enhance scenario execution resilience. Finally, Section 6.4 concludes this chapter.

6.1 Scenario Execution Scheduling

A scenario must be executable even if there is no available service instances (in the environment) for all required services. Moreover, a service failure should not impact the rest of the scenario and should thus be isolated. Therefore, SaS executes a scenario in blocks (that can involve a set of instructions).

These blocks correspond to *steps* in the scenario step-by-step execution. Steps are similar to transactions that respect the *ACID* (atomicity, consistency, isolation, durability) properties [33]. Each step must be executable atomically. Step execution may succeed or fail.

6.1.1 Scenario Structured Representation

To schedule scenario execution, SaS defines a *scenario execution graph*. Such a graph specifies the dynamics of the execution steps and can be compared to Petri nets [85] or finite-state machines [74]. The graph enables SaS to know and anticipate the possible consequences of astep execution.

Figure 6.1 provides a scenario execution graph example that results from the scenario analysis. We can notice that the scenario has been divided into several steps (of different types as detailed in Section 6.1.2.1). Details on the scenario execution graph details and its extraction are presented in the following subsections.

6.1.1.1 Scenario Execution Graph

The scenario execution graph is the result of a syntactic scenario analysis. It is a structured representation of the scenario in a set of atomic executable steps. It is an execution logic representation with control flows. Thus, the graph enables to anticipate and monitor scenario execution.

The scenario execution graph is a simple and directed graph noted $G = (N, E)$ with N being the set of its nodes and E the set of its edges. This graph is acyclic. There is a labeling function φ that attributes a label to each node of G.

$$\varphi : N \rightarrow \{1, 2, ..., n\}, \text{ with n the number of nodes.}$$

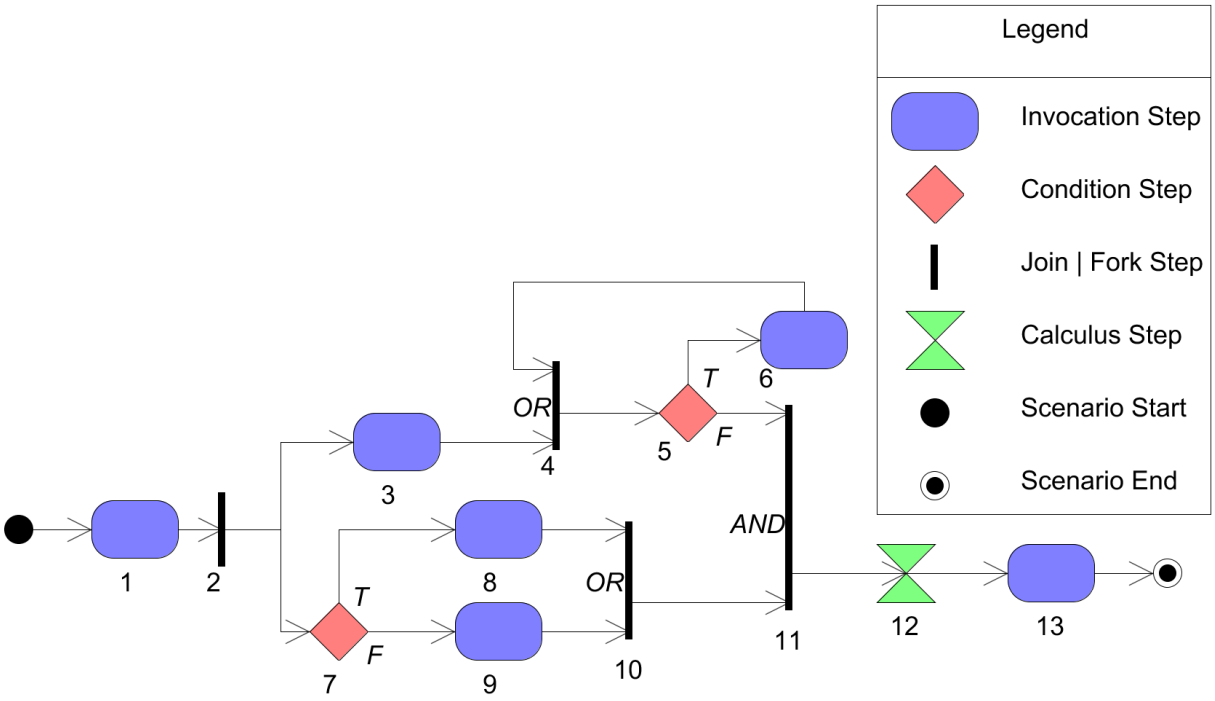

Figure 6.1: Scenario execution graph example

6.1.1.2 Scenario Execution Graph Elements

Nodes. Graph nodes correspond to scenario steps, except for the start and end nodes which are pseudo-steps. *Steps* have an attribute named `status`, that holds the step's state. Thus, there is a function that associates a status value to each node. Besides,the status dynamically evolves during scenario execution. The function takes in parameter an instruction pointer value i, that indicates a specific scenario progress state.

$$f_{status,i} : N \rightarrow \{EXECUTED, WAITING\_SERVICE, LOOP\_NOT\_FINISHED, IN\_EXECUTION, FAILED, PAUSED, TRUE, FALSE\}$$

Steps' status values `FAILED`, `PAUSED`, `IN_EXECUTION` and `WAITING_SERVICE` are inherent to all steps whereas other values are step type specific. Step status enable, inter alia, scenario orchestrators to monitor execution and users to check scenario execution progress. Moreover, steps contain an `execution` attribute, which represents the step's objective. This attribute depends on step types. Typically, a step responsible for a service invocation has for its `execution` value the corresponding *operation invocation*. The `execution` attribute is determined during scenario analysis and does not change.

Edges. Graph edges are the links between scenario steps. We note the set of edges: $E \subseteq N \times N$. To be executed, steps require that previous steps have an appropriate status (*e.g.* in a sequence,

each step needs that the previous step be executed before trying to execute itself). Therefore, we name the edges *precedence links*. Edges are annotated with the required status, that indicates when to pass from one step to the next. We define a f_{edge} function that associates each edge with its status.

$$f_{edge} : E \rightarrow \{EXECUTED, LOOP\_NOT\_FINISHED, TRUE, FALSE\}$$

6.1.2 Correspondences with SaS-SDL elements

The scenario execution graph enables to schedule scenario execution. Scenario execution graph elements are related to scenario definition elements. This subsection details the scenario execution graph elements and their correspondences with the scenario definition elements.

6.1.2.1 Step Types

A scenario is not just a set of services to invoke. Services invocations are aggregated with conditions, loops, etc. We thus define two types of steps: *action steps* that invoke services, and *connection steps* that combine steps when necessary (*e.g.* join step used for a parallel action block). These two step types are divided into four sub-types.

Figure 6.2 depicts the scenario step class hierarchy. We can see the five concrete types at the bottom. Other step type (*Step*, *ConnectionStep* and *ActionStep*) are abstract. All step types share the same two attributes: `status` and `execution`.

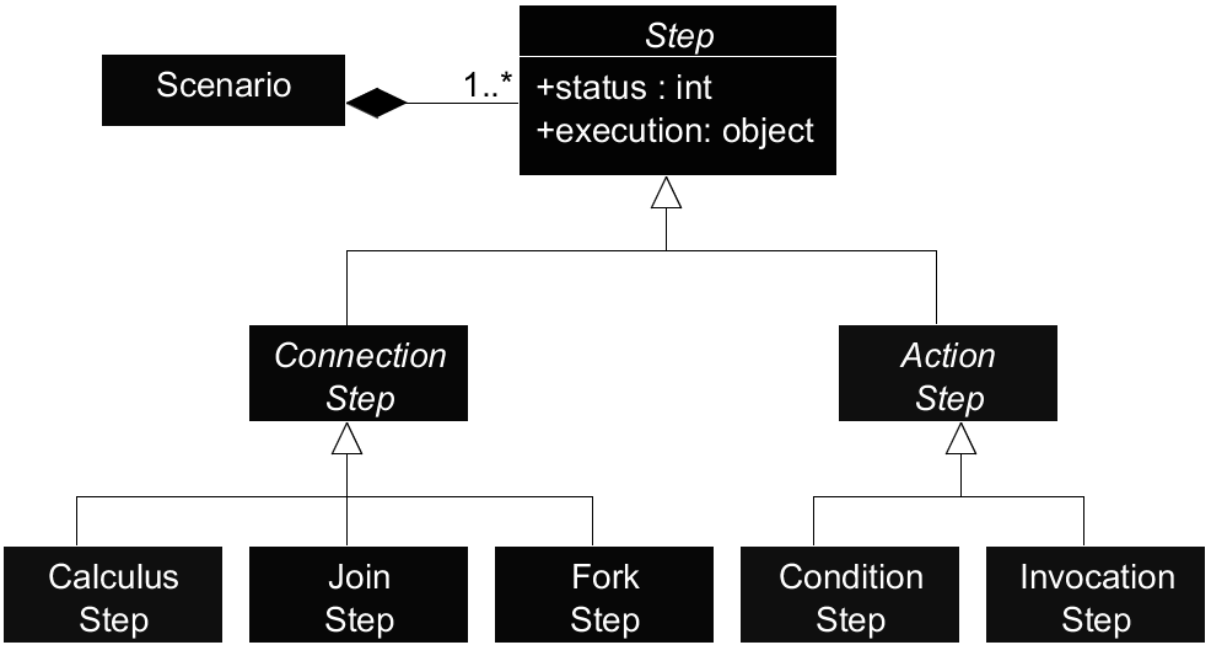

Figure 6.2: Scenario steps type class diagram

We detail here the different step types and their correspondences with the element of SaS-SDL.

- *Action step*. This is an executable step that contains service invocations. In SaS-SDL, services are invoked in *service execution* actions and in *conditions*. Thus, we define two kinds of action steps:
  - *Invocation step*. It is created for *service execution* actions. Its `execution` attribute contains the operation invocation. When the step is executed, its status is set to `EXECUTED`. This step has a single entry edge and and a single output edge.
  - *Condition step*. It is used for *conditions* (used in the *conditional statements*, *while loops* and *conditional events* of scenario actions). Therefore, its `execution` attribute contains a condition, which can be a complex condition (combination of conditions aggregated with binary operators) or a unary condition. Each unary condition contains an operation invocation. The object obtained from the invocation is compared with a specific value to obtain the condition value. When the condition has been successfully evaluated, the status of the condition step is set to the corresponding condition value (*i.e.* `TRUE` or `FALSE`). Condition steps have a single entry edge and two output edges corresponding to the two different values that the step's status can take.
- *Connection Step*. This type of steps is used to connect action steps when necessary. Typically, action steps only have one entry edge. Thus, if an action block is defined as parallel, we need to create a step that joins the parallel action steps. Connection steps do not invoke services but structure the execution flow. We specify three types of connection steps:
  - *Join step*. It is used to join action steps (parallel action blocks, conditional statement actions and loops). A join step could require the execution of several steps to be crossed. Thus, next steps only require the join step to be satisfied. Join steps have an empty *execution* (they do not invoke any service). Join steps have a join mode, that specifies how their precedence links are evaluated. This join mode can take two values: `AND` (all precedence links are required), and `OR` only one precedence link is required). We name a join step in join mode AND (*resp.* OR) an AND (*resp.* OR) join step. They can have several entry edges and only one output edge.
  - *Fork step*. It is used for parallel action blocks to separate the graph into several branches. There are as many branches as elements of the action block defined as parallel. Fork steps have an empty *execution* (they do not invoke any service). They have one entry edge and several outputs edges.
  - *Calculus step*. This type of steps are used by SaS when it necessitates a calculation, without invoking services. Typically, the repeat loop action needs to calculate how

many times actions have been executed (and checks if they need to be executed again). The `execution` attribute contains the calculus to be done. Calculus steps have one entry edge and can have two output edges.

6.1.2.2 Step Preconditions and Step Postconditions Status

Steps are elements of the scenario execution graph. Thus, they are interconnected with edges that symbolize the execution order. Moreover, action steps invoke services and thus, need corresponding service instances. Additionally, we define step postconditions status that correspond to the step's execution end status.

Step preconditions. Preconditions enable to check if a step is ready to be executed. Thus, SaS always checks if the step preconditions are satisfied before executing a step. *Preconditions* are twofold and specific to each step:

- *Step precedence satisfaction*. Some steps can only be executed if preceeding ones have been correctly executed (or have a specific status). Steps are therefore connected by precedence links to other steps. These precedence links correspond to the edges of the scenario execution graph. A precedence link contains an expected status (*i.e.* the edge is annotated). Action steps only have one precedence link whereas join steps can have many.
- *Service presence*. Preconditions check the presence of the services involved in the step. SaS therefore ensures that a step is executed only when all the involved services are simultaneously available.

Step preconditions are automatically computed from its involved services and its precedences. Some precondition examples, based on Listing 4.11, are: step 2 requires the execution of step 1 and the presence of a *Shutter* service (whatever the provider device); step A requires the execution of step 4 or the execution of step 5 (and does not require any service to be present).

Step postcondition status. *Postcondition status* correspond to the step's execution end status. They are threefold:

- Step execution may succeed. The services involved in the steps have been correctly invoked.
- Step execution may also fail. There are several reasons for a step to fail:
  - Service invocation error. The service is present but an error occurs when it is invoked. This can be due to wrong parameters, etc.
  - Service disappearance. The service disappears during the step execution.

- Unresponsive service. The service is available but does not answer.

- Step may also be finished. It can be due to:

  - User interruption. The scenario can be interrupted by a user.
  - Scenario timeout. A scenario timeout can be reached and scenario be aborted.

  Connection steps thus have three possible postconditions status (step succeed, scenario interrupted and scenario timeout reached) whereas action steps have six possible postconditions status (same as connection steps plus the three postconditions status due to a service failure).

Figure 6.3 adds to Figure 6.2 the step's pre-and-post condition status representation. The added classes are represented in white. We can see that steps have several preconditions (which number and nature depend on steps). Connection steps can have several precedence links but do not require any service presence, whereas action steps have a single precedence link but contain several service invocations and thus need several services to be present. A precedence links points to a single step. Moreover, steps contain the list of precedence links that points to them.

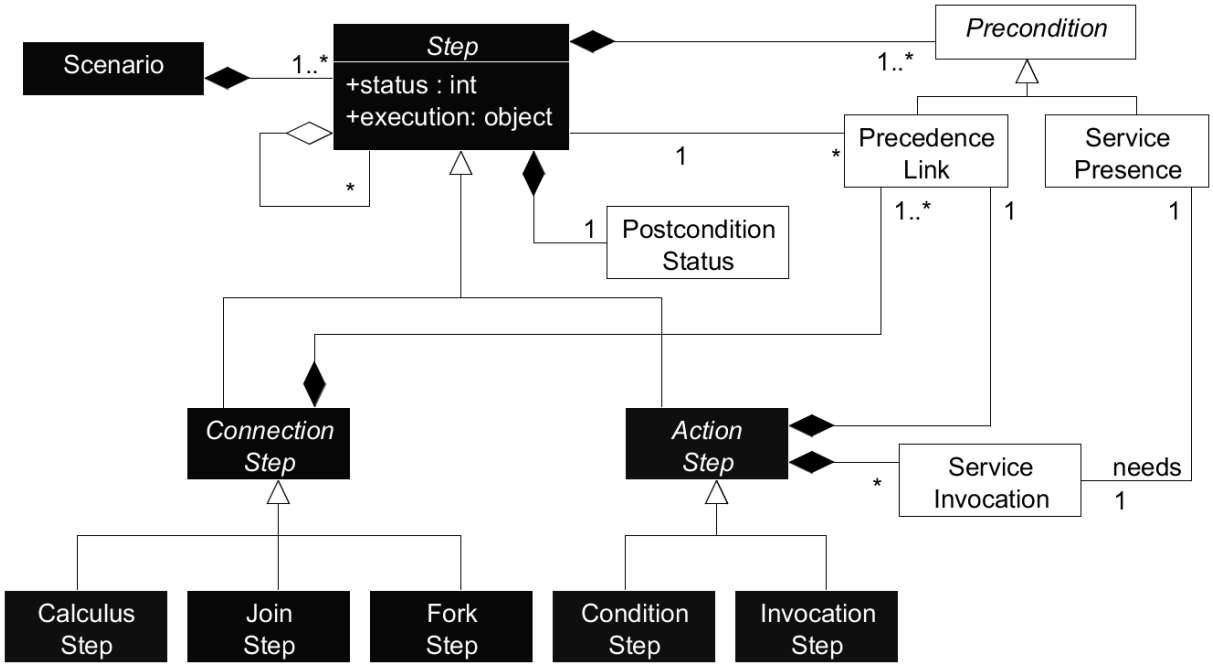

Figure 6.3: Scenario steps class diagram

6.2 Static Scenario Analysis to Prepare its Step-by-Step Execution

The static analysis of a scenario description builds the corresponding scenario execution graph. It is the scheduling of scenario execution and defines what must be executed atomically. We first present here SaS's mechanisms to extract steps from scenario description files, and then, how scenario execution graphs are computed from the extraction of scenario steps. All these processes are automatic and seamless for the user. The algorithms to transform a scenario description into an execution graph are detailed in Appendices A.

6.2.1 Step Extraction

To extract the steps, we do a model transformation from an instance of the SaS-SDL scenario model to the scenario execution graph model. The type of the extracted step depends on the action type (each action implies one or several steps of different possible types). During this extraction, we specify the step `execution` attribute and step-preconditions. Step preconditions are twofold, and thus, differently defined:

- Precedence link. SaS extracts the steps one after the other. When SaS extracts a step, it needs to specify its precedence links. Typically, SaS transforms a sequence action block with a precedence link that points to the previous defined step. Thus, SaS can attach the precedence link to the steps extracted from this action block.

- Service presence. Action steps may invoke services. Services to invoke are specified in the `execution` attribute (that can be a condition or a service execution action). SaS uses this to define the service presence preconditions.

This subsection presents the different model transformations that enable to extract scenario steps. We illustrate each scenario element transformation with a Figure that presents (on the left) the SaS-SDL scenario element and its transformation (on the right).

A scenario contains an action block. An action block contains actions and/or action blocks, and is executed sequentially or in parallel. We first present the different action block transformations (depending on the block's execution type) and then, how SaS transforms a scenario action (depending on the action type).

6.2.1.1 Action Block Transformation

The action block transformation is different when the action block is executed sequentially or in parallel. Thus, we define two different transformations for action block transformation depending on the execution mode.

Sequence Action Block. If an action block is defined as a sequence, we transform each element of the action block. This transformation returns a precedence link that points to the last step defined. We use it for the next element of the action block. Depending on the current element type, we transform it as either an action block or an action.

In the scenario example from Listing 4.11, the scenario action block (from line 4 to line 24) is defined as a sequence. It contains three elements: two actions (at line 4 and from line 20 to 22) and one action block (from line 5 to line 19). Figure 6.4 illustrates the transformation of this sequence action block example. We transform each element with the precedence link obtained by the transformation of the previous action block element.

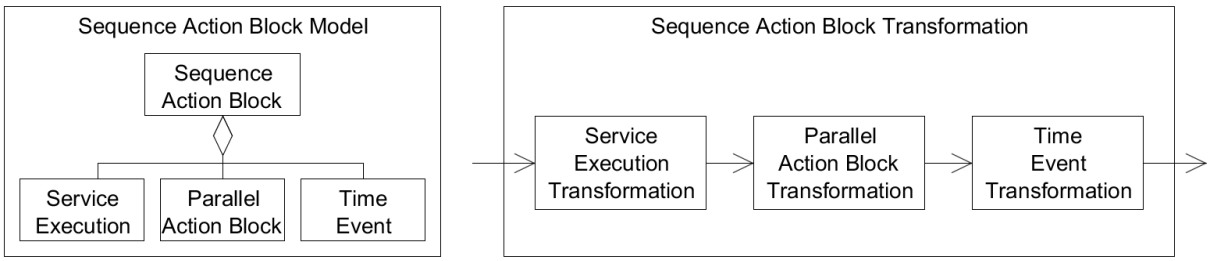

Figure 6.4: Sequence action block model transformation example

Parallel Action Block. If an action block is executed in parallel, we define a fork step at the beginning of the transformation. Then, each element of the transformed action block requires this fork step to be executed. Moreover, we define a AND join step for the end of the action block. Each precedence link extracted from the transformation of the action block elements is added to this join step. Thus, this join step is only executed if all its precedence links are satisfied (*e.g.* all steps computed from the action block are executed). So, scenario execution continues if this join step has the EXECUTED status. Such join step acts as a synchronization step for the content of the parallel block.

In the scenario example from Listing 4.11, the action block from line 5 to line 19 is defined as parallel. It contains an action block (defined as a sequence) and a conditional statement action. Figure 6.5 illustrates the transformation of this parallel action block example. We can notice the fork and join steps, that have outgoing precedence links annotated with `E` for `EXECUTED`.

6.2.1.2 Action Transformation

Action transformation depends on the action's type. Therefore, we define several transformations to extract the different actions. Action transformations return a precedence link, that points to the last step of the action. This precedence link is used to transform next action (or action block).

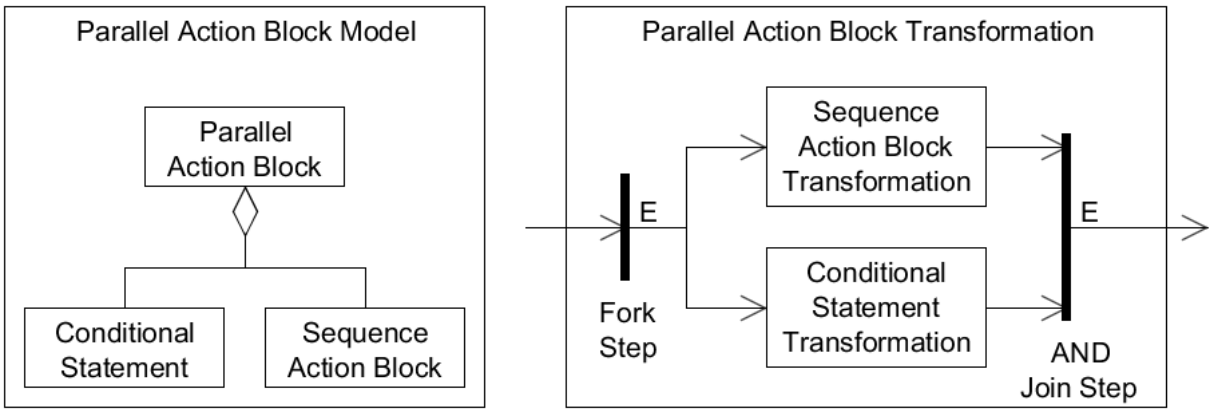

Figure 6.5: Parallel action block model transformation example

Service Execution. The service execution action contains an operation invocation. We define a single invocation step that is initially set to this operation invocation as its `execution` attribute value. The next step requires this step's status to be `EXECUTED`. Figure 6.6 illustrates the service execution model transformation.

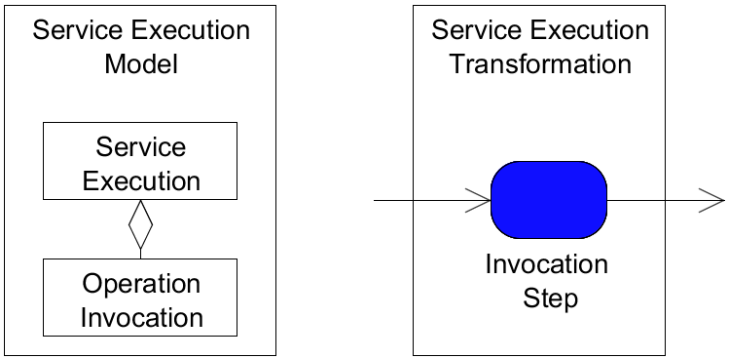

Figure 6.6: Service execution model transformation

Conditional Statement. The conditional statement action contains a condition and two action blocks. Figure 6.7 illustrates the conditional statement transformation. We define a condition step for the condition. The condition step has for its `execution` attribute value the condition. The condition step has two precedence links as outputs, corresponding to the two possible condition evaluation values. Then, we transform the two action blocks (then - else) of the conditional statement. These transformations are done with a precedence link pointing on the condition step but with different required status (`TRUE` or `FALSE`). Thus, the steps representing each of the alternative action blocks are executed only when the condition value corresponds. Transformations return two precedence links that we attach to an OR join step. Therefore,

the OR join step only requires one precedence link to be fulfilled (*i.e.* one of the action block transformation executed). Finally, a precedence link that requires the join step to be executed is defined.

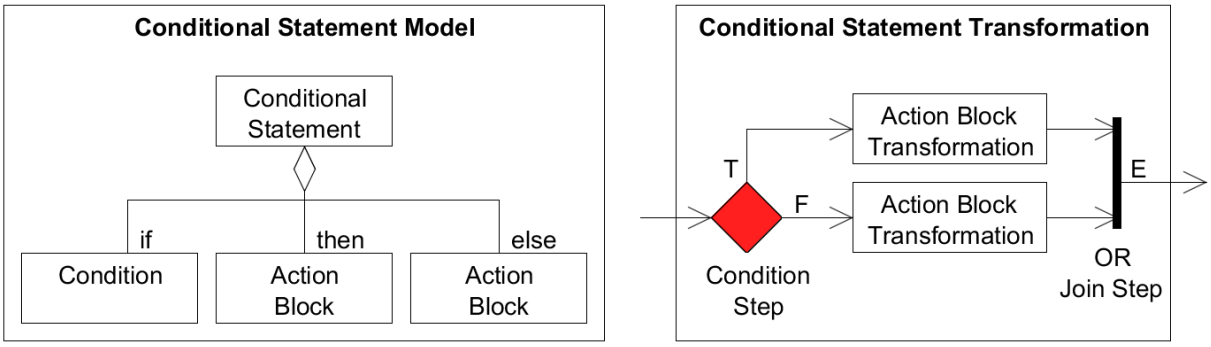

Figure 6.7: Conditional statement model transformation

While Loop. A while loop action contains a condition and an action block to execute while the condition remains `TRUE`. Figure 6.8 illustrates the while loop transformation. SaS first creates an OR join step that is used to create a loop in the execution graph. Then, a condition step is defined for the condition. This condition step requires the previously defined join step to be executed. Then, we transform the action block of the while loop action. This transformation is connected to the condition step by a precedence link that has a status set to `TRUE`. When the condition evaluation equals `TRUE`, the steps defined from the action block transformation are thus executed. The precedence link that results from the action block transformation is connected to the join step. When this precedence link is fulfilled (*i.e.* the last step of the while action block transformed is executed), the join step is executed again and thus, the condition step reevaluates its status. If the condition still is `TRUE`, steps in the while action block are executed again. To finish the while loop, SaS defines a precedence link, that requires the condition step's status to be set to `FALSE`. Thus, when the condition becomes `FALSE`, scenario execution can continue.

Repeat Loop. A repeat loop contains an action block to be executed several times. Figure 6.9 illustrates the repeat loop transformation. SaS creates an OR join step at the beginning. Then, we transform the action block and connect it to the join step with a precedence link that requires the join step to be in the EXECUTED status. We then define a calculus step. We connect it to the action block transformation with the precedence link that results from the action block transformation. This calculus step enables to repeat the set of steps defined in the action block transformation. To do so, the join step has a precedence link with the calculus step. The calculus step calculates if the repeat action block has been executed as many times as

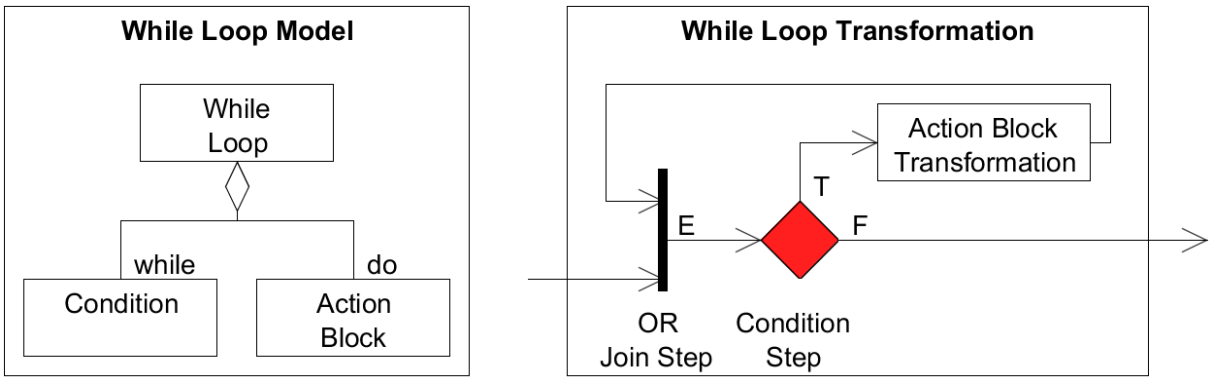

Figure 6.8: While loop model transformation

needed. It thus has two output precedence links which correspond to two possible status values: `LOOP_NOT_FINISHED` (abbreviated as `N_F` in the Figure) and `EXECUTED` (abbreviated as `E`).

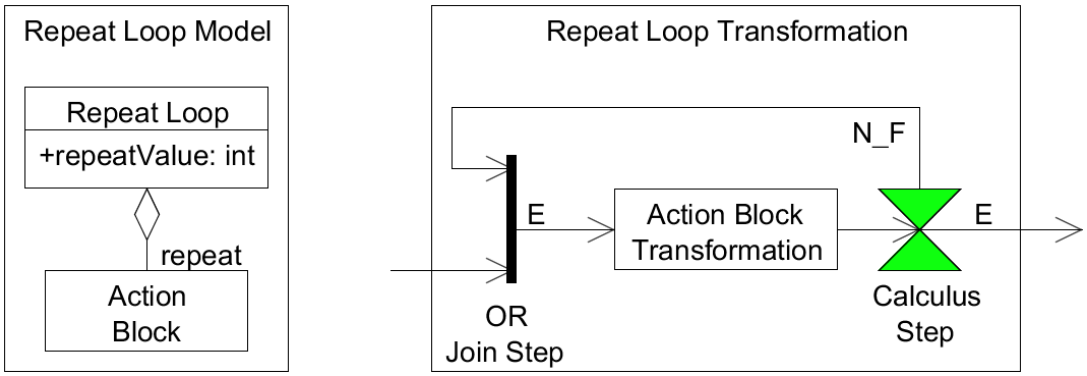

Figure 6.9: Repeat loop model transformation

Conditional Event. In a conditional event action, an action block is executed when a complex condition becomes `TRUE`. Figure 6.10 illustrates the condition event transformation. The execution graph produced by the transformation starts with an OR join step. Then, we define a condition step for the condition. Thereafter, we transform the action block that depends on the event condition and connect it to the condition step with a precedence link with a `TRUE` status. We add to the OR join step a precedence link that points to the condition step with its status set to `FALSE`. If the condition step's status is set to `TRUE`, the action block is executed. Alternatively (step's status set to `FALSE`), the join step is executed again and its condition step is re-evaluated. Thus, we iterate the evaluation of the condition until it becomes `TRUE`.

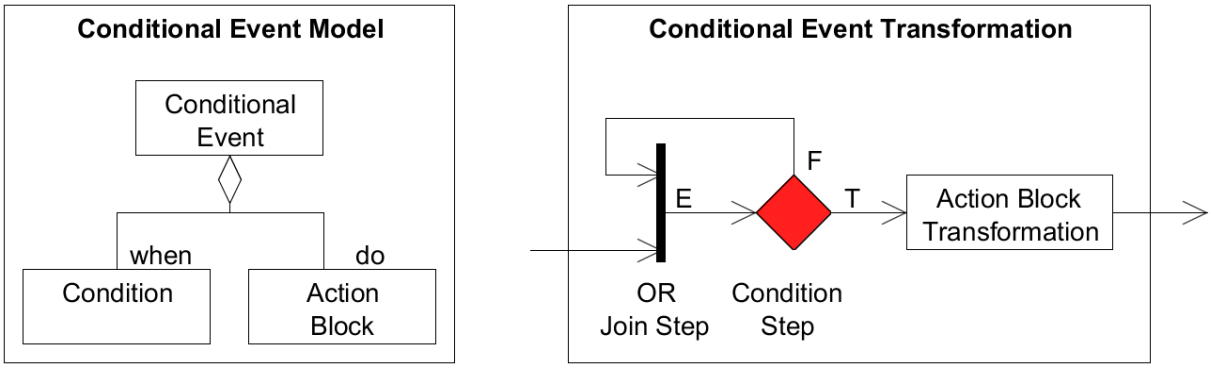

Figure 6.10: Conditional event model transformation

Time Event. A time event action contains an action block to be executed at a specific date. Figure 6.11 illustrates the time event transformation. The specific date to execute the action block is the value of the `timeEventValue` attribute of the Time Event object. The execution graph produced by the transformation starts with a calculus step. This step monitors the occurrence of a specified time event. The status of this step becomes `EXECUTED` when the time event is reached. Then, we transform the action block with a precedence link that points to the calculus step and requires this step to have the `EXECUTED` status.

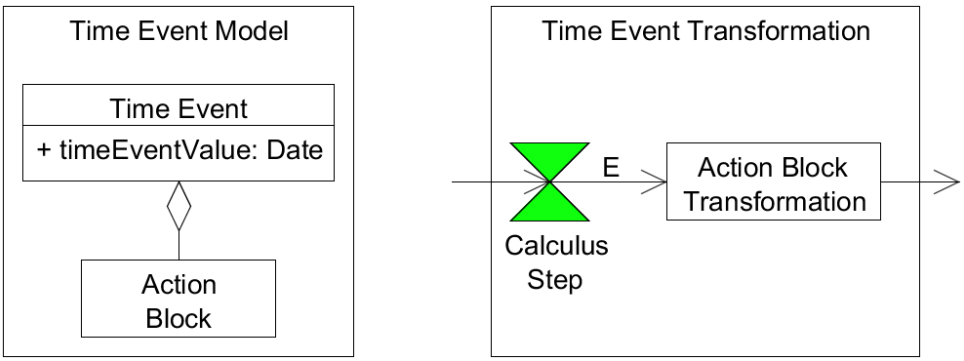

Figure 6.11: Time event model transformation

6.2.1.3 Step Extraction Example

The example scenario (*cf.* Listing 4.11) is decomposed into 13 steps: 6 invocation steps, 2 condition steps, 4 join steps and 1 calculus step. Figure 6.1 illustrates the scenario execution graph defined thanks to the model transformation. We detail here all the steps with their corresponding lines in the example scenario listing.

Step 1: Invocation Step (line 5).

Step 2: Fork Step (begin of the action block executed in parallel).

Step 3: Invocation Step (line 8).

Step 4: Join Step (beginning of the while loop).

Step 5: Condition Step (line 9).

Step 6: Invocation Step (line 10).

Step 7: Condition Step (lines 13 and 14).

Step 8: Invocation Step (line 15).

Step 9: Invocation Step (line 18).

Step 10: Join Step (after the conditional statement).

Step 11: Join Step (after the action block executed in parallel).

Step 12: Calculus Step (line 21).

Step 13: Invocation Step (line 22).

6.2.2 Scenario Execution Life-Cycle

Step division enables SaS to schedule scenario execution. This capability is completed with a *logging system* that records scenario *step-by-step* execution status. In addition, thanks to step-by-step management, scenario execution is not aborted if a required service becomes unavailable. Moreover, the isolation property of the steps' execution also ensures that a failed step will not interrupt the whole scenario execution or alter its other steps.

Scenario execution depends on user actions (*e.g.* start, pause, etc.) and on environmental changes (*e.g.* service appearance, disappearance, etc.). Thus, SaS adapts scenario execution to these events by changing the running state of the scenario, as specified by its life-cycle. Figure 5.2 depicts the state diagram of the scenario *installed* state. In this section, we details the internals of the installed state.

6.2.2.1 Scenario Running

Figure 6.12 depicts the activity diagram the scenario running state. When the scenario is started, SaS schedules the scenario timeout (selected by the user). SaS-SDL enables to use *jokers* for parameter values. Parameter values not defined at scenario definition must be thus specified at scenario execution. So SaS replaces the jokers present in operation invocations with their actual values. Then SaS looks for next steps in the scenario execution graph. At start, the succeeding

steps correspond to the steps that have no precedence link. If the scenario is resumed, the succeeding steps are the steps previously paused.

For each step of the succeeding step list, SaS checks if its precedence links preconditions are satisfied. If not, SaS does not treat this step for the moment. If the step has its precedence links satisfied, SaS checks the step type. If it is a connection step, it does not require any service and can thus be executed. Instead, if it is an action step, SaS searches for available service instances in the service directory. The service selection mechanism is detailed in Section 6.3.1.1. If all required services are available, SaS can execute the step. Instead, to be aware about the missing service and execute the step when an appropriate service appears, SaS sends a request to track the required services' apparition and sets the step's status to `WAITING_SERVICE`. Therefore, SaS knows that this step needs one or more specific services and is waiting for them. When the requested services become available, SaS executes the step.

Before executing the step, SaS sets the step's status to `IN_EXECUTION`. Step execution can succeed or fail (*e.g.* in case of service disappearance, etc.). In both cases, SaS sets the step's status to the postcondition status value when step execution ends. If execution fails, SaS applies recovery strategies as that is detailed in Section 6.3.2.2. Instead, if execution succeeds, SaS checks if the scenario execution is finished. If this is not the case, SaS looks if the executed step has succeeding steps and the scenario execution continues.

6.2.2.2 Scenario Deployed

A scenario installed, finished or aborted falls in the deployed state. Figure 6.13 depicts the activity diagram of the scenario deployed state. In this state, SaS initializes scenario execution. Thus, it resets the scenario steps' status and the scenario timeout. Then, it removes the joker values from a previous execution.

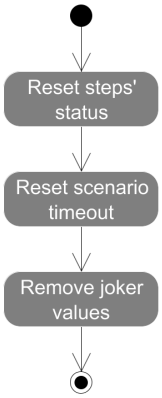

Figure 6.13: Activity diagram of the scenario deployed state

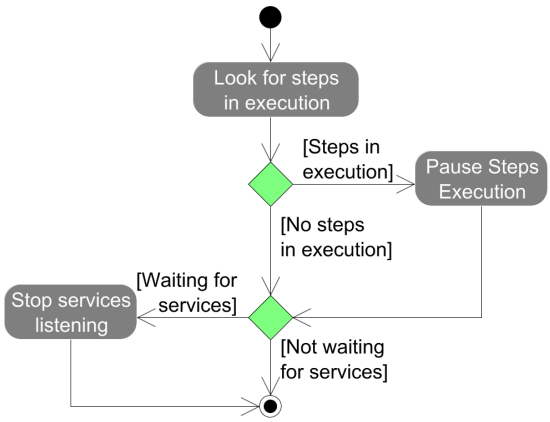

Figure 6.14: Activity diagram of the scenario paused state

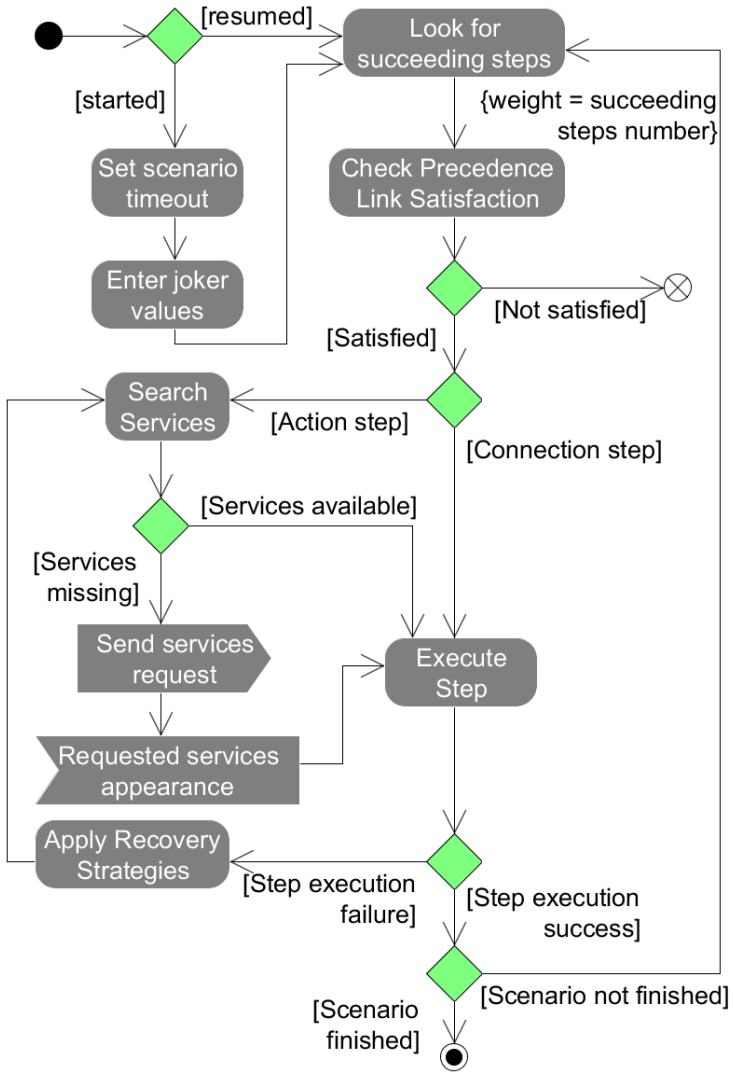

Figure 6.12: Activity diagram of the scenario running state

6.2.2.3 Scenario Paused

A scenario in execution can be paused. Moreover, a scenario automatically aborted traverses *paused* state before it returns in the deployed state. Figure 6.14 depicts the activity diagram of the scenario *paused* state. When the scenario is paused, SaS first retrieves the steps which are currently being executed. These steps have their status set to `IN_EXECUTION`. Steps being executed are paused: their execution is interrupted, and their status set to `PAUSED`.

6.2.3 Step Execution

The scenario orchestrator (presented in Section 7.1.1.4) is responsible for scenario execution and thus, step execution. Steps are executed differently depending on their type. This subsection

presents how the different kinds of steps are executed.

6.2.3.1 Invocation step

An invocation step contains an operation invocation, that SaS tries to execute. If the invocation is well done, step's status is set to `EXECUTED`. If an error appears (*e.g.* service disappearance), the step's status is set to `FAILED`. Algorithm 1 summarizes the invocation step execution.

Algorithm 1 Step Execution: Invocation Step

```
1: function executeInvocationStep(InvocationStep step): void
2:     try
3:         invokeOperationInvocation(step.getExecution())
4:     catch(Exception)
5:         step.setStatus(FAILED)
6:         break
7:     end try-catch
8:     step.setStatus(EXECUTED)
9: end function
```

An operation invocation contains values, that can in turn be obtained by operation invocations. A service is invoked thanks to a *ServiceInvocator*, that implements a specific protocol and executes the appropriate service invocation. Algorithm 2 presents how SaS invokes an operation.

Algorithm 2 Step Execution: Operation Invocation

```
1: function invokeOperationInvocation(OperationInvocation opInvoc): Object
2:     for all (parameter in opInvoc.getParameterList()) do
3:         if (parameter.isInstanceOf(OperationInvocation)) then
4:             parameter.setValue(invokeOperationInvocation(parameter))
5:         end if
6:     end for
7:     return ServiceInvocator.invokeService(operationInvocation.getServiceInvocation())
8: end function
```

6.2.3.2 Condition step

A condition step is defined to execute a condition. Its resulting status is the evaluation of the condition. A condition is defined (*cf.* Section 4.3) as a unary condition or as a complex condition that combines conditions either unary or complex (binary tree decomposition). Algorithm 3 describes condition evaluation.

Algorithm 3 Step Execution: Condition Evaluation

```
1: function evaluateCondition(Condition condition): boolean
2:     switch(condition.isType())
3:         case UNARY_CONDITION
4:             return evaluateUnaryCondition(condition)
5:         case COMPLEX_CONDITION
6:             return evaluateComplexCondition(condition)
7:     end switch
8: end function
```

If the condition is a unary condition (*e.g.* line 6 of scenario example from Listing 4.10), it is defined as a single operation invocation to be compared with a parameter value. SaS retrieves the object returned by the operation invocation (thanks to the *invokeOperationInvocation* function, seen in Algorithm 2). Then, it calls a function that simply evaluates the expression (*e.g.* 25 < 20) and returns a boolean. Algorithm 4 presents how SaS evaluates a unary condition.

Algorithm 4 Step Execution: Unary Condition Evaluation

```
1: function evaluateUnaryCondition(UnaryCondition condition ): boolean
2:     OperationInvocation opInvocation = condition.getOperationInvocation()
3:     Object object = invokeOperationInvocation(opInvocation)
4:     Object parameterValue = condition.getParameter().getValue()
5:     return evaluateExpression(returnValue,parameterValue,condition.getComparisonOperator)
6: end function
```

If the condition is a complex condition, it contains several conditions combined with binary operators (*i.e.* AND/OR). SaS evaluates recursively the conditions, traversing the corresponding binary tree down to unary conditions, calculating their values, and combining them with the used boolean operator. Then, SaS returns a boolean that represents the condition's evaluation value. Algorithm 5 represents the complex condition evaluation.

6.2.3.3 Join step

A join step synchronizes parallel executions, conditional statements and while loops. A join step is used as a control flow connector and thus has no concrete execution, so its status is directly set to EXECUTED.

6.2.3.4 Calculus step

A calculus step can contain a time event value (defined for a time event action) In this case, SaS simply sets the step's status to `EXECUTED` when the specified time has elapsed. Alternatively, SaS also uses a calculus step for repeat loops (*e.g.* `(repeat n times)`), where it is placed at

Algorithm 5 Step Execution: Complex Condition Evaluation

```
1:  function evaluateComplexCondition( ComplexCondition cplxCondition ): boolean
2:      boolean cplxCdtnValue = evaluateCondition(cplxCondition.getConditionList(0));
3:      for (int i=1; i<binCondition.getConditionList().size(); i++) do
4:          boolean binCdtnValue = evaluateCondition(cplxCondition.getConditionList(i))
5:          switch(cplxCondition.getBinaryOperatorList(i-1))
6:              case AND
7:                  cplxCdtnValue = (cplxCdtnValue AND binCdtnValue)
8:              case OR
9:                  cplxCdtnValue =(cplxCdtnValue OR binCdtnValue)
10:     end for
11:     return cplxCdtnValue
12: end function
```

the end of the action. In this case, the join step calculates how many times the loop has been executed. If the loop is not finished, the join step's status is set to `NOT_FINISHED`. Instead, it is set to `EXECUTED`.

Synthesis. In this section, we detailed how SaS transforms a scenario description to obtain a scenario execution graph that enables a step-by-step execution management. During scenario execution, SaS needs to invoke services. To do so, it is provided with a directory of available services. Next section is dedicated to the selection and the invocation of services during scenario execution.

6.3 Dynamic and Adaptive Service Invocation

Step-by-step execution enables, inter alia, to execute a scenario even if all required services are not present. Thus, a scenario can be executed in different places over time. This implies that service invocation has to be dynamic and adapt to environmental changes. Scenario execution can thus leverage as many resources provided by the environment as possible.

6.3.1 The Service Broker

A scenario is composed of service invocations. Therefore, executing a scenario implies to invoke appropriate services. Service invocation can be independent of the provider devices (use of the keyword *any* in scenario definition, *cf.* Section 4.3.2). Additionally, users can define a scenario at a moment in a certain place and decide to (re)execute it later in another place. Thus, a service that is present at scenario definition (discovered with a specific protocol) may be missing at scenario execution, but another one (provided by another device with another protocol) can be a substitute. Therefore, service invocation has to be adaptive. This implies to

dynamically select an appropriate available service. To do so, SaS uses a *service broker*. This service broker is responsible for service matchmaking and invocation.

6.3.1.1 Service Matchmaking

The pervasive system must select a concrete service that matches with an abstract service description present in the service composition (*i.e.* the scenario). This is *service matchmaking*, which is primordial in service composition. Service matchmaking implies to find an available service instance available that proposes the same functionalities as a service description. If several possibilities exist, the pervasive system must select the best one according to their quality of service.

Service selection by service functionality. SaS selects services independently of their technology or management protocol. This enhances *interoperability*. Discovered services are transcribed into SaS-SDL (*cf.* Listing 4.2) and registered in the service directory. SaS looks for appropriate services in this directory. This search depends on the scenario definition. Users can use a quantifier (*any* / *all*) or define a service filter (*cf.* Section 4.3.2). The selection of services according to the functional properties is based on the exact syntactic match between the service description provided by suppliers and the operation invocation required by the consumer. For instance, when SaS searches for an available service inside the service directory example from Listing 4.8, that corresponds to the operation invocation `Kitchen_Lights.SwitchPower.GetTarget()`, it finds that a *SwitchPower* service ,that has been, discovered with UPnP corresponds.

Quality of service. Service selection depends on a quality of service criterion. Quality of service relates to both service characteristics and user preferences. Moreover, a pervasive system should learn from invocation errors to adapt and improve service selection.

Distance preference policy. Since we do not specify a specific network protocol, it is difficult to take into consideration technical aspects such as bandwidth. However, SaS applies a preference policy based on the distance of the service implementation locations to select services. SaS thus prefers closer services (that we assume to be supposedly easier to reach). The preference order is:

1. Local services.
2. Directly reachable services (discovered by the platform).
3. Services shared by another platform (*cf.* Section 5.2.2.3).

Service category preference policy. Users can also define service preferences based on categories (*cf.* Section 4.2.2). Available services registered in a category are considered as

preferable to uncategorized services from devices.

Service invocation error learning. Services can be discovered as being available and fail to answer (or return errors). SaS learns from these errors and blacklists the more frequently faulty services. Such service instances are therefore not invoked anymore. SaS achieves this by creating a specific service category named *blacklist*. Users can easily see the blacklisted service instances and therefore modify the list.

Service equivalence. If a required service is not available, an equivalent service can be used instead. Selection of an equivalent service is handled by these ordered strategies:

1. Same service but without required properties (use of a filter for service invocation).
2. Same service but not from the chosen device.
3. Same service but without required properties and not from the appropriate device.

6.3.1.2 Service Invocation

A SaS platform can implement several discovery protocols. The components deployed on a SaS platform that enable to discover services are also responsible for their invocation. We name these components *gateways*. Thus, a gateway that discovers a service, registers it into the service directory along with its used protocol. When the service broker searches for an available service corresponding to a service description, it can therefore retrieve which gateway to use for service invocation. We distinguish two types of invocations:

- Direct invocations (local or remote services). A gateway directly retrieves and invokes the service instance that it discovered previously.
- Shared Service Invocation. SaS platforms can share service access thanks to the *Collaborate* service (*cf.* Section 5.2.2.5). Thus, the broker tries to remotely invoke the service operation through the *Collaborate* service.

Figure 6.15 illustrates how the service broker interacts with the service directory and gateways. To manage the service directory, SaS provides a *Service Directory Manager*. It is responsible for the management (create, read, update and delete) of service descriptions into the service directory. This figure details that gateways interact with the *Service Directory Manager* which maintains the service directory *updated* and enables the *service broker* to invoke services.

6.3.2 Scenario Fault-Tolerance Mechanisms

Because of both mobility and use of wireless networks, services may disappear, even for a few seconds. The system has to handle these interruptions, especially when the disappeared services are involved in a running scenario.

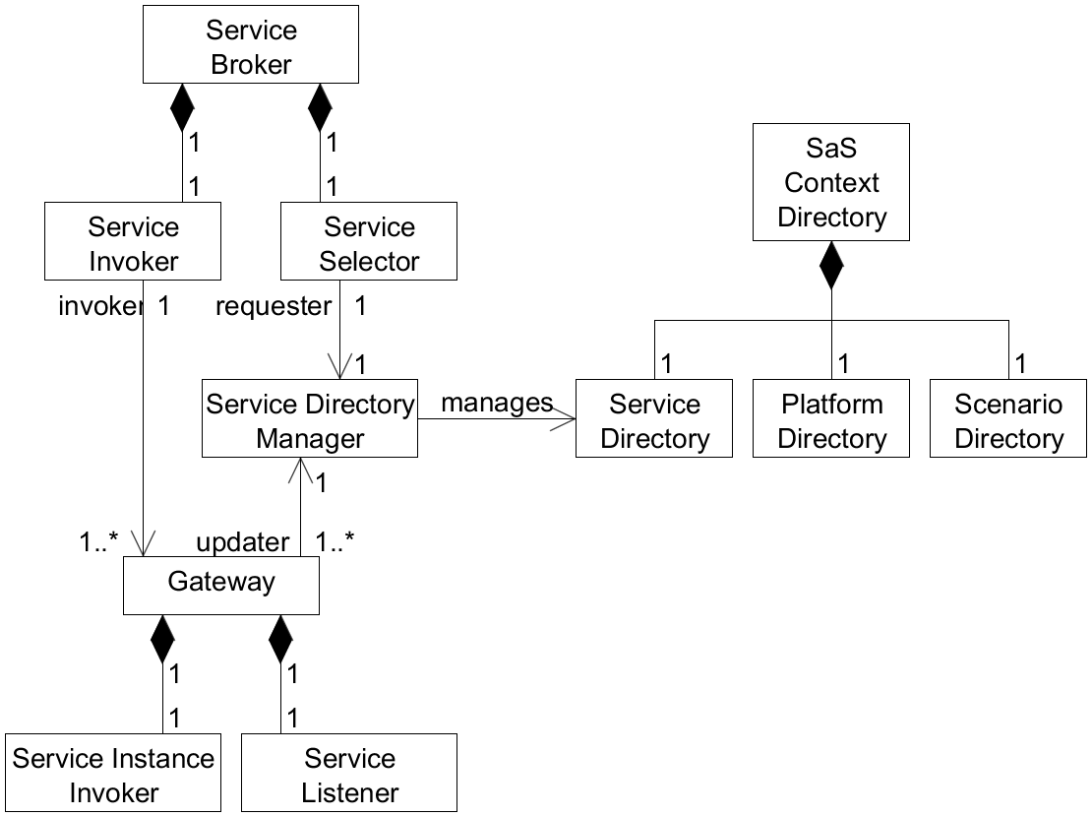

Figure 6.15: Service broker: service matchmaking and invocation

Our idea is to support different strategies to maintain scenario execution on the top of the step-by-step scenario execution mechanism. The strategies we chose are based on the work of Mikic-Rakic and Medvidovic [54] presented in Section 3.4.1.2. SaS uses these strategies to anticipate and / or repair faults. Anticipation strategies are based on the service role in the scenario. Whereas, recovery strategies depend on the nature of the fault.

6.3.2.1 Anticipation Strategies

SaS proposes strategies to anticipate the loss of a service.

- *Caching*. In SaS, when an operation invocation is present several times in a scenario and returns something, the result of its first invocation is cached. The cached result is used when the operation invocation must be invoked again and there is no available service instance that corresponds to the operation invocation.

- *Hoarding*. A condition step that has several conditions to evaluate requires the simultaneous presence of several services. If a condition step precondition is not satisfied because

of the absence of one or several services, the available services also required by this pre-condition are hoarded. Thus SaS invokes the requested operations for available services and caches their results. The cached results are used if the services that were not available appear while the previously invoked ones are no longer available.

- *Replication*. Replication is adapted for a scenario shared by another platform because we can replicate it by getting its scenario description file and deploying it. Thus, when the invoked service is a scenario (SaS recognizes scenarios from simple services), SaS tries to replicate it. SaS gets it description file (when possible) and locally redeploy the scenario if it becomes unavailable. If the shared scenario is updated, the corresponding service is also updated. SaS then retrieves the new version of the scenario descriptor.

- *Queuing*. In SaS, when a needed service is not available, the status of the corresponding step is set to `WAITING_SERVICE`. Thus, the corresponding service invocation (inside the step) is queued, until an appropriate service instance appears.

6.3.2.2 Recovery Strategies

Step execution can fail, and several reasons can cause these failures (*cf.* Section 6.1.2.2). Depending on the error, SaS applies different recovery strategies:

- *Service invocation error*. If the error persists after several attempts, SaS tries to invoke another service instance that can replace it.

- *Service disappearance*. If the service reappears in the specified elapsed time, SaS tries to re-invoke the service instance. If the service disappears again, this service instance is blacklisted. It will not be invoked anymore for this scenario and marked as too volatile in the service directory. Thus, SaS tries to find the same service or an equivalent.

- *Unreachable service*. SaS specifies a service invocation timeout. If a service does not answer after the specified elapsed time, SaS tries to re-invoke the service instance. After several attempts, this service instance is blacklisted. It will not be invoked anymore for this scenario and marked as unreachable in the service directory. Thus, SaS tries to find the same service or an equivalent. The owner of the SaS platform can remove the service from the blacklist whenever he/she wants.

6.4 Synthesis and Conclusion

This chapter concludes the presentation of our contribution and of SaS's mechanisms to execute a scenario. They are designed to schedule scenario execution and to dynamically adapt it to environment changes. Thus, SaS proposes a dynamic and adaptive service invocation scheme and scenario recovery strategies.

6.4.1 Scenario Execution Scheduling

In Section 6.1, we define a scenario execution model, which is an execution graph. Such a graph specifies the operational semantics of the execution steps and can be compared to Petri nets [85] or finite-state machines [74]. This graph enables SaS to calculate and anticipate the consequences of step execution. To schedule scenario execution, SaS realizes a model transformation, from the scenario description model to the scenario execution model.

Advantages of scenario execution scheduling by SaS:

+ Scenario execution scheduling enables to adapt scenario execution to pervasive environment and its evolution. Scenarios can be partially executed, suspended and then resumed when all the required services are not available.

+ Scenario execution scheduling adapts to users' mobility and enables to execute a scenario in several times and several places.

Limits of scenario execution scheduling by SaS:

- SaS enables step-by-step scenario execution but scenario execution is managed by a single platform at a time. It could be interesting, once the scenario has been scheduled in different steps, to distribute the steps execution on several platforms.

- A scenario can imply contradictory orders. We could implement static semantic analysis to avoid scenario execution conflicts.

6.4.2 Dynamic and Adaptive Service Invocation

SaS dynamically selects the services to invoke according to preference rules, that take into account quality of service. Furthermore, SaS defines some fault-tolerance mechanisms that enhance scenario execution resilience.

Advantages of dynamic and adaptive service invocation in SaS:

+ SaS uses a service broker that enables to dynamically adapt to environmental changes. Moreover, interoperability support enables that a service discovered with a protocol and used by the user for scenario creation may be replaced by a corresponding service, discovered with another protocol, at scenario execution.

+ SaS applies different service recovery strategies depending on the service role inside the scenario.

Limits to dynamic and adaptive service invocation in SaS:

- Service matchmaking in SaS requires that services specify the same semantic contract. Service matchmaking thus only relies on syntactic functional correspondence. It could be interesting to look at solutions, such as Larks (Language for Advertisement and Request for Knowledge Sharing) [77], which propose service description languages that use local domain ontologies. Such solutions consider service advertising and request and performs both syntactic and semantic matchmaking.

6.4.3 Requirement Fulfillment

This chapter details how SaS fulfills a functional sub-requirement: scenario execution resilience (R3.b). Table 6.1 synthesizes SaS's functionalities that meet this sub-requirement.

Functional Requirement	**Functionalities Associated**
R3.b Scenario execution resilience	SaS schedules a step-by-step scenario execution that enables to adapt to environmental changes and to users' mobility. Additionally, SaS proposes fault-tolerance strategies.

Table 6.1: Fulfillment of the sub-requirement R3.b detailed in Chapter 6

CHAPTER 7

Implementation and Validation

Contents

The three previous chapters presented all SaS functionalities we proposed to respond to the requirements of a user-centric pervasive system. Chapter 4 presents how SaS enables users to represent their environment and define scenarios. Chapter 5 details scenario management. Chapter 6 is dedicated to scenario execution resilience. It presents the scenario step-by-step execution mechanisms which provide mobile execution and enable to adapt to environmental changes.

This chapter describes the design and implementation of the SaS' prototype and presents a report on experiments that evaluate our contribution. It is organized as follows. We present SaS's prototype architecture and detail the design of the main SaS's functionalities. Then, a second section is dedicated to our experiments. It evaluates our contribution, regarding to the requirements that we established in Section 2.2.

7.1 The SaS' prototype

SaS's prototype has been implemented over industrial standards to prove the feasibility of our contribution. Moreover, this enables to conduct experiments to evaluate our work (*cf.* Section 7.2). The prototype conforms to the Service-Oriented Component architecture [14]. In this section, we first introduce SaS's prototype model and present all its components. These components can be *composite* components. In this case, we simply use the composite word. Then, we detail the different mechanisms they implement (such as context management, scenario control, etc.).

7.1.1 Architecture

The architecture of the prototype is illustrated by Figure 7.1. We can see that the prototype comprises seven composites. The following subsections present SaS's architecture and its composites.

7.1.1.1 Gateway

Gateways enable to discover and invoke services. They implement a specific protocol. A gateway is divided into two sub-composites: the *gateway listener* that listens to service events, and the *gateway communicator*, which interacts with other devices in the environment. Figure 7.2 illustrates the gateway as a composite, with its two sub-composites.

Service listener. This component listens for service events (*i.e.* service appearance, disappearance or update) from the environment. It is used for service discovery at SaS's start and then service event monitoring. Thanks to this component, SaS is aware about the presence of services (that implement a specific protocol) in the environment.

Gateway manager. The `gateway manager` receives service events and service descriptions from the service listener. Service descriptions depend on the protocol implemented by the gateway. Thus, the gateway manager sends service descriptions to the *service declarator* in order to translate it into SaS-SDL. Then, the gateway manager updates the service directory.

Service declarator. The `service declarator` is protocol specific. Its role is to translate a service description from a protocol to a service description into SaS-SDL.

Service instance invoker. Surrounding devices propose functionalities through services. The `service instance invoker` enables to invoke them.

Service exporter. SaS platforms can collaborate by proposing services to surrounding platforms. To do so, the `service exporter` enables to export a service. This component thus receives requests from surrounding SaS platforms, and dispatches these requests through the *collaborate* service provided by the `platform collaborator manager` (described later in this section).

7.1.1.2 Context Manager

The `context manager` is responsible for managing context awareness and must enable users to model context as they wish. SaS's context directory contains three directories. The `context manager` composite maintains directories through its `directory manager` components. Moreover, SaS enables users to define environment elements categories. The `context manager` composite also contains components to manage categories.

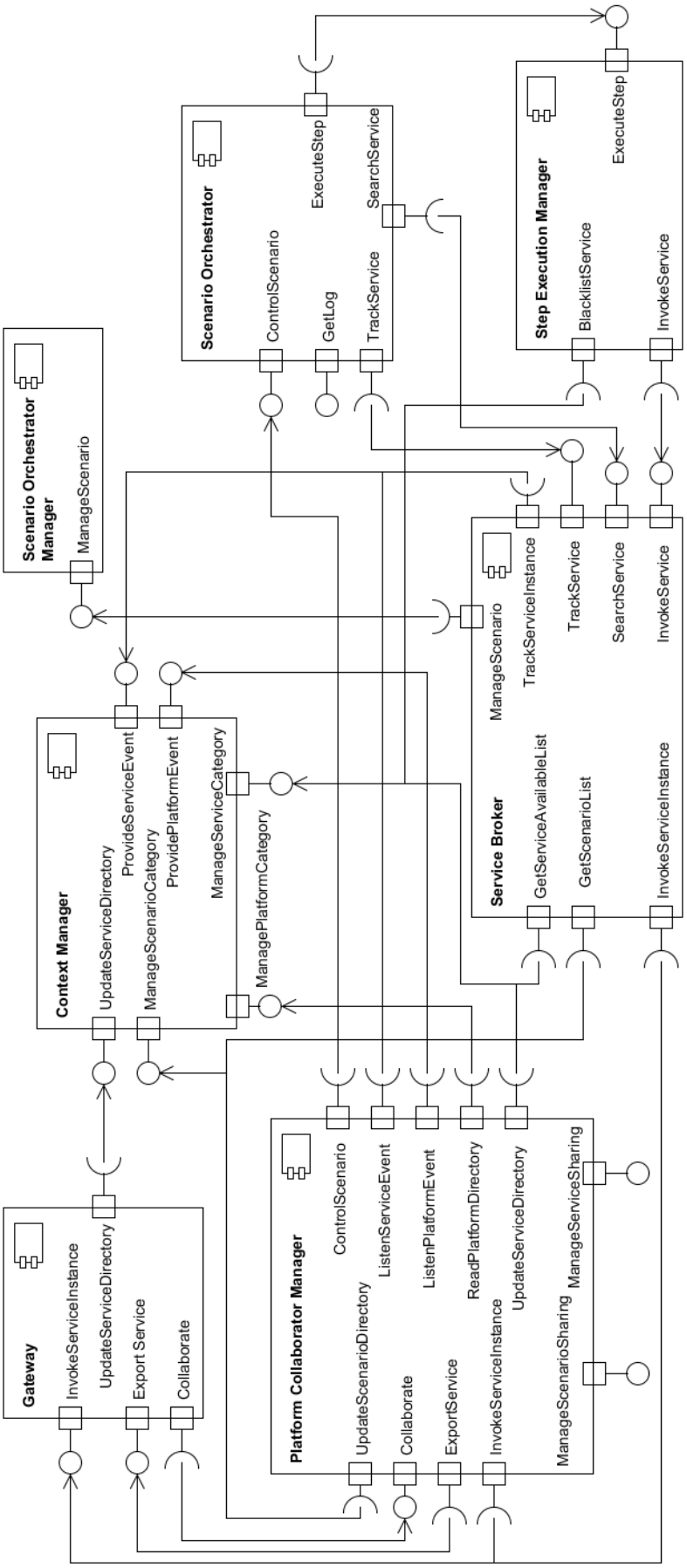

Figure 7.1: SaS's prototype model

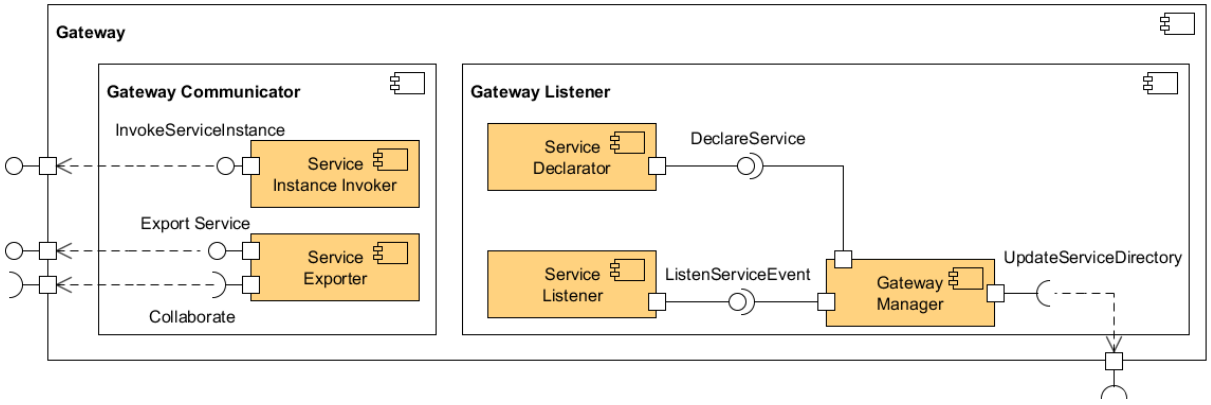

Figure 7.2: Gateway composite

In a pervasive environment SaS discovers services. These services can be a *Collaborate* service that enables, inter alia, to discover surrounding platforms. To do so, the `context manager` composite contains two `awareness manager` components dedicated to services and platforms. Scenarios are not directly discovered from the environment but proposed through the *Collaborate* service. Therefore, the `context manager` composite does not contain a platform awareness manager component. Figure 7.3 depicts the `context manager` composite, with its eight sub-components.

Service awareness manager. The `service awareness manager` component enables the many deployed gateways to update the service directory.

Platform awareness manager. Platforms can export a *Collaborative* service. Thus, the *platform awareness manager* listens for appearance (or disappearance) of other platforms. When a platform event occurs, the `platform awareness manager` component updates the platform directory.

Service directory manager. The `service directory manager` component is responsible of the service directory. It enables to create, read, update and delete service descriptions in the service directory.

Scenario directory manager. The `scenario directory manager` component is responsible of the scenario directory. It enables to create, read, update and delete scenario descriptions in the scenario directory.

Platform directory manager. The `platform directory manager` component is responsible of the platform directory. It enables to create, read, update and delete platform descriptions in the platform directory.

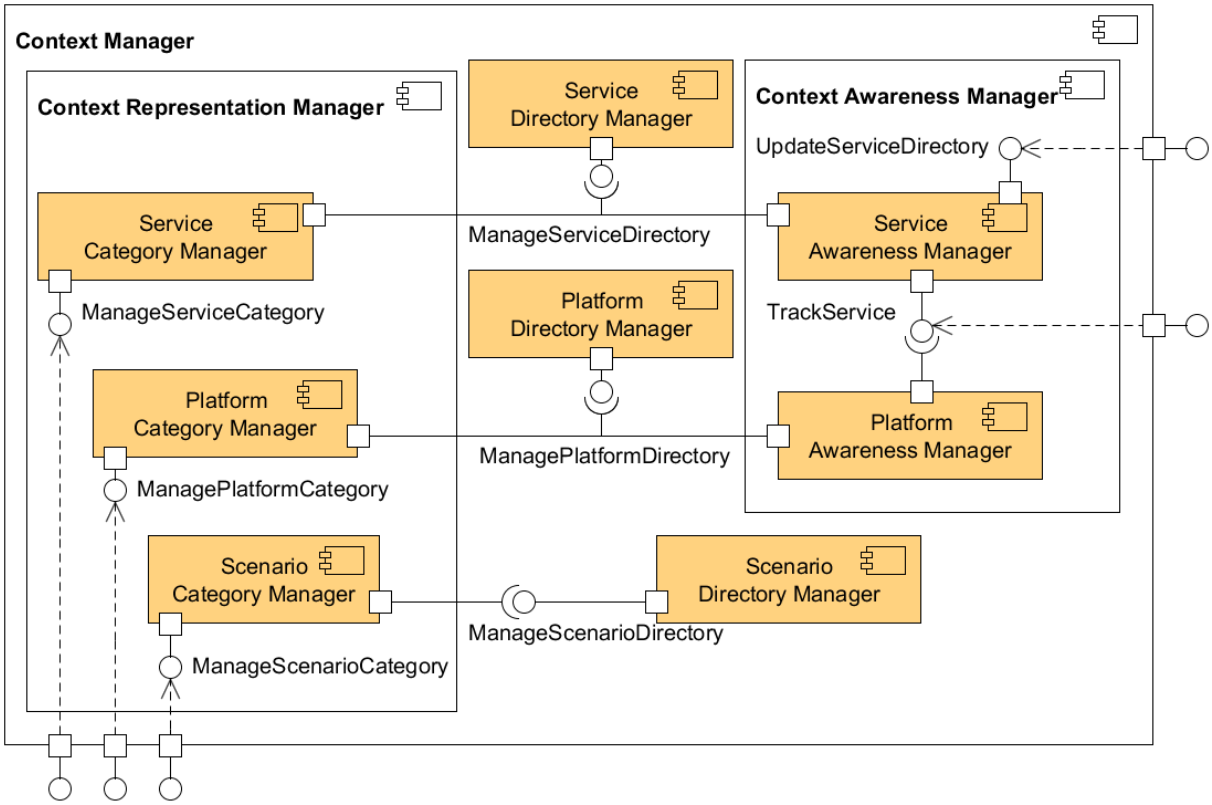

Figure 7.3: Context manager composite

Service category manager. The `service category manager` component enables to create, read, update and delete categories in the service directory. Typically, it enables to retrieve all the services registered under a specific category or add a service description to an existing service category.

Platform category manager. The `platform category manager` component enables to create, read, update and delete categories in the platform directory.

Scenario category manager. The `scenario category manager` component enables to create, read, update and delete categories in the scenario directory.

7.1.1.3 Scenario Orchestrator Manager

The `scenario orchestrator manager` composite is responsible for scenario management. Once a scenario has been defined and memorized as a scenario description file, this composite enables to deploy the scenario. This deployment is realized by the dynamic generation

of a `scenario orchestrator` composite. Figure 7.4 depicts the `scenario orchestrator manager` composite. We can see that the composite is composed of two components.

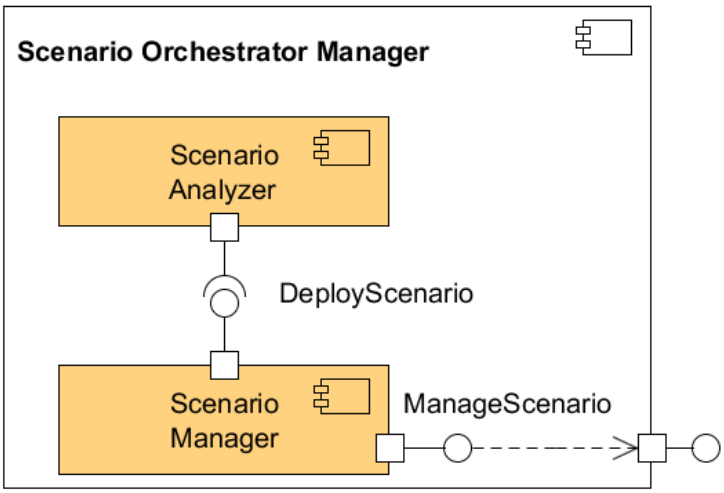

Figure 7.4: Scenario orchestrator manager composite

Scenario manager. The `scenario manager` enables users to install and uninstall a scenario. It handles the dynamic generation of a corresponding `scenario orchestrator` composite which is responsible for scenario execution. Moreover, `scenario manager` component is also responsible for the `scenario orchestrator` destruction (when the scenario is uninstalled).

Scenario analyzer. The `scenario analyzer` comprises all scenario execution scheduling mechanisms (detailed in Section 6.2) such as step extraction.

7.1.1.4 Scenario Orchestrator

A `scenario orchestrator` composite is responsible for the control of a scenario. There is one `scenario orchestrator` composite per deployed scenario. Figure 7.5 describes the `scenario orchestrator` composite. We can see that the composite is composed of six components.

Scenario controller. The `scenario controller` enables users to control a scenario. To do so, it publishes a service into the platform service repository that holds operations to interact with scenario execution (*cf.* Section 5.1.3). Depending on the operation invoked by the user (start, stop, pause, resume), the `scenario controller` executes, pauses or cancels the scenario steps.

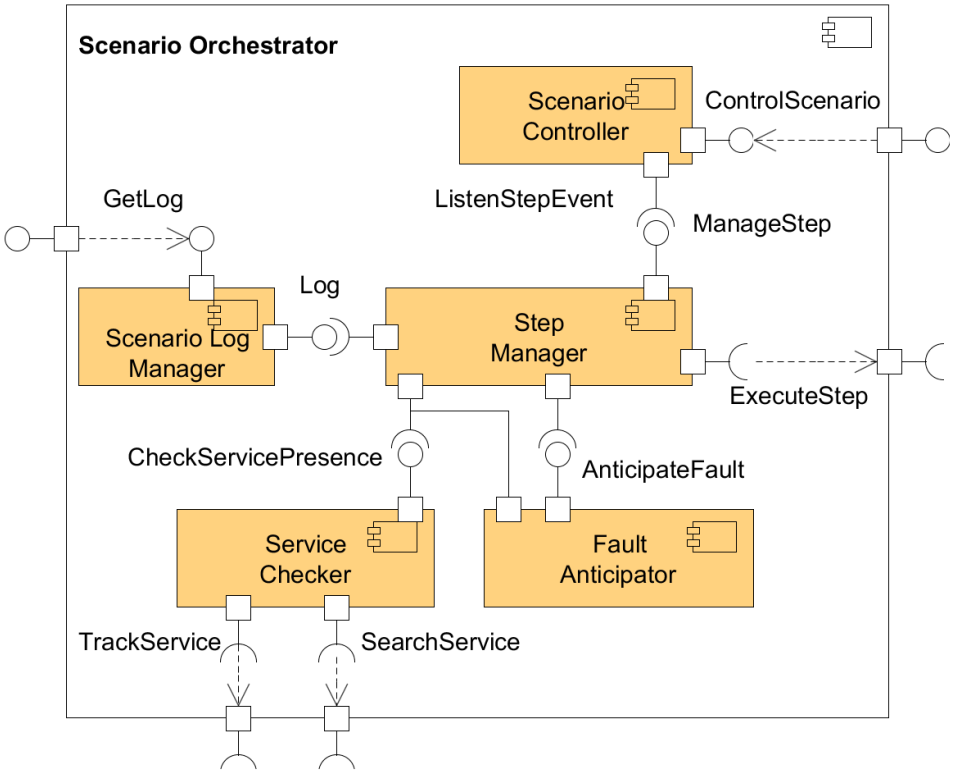

Figure 7.5: Scenario orchestrator composite

Step manager. The `step manager` component manages scenario steps. It handles the selection of the scenario steps to be executed depending on the occurrence of different events (required service availability, user control, etc.). It uses the scenario step list to calculate and anticipate step execution.

Service checker. A step is executable when its preconditions all are satisfied. When a step has its precedence links satisfied, SaS evaluates the service presence preconditions. Thus, the `service checker` components checks if there are available instances for the required services.

Fault anticipator. The `fault anticipator` component implements the fault anticipation strategies (detailed in Section 3.4.1.2). Typically, when a service is invoked several times inside a scenario, the `fault anticipator` caches the first invocation result to reuse it (when necessary) later in the scenario.

Scenario Log Manager. The `scenario log manager` component maintains the log of the scenario execution state. To do so, it provides a *Log* service that enables to log any scenario status change. It is used by the `step manager` component. The `scenario log` manager

component enables to retrieve the log, through the *Log* service. This enables users, inter alia, to check scenario execution advancement.

7.1.1.5 Step Execution Manager

The `step execution manager` composite is responsible for step execution resilience. Figure 7.6 describes it. We can see that the `step execution manager` composite contains two components. One is responsible for step execution (`step executor`) and the other for recovering execution if a problem occurs (`step recovery manager`).

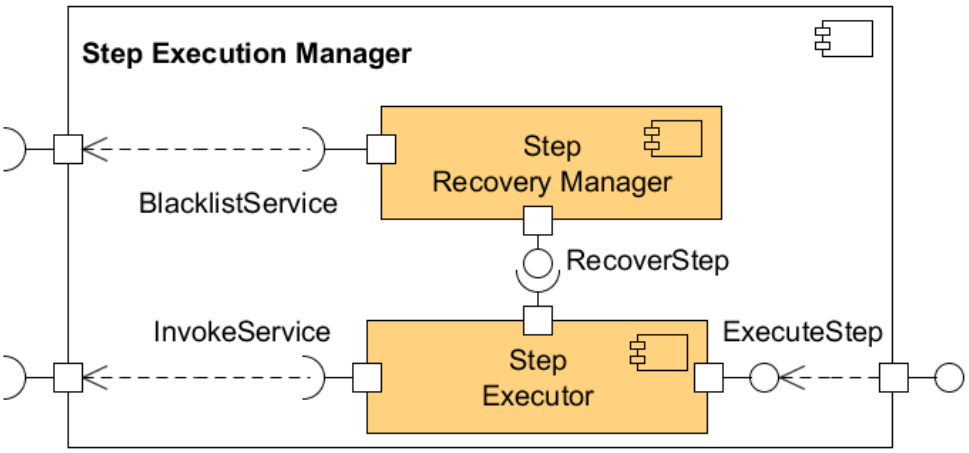

Figure 7.6: Step execution manager composite

Step executor. The `step executor` component tries to execute a step that has its preconditions satisfied. It contains the different algorithms to execute a step (detailed in Section 6.2.3) that depend on the step type. Executing an action step involves invoking services. The component uses the service broker composite to do it.

Step recovery manager. The `step recovery manager` component is responsible for step resilience. When a problem occurs during step execution, the `step recovery manager` applies one of the different recovery strategies (detailed in Section 6.3.2) according to the nature of the error.

7.1.1.6 Service Broker

The `service broker` composite is responsible for searching service instances that correspond to required services and to invoke them. Figure 7.7 presents the `service broker` composite. Two components constitute the composite: the `service selector` that searches service instances and the `service invoker` that invokes service instances.

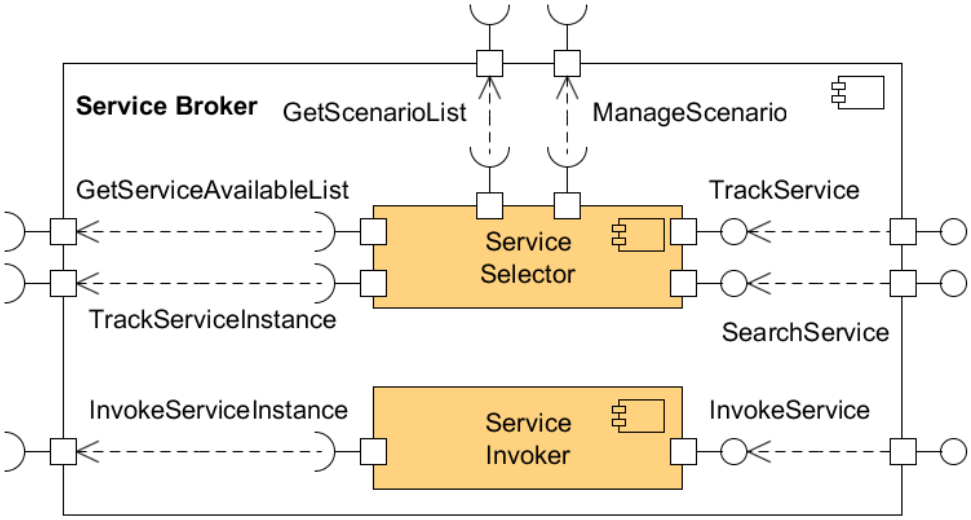

Figure 7.7: Service broker composite

Service selector. The `service selector` component searches for available service instances that match a given operation invocation specification. To do so, it uses the service provided by the `service category manager` to retrieve the corresponding available service list. If the operation is a scenario control operation, the `service selector` looks up in the scenario directory instead. The `service selector` component then uses a selection algorithm (detailed in Section 6.3.1.1) to choose services corresponding to the request.

Service invoker. The `service invoker` component is responsible for service invocation. SaS registers the service description in the service directory with a property specifying the protocol used to reach the service instance. Thus, the `service invoker` selects the appropriate gateway (that provides a *InvokeServiceInstance* service) to invoke the service instance. Then it returns the result of the service invocation or an error if it does occur.

7.1.1.7 Platform Collaborator Manager

The `platform collaborator manager` composite is responsible for platform collaboration management. It enables to share scenarios and services with other platforms thanks to a special service called *Collaborate*. Figure 7.8 depicts the `platform collaborator manager` composite. Its six components are detailed in this subsection.

Sharing manager. The `sharing manager` component manages scenario and service sharing. It receives the chosen access rights, specified thanks to the `scenario sharing manager` and `service sharing manager` components and shares scenarios and services with the selected platforms.

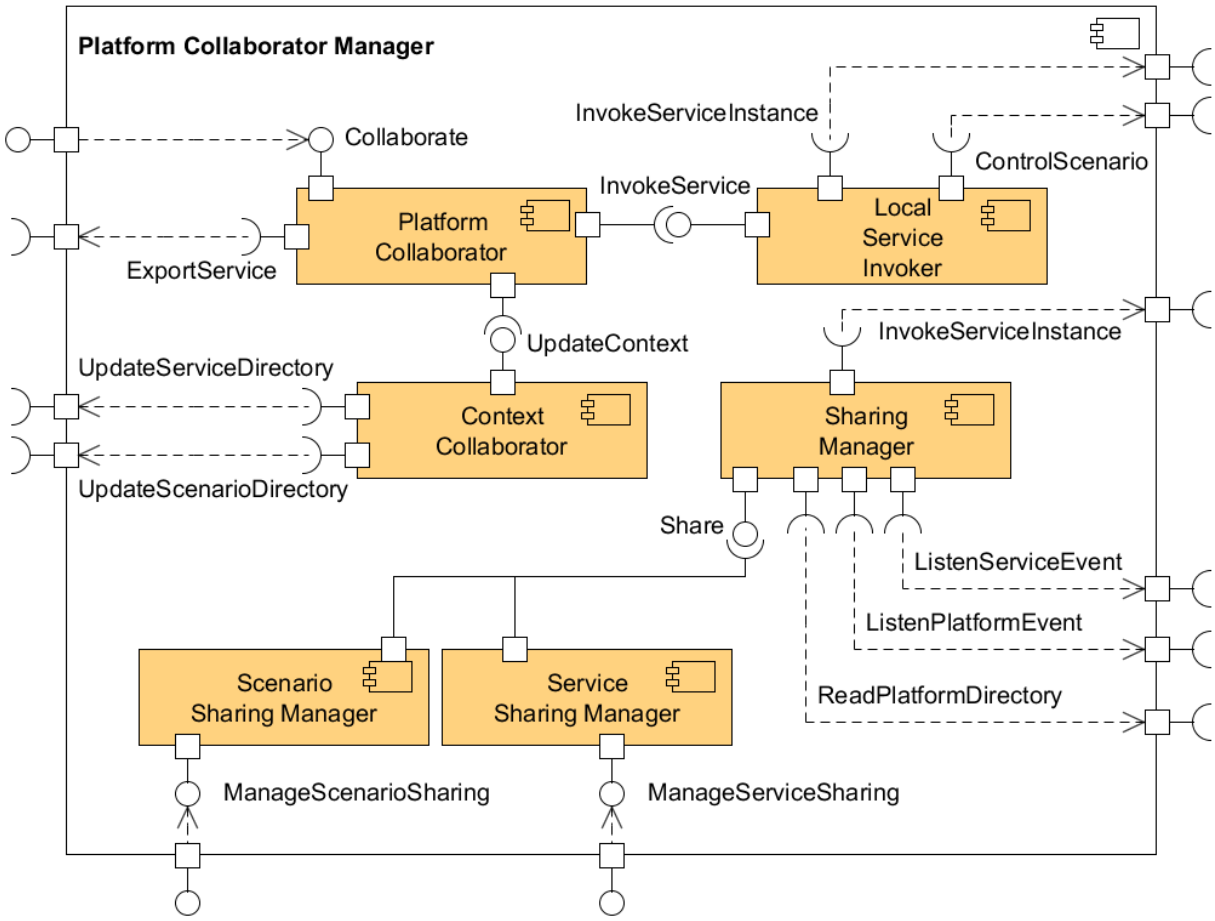

Figure 7.8: Platform collaborator manager composite

Platform collaborator. The `platform collaborator` component provides the special *Collaborate* service (*cf.* Section 5.2.2). To export this service in the environment, and make it available for surrounding platforms, the `platform collaborator` component uses the *ExportService* service provided by available gateways. Then, other platforms that want to share a scenario (*resp.* a service) and/or invoke a scenario (*resp.* a service) can use this *Collaborate* service.

Context collaborator. When a surrounding platform invokes the *Collaborate* service to share a scenario (*resp.* service), the `context collaborator` component updates the scenario (*resp.* service) directory to propose the new scenario (*resp.* service) to users.

Local service invoker. When a surrounding platform invokes the *Collaborate* service to invoke a scenario (*resp.* a service), the `local service invoker` component invokes the appropriate scenario (*resp.* service) which is locally installed and has been shared.

Scenario sharing manager. The `scenario sharing manager` component enables to share a scenario. It proposes a service to define the access rights for a platform (or a platform category) and for a specific scenario.

Service sharing manager. The `service sharing manager` component enables to share access to local services. It proposes a service to attach selected services to a platform (or a platform category).

7.1.2 Insights into the SaS' prototype

This section describes the implementation of SaS's prototype. SaS's prototype is implemented in Java over OSGi [59] with iPOJO [27]. OSGi is a popular development framework that is widely adopted by industry developers to create Java components called bundles. iPOJO is based on OSGi and conforms to the Service-Oriented Component Model [14]. We detailed these two frameworks in Section 3.5. We first introduce the prototype and then present all SaS's mechanisms that enable users to control a pervasive environment.

7.1.2.1 SaS over the OSGi platform

As explained in the previous section, SaS's prototype model follows the Soc architecture and thus, is made of components. Each component is implemented as an OSGi bundle in order to be easily and safely administrated at runtime by the OSGi platform. The OSGi platform proposes a service registry, that bundles use to register and retrieve services. On the top of OSGi, iPOJO provides an API to dynamically generate components and create composites.

7.1.2.2 Context Awareness

Service discovery. Service discovery is handled by gateway composite (*cf.* Section 7.1.1.1). Services can be local (locally registered by bundles installed on the OSGi platform) or provided by other devices (through a network protocol). Thus, we have developed two gateway composites: one that listens to and invokes local services and another one that implements the UPnP protocol (detailed in Section 3.2.1.2).

For the UPnP gateway we use a bundle provided by the Apache Felix project [3] that implements the UPnP protocol. This bundle listens for surrounding UPnP devices. When it discovers a new UPnP device (that provides UPnP services), the bundle registers in the OSGi registry a corresponding OSGi service. Thus, the `service listener` component of the UPnP gateway listens for service events provided by this bundle in the OSGi registry. When a service event occurs, SaS queries the service to retrieve the UPnP service description. Then, the `service declarator` component in the `gateway` composite translates the UPnP service description into a SaS-SDL service description.

The `service awareness manager` component (*cf.* Section 7.1.1.2) provides a service that enables gateways to signal service events. Gateways transmit the nature of the events (service

appearance, disappearance or update) and the description of the related service. The service awareness manager uses the `service directory manager` component to update the service directory. In addition, the `service awareness manager` component provides a *TrackService* service. This service enables components to track the availability of a specific service. When this service appears, the `service awareness manager` component sends a message to the component which is waiting for the service.

Platform discovery. The `platform awareness manager` uses the *TrackService* service to track surrounding SaS platforms. SaS platforms can provide a *Collaborate* service (to manage collaborations). The `platform awareness manager` tracks this kind of service. When a new collaboration service appears, the `platform awareness manager` retrieves its associated platform identifier and user name (provided in the service properties) and sends a message to the `platform directory manager` which can update the platform directory.

7.1.2.3 Context Management

The directories, dedicated to the environment elements (services, platforms and scenarios), are handled by three components (`service directory manager`, `platform directory manager` and `scenario directory manager`) that have a similar role: manage their dedicated directory (*cf.* Section 7.1.1.2). They all provide a service that enables to update the directories.

To avoid duplicates, the directories manager checks if a environment element description is already registered before adding it into the corresponding directory. Moreover, environment elements can be categorized, which enables users to better define and reuse elements of the context. Categories are like tags (an environment element can be linked to several categories). The categories associated with an environment element are listed in their descriptors.

Categorization is also used for registering and retrieving available environment elements. The presence of a service (*resp.* a platform) is transcribed by a special category *available*. Thus, when the `service` (*resp.* `platform`) `awareness manager` sends a message to update the service (*resp.* platform) directory, the `service` (*resp.* `platform`) `directory manager` adds or remove the available category tag in the service (*resp.* platform) directory.

The `context representation manager` composite contains three components that enable to create, read, update and delete categories in the directories: the `service category manager`, the `platform category manager` and the `scenario category manager`.

7.1.2.4 Scenario Definition

SaS-SDL is implemented with Xtext [7]. Xtext is a framework for developing domain-specific languages (DSLs). It enables to generate a parser for the concrete textual syntax of the language (source code) and a class model that implements the concepts of its abstract, conceptual syntax. Moreover, based on these two elements, XText generates a customizable Eclipse-based IDE

dedicated to SaS-SDL. Thus, users can benefit of Eclipse features such as syntax coloration and dynamic syntax checking. As a scenario description file is a simple text file, users can also define a scenario with their favorite text editor. In this case, scenario syntax checking is done during scenario deployment.

7.1.2.5 Scenario Installation

To install a scenario, SaS dynamically generates a `scenario orchestrator` composite which is responsible for scenario execution. Additionally, to uninstall the scenario, SaS uninstalls the corresponding `scenario orchestrator` composite. This is the responsibility of the `scenario orchestrator manager` composite (*cf.* Section 7.1.1.3). To do so, SaS uses an API provided by iPOJO that enables to create a composite of component instances. This composite can be dynamically deployed and removed on the OSGi platform.

A `scenario orchestrator` composite is specific to the scenario that it controls. Thus, SaS needs to analyze the scenario to create a corresponding `scenario orchestrator` composite.

Scenario analysis. The `scenario analyzer` component extracts the scenario steps. The scenario description file parser is implemented with Xtend [25]. Xtend is a programming language implemented on top of Xtext. It enables to parse a file defined in a language implemented with Xtext. Thus, thanks to Xtend, and the SaS-SDL implemented with Xtext, the scenario analyzer parses the description file to obtain an instance of the SaS meta-model (*e.g.* something similar to the one illustrated by Figure 4.8). Then, the `scenario analyzer` component traverses the scenario model and transforms it into a step-by-step execution graph.

To do so, SaS applies model transformation algorithms. Algorithm 6 depicts the conditional statement transformation. The conditional statement action contains a condition and two action blocks. The condition step is created on line 3. On line 4, SaS adds to the condition step the precedence link returned from the previous transformation (received as parameter in the transformation function). On line 5, SaS adds to the step's precedence link list a link to the condition step (with the required status). Thus, the step required by the condition step knows that the condition step needs its execution. The condition step has for execution attribute value the complex condition (line 6). Then SaS adds the condition step to the scenario step list (line 7). Then, we transform the two action blocks (then - else) of the conditional statement (lines 9 to 12). These transformations are done with a precedence link pointing on the condition step but with different required status (TRUE or FALSE). Transformations return two precedence links that we attach to an OR join step (lines 15 and 16). As for the conditional step, we retrieve the steps from these two precedence links and attach to them a link that points to the join step (lines 17 and 18). Finally, a precedence link that requires the join step to be executed is defined and returned (lines 20 and 21).

All the transformation algorithms are depicted in Section A, in Appendices. Steps contain their succeeding steps. Thus, the scenario execution graph that results from the scenario

Algorithm 6 Scenario Steps Extraction: Conditional Statement Transformation

```
1: function transformCondStatement (Action action, PrecLink precLink, StepList scenar-
   ioStepList) : PrecLink
2: begin
3:     ConditionStep conditionStep = new ConditionStep
4:     conditionStep.setPrecedence(precLink)
5:     precLink.getStep().addNextStepLink(conditionStep,precLink.getStatus())
6:     conditionStep.setExec(action.getCondition())
7:     scenarioStepList += conditionStep
8:
9:     PrecLink thenPrecLink = transformActionBlock(action.getThen(),
10: new PrecLink(conditionStep,TRUE), scenarioStepList)
11:    PrecLink elsePrecLink = transformActionBlock(action.getElse(),
12: new PrecLink(conditionStep,FALSE), scenarioStepList)
13:    JoinStep joinStep = new JoinStep
14:    joinStep.setJoinMode(OR)
15:    joinStep.addPrec(thenPrecLink)
16:    joinStep.addPrec(elsePrecLink)
17:    thenPrecLink.getStep().addNextStepLink(joinStep,thenPrecLink.getStatus())
18:    elsePrecLink.getStep().addNextStepLink(joinStep,elsePrecLink.getStatus())
19:    scenarioStepList += joinStep
20:    precLink = new PrecLink(joinStep, EXECUTED)
21:    return precLink
22: end
```

description transformation, is conserved in the form of a step list.

Scenario orchestrator creation. The `scenario manager` component dynamically generates a `scenario orchestrator` composite and its inner components thanks to the iPOJO API (*cf.* Section 7.1.1.4). Listing 7.1 shows a code excerpt of the scenario orchestrator composite creation. It illustrates how the iPOJO API is used to create three components (`fault anticipator`, `scenario log manager` and `service checker`) and the scenario orchestrator composite that contains the components. The `fault anticipator` component has a service dependency: the *CheckServicePresence* service provided by the `service checker` component. Moreover, it provides an *AnticipateFault* service. The `scenario log manager` component provides a *Log* service. The `service checker` component has two dependencies: the *TrackService* and *SearchService* services, both provided by the `service broker` composite. It also provides a *CheckServicePresence* service. iPOJO uses dependency injection to provide components with their required services. The scenario name, used as a property, is used to select the right pair of components to be connected.

Then, we create the `scenario orchestrator` composite with two instances of the previously defined components. The composite exports the *Log* service provided by the `Scenario Log Manager` component. Then, we register the composite instance inside a `scenario orchestrator` composite list. Thus, when the user wants to uninstall the scenario, we retrieve the associated `scenario orchestrator` composite and we delete it.

```
1  //Create Fault Anticipator component
2  PrimitiveComponentType faultAnticip  = new PrimitiveComponentType();
3  faultAnticip.setClassName(IFaultAnticipator.class.getName());
4  faultAnticip.addService(new Service().addProperty("ScenarioName", scenarioName));
5  Dependency checkServiceDependency = new Dependency().setField(checkService)
6                            .setFilter(ScenarioName=scenarioName);
7  faultAnticip.addDependency(checkServiceDependency);
8  faultAnticip.addService(new Service().addProperty("ScenarioName", scenarioName));
9
10 //Create Scenario Log Manager component
11 PrimitiveComponentType scenarLogMngr  = new PrimitiveComponentType();
12 scenarLogMngr.setClassName(IScenarioLogManager .class.getName());
13 scenarLogMngr.addService(new Service().addProperty("ScenarioName",scenarioName));
14
15 //Create Service Checker component
16 PrimitiveComponentType serviceChecker  = new PrimitiveComponentType();
17 serviceChecker.setClassName(IServiceChecker .class.getName());
18 serviceChecker.addService(new Service().addProperty("ScenarioName",scenarioName));
19 serviceChecker.addDependency(new Dependency().setField(trackService));
20 serviceChecker.addDependency(new Dependency().setField(searchService));
21
22 //Create Scenario Orchestrator composite
23 CompositeComponentType composite = new CompositeComponentType();
24 composite.addInstance(new Instance(faultAnticip.getFactory().getName()));
25 composite.addInstance(new Instance(scenarLogMngr.getFactory().getName()));
26 composite.addInstance(new Instance(serviceChecker.getFactory().getName()));
27 composite.addService(new ExportedService().setSpecification(IScenarioLogManager));
28 ComponentInstance instance = composite.createInstance(scenarioName);
29
30 //Register the scenario orchestrator composite
31 scenarioCompositeList.add(instance);
```

Listing 7.1: Extract of scenario orchestrator composite creation

The `scenario orchestrator` composite also exports a service provided by the `scenario controller` component. This service holds the scenario name as a property. It can thus be easily differentiated from the other scenarios registered as services. The dynamically generated `Step Manager` component contains the scenario step list (previously computed), which represents the scenario execution graph.

7.1.2.6 Scenario Execution

The `scenario orchestrator` composite provides a service to control scenario execution. In this subsection, we detail how scenario execution is managed.

Scenario control. When the user invokes a scenario control operation, the `scenario controller` component (*cf.* Section 7.1.1.4) checks if the operation is not contradictory with the scenario status (*e.g.* a user tries to stop a scenario which is not running). Then, if the operation is relevant,

the `scenario controller` component sends a message to the `step manager` component. The `step manager` components adapts scenario execution depending on user control and logs the new scenario status.

Scenario execution. When a scenario is running, the step manager executes the scenario as detailed in Section 6.2.2. To search a service instance, the `step manager` sends a message to the `service checker` component. To execute a step, the `step manager` invokes the *ExecuteStep* service provided by the `step execution manager` composite. The `step executor` component receives the step to execute. Depending on step's type it tries to execute it by invoking some services If an error does occur during service invocation, the `step recovery manager` component applies the appropriate recovery strategies (detailed in Section 6.3.2) and may blacklist a service in the service directory when necessary.

7.1.2.7 Collaborate

SaS platform can provide a *Collaborate* service to other platforms. This service enables the platform to be detected by surrounding platforms and to share scenarios and services. This service is provided by the `platform collaborator` component, and exported to the environment by `gateway` components installed on the platform (*cf.* Section 7.1.1.7). Reciprocally, these `gateway` components are used by the platform to detect the *Collaborate* services exported by the surrounding platforms. Thanks to these *Collaborate* services, the SaS platform collects information about the environment and proposes a representation of surrounding platforms to users. Users can personalize this representation by grouping platforms into categories. In addition, users can decide to share scenarios that they have locally or services locally accessible by assigning access rights to surrounding platforms.

Depending on the access rights defined, thanks to the `scenario sharing` manager and the `service sharing` manager components, the `sharing manager` component retrieves and invokes the *Collaborate* service target to the selected platform to share scenarios and/or services with it.

Users can select a platform (or a platform category) even if it is not available in the environment. Thus, the `sharing manager` component listens for platform events. When a platform selected for sharing appears, the `sharing manager` invokes its *Collaborate* service.

Users can share a service which is currently not available. In this case, the `sharing manager` adapts its sharing depending on the service presence (*i.e.* the service is not shared while it remains unavailable). Thus, it checks the service presence thanks to the *ListenServiceEvent* service. Similarly, a scenario which is initially shared by the user is automatically not shared anymore when the user uninstalls it.

7.2 Experimentations

Using the prototype described previously, we ran experiments to prove the feasibility of our contribution. Moreover, these experiments were used to evaluate SaS on some use cases.

7.2.1 Reports on Experiments

In this sub-section we present the different experiments. Each experiment is dedicated to a SaS prototype functionality.

7.2.1.1 Environment Simulation

The physical environment. To simulate pervasive environments, we connect three PC machines by WI-FI. These machines embedded a Java Virtual Machine that enabled to launch the SaS prototype (based on the Apache Felix platform [3]). Each machine proposed several services locally and to the network. Moreover, we have launched distinct prototype instances on the same machine. We simulated users' mobility by the connection (or disconnection) of a machine that proposed about a dozen services. Thus, the SaS platform had an environment that evolves as if it is moving into different locations.

The devices simulation. Experimentations are, inter alia, based on the Domus simulator, developed by our team, and implemented in JavaFX. Domus, illustrated in Figure 7.9, simulates a smart-home that contains UPnP devices (light, washing machine, air conditioning, shutter, radiator and thermostat). It proposes a realistic interactive view which enables to control directly the UPnP devices by clicking on them. UPnP devices are implemented in Java thanks to the *CyberLink* project for Java[1].

To vary the origins of the UPnP devices we also used the *Developer Tools for UPnP Technologies* software. This software proposes UPnP devices (*e.g.* light device that proposes a *DimmableLightService*) and tools to detect UPnP devices in a network and observe UPnP messages sent through the network. This software is based on Intel UPnP Tools [43]. Thus, the implemented devices are fully compliant with the UPnP specification (Intel is a member of the UPnP forum, responsible of the UPnP norm).

Moreover, we developed OSGi bundles that propose services involved in the scenario example of Listing 4.10 and that are not already provided by the UPnP devices. Thus, we implemented a *LuminosityService* service, a *Player* service, a *Thermometer* service, a *DoorService* service and a *RadioService* service. To simulate a real environment, the *LuminosityService* comprises a luminosity attribute that we can dynamically modify. Moreover, we define an *AdjustTemperatureScenario* that simply invokes the operation *setValue(**Integer**)* of the *ThermostatService* service. Additionally, the *PlayerService* and the *RadioService* simulate a multi-

[1] http://www.cybergarage.org/twiki/bin/view/Main/CyberLinkForJava

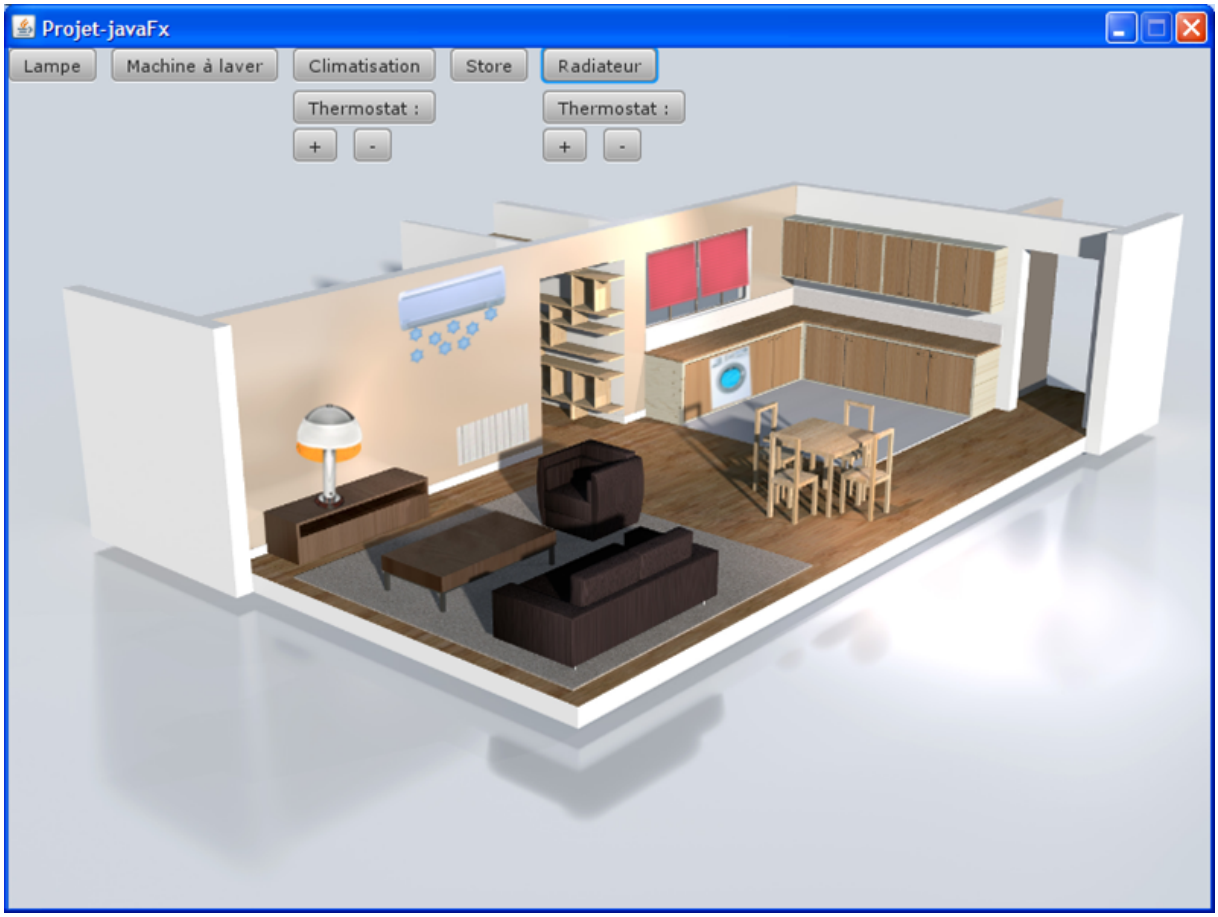

Figure 7.9: Domus simulator

media service. We define a *Multimedia* object that contains attributes (*e.g.* name, type) that the *RadioService* provides and the *PlayerService* uses.

7.2.1.2 Context awareness

To prove that SaS prototype supports interoperability, we developed two service gateways: one that implements the UPnP protocol and one that listens and invokes OSGi-based services.

Service discovery. A first experiment consists in discovering services and transforming them into SaS-SDL service descriptions. The SaS prototype automatically discovers OSGi services (implemented locally) and UPnP services (present in the network). Then, SaS updates the service directory. All service declarations are translated into SaS-SDL and registered into the service directory, defining their corresponding protocol as a property and their category as available. Listing 7.2 shows an extract from the obtained service directory.

```
service_directory {

   service DimmableLightService :
      device RoomLight;
      property deviceType : DimmableLightService;
      property protocol : UPnP;
      operation SetValue(Integer) : void;
      operation GetTarget() : Integer;
      category available;

   service Player :
      device Lounge_PC;
      property deviceType : Computer;
      property protocol : Local;
      operation play(Object) : void;
      category available;

   service RadioService :
      device WebRadio;
      property protocol : Local;
      operation getChanngel(String) : Object;
      category available;

   service ShutterService :
      device ShutterDevice;
      property deviceType : Shutter;
      property protocol : UPnP;
      operation open() : void;
      operation close() : void;
      category available;
}
```

Listing 7.2: Service directory automatically obtained by experimentation

If services appear once the prototype is launched, the SaS prototype discovers and memorizes them as well.

Service disappearance. When a service disappear, the service directory is dynamically updated. SaS detects the corresponding service event, retrieves the service description inside the service directory and removes its *available* category entry.

Platform representation. We launched two platforms on two different machines connected by Wi-Fi to simulate platforms collaboration. In our experiment, the platforms propose the *Collaborate* service through UPnP. Thus, platforms discover *Collaborate* services automatically, use them to retrieve surrounding platform information (platform id and user name). Platform information are memorized into the platform directory.

7.2.1.3 Scenario Installation

We created the scenario example of Listing 4.11. This scenario enables to test the different functionalities of SaS-SDL (*i.e.* parallel execution, dynamic service selection, etc.) The time event defined in the scenario description is modified in order to wait just one minute (and not

until 8pm). Once created, SaS memorizes the scenario into the scenario directory. Scenario is then ready for installation. Scenario installation leads to scenario orchestrator generation. The scenario description file is parsed with our Xtend parser. The result is an instance of the SaS-SDL scenario declaration model. Thanks to the algorithms implemented by SaS, this scenario declaration model is transformed into a scenario execution graph. Then, SaS generates the scenario orchestrator composite with iPOJO. The scenario orchestrator provides a service to control scenario execution. The start operation requires a parameter value for the joker used in the scenario. The scenario is dynamically discovered as a service by the local gateway (that listens for OSGi service events).

7.2.1.4 Scenario Execution

To test scenario execution resilience we try to execute the above scenario in different execution conditions. The scenario contains a joker (specified on line 9 of Listing 4.11). Thus, the scenario start operation requires a parameter value. We enter the value 500.

Execution without interruption. In this first test, all the required services are available in the environment. The *AdjustTemperatureScenario* scenario is available as a service. Services are easily retrieved by the service broker and invoked by the appropriate gateways. The step manager executes the steps, following the scenario execution graph. SaS executes the scenario which remains in the while loop (steps 4, 5 and 6). We manually change the luminosity value to make the while condition false. Now that we have tested the while loop, we maintain the while condition to false for the next experiments. Scenario execution continues until the end.

Execution with missing required services. In this test we try scenario execution without some required services. The *AdjustTemperatureScenario* scenario is stored in the scenario directory but not installed.

We begin by removing the *DoorService* service (used in step 1). The scenario can be launched but does not progress. Step 1 cannot be executed while the service is missing. Thus, its status is set to `WAITING_SERVICE`. When the *DoorService* service appears, it is automatically detected and step 1 is executed.

Then we remove the *ShutterService* service (used in step 3). This service is involved in a parallel action block. Thus, step 7 and following steps are executed until step 11 that transitively depends on the execution of step 3. When the *ShutterService* service appears, step 3 is executed and the execution goes on.

We set the environment parameters (such as Temperature) to a specific value in order to obtain a negative evaluation of the condition step 7. SaS then needs to invoke the *AdjustTemperatureScenario* scenario. This scenario is not available as a service but exists in the scenario directory. Thus, SaS dynamically installs it. The generated scenario orchestrator proposes a service to control the *AdjustTemperatureScenario* scenario execution. This service is detected and the start operation is invoked.

7.2.1.5 Platform Collaboration

Thanks to the *Collaborate* service, two platforms can share scenarios and services. To test this functionality, a first platform invokes the *Collaborate* service of a second to share a scenario.

We first shared the scenario in descriptive mode. The second platform receives a service description (that correspond to the scenario) which provides an operation to get the scenario description file: `getDescriptor`. The second platform invokes this operation and retrieves the scenario description.

Then, we shared the same scenario in collaborative mode. The second platform receives a service description that, this time, provides the control commands operations instead. The second platform invokes the scenario start operation. The scenario installed on the first platform executes accordingly.

The copied mode is a combination of the first two and thus, works fine.

7.2.2 Experimental Validation

Thanks to these experiments, we have proven the feasibility of our contribution. This validates our proposed solutions. However, this prototype implementation has limits, detailed in this sub-section.

7.2.2.1 Prototype Validation

We can argue that our SaS contribution is adapted to pervasive environments with heterogeneous and volatile services. We tested SaS's functionalities on realistic uses cases (service and device heterogeneity, different environments, service volatility, presence of several users). SaS features such as context representation, scenario control reuse or sharing are fully functional. Scenario execution adapts to environmental changes and enables to execute a scenario without the need for all required services to be simultaneously available.

7.2.2.2 Prototype Limits

Our prototype has been yet tested in simple cases that that are representative of smart-home contexts (few services, limited service volatility, small number of platforms, etc.). We thus still have to experiment the scalability of our proposal, to validate its ability to manage larger scale contexts (buildings, airports, companies, ...). Besides, we have not yet implemented SaS' fault anticipation strategies (except for queuing). Moreover, the implemented recovery strategies do not yet include service blacklisting.

Part III

Conclusion

CHAPTER 8

Conclusion and Perspectives

Contents

In the previous chapters, we have defined what a user-centric pervasive system is and identified the requirements it must fulfill. Then we presented our contribution named SaS and its functionalities that enable users to fully benefit from pervasive environment. This chapter concludes this thesis and draws some perspectives.

8.1 Conclusion

This section concludes this thesis. We first make a synthesis of our problematics. Then, we detail how SaS meets the requirements for a user-centric pervasive system.

8.1.1 Synthesis

In the context of collaborative pervasive environments, this thesis advocates that systems should be *user-centric i.e.* enable users to fully benefit from such environments. The study of existing pervasive systems, their particularities and their constituting elements enabled us to establish a requirement list for user-centric systems (detailed in Section 2.2). These requirements raise a

number of major issues to build such systems. Indeed, device heterogeneity and distribution and service volatility necessitate an adaptive and interoperable approach. Moreover, in a pervasive environment users express their needs as scenarios, which implies a composition of multiple services provided by multiple devices. Additionally, users are mobile, they can have needs that combine functionalities present in different locations. In addition, multiple users can share the same environment and may want to collaborate. To answer to all these issues, the contribution of this thesis is SaS (Scenarios as Services): a model of user-centric system to manage scenarized service compositions for collaborative pervasive environments which fulfills the established requirements list.

8.1.2 Requirements Fulfillment

In this section, we synthesize how our SaS solution fulfills all the requirements defined in Section 2.2.

8.1.2.1 Functional Requirements

We have defined five functional requirements and each of them implies two functional sub-requirements. At the end of each previous chapter dedicated to our contribution, we proposed a table that synthesizes how SaS fulfills several functional requirements. Table 8.1 synthesizes all the functional requirements and how SaS fulfills them.

8.1.2.2 Non-Functional Requirements

We defined four non-functional requirements that impact functional requirements. This section presents SaS solutions for each non-functional requirement depending on the functional requirement that they impact.

User friendliness. SaS provides users with a context representation. This context representation corresponds to environment elements necessary for users (services, devices, platforms, platform owners, scenarios). Moreover, this representation is easily personalizable thanks to categories. Users can easily categorize environment elements as they wish.

Users can declare scenarios with SaS-SDL. SaS-SDL contains only the elements necessary for users to express their needs. Complex concepts are transparently handled by the system. A scenario description is easily readable.

SaS provides users with easily understandable operations to control scenario execution. Scenario execution mechanisms are transparently handled. SaS also enables users to easily check scenario execution advancement.

In addition, scenarios are stored as description files and their corresponding descriptions are memorized into the scenario directory. Thus, users can easily retrieve scenarios in the future. They can also categorize them, similarly as services and platforms. Additionally, a scenario is

Functional Requirements	Functionalities Associated
R1.a Context representation	SaS provides users with a persistent and personalizable context directory that contains services, devices, platforms, users and scenarios.
R1.b Context awareness	SaS proposes an interoperable approach that enables to use various service discovery protocols thanks to model transformations.
R2.a Service composition	SaS comprises a scenario description language (SaS-SDL) that enables service composition and contains only elements necessary for users.
R2.b Scenario customization	The SaS-SDL enables to customize scenarios in order to better correspond to users' needs.
R3.a Scenario user control	The scenario orchestrator provides users with scenario execution control commands.
R3.b Scenario execution resilience	SaS schedules a step-by-step scenario execution that enables to adapt to environmental changes and to users' mobility. Additionally, SaS handles fault-tolerance strategies.
R4.a Scenario description availability	Scenarios are memorized into scenario description files and stored into the scenario directory.
R4.b Hierarchical composition	Scenarios are considered as services and thus, hierarchically composable.
R5.a Select what to share	SaS provides scenario sharing modes that enable to select what to share.
R5.b Select who to share with	Platforms export a *Collaborate* service, that enable users to collaborate selectively.

Table 8.1: Fulfillment of requirements by SaS

registered as a service thanks to a generated scenario orchestrator. Therefore, users can easily hierarchically compose a scenario into a new scenario.

SaS provides users with scenario sharing modes. This enables users to easily select who they want to share scenarios with and which part of the scenario they want to share.

Collaborativeness. SaS provides users with a context representation that includes surrounding platforms and their owners. Thus, SaS detect users that have a platform in the environment.

Thanks to SaS-SDL, users can define parameterizable scenarios. A scenario can therefore be used differently by several users.

Users can share scenario execution control with its surrounding platforms. SaS enables users to keep control of scenario execution thanks to two extra sharing-modes: veto and free modes. The scenario provider thus decides if a requested control operation, depending on the requester, can be effected or not.

Furthermore, a platform responsible of a scenario (availability or execution) can ask to another one to be its substitute. The other platform may accept to take the responsibility of the scenario. This enhances collaborativeness.

Scenario sharing modes enable users to select who they want to share scenarios with. In addition, a platform can have access to services that are not reachable for other platforms in the same environment. For instance, services locally deployed or reachable with a service discovery protocol that has a small scope (*e.g.* Bluetooth). Thus, SaS enables platforms to share service access.

Adaptability. SaS adapts its context representation to the different elements of the environment (services, users, etc.). This representation also adapts to environmental changes. Moreover, SaS is interoperable and can use various service discovery protocols. Thus, it adapts to the heterogeneity of the environment elements.

SaS-SDL enables to define scenarios which are customizable at runtime. Thus, users can adapt scenario definition to the context (by specifying certain parameters value at runtime for example).

The step-by-step scenario execution adapts to environmental changes and to users' mobility. Moreover, SaS applies fault-tolerance strategies to anticipate and recover from errors.

Scenario descriptions are registered into scenario description files and thus, persist. In addition, SaS enables a platform responsible of a scenario to be substituted by another one. Scenario availability thus adapts to platform disappearance.

SaS provides a platform directory capability, which enables users to select surrounding platforms for scenario sharing. The directory persistence enables to adapt sharing to platform appearance or disappearance. For instance, a platform can be selected for sharing even if it is not available in the environment. When this platform appears, it automatically has access to the shared scenario.

Mobility. The context representation (that contains three directories) is persistent. Thus, it adapts to device and user mobility. Moreover, users can customize their directories. They can therefore represent different contexts that they have encountered such as locations.

Furthermore, users still have the representation of elements that are not currently available. They can define scenarios with services that are not available simultaneously.

Step-by-step scenario execution is used to reach scenario execution continuity in different locations.

Scenario responsibility can be delegated to another platform. Thus, the scenario can move from a platform to another and thus, remain available.

Finally, thanks to the persistent platform directory, users can select platforms that are not currently available for sharing. The sharing mechanism automatically adapts to platform appearance or disappearance and thus, to their mobility.

8.1.3 SaS Functionalities Synthesis

In the prior section, we have seen how SaS fulfills all the identified requirements. This section details SaS functionalities more.

8.1.3.1 Context Management

To handle device heterogeneity, SaS adopts a generic approach. Thus, it proposes a context representation model that contains main elements of a pervasive environment: services, devices, SaS platforms, users and scenarios. SaS is not restricted to the use of a single service discovery protocol. This enhances interoperability. Its generic approach enables to define model transformations between the service model adopted by a protocol to the SaS' model. We illustrate the model transformation (*cf.* Section 4.2.1.2) with UPnP and the Web Service Description Language (WSDL) used by the Web Services Dynamic Discovery (WS-Discovery) protocol.

Thanks to service discovery (realized with multiple service discovery protocols), SaS dynamically provides users with a context representation. This context representation is composed of three directories: the service directory (that also contains devices), the platform directory (where platform owners are listed) and the scenario directory. SaS listens for environmental changes and thus, dynamically adapts the representation. Moreover, this representation is persistent (*cf.* Section 4.2.2.3). The descriptions of environment elements that are not currently available remain accessible. Thus, context representation adapts to user mobility. This enables, inter alia, users to retrieve and use services in scenarios even if they are not available simultaneously.

Some information that users may want (such as services location) are not present in the context representation that SaS dynamically generates. Thus, users can create categories and attach environment elements to them (*cf.* Section 4.2.2.2). Categories are like tags, an environment element can belong to several categories.

8.1.3.2 Scenario Definition

SaS comprises a scenario description language, namely SaS-SDL, that enables to define scenarios by composing services as basic workflows (*cf.* Section 4.3.1). This language only contains necessary elements for users to express their goals (*e.g.* conditional statements, loops, etc.). Complex mechanisms (*e.g.* asynchronous invocations, exception handling, etc.) are transparently handled by SaS when necessary. Moreover, SaS-SDL also enables users to customize their scenarios to make them parameterizable, more dynamic and representative of their needs (*cf.* Section 4.3.2).

8.1.3.3 Scenario Life-Cycle

SaS manages scenario life-cycle from their creation to their removal (*cf.* Section 5.1). Scenarios are memorized into description files and SaS registers scenarios description into the scenario directory. This directory is part of the context directory. Thus, scenarios description are persistent and can be categorized (such as services and platforms).

Moreover, SaS dynamically generates a scenario orchestrator per scenario This scenario orchestrator enforces, inter alia, the scenario execution life-cycle. Moreover, it provides a service to control the scenario execution. This service contains operations that correspond to execution control commands (*i.e.* start, pause, resume and abort) but also that check scenario execution advancement and retrieve scenario description file. As a service, scenarios are easily used and can be hierarchically composed.

8.1.3.4 Platform Collaboration

SaS platforms export a *Collaborate* service into the environment (*cf.* Section 5.2.2). This service provides a platform's identifier and its owner's name and enables platforms to detect each other. Moreover, this *Collaborate* service contains operations to share scenarios and services among platforms. Users can thus select surrounding platforms to share their creations or services that they have access to. This enhances collaborativeness. Thanks to the use of the platform directory, scenario sharing is easy (selection of platform categories) and adapts to users mobility. A scenario can be shared with a platform, even if it is not currently available. When the selected platform appears in the environment, it automatically receives the shared scenario without any need of further intervention from users.

Moreover, a scenario comprises properties (such as a brief textual description), the description file and the execution control. Therefore, SaS provides users with scenario sharing modes to select which part of the scenario to share (*cf.* Section 5.2.1.2). Users can thus select who to share with and what to share. Users can thus share scenario execution control. However, to keep control of the scenario and avoid execution conflicts, SaS provides users with scenario execution conflict management modes. These extra-modes for shared scenarios enable scenario owners to approve a request for scenario execution control or not.

Additionally, SaS provides a *platform substitution* mechanism, through the *Collaborate* service. It enables to maintain scenario availability and execution when the original platform provider cannot be reached.

8.1.3.5 Scenario Execution Resilience

SaS schedules a step-by-step scenario execution that adapts to the pervasive environment and its evolutions. Thus, scenarios can be executed even if all required services are not simultaneously available. Moreover, scenario execution adapts to users' mobility and enables to execute a scenario in different times on several places. To do so, we define a scenario execution model, which is an execution graph (*cf.* Section 6.1). Such a graph specifies the dynamics of the execution steps and can be compared to Petri nets [85] or finite-state machines [74]. The graph enables SaS to know and anticipate the consequences that a step execution could have. To schedule scenario execution, SaS realizes a transformation model from the scenario description model to the scenario execution model.

8.1.3.6 Dynamic and Adaptive Service Invocation

SaS uses a service broker that enables to dynamically benefit from environment changes. Moreover, this interoperable approach enables a service discovered with a protocol to be used by the user for scenario creation and a similar service, discovered with another protocol, to be used at scenario execution.

Furthermore, SaS defines fault-tolerance mechanisms that enhance scenario execution resilience. These strategies depend on the service's role inside the scenario.

8.2 Perspectives

Our perspectives are plural. We plan to enhance each SaS's functionality and advance our system towards ambient intelligence.

8.2.1 Perspectives for Context Management

It could be interesting to provide users with a device type model, such as the one already proposed by UPnP. The nature of devices, specially in home-automation, can be easily be listed (typically: light, radiator, fan, etc.). Thus, we may propose users a better representation of the surrounding devices, classified by types. In addition, the model may comprise the services and operations proposed by the devices. Thus, we could normalize functionalities that we discover and enable users to express simple needs such as *all the lights off*.

We plan to study multi-agent systems such as *Context Broker Architecture* [16] (CoBrA). In CoBrA, agents maintain a model of the present context and enable to share this model of context knowledge with other agents, services and devices.

Moreover, we may use a context-aware ontology (such as GCOMM [21]) to organize collected data. The context representation would integrate network context, location context, etc. Thus, the context would be better described and understood and the pervasive system might better adapt to its particularities. Typically, it could be interesting to study systems based on ontologies such as PERSEWS [48], that proposes a semantic context representation and thus, considers composition rules by inference.

8.2.2 Perspectives for Scenario Definition

In perspectives, we plan to develop a graphical programming language based on SaS-SDL. A composition language enables to define scenarios. However, it is still too complex for users without any technical knowledge. Tools, such as Yahoo Pipes [92], Automator [4] and Scratch [65], provide non-technical end-users the capability to graphically develop small applications by composing elements.

8.2.3 Perspectives for Scenario Execution

We wish to enable the execution of a specific scenario part. Thus, users may be able to execute the part that they want but also to share a scenario part. Moreover, this mechanism would enable to distribute scenario execution. We can imagine several platforms distribute scenario execution between them, depending on what services each platform can execute. Moreover, we plan to enable multiple instances of a scenario to run on the same platform.

A scenario can imply contradictory orders. We think that it must be left to the devices themselves to implement mechanisms to avoid conflicts. For instance, a device that receives two contradictory orders in a short time may send a specific error message. It could be interesting to develop mechanisms considering this possibility. Meanwhile, it could also be interesting to implement static semantic analysis to avoid scenario execution conflicts. This would necessitate that services furnish extra informations about the effects of their operations.

8.2.4 Perspectives for Scenario and Service Sharing

We may also consider that platforms comprise advanced recognition techniques. Such mechanisms would enable platforms to automatically select platforms for sharing and enhance collaboration.

Additionally, it could be interesting to enable platforms to share service descriptions that they know they can achieve in the future. Therefore, they could participate in a large collaborative scenario. Typically, in robotic science, a robot can announce that it can access to a service somewhere else, and thus can be given the order to invoke this service.

In addition, we would like to develop automatic platform substitution mechanisms. Some research works, such as the user Centric REplicAtion Model (CReaM) [81], already propose a user-centric replication model. We plan to study this works to implement replication strategies

for scenario description and log files. This mechanism, combined with an automatic platform selection (*e.g.* avoid overloading mechanism, by vote, etc.), would enable to maintain scenario availability and execution when a SaS platform brusquely disappears. It would thus enhance fault tolerance.

8.2.5 Perspectives for the Service Broker

It could be interesting to improve service matchmaking (limited to syntactic functional correspondence for the moment). Typically, SaS may use domain ontologies to perform both syntactic and semantic matchmaking.

8.2.6 Security Perspectives

In this thesis, we do not integrate security as a non-functional requirement. Security generally imply low-levels mechanisms which are out-of-scope. Security is usually implemented on the network level. Thus, when we access network capabilities, we can assume that exchanges are safe.

Moreover, our solutions do not imply a high level of security management. We do not enable code mobility or execute locally something that we could not assume the safety of. SaS enables to share text files that must correspond to a particular syntax. Theses files are parsed and interpreted, not executed. In addition, platform collaboration is not automatic and thus, users select who they want to collaborate with. However, it could be interesting to develop some authentication mechanisms and to crypt the exchanged messages.

Part IV

Bibliography and Appendices

Bibliography

[1] Emile Aarts and Boris de Ruyter. New research perspectives on Ambient Intelligence. volume 1, pages 5–14. IOS Press, 2009. (Cited in page 16.)

[2] Anupriya Ankolekar, Mark H. Burstein, Jerry R. Hobbs, Ora Lassila, David L. Martin, Sheila A. McIlraith, Srini Narayanan, Massimo Paolucci, Terry R. Payne, Katia P. Sycara, and Honglei Zeng. DAML-S: Semantic Markup for Web Services. In *Semantic Web Working Symposium, SWWS'01*, pages 411–430, 2001. (Cited in page 46.)

[3] Apache Foundation. Apache Felix UPnP. http://felix.apache.org/site/apache-felix-upnp.html, 2008. [Last access: October 2012]. (Cited in pages 149 et 155.)

[4] Apple. Automator: Your personal Automation Assistant. http://www.macosxautomation.com/automator, 2012. (Cited in page 170.)

[5] Gerd Aschemann, Roger Kehr, and Andreas Zeidler. A Jini-based Gateway Architecture for Mobile Devices. *In Proc. of the Java-Informations-Tage (JIT99)*, pages 203–212, 1999. (Cited in pages 39 et 76.)

[6] Mohamed Bakhouya and Jaafar Gaber. Service composition approaches for ubiquitous and pervasive computing environments: A survey. In Eldon Li and Soe-Tsyr Yuan, editors, *Agent Systems in Electronic Business*, pages 323–350. Information Science Reference/IGI Publishing, 2007. (Cited in page 19.)

[7] Heiko Behrens, Michael Clay, Sven Efftinge, Moritz Eysholdt, Peter Friese, Jan Köhnlein, Knut Wannheden, and Sebastian Zarnekow. Xtext User Guide. https://eclipse.org/Xtext/documentation/1_0_1/xtext.pdf, 2010. [Last access: October 2012]. (Cited in page 150.)

[8] Christian Bettstetter and Christoph Renner. A comparison of service discovery protocols and implementation of the service location protocol. In *6th EUNICE Open European Summer School: Innovative Internet Applications*, pages 13–15, 2000. (Cited in pages 39 et 76.)

[9] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, 2(1):39–59, February 1984. (Cited in page 58.)

[10] Harold Boley, Said Tabet, and Gerd Wagner. Design Rationale for RuleML: A Markup Language for Semantic Web Rules. In *Semantic Web Working Symposium, SWWS'01*, pages 381–401, 2001. (Cited in page 46.)

[11] André Bottaro, Anne Gérodolle, and Philippe Lalanda. Pervasive service composition in the home network. In *21st Int. Conf. on Advanced Networking and Applications*, pages 596–603, 2007. (Cited in pages 20 et 38.)

[12] Jeppe Bronsted, Klaus Marius Hansen, and Mads Ingstrup. Service composition issues in pervasive computing. *IEEE Pervasive Computing*, 9:62–70, 2010. (Cited in page 19.)

[13] Janis Bubenko, Colette Rolland, Pericles Loucopoulos, and Valeria De Antonellis. Facilitating 'Fuzzy to Formal' Requirements Modelling. In *Proceedings of International Conference on Requirement Engineering (ICRE'94)*, pages 154–158, USA, 1994. (Cited in page 27.)

[14] Humberto Cervantes and Richard S. Hall. Autonomous adaptation to dynamic availability using a service-oriented component model. In *ICSE*, pages 614–623. IEEE, 2004. (Cited in pages 139 et 149.)

[15] Dipanjan Chakraborty, Filip Perich, Anupam Joshi, Timothy W. Finin, and Yelena Yesha. A reactive service composition architecture for pervasive computing environments. In *PWC*, pages 53–62, 2002. (Cited in page 36.)

[16] Harry Chen, Tim Finin, and Anupam Joshi. A context broker for building smart meeting rooms. In *Proceedings of the Knowledge Representation and Ontology for Autonomous Systems Symposium, 2004 AAAI Spring Symposium*, 2004. (Cited in page 169.)

[17] Daniel Cheung, Jean-Yves Tigli, Stéphane Lavirotte, and Michel Riveill. Wcomp: a multi-design approach for prototyping applications using heterogeneous resources. *Rapid System Prototyping, IEEE International Workshop on*, 0:119–125, 2006. (Cited in page 38.)

[18] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web Services Description Language (WSDL) 1.1. `http://www.w3.org/TR/wsdl`, March 2001. [Last access: October 2012]. (Cited in page 45.)

[19] James Clark and Steve DeRose. XML Path Language (XPath), Version 1.0. `http://www.w3.org/TR/xpath/`, November 1999. [Last access: October 2012]. (Cited in page 45.)

[20] Joëlle Coutaz, James Crowley, Simon Dobson, and David Garlan. Context is key. volume 48, pages 49–53, 2005. (Cited in page 27.)

[21] Ejigu Dejene, Vasile-Marian Scuturici, and Lionel Brunie. An Ontology-Based Approach to Context Modeling and Reasoning in Pervasive Computing. In *Fifth Annual IEEE International Conference on Pervasive Computing and Communications Workshops (PerComW'07)*, pages 14–19, March 2007. (Cited in page 170.)

[22] Troy Bryan Downing. *Java RMI: Remote Method Invocation*. IDG Books Worldwide, Inc., Foster City, CA, USA, 1st edition, 1998. (Cited in pages 39 et 59.)

[23] Ecma International. ECMA-262: ECMAScript Language Specification. http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-262.pdf, 2009. [Last access: October 2012]. (Cited in page 37.)

[24] Keith Edwards. Discovery systems in ubiquitous computing. volume 5, pages 70–77, Piscataway, NJ, USA, April 2006. IEEE Educational Activities Department. (Cited in page 39.)

[25] Sven Efftinge and Sebastian Zarnekow. Xtend User Guide. http://www.eclipse.org/xtend/documentation/2.3.0/Documentation.pdf, 2012. [Last access: October 2012]. (Cited in page 151.)

[26] José L. Encarnaçao and Thomas Kirste. Ambient intelligence: Towards smart appliance ensembles. *From Integrated Publication and Information Systems to Information and Knowledge Environments*, pages 261–270, 2005. (Cited in page 37.)

[27] Clement Escoffier and Richard S. Hall. Dynamically adaptable applications with iPOJO service components. *6th Int. Conf. on Software composition*, pages 113–128, 2007. (Cited in pages 54 et 149.)

[28] David F. Ferraiolo and D. Richard Kuhn. Role-based access controls. In *Proc. of the 15th National Computer Security Conference (NCSC)*, pages 554–563, October 1992. (Cited in page 58.)

[29] OpenLDAP Foundation. Lightweight Directory Access Protocol (LDAP): Technical Specification Road Map. http://tools.ietf.org/html/rfc4510, 2006. [Last access: October 2012]. (Cited in page 42.)

[30] Maria Fox and Derek Long. Pddl2.1: an extension to pddl for expressing temporal planning domains. volume 20, pages 61–124, USA, December 2003. AI Access Foundation. (Cited in page 46.)

[31] Narain H. Gehani, Hosagrahar Visvesvaraya Jagadish, and Oded Shmueli. Event specification in an active object-oriented database. In *Proceedings of the 1992 ACM SIGMOD international conference on Management of data*, SIGMOD '92, pages 81–90, New York, NY, USA, 1992. ACM. (Cited in page 45.)

[32] Angela Goh, Y.-K. Koh, and Dragan S. Domazet. Eca rule-based support for workflows. volume 15, pages 37 – 46, 2001. (Cited in page 45.)

[33] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. volume 15, pages 287–317, New York, NY, USA, December 1983. ACM. (Cited in page 114.)

[34] Fady Hamoui. *A component-based multi-agent system for autonomous context-aware adaptation - Application to smart homes*. PhD thesis, University Montpellier II, 2010. (Cited in page 37.)

[35] Fady Hamoui, Marianne Huchard, Christelle Urtado, and Sylvain Vauttier. Specification of a component-based domotic system to suport user-defined scenarios. In *21st SEKE Int. Conf.*, pages 597–602, 2009. (Cited in page 37.)

[36] Thomas Heider and Thomas Kirste. Multimodal appliance cooperation based on explicit goals: concepts & potentials. In *Joint Conf. on Smart objects and ambient intelligence: innovative context-aware services: usages and technologies*, pages 271–276. ACM, 2005. (Cited in page 37.)

[37] Ian Horrocks, Peter F. Patel-Schneider, Harold Boley, Said Tabet, Benjamin Grosof, and Mike Dean. SWRL: A semantic web rule language combining OWL and RuleML, 2004. (Cited in page 46.)

[38] Vincent Hourdin, Jean Y. Tigli, Stéphane Lavirotte, Gaëtan Rey, and Michel Riveill. SLCA, composite services for ubiquitous computing. In *Int. Conf. on Mobile Technology, Applications, and Systems*, pages 1–8. ACM Press, 2008. (Cited in page 47.)

[39] Tim Howes. A String Representation of LDAP Search Filters, 1996. (Cited in page 88.)

[40] IBM. IBM 305 RAMAC. http://www.ed-thelen.org/comp-hist/BRL61-ibm03.html, 1961. [Last access: October 2012]. (Cited in page 15.)

[41] Noha Ibrahim and Frédéric Le Mouël. A Survey on Service Composition Middleware in Pervasive Environments. *Int. Journal of Computer Science Issues*, 1:1–12, 2009. (Cited in page 19.)

[42] Information Sciences Institute. RFC 791 : Internet Protocol. http://tools.ietf.org/html/rfc791, 1981. [Last access: October 2012]. (Cited in page 41.)

[43] Intel. Intel Software for UPnP Technology. http://software.intel.com/en-us/articles/intel-software-for-upnp-technology-download-tools/, 2009. [Last access: October 2012]. (Cited in page 155.)

[44] Diane Jordan and John Evdemon. Web services business process execution language version 2.0. http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf, January 2007. [Last access: October 2012]. (Cited in pages 45 et 84.)

[45] Patrick Kinney. ZigBee Technology: Wireless Control that Simply Works, October 2003. Communication Design Conference. (Cited in page 42.)

[46] Thomas Kirste, Thorsten Herfet, and Michael Schnaider. EMBASSI: multimodal assistance for universal access to infotainment and service infrastructures. In *Proceedings of the 2001 EC/NSF workshop on Universal accessibility of ubiquitous computing: providing for the elderly*, WUAUC'01, pages 41–50, New York, NY, USA, 2001. ACM. (Cited in page 37.)

[47] Wojtek Kozaczynski and Grady Booch. Component-Based Software Engineering. *IEEE Softw.*, 15(5):34–36, September 1998. (Cited in page 53.)

[48] Julien Lancia. *Infrastructure orientée service pour le développement d'applications ubiquitaires*. PhD thesis, Université Bordeaux 1, 2008. (Cited in page 170.)

[49] Frank Leymann. Web Services Flow Language (WSFL 1.0). http://xml.coverpages.org/WSFL-Guide-200110.pdf, May 2001. [Last access: October 2012]. (Cited in page 45.)

[50] David Martin, Massimo Paolucci, Sheila Mcilraith, Mark Burstein, Drew Mcdermott, Deborah Mcguinness, Bijan Parsia, Terry Payne, Marta Sabou, Monika Solanki, Naveen Srinivasan, and Katia Sycara. Bringing semantics to web services: The owl-s approach. In *Semantic Web Services and Web Process Composition*, pages 26–42. Springer, 2004. (Cited in page 46.)

[51] Anne McCrory. Ubiquitous? Pervasive? Sorry, they don't compute. http://www.computerworld.com/s/article/41901/Ubiquitous_Pervasive_Sorry_they_don_t_compute, October 2000. [Last access: October 2012]. (Cited in page 16.)

[52] Drew Mcdermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. PDDL - The Planning Domain Definition Language. Technical report, Yale Center for Computational Vision and Control, 1998. (Cited in page 46.)

[53] Sun Microsystems. RPC: Remote Procedure Call Protocol Specification Version 2. http://www.ietf.org/rfc/rfc1057.txt, June 1988. [Last access: October 2012]. (Cited in page 59.)

[54] Marija Mikic-Rakic and Nenad Medvidovic. A Classification of Disconnected Operation Techniques. In *32nd EUROMICRO Conf. on Software Engineering and Advanced Applications*, pages 144–151. IEEE, 2006. (Cited in pages 49, 51 et 134.)

[55] Gordon Moore. Cramming More Components onto Integrated Circuits. *Electronics*, 38:114–117, April 1965. (Cited in page 15.)

[56] Tatsuo Nakajima and Ichiro Satoh. Personal home server: Enabling personalized and seamless ubiquitous computing environments. In *Proceedings of the Second IEEE International Conference on Pervasive Computing and Communications (PerCom'04)*, PER-

COM '04, pages 341–345, Washington, DC, USA, 2004. IEEE Computer Society. (Cited in page 37.)

[57] Eric Newcomer and Greg Lomow. *Understanding SOA with web services*. Addison-Wesley, 2005. (Cited in pages 38 et 41.)

[58] OASIS. Reference Model for Service Oriented Architecture 1.0. http://docs.oasis-open.org/soa-rm/v1.0/soa-rm.html, Oct 2006. [Last access: October 2012]. (Cited in pages 18, 26 et 38.)

[59] OSGi Alliance. OSGi Service Platform Core Specification Release 4. http://www.osgi.org/download/r4v40/r4.core.pdf, 2005. [Last access: March 2012]. (Cited in pages 54 et 149.)

[60] Petri Palmila. Zeroconf and UPnP techniques. Technical report, Helsinki University of Technology, 2007. (Cited in page 39.)

[61] Michael Papazoglou. Service-Oriented Computing : Concepts , Characteristics and Directions. In *4th Int. Conf. on Web Information Systems Engineering*, pages 3–12. IEEE, 2003. (Cited in pages 18 et 26.)

[62] Michael Papazoglou. *Web Services: Principles and Technology*. Prentice Hall, September 2007. (Cited in pages 40, 41 et 77.)

[63] Michael Papazoglou, Paolo Traverso, Schahram Dustdar, and Frank Leymann. Service-oriented computing. volume 46, pages 25–28, 2003. (Cited in page 38.)

[64] Chris Peltz. Web services orchestration and choreography. volume 36, pages 46–52, Los Alamitos, CA, USA, October 2003. IEEE Computer Society. (Cited in page 48.)

[65] Mitchel Resnick, John Maloney, AndrÃl's Monroy-Hernandez, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and YYasmin Kafai. Scratch: Programming for Everyone. *Comm. of the ACM*, 52(11):60–67, 2009. (Cited in page 170.)

[66] Golden G. Richard. Service advertisement and discovery: Enabling universal device cooperation. volume 4, pages 18–26, Piscataway, NJ, USA, September 2000. IEEE Educational Activities Department. (Cited in page 39.)

[67] Daniel Romero, Gabriel Hermosillo, Amirhosein Taherkordi, Russel Nzekwa, Romain Rouvoy, and Frank Eliassen. The DigiHome Service-Oriented Platform. *Software: Practice and Experience*, 2011. 14 pages. (Cited in page 36.)

[68] Stephen Ross-Talbot. Orchestration and Choreography: Standards, Tools and Technologies for Distributed Workflows. In *Proc. Workshop Network Tools and Applications in Biology (NETTLAN'05)*, October 2005. 8 pages. (Cited in page 48.)

[69] Salutation Consortium. Salutation Architecture: Overview. http://salutation.org/wp-content/uploads/2012/05/originalwp.pdf, 1998. [Last access: October 2012]. (Cited in pages 39 et 40.)

[70] Salutation Consortium. Salutation Architecture Specification (Part-1). http://www.salutation.org/, 1999. [Last access: October 2012]. (Cited in pages 11 et 40.)

[71] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-Based Access Control Models. volume 29, pages 38–47, 1996. (Cited in page 58.)

[72] Mahadev Satyanarayanan. A catalyst for mobile and ubiquitous computing. volume 1, pages 2–5, Los Alamitos, CA, USA, 2002. IEEE Computer Society. (Cited in page 17.)

[73] SCA Consortium. Building Systems using a Service Oriented Architecture. http://www.ibm.com/developerworks/library/specification/ws-sca/, 2005. [Last access: October 2012]. (Cited in pages 54 et 55.)

[74] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. volume 22, pages 299–319, New York, NY, USA, December 1990. ACM. (Cited in pages 44, 114, 136 et 169.)

[75] Lionel Seinturier, Philippe Merle, Damien Fournier, Nicolas Dolet, Valerio Schiavoni, and Jean-Bernard Stefani. Reconfigurable sca applications with the frascati platform. volume 0, pages 268–275, Los Alamitos, CA, USA, 2009. IEEE Computer Society. (Cited in pages 36 et 55.)

[76] Evren Sirin, Bijan Parsia, Dan Wu, James A. Hendler, and Dana S. Nau. Htn planning for web service composition using shop2. volume 1, pages 377–396, 2004. (Cited in page 46.)

[77] Katia Sycara, Matthias Klusch, Seth Widoff, and Jianguo Lu. Dynamic Service Matchmaking Among Agents in Open Information Environments. volume 28, pages 47–53, March 1999. (Cited in page 137.)

[78] Clemens Szyperski, Dominik Gruntz, and Stephan Murer. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 2002. (Cited in page 53.)

[79] Satish Thatte. XLANG: Web services for business process design, 2001. (Cited in page 45.)

[80] Jean-Yves Tigli, Stéphane Lavirotte, Rey Gaetan, Hourdin Vincent, Cheung-Foo-Woo Daniel, Callegari Eric, and Michel Riveill. WComp middleware for ubiquitous computing: Aspects and composite event-based Web services. volume 64, pages 197–214, Apr 2009. (Cited in page 38.)

[81] Zeina Torbey, Nadia Bennani, Lionel Brunie, and David Coquil. A decentralized and autonomous replication model for mobile environments . In F. Laforest, editor, *UBIMOB'10 Actes informels*, 2010. (Cited in page 170.)

[82] UPnP Forum. Understanding UPnP: A White Paper. http://www.upnp.org/download/UPNP_UnderstandingUPNP.doc, 2000. [Last access: October 2012]. (Cited in pages 40, 41, 76 et 77.)

[83] Aitor Urbieta, Guillermo Barrutieta, Jorge Parra, and Aitor Uribarren. A survey of dynamic service composition approaches for ambient systems. In *Ambi-Sys Wkshp on Software Organisation and MonIToring of Ambient Systems*, pages 1–8. ICST, 2008. (Cited in page 19.)

[84] Dimitar Valtchev and Ivailo Frankov. Service gateway architecture for a smart home. *Communications Magazine, IEEE*, pages 126–132, 2002. (Cited in pages 25 et 38.)

[85] Wil Van der Aalst and Arthur Ter Hofstede. Workflow Patterns: On the Expressive Power of (Petri-net-based) Workflow Languages. In D. Moldt, editor, *4th Wkshp on the Practical Use of Coloured Petri Nets and CPN Tools*, pages 1–20. University of Aarhus, 2002. (Cited in pages 44, 85, 114, 136 et 169.)

[86] John Veizades, Erik Guttman, Charles Perkins, and Scott Kaplan. Service Location Protocol. www.ietf.org/rfc/rfc2165.txt, 1997. [Last access: October 2012]. (Cited in pages 40 et 42.)

[87] W3C. XML Schema 1.1. http://www.w3.org/XML/Schema, April 2012. [Last access: October 2012]. (Cited in page 45.)

[88] Jean-Baptiste Waldner. *Nanocomputers and Swarm Intelligence*. Wiley-IEEE Press, 1st edition, 2008. (Cited in page 16.)

[89] Mark Weiser. The computer for the 21st century. *Scientific American*, pages 78–89, 1995. (Cited in page 16.)

[90] Stephen A. White. Using BPMN to Model a BPEL Process. 3:1 – 18, 2005. (Cited in page 45.)

[91] Chao-Lin Wu, Chun-Feng Liao, and Li-Chen Fu. Service-Oriented Smart-Home Architecture Based on OSGi and Mobile-Agent Technology. volume 37, pages 193–205, Mar 2007. (Cited in pages 20, 37 et 38.)

[92] Yahoo. Rewire the Web. http://pipes.yahoo.com/pipes, 2012. (Cited in page 170.)

[93] Fen Zhu, Matt W. Mutka, and Lionel M. Ni. Service discovery in pervasive computing environments. volume 4, pages 81–90, Piscataway, NJ, USA, October 2005. IEEE Educational Activities Department. (Cited in page 39.)

APPENDIX A

Scenario Transformation Algorithms

A.1 Action Block Transformation

Algorithm 7 summarizes the algorithm to transform an action block.

Algorithm 7 Scenario Steps Extraction: Action Block Transformation

```
1: function transformActionBlock(ActionBlock aBlock, PrecLink precLink, StepList scenarioStepList) : PrecLink
2: begin
3:     if (!aBlock.isParallel()) then
4:         return transformSequenceActionBlock(aBlock, precLink, scenarioStepList)
5:     else
6:         return transformParallelActionBlock(aBlock, precLink, scenarioStepList)
7:     end if
8: end
```

A.1.1 Sequence Action Block

Algorithm 8 presents the sequence action block transformation.

Algorithm 8 Scenario Steps Extraction: Sequence Action Block Transformation

```
1: function transformSequenceActionBlock(ActionBlock aBlock, PrecLink precLink, StepList scenarioStepList) : PrecLink
2: begin
3:     for all element in aBlock.getElements() do
4:         switch element.isType() :
5:           case ACTIONBLOCK
6:             precLink = transformActionBlock(element, precLink, scenarioStepList)
7:           case ACTION
8:             precLink = transformAction(element, precLink, scenarioStepList)
9:         endswitch
10:    end for
11:    return precLink
12: end
```

A.1.2 Parallel Action Block

Algorithm 9 represents the parallel action block traversal.

Algorithm 9 Scenario Steps Extraction: Parallel Action Block Transformation

```
1: function transformParallelActionBlock(ActionBlock aBlock, PrecLink precLink,
   StepList scenarioStepList) : PrecLink
2: begin
3:     Step forkStep = new ForkStep
4:     forkStep.setPrecedence(precLink)
5:     precLink.getStep().addNextStepLink(forkStep,precLink.getStatus())
6:     precLink = new PrecLink(forkStep, EXECUTED)
7:     scenarioStepList += forkStep
8:     Step joinStep = new JoinStep
9:     joinStep.setJoinMode(AND)
10:    for all element in aBlock.getElements() do
11:        switch element.isType() :
12:          case ACTIONBLOCK
13:            joinStep.addPrec(transformActionBlock(element, precLink, scenarioStepList))
14:          case ACTION
15:            joinStep.addPrec(transformAction(element, precLink, scenarioStepList))
16:        endswitch
17:        precLink.getStep().addNextStepLink(joinStep,precLink.getStatus())
18:    end for
19:    scenarioStepList += joinStep
20:    precLink = new PrecLink(joinStep, EXECUTED)
21:    return precLink
22: end
```

A.2 Action Transformation

Algorithm 10 presents the function that dispatches action transformation depending on the action type.

Algorithm 10 Scenario Steps Extraction: Action Transformation

```
1: function transformAction (Action action, PrecLink precLink, StepList scenarioStepList) : PrecLink
2: begin
3:     switch action.isType :
4:         case SERVICE_EXECUTION
5:             return transformServiceExecution(action, precLink, scenarioStepList)
6:         case COND_STATEMENT
7:             return transformCondStatement(action, precLink, scenarioStepList)
8:         case WHILE_LOOP
9:             return transformWhileLoop(action, precLink, scenarioStepList)
10:        case REPEAT_LOOP
11:            return transformRepeatLoop(action, precLink, scenarioStepList)
12:        case COND_EVENT
13:            return transformCondEvent(action, precLink, scenarioStepList)
14:        case TIME_EVENT
15:            return transformTimeEvent(action, precLink, scenarioStepList)
16: end
```

A.2.1 Service Execution

Algorithm 11 illustrates the operation invocation transformation.

Algorithm 11 Scenario Steps Extraction: Service Execution Transformation

```
1: function transformServiceExecution (Action action, PrecLink precLink, StepList scenarioStepList) : PrecLink
2: begin
3:     InvocationStep invocationStep = new InvocationStep
4:     invocationStep.setPrecedence(precLink)
5:     precLink.getStep().addNextStepLink(invocationStep,precLink.getStatus())
6:     invocationStep.setExec(action.getOpInvocation())
7:     scenarioStepList += invocationStep
8:     precLink = new PrecLink(invocationStep, EXECUTED)
9:     return precLink
10: end
```

A.2.2 Conditional Statement

Algorithm 6 (detailed in Section 7.1.2.5) presents the conditional statement transformation.

A.2.3 While Loop

Algorithm 12 presents the while loop transformation.

Algorithm 12 Scenario Steps Extraction: While Loop Transformation

```
1: function transformWhileLoop (Action action, PrecLink precLink, StepList scenar-
   ioStepList) : PrecLink
2: begin
3:     JoinStep joinStep = new JoinStep
4:     joinStep.setJoinMode(OR)
5:     joinStep.setPrecedence(precLink)
6:     precLink.getStep().addNextStepLink(joinStep,precLink.getStatus())
7:     scenarioStepList += joinStep
8:     ConditionStep whileConditionStep = new ConditionStep
9:     whileConditionStep.setExec(action.getCondition())
10:    whileConditionStep.addPrec(new PrecLink(joinStep, EXECUTED))
11:    joinStep.addNextStepLink(whileConditionStep,EXECUTED)
12:    scenarioStepList += whileConditionStep
13:    PrecLink thenPrecLink = transformActionBlock(action.getActionBlock(),
14: new PrecLink(whileConditionStep,TRUE), scenarioStepList)
15:    joinStep.addPrec(thenPrecLink)
16:    thenPrecLink.getStep().addNextStepLink(joinStep,thenPrecLink.getStatus())
17:    precLink = new PrecLink(whileConditionStep, FALSE)
18:    return precLink
19: end
```

A.2.4 Repeat Loop

Algorithm 13 presents the repeat loop transformation.

Algorithm 13 Scenario Steps Extraction: Repeat Loop Transformation

```
1: function transformRepeatLoop (Action action, PrecLink precLink, StepList scenar-
   ioStepList) : PrecLink
2: begin
3:     JoinStep joinStep = new JoinStep
4:     joinStep.setJoinMode(OR)
5:     joinStep.addPrec(precLink)
6:     precLink.getStep().addNextStepLink(joinStep,precLink.getStatus())
7:     scenarioStepList += joinStep
8:     CalculusStep calculusStep = new CalculusStep
9:     calculusStep.setCalculusValue(action.getRepeatValue())
10:    PrecLink doPrecLink = transformActionBlock(action.getActionBlock(),
11: new PrecLink(joinStep,EXECUTED), scenarioStepList)
12:    calculusStep.addPrec(doPrecLink)
13:    thenPrecLink.getStep().addNextStepLink(calculusStep,doPrecLink.getStatus())
14:    scenarioSteps += calculusStep
15:    joinStep.addPrec(new PrecLink(calculusStep, NOT_FINISHED))
16:    calculusStep.addNextStepLink(joinStep, NOT_FINISHED)
17:    precLink = new PrecLink(calculusStep, EXECUTED)
18:    return precLink
19: end
```

A.2.5 Conditional Event

Algorithm 14 presents the conditional event transformation.

Algorithm 14 Scenario Steps Extraction: Conditional Event Transformation

```
1: function transformCondEvent (Action action, PrecLink precLink, StepList scenarioStepList) : PrecLink
2: begin
3:     JoinStep joinStep = new JoinStep
4:     joinStep.setJoinMode(OR)
5:     joinStep.addPrec(precLink)
6:     precLink.getStep().addNextStepLink(joinStep,precLink.getStatus())
7:     scenarioStepList += joinStep
8:     ConditionStep conditionStep = new ConditionStep
9:     conditionStep.setPrecedence(new PrecLink(joinStep, EXECUTED))
10:    joinStep.addNextStepLink(conditionStep, EXECUTED)
11:    conditionStep.setExec(action.getCondition())
12:    scenarioStepList += conditionStep
13:    joinStep.addPrec(new PrecLink(conditionStep, FALSE))
14:    conditionStep.addNextStepLink(joinStep, FALSE)
15:    precLink = transformActionBlock(action.getActionBlock(), new PrecLink(conditionStep, TRUE), scenarioStepList)
16:    return precLink
17: end
```

A.2.6 Time Event

Algorithm 15 constitutes the time event transformation.

Algorithm 15 Scenario Steps Extraction: Time Event Transformation

```
1: function transformTimeEvent (Action action, PrecLink precLink, StepList scenarioStepList) : PrecLink
2: begin
3:     CalculusStep calculusStep = new CalculusStep
4:     calculusStep.setPrecedence(precLink)
5:     precLink.getStep().addNextStepLink(calculusStep,precLink.getStatus())
6:     calculusStep.setExec(action.getTimeEventValue())
7:     scenarioStepList += calculusStep
8:     precLink = transformActionBlock(action.getActionBlock(), new PrecLink(calculusStep, EXECUTED), scenarioStepList)
9:     return precLink
10: end
```

Appendix B

Publications

A service component framework for multi-user scenario management in ubiquitous environments.
Matthieu Faure, Luc Fabresse, Marianne Huchard, Christelle Urtado and Sylvain Vauttier.
Proceedings of the 6th International Conference on Software Engineering Advances (ICSEA'11).
Barcelona, Spain, October 2011. AR 30%

User-defined scenarios in ubiquitous environments: creation, execution control and sharing.
Matthieu Faure, Luc Fabresse, Marianne Huchard, Christelle Urtado and Sylvain Vauttier.
Proceedings of the 23rd International Conference on Software Engineering and Knowledge Engineering (SEKE'11), pages 302-307.
Miami, USA, July 2011. AR 31%

Towards scenario creation by service composition in ubiquitous environments.
Matthieu Faure, Luc Fabresse, Marianne Huchard, Christelle Urtado and Sylvain Vauttier.
Proceedings of the 9th BElgian-NEtherlands software eVOLution seminar (BENEVOL'10), S. Ducasse, L. Duchien and L. Seinturier editors, pages 145-155.
Lille, France, December 2010.

Mission-oriented Autonomic Configuration of Pervasive Systems.
Guillaume Grondin, Matthieu Faure, Christelle Urtado and Sylvain Vauttier.
Proceedings of the 7th International Conference on Software Engineering Advances (ICSEA'12).
Lisbon, Portugal, November 2012. AR 30%

Management of Scenarized User-centric Service Compositions for Collaborative Pervasive Environments.

Pervasive (or ubiquitous) computing is a paradigm for environments containing distributed interconnected devices that embed electronics. These devices provide a remote access to numerous functionalities that assist us in our daily life. Service-Oriented Architectures are suitable to design software for pervasive environments. Indeed, each device provides its own set of functionalities as services. Without any extra mechanism, users can only use a single service at a time. Nevertheless, their needs usually correspond to scenarios which involve a composition of multiple services, provided by multiple devices.

In this thesis, we advocate that a pervasive system must, on the one hand, enable users to easily express their needs through scenario creation and, on the other hand, propose to users a representation of their context so that they can benefit from both their environment and its changes. In addition, the presence of several users implies that users must be able to collaborate.

Our contribution, named SaS (Scenarios as Services), fulfills these requirements. It proposes an interoperable approach that adapts to its environment. It provides users with a customizable and persistent representation of their context and includes a scenario description language targetted to users. Scenarios are easy to control, customize and reuse. SaS schedules the step-by-step execution of scenarios to adapt to environmental changes and benefit from user mobility (scenario execution splitted over time on successive distinct sites). Finally, SaS includes scenario sharing mechanisms which are a basis for collaboration. A prototype of SaS, based on industrial standards (e.g., OSGi), proves the feasibility of our contribution and serves for its evaluation on a simple use case.

Keywords: Pervasive / ubiquitous computing, Service oriented architecture, Service composition, Software Engineering

Mise en œuvre de la composition de services scénarisée et centrée utilisateur pour les environnements pervasifs collaboratifs.

L'informatique pervasive (ou ubiquitaire) est un support pour des environnements contenant de nombreux objets (ou dispositifs) disséminés, équipés d'électronique et interconnectés. Ces dispositifs fournissent un accès distant à une multitude de fonctionnalités qui nous aident dans notre vie quotidienne. Les Architectures Orientées Services sont adaptées à la conception de logiciels pervasifs. En effet, chaque dispositif fournit son propre ensemble de fonctionnalités sous la forme de services. Ainsi, en l'absence de mécanisme complémentaire, les utilisateurs se trouvent limités à utiliser les services isolément alors que leurs besoins correspondent à des scénarios qui impliquent une composition de multiples services offerts par plusieurs appareils.

Dans cette thèse, nous défendons qu'un système pervasif doit : d'une part, permettre aux utilisateurs d'exprimer facilement leurs besoins en créant des scénarios et d'autre part, proposer à ses utilisateurs une représentation et des moyens de gestion de leur contexte afin qu'ils puissent tirer le meilleur parti de leur environnement et de ses changements. De plus, la présence de plusieurs utilisateurs implique la nécessité de collaborer. Par ailleurs, l'exécution de scénarios doit être résiliente aux changements environnementaux et aux actions des utilisateurs. Elle doit ainsi s'adapter dynamiquement et, si possible, tirer profit du contexte et des changements de l'environnement.

Notre contribution, nommée SaS (Scenarios as Services), répond à ces objectifs. Elle propose une approche interopérable capable de s'adapter à l'environnement. Elle fournit une représentation persistante et personnalisable du contexte et inclut un langage de description de scénarios destiné aux utilisateurs. Ces scénarios sont facilement contrôlables, personnalisables et réutilisables. Elle planifie l'exécution pas à pas des scénarios, afin de s'adapter aux changements de l'environnement et de bénéficier des avantages de la mobilité des utilisateurs (exécution d'un scénario, dans la durée, sur plusieurs lieux). Enfin, elle inclut le partage de scénarios qui permet aux utilisateurs de collaborer. Un prototype de SaS, basé sur des normes industrielles (telle qu'OSGi), prouve la faisabilité de notre contribution et nous permet de l'évaluer sur un cas d'étude simple.

Mots clés : Informatique pervasive/ubiquitaire, Architecture orientée service, Composition de services, Génie logiciel