Engineering Reconfigurable Distributed Software Systems: Issues Arising for Pervasive Computing

Apostolos Zarras, Manel Fredj, Nikolaos Georgantas, Valérie Issarny

Engineering Reconfigurable Distributed Software Systems: Issues Arising for Pervasive Computing

Apostolos Zarras[1], Manel Fredj[2], Nikolaos Georgantas[2], and Valerie Issarny[2]

[1] Dept. of Computer Science, University of Ioannina, Greece
zarras@cs.uoi.gr
http://www.cs.uoi.gr/ zarras
[2] INRIA-Rocquencourt, France
{Manel.Fredj, Nikolaos.Georgantas, Valerie.Issarny}@inria.fr
http://www-rocq.inria.fr/arles/

Abstract. This chapter establishes a common base for discussing reconfigurability in distributed software systems in general and in pervasive systems in particular, by introducing a generic reconfiguration cycle. Following this cycle, we discuss in detail three former efforts on reconfigurable pervasive systems, and draw conclusions about the capacity of existing approaches to deal with open, dynamic, ad hoc environments. We, then, outline our approach towards uncontrolled reconfiguration targeting environments in which no centralized coordination or prior awareness between services being composed is assumed. Our solution supports awareness of service semantics and related service discovery, configuration change detection and state transfer, interface-aware dynamic adaptation of service orchestrations and conversation-aware checkpointing and recovery.

1 Introduction

Dynamic reconfiguration proved, along the years, to be a major issue towards the development of dependable distributed software systems. In principle, we may distinguish three basic types of reconfiguration situations based on the targeted needs [1]. First, we have *corrective* reconfiguration that aims at dealing with faults causing failures in the constituents of a system. Second, we have *perfective* reconfiguration that targets changes performed towards meeting the evolving functional and non-functional requirements of the system. Finally, we have *adaptive* reconfiguration aiming at the proper functioning of devices and their hosted applications that are dynamically integrated in a computing system without prior knowledge of the functional constraints (e.g., available functionalities and resources) imposed by this system. The first two types of reconfiguration were typically targeted by stationary distributed systems. On the other hand, the need for the last type of reconfiguration arose with the latest emergence of pervasive computing systems. An in between evolution with respect to these two system domains were nomadic computing systems, which added wide area

M. Butler et al. (Eds.): Fault-Tolerant Systems, LNCS 4157, pp. 364–386, 2006.

mobility to stationary distributed systems and were a precursor to pervasive computing systems. There, mobility makes the computing environment less predictable than in stationary systems, thus as well implying the need for adaptive reconfiguration, to a lesser extent, however, than in pervasive systems.

Reconfiguration in stationary distributed systems – architecturally modelled in terms of components and connectors [2] – concerns adding, removing or substituting components or connectors. Changes should take place at runtime to avoid compromising the availability of the overall system. The basic techniques to achieve this goal rely on a main authority that is often called *reconfiguration manager* [3]. This authority has knowledge of the changes that are going to take place and its main responsibility is to perform them, whilst not jeopardizing the overall system integrity. Techniques proposed for handling reconfiguration aim at isolating a component of the system that is to be removed or substituted by enforcing request blocking [3,4] or request redirection [5] on components that use this component. By request blocking, this component eventually reaches a state where it is not used, and reconfiguration can be safely performed. Request redirection supports immediate component replacement: a connector can direct all communication after a certain point in time from the old component to the new one. In this case and if the components are statefull, state transfer [4,6,7] from the old component to the new one enables a seamless transition.

Evolution to nomadic computing systems enabled users to be mobile and to carry around wireless devices. The key concept in such systems is that a client software entity resides on the user's device and is connected to some remote server software entity. Connectivity between client and server may be intermittent due to insufficient wireless network coverage or limited bandwidth shared between multiple users. However, it is assumed that eventually the client will reconnect to the same server or to some replica of the server. Then, the objective is to enable users to use their mobile devices even during periods of low or non-connectivity. The basic technique applied is to emulate locally at the client the remote server, e.g., by locally replicating server data [8,9] or code [10], or by just buffering client requests, and by synchronizing client and server upon reconnection [11]. Further attention may be required when a server is updated by multiple clients, or when clients connect to and disconnect from more than one server replica [8]. Data integrity should be maintained when data is replicated on multiple hosts. In terms of architectural modelling, connectors can handle transparently for components the disconnection, reconnection, or connection to replicated servers.

Being one step further, pervasive computing systems aim at making computational power available everywhere. Mobile and stationary devices will dynamically connect and coordinate to seamlessly help people in accomplishing their tasks. For this vision to become reality, systems must adapt themselves with respect to the constantly changing conditions of the pervasive environment: (i) the highly dynamic character of the computing and networking environment due to the intense use of the wireless medium and the mobility of the users; (ii) the resource constraints of mobile devices, e.g., in terms of CPU, memory

and battery power; and (iii) the high heterogeneity of integrated technologies in terms of networks, devices and software infrastructures. In response to such challenges, the Service-Oriented Architecture (SOA) paradigm [12] provides an attractive solution. A service is a consistent piece of functionality made available over the network by a software entity and accessed by other – customer – software entities. A service is accessible at a specific network address, via a well-defined interface, i.e., a set of supported operations, and over a specific middleware communication protocol. Besides this generic definition, no restriction is imposed on the implementation of a service, which enables integration of diverse technologies and loose coupling between interacting services, making SOA suitable for dynamic, heterogeneous environments. Further, a service supports a set of valid service *conversations*, which are *processes* in the form of *workflow* that define the behavior of the service. All the above information concerning a service constitutes the service specification, which may be published on the network, thus, made discoverable by customers via a service discovery protocol. Discovering a service means matching a required service specification with a provided service specification. A direct matching technique constitutes in comparing – among others – the required and provided interface specifications syntax, assuming that two syntactically compatible interfaces imply semantically – i.e., concerning their meaning – compatible services. However, enforcing an agreement on a common syntax for denoting semantics is impossible to achieve in open environments, such as pervasive computing environments. Thus, the latest tendency is towards adopting semantic representation paradigms for specifying and matching services even when these differ in their syntactic interfaces. Such paradigms employ *ontologies* to represent concepts and related well-founded formalisms to enable machine reasoning about them [13]. Finally, services may be composed towards realizing complex functionalities. Two essential models for service composition are: (i) service *orchestration*, where a customer invokes a set of services in a coordinated way, and (ii) service *choreography*, where a set of services interact with each other in a peer-to-peer fashion.

Regarding reconfiguration, the distinctive feature of pervasive systems is that software entities making up a system may have no *a priori* knowledge of each other before their dynamic composition. Bindings between entities are ad hoc and temporary, which is served pretty well by the loosely coupled interaction model of SOA. However, unawareness not only concerns which concretely the entities are, but is further extended to the specification of entities, e.g., in terms of interfaces and conversations. This means that entities composing pervasive systems have not necessarily been developed to be syntactically compatible. In the same direction, after a disconnection, a client software entity will most likely not reconnect to the same server software entity or even a replica of it, but rather to another server. This new server should be semantically equivalent or similar to the old one, and thus compatible with the client, but it will not necessarily be syntactically compatible with it. Thus, semantic paradigms prove to be essential for pervasive systems. Semantic matching enables associating semantically compatible software entities, but this is not sufficient. To integrate such

entities, adaptation is further needed in terms of interfaces and conversations. Furthermore, no central reconfiguration management can be established in such systems. We call such reconfiguration *uncontrolled*. Uncontrolled reconfiguration in pervasive systems distinguishes itself from the controlled one in stationary and nomadic systems, where prior awareness is a basic assumption.

Uncontrolled reconfiguration in pervasive systems presents numerous challenges as made clear in the above. In this chapter, we particularly contribute with an approach for reconfiguration in pervasive environments, which comprises syntactic and semantic dynamic service discovery, change detection, state transfer, interface-aware orchestration adaptation and conversation-aware checkpointint and recovery mechanisms. Before presenting our approach to uncontrolled reconfiguration in pervasive computing systems (Section 3), we examine in detail three related efforts on reconfigurable pervasive systems (Section 2), which gives a concrete view of ongoing research in the domain, and discuss goals, strong points and constraints in current approaches. We particularly examine the capacity of these approaches to deal with open, dynamic, ad hoc pervasive environments. In the beginning of the latter section, we introduce a general view of the reconfiguration procedure in distributed software systems, which establishes a common base for discussing both existing approaches and the proposed one. Finally, we conclude with a summary of this chapter, and point out open issues and future work (Section 4).

2 Reconfigurable Systems

In this section, we discuss in detail three former efforts related to dynamic reconfiguration of pervasive computing systems (Section 2.2). To allow comparative study of such systems, we introduce a *generic reconfiguration cycle*, which provides an abstract descriptive view of the reconfiguration procedure of a system; this cycle can pretty well apply to different distributed software systems – stationary, nomadic, pervasive – and related reconfiguration techniques (Section 2.1). Our detailed presentation of the three approaches allows a comprehensive view of the whole reconfiguration procedure, each time for a complete, consistent system.

2.1 Generic Reconfiguration Cycle

To allow a separation of concerns, we distinguish between the *Reconfigurable System (RS)*, its *Context* or *Environment (CE)*, and the *Reconfiguration Management System (RM)*, as depicted in Figure 1. CE is in constant interaction with RS, for example, affecting RS functioning or hosting some functional entity that may join RS as a result of reconfiguration. RM integrates all functionality necessary for RS reconfiguration, while RS should only hold some architectural and functional properties supporting its reconfiguration along with the capacity to respond to RM actions; otherwise, RS is not aware of its reconfiguration. We further assume that the architecture of RS, CE and RM can be described at

a generic level in terms of components and connectors [2]. Based on that, we deal with architectural reconfiguration of RS in terms of adding, removing and substituting components [4] and connectors [6,7].

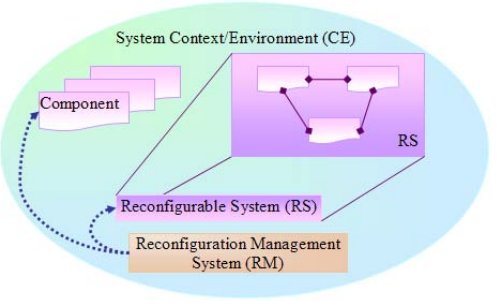


Fig. 1. Separation of concerns for reconfiguration

We call *reconfiguration cycle* a complete sequence of phases that takes place during the execution of RS and reconfigures it taking it from a consistent state to another consistent state, i.e., one from which RS can continue normally its execution rather than progressing towards an error state. We introduce the generic reconfiguration cycle depicted in Table 1, where lines are associated to phases succeeding one another in time in ascending order, and columns are associated to functional entities that may act or be acted upon concurrently, specifically RS, CE and RM. RM's activity is presented in two columns: the first one indicates RM's overall functions, while the second one is dedicated to RM's data processing concerning reconfiguration. Our reconfiguration cycle aims at enabling a common, abstract descriptive view of the reconfiguration procedure for a large variety of systems. Representing the exact state transitions of RS, CE and RM and the eventually complex interactions taking place between them for any of these systems with a single reconfiguration cycle is certainly not possible. Thus, we make no strict assumptions about the functional entities and phases of the reconfiguration cycle, other than the ones stated in the above. In the following, we introduce in detail the various phases of the reconfiguration cycle.

In Phase 1, RS executes normally, while RM monitors both RS and its CE. RM holds a set of data concerning RS, which were produced at RS development or deployment time. Thus, RS configuration description represents the current configuration of RS, which includes, for example, the functional dependencies between components. Further, the normal execution of the combination RS-CE is delimited by a set of execution constraints, for example, which components need to be up and running, or minimum communication bandwidth available to RS [14,15]. Definition of normal execution naturally directly defines as well abnormal execution. Along with this, enhanced execution may be identified, enabling automated perfective reconfiguration. Finally, a set of possible reconfiguration strategies and actions may be provided, specifying the scope of RM's role [14,15].

Table 1. Generic reconfiguration cycle

Phase	Reconfigurable system (RS)	System context/ environment (CE)	Reconfiguration management system (RM)	
			Function	Data processing
1	RS executes normally		RM monitors RS and CE	• Holds RS configuration description • Holds description of RS-CE constraints for normal, abnormal, enhanced execution • Holds set of reconfiguration strategies and actions • Produces RS and CE monitoring data • Periodically saves dynamic RS data
2	• RS executes normally/abnormally • RS generates cause for reconfiguration	CE generates cause for reconfiguration		
3			RM detects cause for reconfiguration	• Uses RS and CE monitoring data • Uses description of RS-CE constraints for normal, abnormal, enhanced execution
4			RM specifies reconfiguration	• Uses RS configuration description • Uses set of reconfiguration strategies and actions • Produces sequence of actual reconfiguration actions
5	RM prepares RS for reconfiguration	RM determines participation of CE to new configuration	RM applies reconfiguration	• Uses sequence of actual reconfiguration actions • Saves dynamic RS data
6	RM adapts RS to new configuration	RM adapts CE to new configuration		Uses saved dynamic RS data
7	RM reconfigures RS	RM reconfigures CE		Produces new RS configuration description

Moreover, during its execution, RM manages some dynamic RS data. Thus, it produces monitoring data concerning RS and CE. It may as well periodically save dynamic RS data, such as the state of RS components, thus checkpointing RS. Another example of such activity is the local caching or pre-fetching of remote server data by a client entity in nomadic systems [8,9].

In Phase 2, a cause for reconfiguration emerges, generated by either RS or CE. This cause may be accompanied by abnormal RS execution or not. An example of the first case may be the disconnection or failure of an essential component of RS, or the drop in the available bandwidth, while an example of the second case may be the availability of a new component that offers enhanced quality of service. In Phase 3, RM detects the emerging cause for reconfiguration after having observed current monitoring data and compared it with execution constraints. In Phase 4, RM decides its way of intervention to reconfigure RS. To this aim, RM uses the current RS configuration description and the set of possible reconfiguration strategies and actions in order to produce the sequence of actual reconfiguration actions that it will take. For example, based on dependencies between components, RM may identify components affected by the intended reconfiguration and take some preventive action before applying reconfiguration.

In Phases 5, 6 and 7, RM applies the sequence of actual reconfiguration actions. In Phase 5, RM prepares RS for reconfiguration. This preparation concerns components affected by the intended reconfiguration and may take several forms. For example, request blocking [3,4], request redirection [5] or request queuing may be enforced on components that interact with a component that is about to leave RS. RM may save the state of a leaving mobile component just in time if the component issues a warning before leaving; this provides a perfectly up-to-date state, which may not be the case for state saved by periodic checkpointing. A similar last minute action may be taken by a client entity in a

nomadic system to locally pre-fetch remote server data just before disconection [9]. Further in the same phase, RM determines the participation of CE to the new configuration, i.e., whether some new components coming from CE will join RS. This task heavily depends on whether RM has *a priori* or not knowledge of the new components that will be introduced into RS. Such awareness may range: from concretely knowing already from the deployment of RS which these components are, to having to carry out dynamic component discovery based on syntactic or semantic descriptions of the interfaces and supported conversations of the components. In Phase 6, RM may have to adapt either one or both of RS and CE to the new configuration, so that their integration be possible. RS adaptation may involve coordination workflow adaptation to be compatible with a new component being introduced, or workflow rollback to cancel some interrupted transaction. CE adaptation may involve transferring the saved state of the leaving component to its substitute component [6,7,16]. In the case of a nomadic system upon reconnection, CE adaptation may involve synchronizing the remote server with updates maintained locally on the client, or submitting to the server all locally queued client requests [8]. Finally, in Phase 7, RM carries out the final reconfiguration action on RS, possibly integrating some new components coming from the CE. Now, the new RS configuration description is available, and RS may go back to normal execution (Phase 1).

In the next section, we study in detail three approaches to reconfigurable pervasive systems on the basis of the above discussion. We highlight the relation of the presented efforts to the introduced generic reconfiguration cycle by referencing specific phases of the cycle.

2.2 Reconfigurability in Pervasive Computing Systems

The first two reconfiguration approaches that we discuss in this section, *RAPIDware* and *CASA*, focus on techniques enabling a smooth transition of the system from its initial to its target configuration, where no loss of component processing or data occurs during reconfiguration. RAPIDware calculates a safe reconfiguration strategy and employs request blocking based on dependencies between components, while CASA manages state transfer at object programming level for dynamically replaceable local objects. The third approach, *Polymorphic applications*, enables migration of distributed component-based applications between pervasive environments. The combined presentation of these three approaches allows looking into how well-established techniques coming from stationary distributed systems are applied to pervasive systems, as well as pointing out new requirements and solutions specific to pervasive systems.

RAPIDware. This project addresses perfective reconfiguration of component-based pervasive systems [17]. The reconfiguration approach is applied to a wireless video streaming application, which involves a video server multicasting video streams to clients residing on laptops and handheld devices. Streaming is secure

via encryption of the wireless stream. Reconfiguration aims at enhancing system properties: in the specific application, encoder and decoder components are substituted by alternative ones in order to enforce a higher encryption scheme, thus enhancing security. All available encoder and decoder components are known before system execution and have been developed to directly fit together in terms of interfaces and behavior. Reconfiguration is executed by a central *reconfiguration manager (RM)*, which coordinates a set of *reconfiguration agents (RAs)* attached to system components involved in the reconfiguration.

RM holds the system configuration description, which is in terms of (Phase 1): (i) *dependency relationships* between components, i.e., the correct functionality of a component may require the correct functionality of other components; and (ii) *critical communication segments* between components, i.e., communication segments whose interruption may cause errors in the system. Further, the reconfiguration manager holds the set of all possible reconfiguration actions. A fixed cost is associated to each reconfiguration action, depending on associated system blocking time, delay of data delivery and resource usage.

Upon some external command, e.g., by the user, RM obtains the target configuration (Phases 2-3). A reconfiguration procedure is *safe* if (a) it does not violate dependency relationships and (b) it does not interrupt critical communication segments. Based on that, RM specifies reconfiguration in three steps (Phase 4):

1. Based on the source and target configurations and the dependency relationships, RM produces a set of safe configurations. A safe configuration is one that satisfies all the dependency relationships.
2. RM constructs a safe reconfiguration graph, where vertices are all safe configurations and edges are all possible reconfiguration actions connecting safe configurations. This graph can be deduced from available reconfiguration actions. To ensure a safe reconfiguration procedure, reconfiguration actions should not interrupt critical communication segments.
3. RM applies Dijkstra's shortest path algorithm on the graph to find a safe reconfiguration path with minimum weight, where the weight of a path is the sum of the costs of all the edges along the path.

Finally, RM applies the calculated reconfiguration path. For each reconfiguration action in the path:

1. RM sends block commands via RAs to affected components to enforce suspension of their functioning. Block commands are applied after waiting for the last critical communication segment to be completed (Phase 5).
2. RM/RAs carry out the actual reconfiguration action, e.g., replacing a component by another one. When the adaptation action is done, RM/RAs send resume commands to blocked components reactivating them (Phase 7).

The interest of the RAPIDware approach lies in the systematic way for calculating a safe reconfiguration path. Certainly, even if applied to a wireless mobile application, a well-controlled environment is required, where all information about component functionality and interaction, both for current and for new system components, is known in advance.

CASA (Contract-based Adaptive Software Architecture). This framework enables dynamic adaptation of a component-based software application executing on a mobile device in response to changes in contextual information such as user's location, or to changes in resource availability such as bandwidth [18]. A device hosting adaptive applications is required to run an instance of the CASA runtime system, which is responsible for monitoring the changes in the environment and adapt these applications accordingly. Components in CASA-enabled applications are objects in an object-oriented programming language. This reconfiguration approach is realized at object programming level and consequently inherits all restrictions coming from the strong coupling inherent in the object-oriented paradigm – which was relaxed in the descendant component-oriented and service-oriented paradigms. Nevertheless, it presents a number of features that can be of interest as well to reconfigurable systems based on the latter two paradigms.

Replaceability of objects is determined based on the notion of a *set of alternative classes*, which is a collection of classes whose instances can dynamically replace each other. This means that these classes: (1) conform to the same interface; (2) the pre- and post-conditions of the methods of their interfaces are the same; and (3) a valid *persistent* state of an instance of one such class can be mapped to a valid persistent state of an instance of another such class. To enable replacement, the *Bridge* software architectural pattern [19] is used, where every set of alternative classes is associated with a unique *Handle* class, which conforms to the same interface as the classes of the set. Clients of an object are actually bound to an instantiation of the Handle class, which allows hiding from them the fact that the object may be dynamically replaced.

CASA adaptation is based on an *application contract*, which is divided into *context elements*. Each context element represents a state of contextual information of interest to the application and contains a list of alternative configurations of the application, suited to the particular state of contextual information. Thus, reconfiguration is decided and carried out in order to respect the application contract (Phases 1-3). Regarding specification of reconfiguration (Phase 4), two replacement strategies are defined: (a) in *lazy replacement*, an already running component is allowed to complete its current execution before being replaced; (b) in *eager replacement*, the execution of a running component is suspended, and the execution resumes again from the point where it was suspended, after the component is replaced. To *eagerly replace* an object objA by an object objB, where both are handled by a Handle object objH, the following steps are taken:

1. objH starts queueing calls made to objA (Phase 5).
2. objA is notified to suspend execution of the current call. Suspension can be done only when execution of objA has reached one of the explicitly predefined *safe points* at which execution can be resumed correctly by objB. The information about the safe point where the call is suspended is passed to objH (Phase 5).
3. objH creates objB (Phase 5).

4. objH reinvokes the suspended call on objB, passing the information about the safe point where the call was previously suspended, in order to enable objB to resume the execution correctly (Phases 6-7).

A necessary condition for valid eager replacement is that the transferred state of objA can get transformed into a reachable state of objB. However, this may not be possible for a *transient* state of objA. In this case, lazy replacement is applied, where objA is not running at the time of replacement, and thus the state to be transferred is the *persistent* state of objA, which, as indicated above, can become a valid persistent state of objB.

As already indicated, even if the object-oriented architectural style may be restrictive, the interest of CASA lies in its management of state transfer which is a general issue concerning reconfiguration. As also observed for RAPIDware, reconfiguration in CASA requires as well a well-controlled local environment. Response to context changes based on an application contract is also worth noting in CASA, even if the alternative application configurations suited to a particular contextual state are pre-defined.

Polymorphic applications. This approach addresses application migration with the user across pervasive environments that may differ in terms of available devices and services as well as context [20]. Migration consists in suspending an executing application and resuming it later in a new environment. The targeted pervasive environment, called an *Active Space*, is situated in a physical space like a room or a building, and consists of various entities including users, applications, services and devices. An example polymorphic application is one that supports a user's slide show by integrating distributed resources, such as a PowerPoint viewer component, a wall-mounted display and a GUI component. Application structure is based on the Model-View-Controller framework [6], consisting of input (controller), output (view) and logic (model) components. Applications execute on top of Gaia, a CORBA-based meta operating system that manages all physical and digital entities in an Active Space.

Reconfiguration concerns three kinds of application adaptation: change in the type of components, change in the number of components, and change in the devices on which these components execute. These types of adaptation are based on the notion of *semantic similarity* of application components, stating that an application component can be substituted by another component if it allows the user to perform the same tasks in some manner. Thus, a PowerPoint viewer can be replaced by an Acrobat Reader viewer (if appropriate data transcoding is done) or by a Speech Engine that reads out the text in the slides. Certainly, Acrobat Reader is semantically closer to PowerPoint. Semantic similarity between components is determined with the help of ontologies that define a hierarchy of components based on the kinds of tasks that they help users to perform. Application migration between two Active Spaces is performed by two collaborating instances of the *Migration Service (MS)*, a central coordinating entity that controls an Active Space.

MS holds the current structure of an executing application stored in an *Application Customized Description (ACD)* file. Further, MS has access to the *Space*

Repository, which maintains information concerning all devices, components and services available in the Active Space (Phase 1). The migration procedure is triggered by the user through a GUI; the user specifies the Active Space to which the application is to be migrated (Phases 2-3). Then, MS saves the current state of the application along with its structure in the ACD file, and communicates the file to the new Active Space over the network. MS in the new Active Space takes the old ACD of the application and generates a new ACD for the application, after performing appropriate adaptation in three steps (Phases 4-6):

1. MS consults the Active Space ontologies to identify classes of components that are semantically similar to the components listed in the old ACD, as well as classes of devices that can host these component classes. Some additional components that should do, e.g., data transcoding, may be needed.
2. MS queries the Space Repository to get instances of the classes of devices obtained from the previous step that are available in the new Active Space.
3. For each identified component, MS decides the cardinality and the devices on which the components must be instantiated using rules involving the context of the new Active Space and preferences of the user. Context includes the location of the user in the room, the location of devices, the presence of other people in the room, the current activity of the user and so on.

Finally, once MS arrives at a new application structure, it instantiates this application in the new Active Space (Phase 7).

The approach of polymorphic applications is very interesting, as it highlights several issues of pervasive applications, such as mobility of users between pervasive environments, on-the-fly integration of available resources and adaptation to them, semantic similarity between resources, and context-awareness. Nevertheless, even if resources differ between Active Spaces, composition of resources within an Active Space is pretty direct in terms of interfaces, and only data encoding adaptation needs to be dealt with. While this is reasonable for stationary resources, it cannot be assumed for mobile resources present on devices of mobile users, which also make part of the pervasive environment.

Concluding this section, we point out that the presented efforts on reconfigurable pervasive systems largely assume a well-controlled environment: a central coordinating entity is responsible for reconfiguration, and has absolute control and, mostly, full *a priori* knowledge over available resources. In the next section, we deal with reconfiguration in *uncontrolled* environments, in an attempt to come closer to the realization of the concepts of pervasive and ubiquitous computing.

3 Uncontrolled Reconfiguration in Pervasive Computing Systems

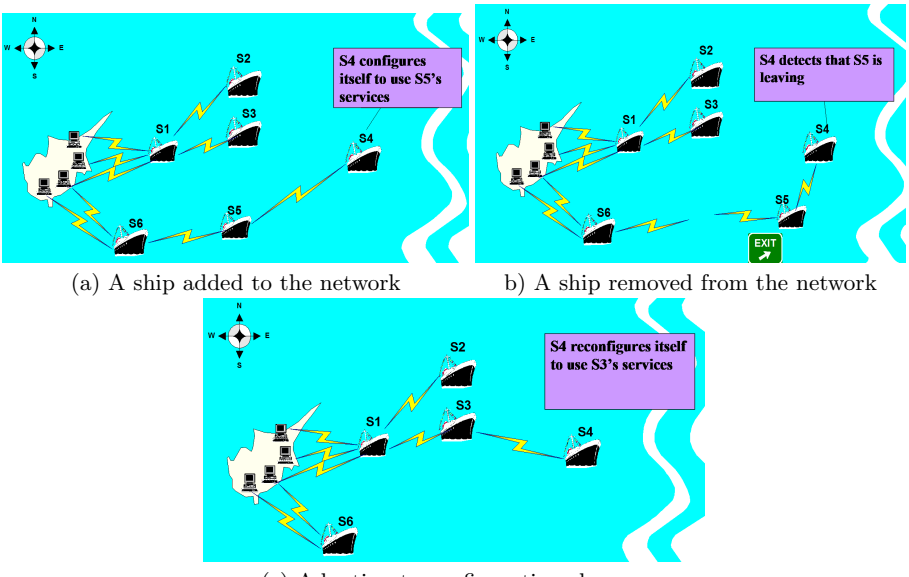
In this section, we present our vision of *uncontrolled* reconfiguration targeting open, dynamic, ad hoc pervasive environments. Our approach adopts the SOA

paradigm. To discuss in more detail the basic functional requirements for dealing with uncontrolled reconfiguration in SOA-based environments, we employ a motivating scenario inspired by [21] (Section 3.1). Based on this scenario, we introduce a service-oriented pervasive environment enhanced with awareness of semantics of services (Section 3.2), and we outline the essential mechanisms supporting reconfiguration in such an environment (Section 3.3). Throughout the present section, we relate our approach to the discussion of Section 2 by referencing specific phases of the generic reconfiguration cycle.

3.1 Scenario and Requirements

In our scenario, we are placed in the near territory of the island of Cyprus. Our pervasive environment consists of several services offering, e.g., tourist information, hotel reservation and car reservation. These services execute on stationary hosts located onshore. The environment that we consider further comprises mobile hosts located on cruise ships, yachts, and other boats. At a short distance from the island, software entities residing on mobile hosts may have access to the services located onshore through a wireless network. If moving further from the island, however, their only possibility to access the services is through satellite-based connections, which are usually expensive and inefficient (especially in the case of GEO networks). To confront this problem, the island's local authorities realized the following setup. The stationary software entities located onshore may actually recruit volunteer mobile entities that can serve as their proxies. Proxies provide indirect wireless access to the onshore services to mobile entities that do not have direct access to these services. As an exchange, the crew members and the tourists onboard may benefit from more favorable hotel, restaurant and car rental prices. Figure 2 gives three snapshots of our pervasive environment resulting from the mobility of the participating entities. In Figure 2(a), the mobile entity S4 is added to the pervasive environment. The entity requires using a hotel reservation service. Since S4 does not have direct access to the stationary services, it selects S5 as a proxy to the required service. In Figure 2(b), the geographical location of S5 obliges it to leave the pervasive environment. In Figure 2(c), S4 has to deal with the change triggered by the S5 entity. The removal of S5 may take place while S4 is trying to use the proxy services provided by S5.

Preserving the environment's consistency in the presence of the aforementioned changes involves dealing with the following issues. In Figure 2(a), the newly added entity should be able to execute its orchestration processes. Consequently, it should discover services suitable for the realization of these orchestration processes. In the open environment of the scenario, discovering services that are syntactically suitable should be considered as the exception rather than the rule. Thus, syntactic discovery of services is not sufficient; supporting semantic description and matching of services is an essential requirement. Then, to be able to use the discovered, semantically compatible services, adaptation is further needed in terms of interfaces and conversations. The newly added entity should adapt its orchestration processes to both the interfaces and



(a) A ship added to the network

b) A ship removed from the network

(c) Adapting to configuration changes

Fig. 2. A pervasive environment formed around the island of Cyprus

conversations of the discovered services. Getting to Figures 2(b) and (c), entities that use the leaving entity should detect its departure, so as to properly adapt their affected orchestration processes. The affected processes are distinguished into *pending* and *inactive*. Pending processes are ones whose execution started before the mobile entity decides to leave and involve the removed entity. Inactive processes are non-instantiated processes that involve the removed entity. In the case of a pending process, the affected entity should discover new services that can substitute the ones of the removed entity, semantically and, if possible, syntactically suitable, and adapt the process to the interfaces and conversations of the new services. Furthermore, the process should be rolled back to a point where it is possible to resume its execution, now with the new services in the place of the removed ones. State transfer between the old and the new services, if and whenever possible, could minimize rolling back or make it unnecessary. Finally, entities used by the leaving entity should also detect this incident so as to terminate all the pending conversations initiated by this entity.

The main outcome from the above discussion is that the effective support for dynamic reconfiguration in pervasive environments requires mechanisms for:

(1) semantic and syntactic service discovery; (2) configuration changes detection and state transfer; (3) process dynamic adaptation; and (4) checkpointing and recovery.

Apart from our motivating example, several other scenarios may involve the requirements we identified here. Consider for instance some of the scenarios identified by ISTAG (Information Society Technologies Advisory Group) for

ambient intelligence (AmI) environments [22]. In the scenario that concerns AmI environments for business, employees who used to work at a fixed location, today change working locations and environments frequently. In such cases, the employees that are *added* in a new working environment would require access to location-specific, syntactically or semantically compatible services. Similarly, in AmI environments supporting E-Government, people may *migrate* from one country to another one. Different countries may employ semantically equivalent procedures for these people (e.g., for validating a driver's licence), supported by semantically compatible E-Government services offered by each community. Confronting the previous, involves transparently adapting the processes used by the immigrants with respect to the E-services of the new country that they visit. In AmI for intelligent transport systems the goal is to develop intelligent vehicles able to monitor traffic conditions using services offered either by the environment or by other vehicles. *Moving* from one area to another implies adapting the processes used by intelligent vehicles with respect to the interfaces offered by the AmI environment that supports the new area.

In the following sections we concentrate on our sailing example. In particular, we introduce a semantics-aware service-oriented environment that can effectively represent the pervasive environment of our sailing scenario, and we outline the required mechanisms in the context of this environment.

3.2 Semantic Service-Oriented Pervasive Environment

Adopting the service-oriented architectural style in the context of pervasive computing systems implies employing a middleware infrastructure that supports it. SOA middleware infrastructures for pervasive environments should support the execution of services on top of resource-constrained devices. As an appropriate such middleware platform, we employ WSAMI [23], which is a lightweight Web Services middleware suitable for mobile devices with limited resources. A Web service is identifiable by a URI (Unified Resource Identifier), has its interface described in the XML-based WSDL language, and is accessible over the XML-based SOAP communication protocol on top of standard Internet protocols like HTTP.

To deal with dynamic reconfiguration in a pervasive environment, we introduce the notion of a pervasive configuration C, which consists of entities *available* in the environment: a set ME of networked mobile entities, and a set SE of networked stationary entities, where an entity e (mobile or stationary) is a collection of software functionalities – which will be specified in the following – executing on a host over WSAMI. For the sailing scenario introduced in the previous section, *availability* is defined with respect to a specific entity e, i.e., $C.ME \cup C.SE$ are entities accessible to e thanks to network coverage; we also call them e's *neighboring* entities in the following. In terms of the generic reconfiguration cycle (Section 2.1), C is the union of the reconfigurable system (RS) and its context or environment (CE). We specify the addition (e.g., may apply to the case of an entity joining the environment) and removal (e.g., may apply to the case of an entity leaving the environment, or the case of an entity that

fails) of an entity e as two actions that cause, respectively, the inclusion and exclusion of e in $C.ME$ ($C.SE$). The addition and the removal actions may be events generated by either the environment or the entities themselves.

An entity e (mobile or stationary) consists of: a set PS of provided application services ws, a set PR of orchestration processes pr, a service discovery service SD, a process execution engine PEE, a changes detection service D, a checkpointing service CH, a recovery service RE and possibly, a state transfer service ST. The $e.SD$, $e.D$, $e.CH$, $e.RE$ and $e.ST$ are system services, as well deployed over the WSAMI platform. In terms of the generic reconfiguration cycle, these system services, as well as part of the functionalities of $e.PEE$, constitute the reconfiguration manager (RM), which is completely distributed: there is an RM instance included in each entity of C. The $e.PS$ comprises concrete service specifications. A concrete service specification is a tuple consisting of the WSDL interface specification of the service, the URI identifying where the service is deployed, and the service conversations, which follow the standard, XML-based, Business Process Execution Language (BPEL) [24].

Orchestration processes are also defined in BPEL. Each process pr that belongs to $e.PR$ is a tuple that contains: a set of activities a, and a set of services ws required for the execution of these activities. Activities may be either simple ones, involving the invocation of an operation provided by a service, or complex ones, consisting of more than one constituent activity. Complex activities include *sequence* activities, comprising the sequential execution of two or more constituent activities; *while* activities, consisting of the iterative execution of a constituent activity; *flow* activities, involving the concurrent execution of two or more constituent activities; and *switch* activities, allowing the selection amongst two or more alternative activities. Required services ws are identified in terms of required WSDL interface specifications. URIs of concrete provided services that syntactically match these required services are resolved by service discovery. We assume that concrete services that provide interfaces syntactically compatible to the ones required by the orchestration process, also provide syntactically compatible conversations.

Finally, to enable semantic service specification and discovery, C is further characterized by an ontology O. The ontology O is defined for the purpose of this chapter as a graph whose nodes represent different semantic classes sc of services that may be provided by the entities of C. The edges between the nodes represent semantic relations between the classes. Currently, we assume *generalization* and *aggregation* relationships. Each semantic class aggregates the syntactic WSDL specifications of alternative standard interfaces which may be provided by different, semantically compatible, services ws that belong to this class; along with each service interface specification, the service conversations specification is also included. Different, semantically compatible, services that belong to the same semantic class provide alternative service conversations. Figure 3 gives an ontology that corresponds to the scenario discussed in the previous section. Specifically, we have the HotelReservation, CarReservation and TouristInformation classes,

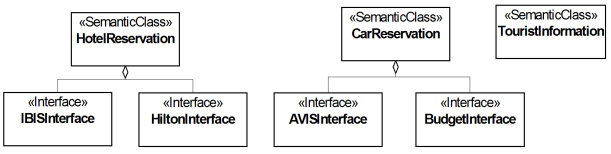


Fig. 3. An ontology for the sailing scenario

comprising services that may provide various kinds of interfaces (e.g., the IBISInterface and the HiltonInterface interfaces for the case of HotelReservation).

3.3 Mechanisms Supporting Reconfiguration

In the context of the semantic service-oriented pervasive environment introduced in the previous section, we now sketch the mechanisms elicited in Section 3.1.

Semantic and Syntactic Service Discovery. At the time when the entity e is added in a pervasive configuration C, e's service discovery $e.SD$ obtains information about services provided by e's (mobile and stationary) neighboring entities (i.e., $C.ME \cup C.SE$). More specifically, $e.SD$ periodically checks the environment for other instances of SD services hosted by neighboring entities, and maintains a related registry. This task is realized by multicasting a discovery request in a standard discovery protocol (e.g., the Service Location Protocol - SLP). Then, $e.SD$ provides two basic operations for syntactic and semantic service search.

The syntactic search takes as input the WSDL interface specification of a required service ws. When invoked by e, $e.SD$ makes corresponding calls to the SD services of e's neighbors. The replies of all neighbors concerning provided services that syntactically match ws are merged into a single set RES_{ws}, which is returned back to e. Caching the most recent replies enables optimizing service discovery latency and bandwidth consumption. The semantic search takes as input a required semantic class sc from the pervasive configuration ontology $C.O$, or the WSDL interface specification of a required service ws. In the second case, the semantic class sc to which ws belongs has to be resolved. The semantic search is executed in the same way as above discussed for the syntactic search. Now, replies contained in RES_{sc} or RES_{ws} concern provided services that belong to the semantic class sc. In the second case, this means that services contained in RES_{ws} semantically match ws. Optionally, to increase the possibility of discovering a provided service that can be employed, we may have supplementary semantic search calls for services that belong to *specializations* of the semantic class sc.

In our sailing scenario, a possible syntactic search could be for hotel reservation services which provide an interface that follows the IBISInterface WSDL specification (Figure 3). Similarly, a possible semantic search could be for any services belonging to the HotelReservation semantic class.

Configuration Changes Detection and State Transfer. The changes detection service (D) is a simple push-based notification service. When an entity e

is removed from a pervasive configuration, a corresponding event may be pushed in the change detection services of all of e's neighbors (mobile and stationary), which are thus notified about the fact that e is being removed or has already been removed from the pervasive configuration C (this depends on the particular network latency). A broadcast-based approach is employed for (D) instead of a unicast one that would involve only the entities that are affected by the removal, since it is not possible to know all of them in advance. Actually, the entity being removed knows only the entities that are engaged in a pending conversation with it. It can not possibly know the entities that intend to begin a conversation with it. Getting to our sailing scenario, at the time when S5 is leaving, notification events may be sent to all of S5's neighbors (including S4, who is actually using S5). Issuing a notification before departing or not may depend on the good will of the leaving entity or simply on its consciousness of its departure. In the case of no warning, pending connections with the leaving entity will be broken and new connection attempts will fail; thus, the changes detection services of affected entities will eventually be notified by the underlying middleware. Certainly, when applied, pre-departure notification enables detecting the change and dealing with it as early as possible. Moreover, it enables communicating the state of the removed entity. The state of e is the aggregate of the states of all services provided by e. When e is removed from a pervasive configuration C, its (current or logged) state may be exported with a corresponding event to the ST services of all of its neighbors. This information may be directly discarded or used afterwards so as to initialize compatible entities that are going to take e's place in the execution of orchestration processes that use e's services.

Process Dynamic Adaptation. The process execution engine PEE of an entity e has two main functionalities. The first one is to execute the orchestration processes of e. This execution may be triggered by a user in an application-dependent way (e.g., through a GUI). The second functionality amounts in adapting the orchestration processes dynamically in response to changes that occur in the pervasive configuration C that includes e. The first functionality of PEE is a typical one provided by various process execution engines that already exist for non-mobile service-oriented systems (e.g., ActiveBPEL[1]). On the other hand, the second functionality is introduced specifically to deal with the problem of dynamic reconfiguration in pervasive computing configurations. The realization of the second functionality involves the service discovery (SD), changes detection (D), state transfer (ST), checkpointing (CH) and recovery (RE) services.

We first consider the case of reconfiguration upon addition of an entity e to the pervasive configuration C. In terms of the generic reconfiguration cycle, this is actually an initial configuration of RS, which was not executing before. Upon entering in C and if triggered in an application-dependent way, e is requested to adapt its orchestration processes with respect to the services provided by the entities of C (Phases 2-4). Accordingly, e searches for syntactically compatible services required for the execution of its processes (Phase 5). If for every required

[1] http://www.activebpel.org/

ws_j service used in a process pr_i the syntactic search returns a non-empty set RES_{ws_j} of matching services, then one of them is selected. Following, e's process execution engine ($e.PEE$) should adapt pr_i with respect to the URI of the selected service (Phases 6-7). Suppose now that for a service ws_j, required by pr_i, the syntactic search returns an empty set of results. Following, e performs a semantic search, which may also return a set of alternative services (Phase 5). Suppose that a service ws_j^{sem} is selected. Following, $e.PEE$ should adapt pr_i with respect to the interface, the conversations and the URI of ws_j^{sem} (Phases 6-7). Achieving this step in a systematic manner involves using the concept of *refinement rules*.

In general, the refinement rules are a part of the overall reconfiguration policy/strategy [25] used upon an event that signals a configuration change (entity addition, removal). Specifically, a set of refinement rules is specified along with every pair of services (ws_j, ws_j^{sem}) which provide alternative standard interfaces that are aggregated by the same semantic class sc of the pervasive configuration ontology $C.O.$ A refinement rule is a mapping relation between the activity a_i of a conversation process realized using ws_j and a corresponding activity a_j of a conversation process realized using ws_j^{sem}. In the simplest case, a refinement rule may directly map an invocation activity to another invocation activity. However, it is also possible that a refinement rule maps an invocation activity to a more complex activity (e.g., a sequence activity), or the inverse (e.g., a sequence activity to a simple invocation activity). We may envision even more complicated refinement rules, mapping complex activities (e.g., a sequence activity) to other complex activities (e.g., a while activity). Hence, to adapt the processes that use the ws_j service into corresponding ones that use the ws_j^{sem} one, $e.PEE$ uses the refinement rules defined for the (ws_j, ws_j^{sem}) pair.

Getting to our sailing scenario, let us assume that when S4 joins the environment (Figure 2(a)), it requires a service that provides the IBISInterface towards the realization of the orchestration process that is given in Figure 4(a). According to this process, the customer at some point confirms a reservation by executing a sequence of two invocations, involving the Book and the PrePayment operations. The PrePayment operation is required by IBIS hotels to deposit a percentage of the overall amount to pay for the room. Suppose now that there are no available proxies providing the IBISInterface and the semantic search returns among others a semantically compatible service that provides the HiltonInterface. The HiltonInterface provides operations that are semantically compatible with the operations of the IBISInterface, may differ, however, in terms of operation names and parameter names and data types. With regard to process structure, interaction with the service providing the HiltonInterface is simpler given that there is no need for advance payment. Consequently, to adapt S4's orchestration process to the conversation of the service providing the HiltonInterface, besides adapting semantically matching operations in terms of names and parameters, the sequence of the Book and the PrePayment operations should be reduced into a simple invocation that involves the Confirm operation of the HiltonInterface (Figure 4(b)).

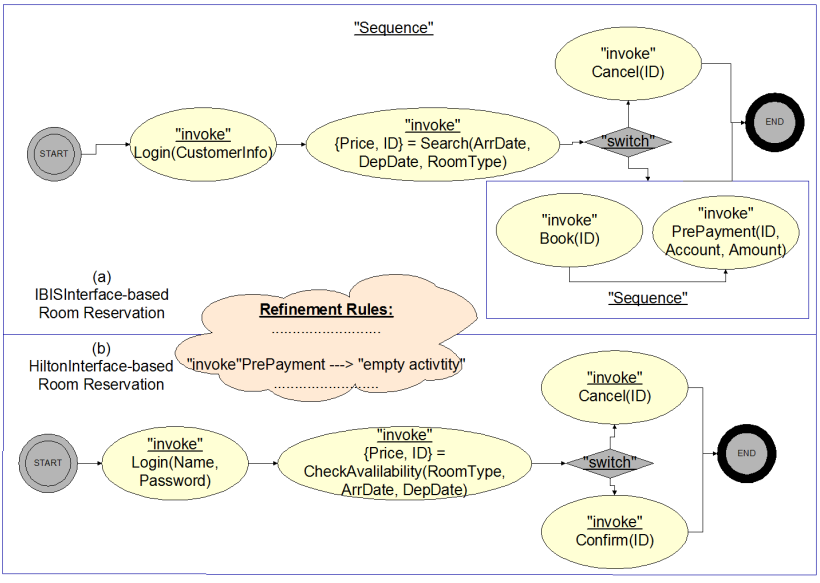


Fig. 4. Inactive process adaptation

We now consider the case of reconfiguration upon removal of an entity *e'* from the pervasive configuration *C*. As previously discussed, the changes detection service *D* of an entity *e* will receive a notification (either soft or hard) about entity *e'* removed from the pervasive configuration *C* (Phases 2-3). If *e* uses *e'* in some of its orchestration processes (i.e., the affected processes), these processes should be adapted as in the case where *e* is added to *C* (Phase 4). Specifically, for every affected process pr_i, a syntactic search, possibly followed by a semantic one, is performed for a substitute entity (Phase 5). Following, the affected process is adapted by *e.PEE* in the way introduced earlier (Phase 6). At this point, the role of *e.PEE* is done if pr_i is an inactive process. Otherwise, if pr_i is pending, the following steps are further followed (Phases 6-7): If the removed entity issues a notification before departing, and both the removed and the substitute entities provide state transfer capabilities and are *state-compatible*, then the part of the state of the removed entity that concerns pr_i is imported in the substitute entity. Then, the execution of pr_i resumes from a point that depends on the previous step. This particular step is realized based on the checkpointing and recovery mechanisms detailed in the following paragraph. In the worst case, all the activities of pr_i that involve the removed entity may have to be restarted.

Checkpointing and Recovery. The checkpointing and recovery mechanisms discussed here are primarily inspired by traditional mechanisms used in conventional distributed systems [26], adapted to the concepts of orchestrations and conversations. Specifically, the checkpointing and recovery mechanisms take charge of rolling back a pending orchestration process to a point that preserves the process *consistent* execution. These particular mechanisms are triggered if

an entity e' is substituted by another one e" and the entities do not provide any state transfer capabilities. As previously discussed, an orchestration process in BPEL consists of different types of activities executed using operations offered by Web services deployed in the environment. Moreover, a Web service specification comprises the service's interface (i.e., the operations provided), a service URI and the valid conversations that can be realized by invoking the service's operations (Section 1). Therefore, an orchestration process actually consists of a set of valid conversations executed over a set of Web services. Based on this observation, we define *consistency* for a pending orchestration process pr of e as follows: The execution of the pending orchestration process after reconfiguration is consistent if there exist no pending constituent conversations of this process that execute using data produced by conversations involving e'. Based on this definition, we discuss in the following the basic responsibilities of the checkpointing and recovery mechanisms.

The checkpointing mechanism is used on the side of Web services that participate in the execution of certain conversations. The checkpointing mechanism requires from a Web service to specify along with its valid conversations the decomposition of these conversations into atomic sub-conversations. An atomic sub-conversation is a subset of the activities of a conversation that must be rolled-back as a whole. Based on this specification, the mechanism checkpoints the state of the service at the beginning of each atomic sub-conversation triggered.

The recovery mechanism is used on the side of entities performing certain orchestration processes. Specifically, before the beginning of each orchestration pr performed by e that consists of conversations with a set of Web services, the orchestration is divided into atomic sub-conversations, based on the Web services specifications. Following, the mechanism discovers the possible data dependencies that exist between sub-conversations performed with Web services offered by different entities. A data dependency exists between two atomic sub-conversations spr_i and spr_j if the entity that performs the orchestration process uses data resulting from output messages of operations invoked during spr_i to construct input messages issued during the invocation of operations performed during spr_j.

Taking, now the case where e' is substituted by e", the following actions are taken by the recovery mechanism of e. If pr is pending, the recovery mechanism locates every pending sub-conversation spr_j that depends on conversation spr_i performed with e'. Following, it notifies the checkpointing mechanism responsible for spr_i that spr_i must be rolled-back to the beginning of its execution. Regarding the overall orchestration pr, the recovery mechanism rolls it back to the beginning of the execution of the first sub-conversation performed with e'.

Getting back to our sailing scenario, suppose that S5 is leaving the environment and issues a related notification. S4's changes detection service will receive this notification. Suppose that at this time S4 is executing a pending orchestration process that consists of a flow of two conversations that execute concurrently (Figure 5). The first one is the HiltonInterface-based conversation of Figure 4(b)

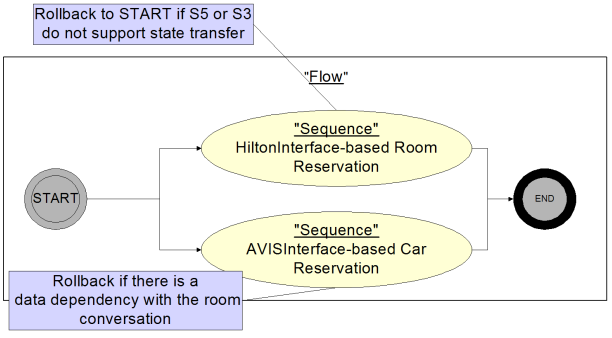


Fig. 5. Pending process adaptation

that executes on S5 and the second one is an AVISInterface-based car reservation conversation executed on S2. The syntactic search that follows the notification of S4 results in selecting S3 as S5's substitute. Suppose that both conversations are specified as being atomic. This means that before their beginning the states of S5 and S2 are saved by the checkpointing mechanisms deployed on the aforementioned entities. In the absence of state transfer, the whole HiltonInterface-based conversation should be restarted. Moreover, if there exists a data dependency between the two conversations the AVISInterface-based conversation must be rolled-back. Otherwise, the execution of the latter continuous normally.

4 Conclusion

In this chapter we established a common base for investigating reconfigurability in distributed software systems, by introducing a generic reconfiguration cycle. Based on this cycle, we investigated in detail former efforts on reconfigurable pervasive systems. The main outcome of this study was that these approaches are strongly influenced by principles, assumptions and techniques proposed in the context of stationary systems, where reconfiguration is *controlled* in the sense that a central reconfiguration manager is in control, *a priori* aware of entities currently present in the system and entities that are candidate to join the system. Finally, we discussed our approach towards uncontrolled reconfiguration targeting environments in which no centralized coordination or prior awareness between services being composed is assumed. The proposed solution comprises syntactic and semantic dynamic service discovery, change detection, state transfer, interface-aware orchestration adaptation and conversation-aware checkpointing and recovery mechanisms.

A number of issues are still open in our approach, which are to be dealt with in our current and future work. A language for specifying refinement rules and the process adaptation mechanism are currently under development. Particularly, we focus on an aspect-oriented approach that relies on our prior work in this field [27]. The development and global interconnection of ontologies proposed by paradigms such as the Semantic Web [28] may prove useful for our approach.

The issue of QoS-aware process adaptation is also an interesting direction for future research [29]. Finally, till now, we have considered service compositions in the form of orchestrated processes. Extending the proposed approach to deal with services choreography is challenging as it may possibly involve distributed coordination mechanisms for service discovery, changes detection, checkpointing, recovery, state transfer, and process adaptation.

References

1. Oreizy, P., Medvidovic, N., Taylor, R.N.: Architecture-based runtime software evolution. In: Intl. Conf. on Software Engineering, Kyoto, Japan (1998)
2. Garlan, D., Shaw, M.: An introduction to software architecture. Technical Report CMU-CS-94-166, Carnegie Mellon University (1994)
3. Kramer, J., Magee, J.: The evolving philosophers problem: Dynamic change management. IEEE Transactions on Software Engineering **16**(11) (1990) 1293–1306
4. Bidan, C., Issarny, V., Saridakis, T., Zarras, A.: A dynamic reconfiguration service for corba. In: ICCDS '98: Proceedings of the 4th IEEE International Conference on Configurable Distributed Systems. (1998) 35–42
5. Minsky, N., Ungureanu, V., Wang, W., Zhang, J.: Building reconfiguration primitives into the law of a system. In: ICCDS '96: Proceedings of the 3rd International Conference on Configurable Distributed Systems. (1996) 62–69
6. Blair, G.S., Blair, L., Issarny, V., Tuma, P., Zarras, A.: The role of software architecture in constraining adaptation in component-based middleware platforms. In: Proceedings of MIDDLEWARE'00. (2000) 164–184
7. Zarras, A.: Online upgrade of object-oriented middleware. Journal of Object Technology **3**(7) (2004) 121–140
8. Kistler, J.J., Satyanarayanan, M.: Disconnected operation in the coda file system. In: Thirteenth ACM Symposium on Operating Systems Principles. Volume 25., Asilomar Conference Center, Pacific Grove, U.S., ACM Press (1991) 213–225
9. Kuenning, G.H., Popek, G.J.: Automated hoarding for mobile computers. In: SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles, New York, NY, USA, ACM Press (1997) 264–275
10. Fuggetta, A., Picco, G.P., Vigna, G.: Understanding Code Mobility. IEEE Transactions on Software Engineering **24**(5) (1998) 342–361
11. Joseph, A.D., deLespinasse, A.F., Gifford, J.A.T.D.K., Kaashoek, M.F.: Rover: a toolkit for mobile information access. In: Proceedings of the 15th ACM Symposium on OperatingSystems Principles (SOSP '95), Copper Mountain Resort, Colorado (1995) 156–171
12. Papazoglou, P., Georgakopoulos, D., eds.: Service-oriented computing. In: Communications of the ACM. Volume 46. ACM Press (2003)
13. Martin, D., Paolucci, M., McIlraith, S., Burstein, M., McDermott, D., McGuinness, D., Parsia, B., Payne, T., Sabou, M., Solanki, M., Srinivasan, N., Sycara, K.: Bringing semantics to web services: The owl-s approach. In: First International Workshop on Semantic Web Services and Web Process Composition (SWSWPC 2004), San Diego, California, USA. (2004)
14. Cheng, S.W., Garlan, D., Schmerl, B.R., Sousa, J.P., Spitznagel, B., Steenkiste, P., NingningHu: Software architecture-based adaptation for pervasive systems. In: ARCS. (2002) 67–82

15. Garlan, D., Cheng, S.W., Huang, A.C., Schmerl, B., Steenkiste, P.: Rainbow: Architecture-based self-adaptation with reusable infrastructure. Computer **37**(10) (2004) 46–54
16. Soules, C., Appavoo, J., Hui, K., Silva, D., Ganger, G., Krieger, O., Stumm, M., Wisniewski, R., Auslander, M., Ostrowski, M., Rosenburg, B., Xenidis, J.: System support for online reconfiguration (2003)
17. Zhang, J., Cheng, B.H., Yang, Z., McKinley, P.K.: Enabling safe dynamic component-based software adaptation. In: Architecting Dependable Systems III, Springer Lecture Notes in Computer Science (2005)
18. Mukhija, A., Glinz, M.: Runtime adaptation of applications through dynamic recomposition of components. **16**(11) (2005) 124–138
19. Gamma, E., Helm, R., Johnson, R.: Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley Professional Computing Series. Addison-Wesley (1995) GAM e 95:1 1.Ex.
20. Ranganathan, A., Chetan, S., Campbell, R.: Mobile polymorphic applications in ubiquitous computing environments. In: Mobiquitous 2004 : The First Annual International Conference on Mobile and Ubiquitous Systems:Networking and Services, Boston, Massachusetts, USA (2004)
21. Pitkranta, T., Riva, O., Toivonen, S.: Designing and implementing a system for the provision of proactive context-aware services. In: CAPS '05: Proceedings of the Workshop on Context Awareness for Proactive Systems. (2005) 21–30
22. IST Advisory Group (ISTAG): Software Technologies, Embedded Systems and Distributed Systems - A European Strategy Towards Ambient Intelligent Environment. Technical report, IST (2002) http://www.cordis.lu/ist/istag.html.
23. Issarny, V., Sacchetti, D., Tartanoglu, F., Sailhan, F., Chibout, R., Levy, N., Talamona, A.: Developing ambient intelligence systems: A solution based on web services. Automated Software Engineering **12**(1) (2005) 101–137
24. IBM, Microsoft Corporation and BEA: Business Process Execution Language for Web Service (BPEL4WS) v.1.0. Technical report, IBM, Microsoft Corporation, BEA (2002) http://www.ibm.com/developerworks/webservices/library/ws-bpel/.
25. Porcarelli, S., Castaldi, M., Giandomenico, F.D., Bondavalli, A., Inverardi, P.: An Approach to Manage Reconfiguration in Fault Tolerant Distributed Systems. In: Proceedings of the ICSE 2003 Workshop on Software Architectures for Dependable Systems. (2003) 71–76
26. Babaoglu, O., Marzullo, K.: Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms. In Mullender, S., ed.: Distributed Systems. Addison-Wesley (1993) 55–96
27. Zarras, A.: Applying Model Driven Architecture to Achieve Distribution Transparencies. Information and Software Technology **48**(7) (2006) 498–516
28. Berners-Lee, T., Hendler, J., Lassila, O.: The Semantic Web. In: Scientific American. (2001)
29. Mokhtar, S.B., Liu, J., Georgantas, N., Issarny, V.: Qos-aware dynamic service composition in ambient intelligence environments. In: ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering, New York, NY, USA, ACM Press (2005) 317–320