Service-Oriented Autonomic Multimodal Interaction in a Pervasive Environment

Pierre-Alain Avouac, Philippe Lalanda and Laurence Nigay
Université Joseph Fourier Grenoble 1
Laboratoire d'Informatique de Grenoble LIG UMR 5217, Grenoble, F-38041, France
{Pierre-Alain.Avouac, Philippe.Lalanda, Laurence.Nigay}@imag.fr

ABSTRACT

Heterogeneity and dynamicity of pervasive environments require the construction of flexible multimodal interfaces at run time. In this paper, we present how we use an autonomic approach to build and maintain adaptable input multimodal interfaces in smart building environments. We have developed an autonomic solution relying on partial interaction models specified by interaction designers and developers. The role of the autonomic manager is to build complete interaction techniques based on runtime conditions and in conformity with the predicted models. The sole purpose here is to combine and complete partial models in order to obtain an appropriate multimodal interface. We illustrate our autonomic solution by considering a running example based on an existing application and several input devices.

Categories and Subject Descriptors

D.2.2 **[Software Engineering]**: Design Tools and Techniques – User interfaces

General Terms

Algorithms

Keywords

Multimodal interaction, service-oriented components, autonomic computing, model-based engineering.

1. INTRODUCTION

The pervasive computing area has recently gained major importance from both industry and academia and is changing the way we interact with our environment [20, 16]. This computing domain emphasizes the use of small, intelligent and communicating everyday life objects to interact with the computing infrastructure. These devices tend to blend in their environment. This is especially true in homes and buildings where new electronic devices such as photo frames aim to be decorative as well as powerful in terms of interaction. This new equipment has the ability to communicate with other devices, to configure itself, and perform context-based cognitive and physical actions. The vision of coordinated or cooperating devices teaming up transparently to provide human beings with services of all sorts is actually getting closer and closer. However, the main part of the research effort has focused so far on providing hardware that can actually enable such interactions and on developing ad-hoc software solutions for predefined devices (e.g., controlling the home's lighting or the TV with a mobile phone). Consequently, plenty of electronic devices providing these kinds of features are already commercialized, whereas very few interesting applications take full advantage of this new infrastructure. Indeed, the complexity of building software that can actually benefit from these devices is often underestimated. Usual software engineering techniques and tools are not suitable. Indeed several software engineering challenges remain to be solved before fulfilling the vision of a true pervasive world. Notably the high degree of dynamism, distribution, heterogeneity and autonomy of the electronic devices involved raises important challenges. The envisioned environment is open to dynamic connections: devices may enter and leave the network spontaneously, providing context-dependent features (e.g. according to user's activity). It is also open to heterogeneous devices: protocols and devices' types differ according to application domains and service providers.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICMI'11, November 14–18, 2011, Alicante, Spain.
Copyright 2011 ACM 978-1-4503-0641-6/11/11...$10.00.

It is envisioned that, in the mid-term, devices and applications will disappear from the users' awareness. Users will then reason in terms of services and no longer in terms of concrete computing elements. At the same time, it is foreseeable that unimodal, well-identified interfaces will also disappear. Users will simply express their needs or desires with any available interaction modalities and the environment and its objects/devices will react accordingly. As defined by Oviatt [13]:

> "Multimodal interfaces process two or more combined user input modes in a coordinated manner with multimedia system output. They are a new class of interfaces that aim to recognize naturally occurring forms of human language and behavior, …"

Multimodal interfaces fit well in the pervasive landscape. They offer a more natural/efficient way for users to interact with device-stuffed environments by means of speech, gestures or other interaction modalities. They also leave the ability for humans to use a variety of modalities to interact with an application, depending on the context (e.g. availability and reliability of interaction devices, user's mood, etc.). Moreover multimodal interfaces have been demonstrated to offer better flexibility and reliability than WIMP interfaces (interfaces based on Windows, Icons, Menus, Pointing device) [14].

The dark side of multimodal interfaces in pervasive environments is their management complexity. Recent work has proposed component-based or agent-based frameworks to support the development of such interfaces [3, 17]. These approaches are based on the definition of an interaction modality as the coupling of an interaction device with an interaction language (i.e., a set of transformations of raw data from input devices). These approaches allowed a better separation of concerns, clearly distinguishing functional aspects like data fusion and non-functional aspects like communication and synchronization. Most of these approaches are however not flexible enough to deal with highly dynamic environments where interaction devices, applications, and the way multimodal interactions take place, are rapidly evolving. We thus believe that, one of the great challenges of multimodal interfaces in pervasive environments, is to build reliable and autonomic processing systems able to analyze and understand multiple communication means and reconfigure itself in real-time.

In this paper we present an autonomic framework for the development and runtime management of input multimodal interfaces in pervasive environments. Our discussion will concentrate on input (i.e., from the user to the system) although our model holds for output as well. Nevertheless we did not test our approach for output so far. This framework leverages recent advances in service-oriented components [15, 17] and in model-based engineering. The paper is organized as follows. First we recall the key characteristics of pervasive multimodal interfaces. We then present our framework by describing the execution machine, the multimodal process and the autonomic manager. The autonomic manager is driven by partial models to manage multimodal processes: we present the two types of models that we defined and in particular the interaction model. We conclude with a simple example that illustrates how the autonomic manager works.

2. PERVASIVE MULTIMODALITY

Pervasive computing systems typically consist of multiple devices and software entities that are capable of interacting with one another. Various types of software-equipped devices may be available for various purposes, such as for interaction with the real environment (i.e., interaction output/input device), providing display and control services to users, or for exposing data and application interfaces to other devices. The main challenge of the pervasive computing domain is to provide coherent pervasive environments, offering useful applications and services, based on an entanglement of heterogeneous, distributed and dynamic devices and software services, communicating via various technologies and protocols. In this context, several characteristics specific to pervasive devices make this domain appealing from a business and customer perspective, while raising difficult problems for system development, runtime interactions and management practices. Such device properties include:

• **Distribution**. Devices are typically scattered across the physical environment and are accessible via various communication protocols, generally over a wireless communication support.

• **Heterogeneity**. A vast range of devices, software technologies and communication protocols are currently available for the pervasive computing domain. A common consensus on uniform and compatible implementations is not presently foreseen: today, more than fifty candidate protocols, working groups and standard specifications for home networking already exist for providing communication and interoperation between access and indoor networks (e.g., ZigBee, HomePlug).

• **Plural authority**. Devices present in a pervasive environment generally belong to different vendors. In addition, applications deployed and run on such devices may be provided by yet different vendors and may require cooperation with other services and/or devices. In this context, it is foreseeable that equipment vendors and service providers will prefer to maintain a certain control over their devices and software and may consequently limit access requests from external entities.

• **Dynamism**. Availability of devices is by far the most volatile in pervasive systems with respect to other computing system types. This is due to several facts, including: i) users may freely and frequently change their locations and hence the locations of the devices they carry; ii) users may voluntarily activate and deactivate devices, or devices may unexpectedly run out of battery; iii) users and providers may periodically update deployed software services.

In addition to device and software dynamism, pervasive systems are constantly confronted with the evolution of their execution contexts. This may include modifications in the user's current behavior, location, mood, or general routine, as well as changes in other software applications' availability and behaviors. Another important characteristic of pervasive environments is the generally limited resource availability on the physical execution platforms involved. Typically, in the home context, software applications will run on a small gateway, with little memory space and low processing capabilities. Finally, an essential requirement for any successful pervasive system is the simplicity of operating and managing the system, by users not necessarily knowledgeable in the computer science domain. Furthermore, privacy, security and safety concerns represent major challenges for the pervasive computing community.

Technically speaking, pervasive computing is today much influenced by recent advances in service-oriented computing. Service-oriented Computing (SOC) has recently appeared in the software engineering landscape. The very purpose of this reuse-based approach is to build applications through the late composition of independent software elements, called services. Many smart devices are today exposed as services. More precisely, their capabilities are described and dynamically published by service providers; at runtime they are chosen and invoked by service consumers. This is achieved within a service-oriented architecture (SOA), providing the supporting mechanisms. Service orientation brings major software qualities. It promotes weak coupling between consumers and providers, reducing dependencies among composition units. Late binding and substitutability improve adaptability. Since a service can be chosen or replaced at runtime, it is possible to optimize the way requirements are met. A number of implementations have been proposed in the last years. Web Services (www.w3c.org), for instance, represent a solution of choice to expose software applications on the Web. UPnP (www.upnp.org) and DPWS (Devices Profile for Web Services) are heavily used in order to implement volatile devices. OSGI (www.osgi.org) and iPOJO (www.ipojo.org) provide advanced dynamic features that are exploited to build component-based pervasive applications.

Service orientation brings in solutions to deal with environment dynamicity. It also allows the development of more flexible software, regarding the core functions (the functional core component of the Arch software architectural model [1]) or the interaction functions (the interaction components of the Arch model [1]). However, these solutions are still complex, hard to manage and heterogeneous. Devices and applications still belong to different authorities and there is no real hope to see reifying standards in the short term.

The presented characteristics and desired properties of pervasive environments and applications must be taken into consideration when building and administering pervasive multimodal interfaces. Most importantly, the dynamicity property and requirement must be constantly dealt with by adapting pervasive systems to internal and external modifications. Such modifications may include changes in a system's constituent services and execution platform, or into its constantly evolving physical and social context (e.g. availability of resources, or user's location, habits and preferences). In the pervasive computing context, system adaptation should remain largely transparent to users, while seamlessly coping with resource heterogeneity, distribution and plural authority. As such, necessary management operations should require minimum human intervention, while meeting specific performance and dependability constraints.

Multimodal interfaces should then be designed to adapt easily to different contexts, user's profiles and application needs. Adaptability is always a challenging requirement. It requires first to prepare adaptation points in the code, which can be at different levels of abstraction (parameter, function, object, component, etc.). It also requires the definition of a language to specify the possible runtime adaptations. Finally, it requires some intelligence to decide when, how and where in the code to adapt multimodal interfaces. In pervasive multimodal interfaces, this intelligence cannot be entirely provided by end-users who are not supposed to play a heavy administrative role. Adaptations have to be undertaken by the system itself, in conformance with the users' current goals. As a consequence, the system has to be able to deal with a great range of computing elements, being input interaction devices or applications, with little help from users.

3. APPROACH: DYNAMO

The autonomic computing initiative [9] aims to limit the need for human intervention in computer management processes by enriching software applications with self-management capabilities, such as self-building, self-optimization, self-configuration, self-repair and self-protection. Conforming to this approach, autonomic management abilities would enable software applications to seamlessly and transparently self-adapt to their changing environments and evolving business goals. Therefore, autonomic computing seems to provide a viable solution to the difficult concerns and constraints specific to the pervasive multimodal interfaces.

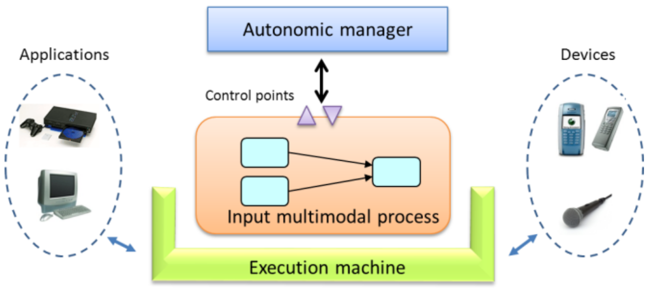

Figure 1. Overall approach

In that spirit, we have developed a complete suite for the development, execution and management of multimodal processing, called "*DynaMo*" (*Dynamic multimodality*). The management is autonomic in the sense that the whole multimodal processing system is generated and maintained at runtime by an autonomous manager. The processing system is modular. It is made of service-oriented components in interaction.

As illustrated by figure 1, our purpose is to clearly separate the multimodal processes, the volatile input interaction devices joining and leaving the environment, and the applications that may also appear or disappear dynamically. The schema highlights the main architectural elements of DynaMo, which are:

- The execution machine supporting multimodal processes. The purpose of this machine is to provide a flexible, context aware runtime environment. It is based on iPOJO, a dynamic service-oriented component framework built on top of OSGi. The goal of this machine is also to monitor the environment in order to trace any computing evolution.

- The multimodal process itself. Input interaction modalities are developed and executed with a dedicated component model. The purpose of this domain-specific component model is to provide the right level of abstraction for developers and maintainers of multimodal interfaces. Low-level technical aspects like synchronization are hidden away by the model.

- An autonomic manager whose purpose is to build and manage the multimodal interfaces. To make its decisions, it uses contextual information provided by the execution machine. It builds multimodal interfaces through the composition of pre-defined components conforming to the component model presented above.

These three architectural elements of DynaMo are presented in detail in the next subsections.

3.1 Execution machine

The purpose of the execution platform is twofold. First, it is a supporting infrastructure for the execution of dynamic, component based applications. Then, it also monitors the environment and integrates the discovered entities seamlessly within the running applications. In particular, the execution machine captures services arrival and departure. Services expose both input interaction devices and applications.

As illustrated by figure 2, the execution machine is built on top of OSGi and iPOJO, the Apache service-oriented component model, and ROSE, available on ObjectWeb [2]. ROSE is an OSGi-based open source middleware dealing with distribution. It includes communication drivers for different technologies (Web Service, DPWS, UPnP) and is able to trace the availability of services. ROSE builds and maintains an advanced service registry. Service registries play an important role in service-oriented architectures, allowing late-binding and weak-coupling. It is a means for providers to publish their services and consumers to be aware of available services and, possibly, select one of them for execution. Each service technology has a particular standard for the registry. Web Services, for instance, are described on WSDL files and published in UDDI compliant registries. In our architecture, we have introduced an advanced registry, which supports many service technologies and a detailed description of services including functional and non-functional properties. The very purpose of this advanced registry, of course, is to support the dynamic selection of services.

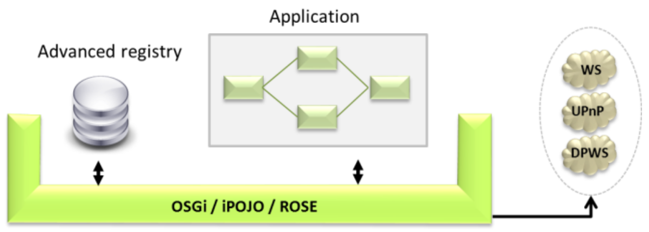

Figure 2. Execution machine

Finally, the principle of this advanced registry is to have a global view of the runtime environment. The registry is updated regularly by the work engine and ROSE according to the arrival and/or the departure of the services. More precisely, it is implemented by a set of registries enhanced with service runtime properties. A registry contains the information for a particular technology. This advanced registry is defined in the model-driven engineering articles as a runtime model [6]. The main advantages of using a runtime model are 1) to store only the relevant information required for selection 2) to hide the technological characteristics of concrete services and 3) to hide the dynamic context with arrival and departure of services. In addition, since the runtime model is modular, with a clear separation between functional and security concerns, it can be extended. In fact, we could support new non-functional properties such as quality of service with non-functional runtime extensions linked to the functional part of the global runtime model.

OSGI, iPOJO, and ROSE are all heavily used and validated in industrial applications. The execution machine integrating these technologies is very robust and used by our industrial partners (Schneider Electric and France Telecom in particular).

3.2 A domain-specific component model

The activity of integrating disparate information sources in a timely fashion is known under the name of mediation. Mediation has been historically used to integrate data stored in IT resources [12]. Recent work has been using mediation to allow interoperation between applications and services [7]. Service mediation implements all the operations that are necessary to enable the actual communication within service-based applications. The most common functions to be provided are:

• Communication. The primary purpose of mediation is to enable applications and devices using different communication protocols to interoperate. This is implemented by means of protocol transformations as in a network bridge. This function can also play the role of a broker, hiding for instance the applications network addresses.

• Synchronization. Time is a major aspect in mediation. Data shared by different elements have to be time-stamped, organized, synchronized.

• Syntactic alignment. The purpose of this function is to align data formats. This can be done between each application or through an intermediary format. In the latter case, the number of data transformations to be made is obviously reduced.

• Semantic alignment. The purpose of this function is to align data semantics. In the absence of recognized and used standards, applications develop different ontologies to represent (static and dynamic) knowledge.

• Non-functional properties. The purpose of this function is to ensure certain quality properties in the application exchanges, as for instance security or availability.

These functions are obviously at the heart of multimodal processes. As a matter of fact, developing a multimodal interface requires dealing with communication, synchronization, alignment and non-functional aspects.

Encapsulating mediation operations in dedicated software is clearly a good practice. Indeed mediation software provides a single point of interface to the different applications implied in the communication. This reduces the number of connections needed and facilitates change management. Mediation also provides an isolation layer from software details and, if appropriately configurable, permits the quick and cost-effective development of new applications. The mediator layer improves reusability and evolution of applications. It also permits the transparent addition of new QoS properties such as security and reliability.

Modern mediation frameworks are modular, mostly based on component-based engineering. They however lack flexibility. Adaptation in the mediation process generally requires stopping and restarting the process, which is hardly acceptable in pervasive environments. In order to deal with dynamic mediation as needed for multimodal processes, we have developed a domain-specific component model called Cilia. A mediation process in Cilia is a set of components interacting in a loosely coupled way through, but not limited to, event-based protocols. As with any component-based model, Cilia relies on two main models, the specification model and the composition model. The specification model is used to define components. The composition model defines the way components are combined. Components are specified at development time. They are made of three java classes and an XML-based specification. More precisely, a component includes the following Java classes (figure 3):

• A scheduler class. The purpose of this constituent is to synchronize data reception. It intercepts incoming data, store them and initiates their processing. The processing decision can be time-based, content-based or, any other condition in relation with the mediation context.

• A processor class. The processor performs the mediation algorithm per se. When notified by the scheduler, it processes the collected data and passes them to the dispatcher.

• A dispatcher class. The dispatcher receives the processed data from the processor. This constituent decides on the data destination and triggers their delivery. The dispatcher choice is a logical destination because of loosely coupled relations between mediators.

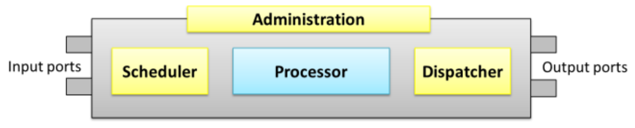

Figure 3. Component model

Developers of a mediation process concentrate on the processor class where they express the way data should be processed. The scheduler and dispatcher can also be entirely specified but are generally simply reused. Most frequent operations like periodic consumption are provided. Moreover we defined generic mediation processes that include fusion algorithms as defined in multimodal interfaces. Such mediation processes represent high-level reusable abstractions. They correspond to generic composition operations, defining elementary temporal fusion operations, independent from devices, modalities and application tasks. To define such mediation composition operations we draw on the CARE properties [19]. The CARE properties were proposed as a simple way of characterizing and assessing aspects of multimodal interaction: the Complementarity, Assignment, Redundancy, and Equivalence that may occur between the interaction modalities available in a multimodal interface. We define mediation operations related to Redundancy and Complementarity of CARE.

Mediation components are connected through typed ports. Bindings are also defined at development time. A binding specification describes how communication is established. Binding specifications are independent of mediators logic, thus mediators could use any binding specification, essentially event-based and RPC based. Figure 4 provides a simplified example where three mediation components are used to allow a multimodal interaction with an embedded application. Each mediation component is composed of three parts as defined in figure 3. The middle parts of the mediation components correspond to alignment and fusion code provided by the interface developers. Such an example could correspond to the combination of speech commands with gestures on a tactile surface to control a multimedia player: one complementary mediation component is used to combine speech and gesture events processed by two mediation components.

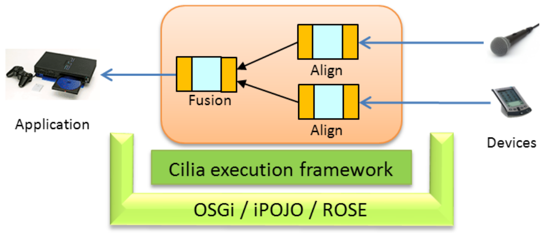

Figure 4. Example of mediation chain

Figure 5 presents the Cilia stack, detailing the computing responsibilities of each layer. Cilia has been developed in iPOJO (more precisely, a component specification is transformed into a set of iPOJO components). Cilia is fully adaptable: it provides interfaces allowing us to add, remove, replace, configure a mediation component at runtime. To do so, the Cilia execution framework maintains internal states and deals with components quiescence. This means that, when a component is replaced, the new component gets the state of the replaced one (i.e., the data to be processed).

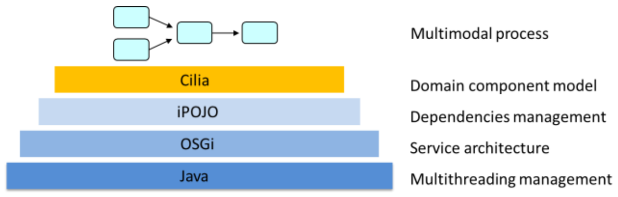

Figure 5. Cilia execution stack

Regarding quiescence, computing threads are controlled at a fine grain: a component can be changed or replaced only when there is no running client (proxies are used to intercept calls to a component to be replaced).

3.3 Autonomic manager

The runtime management of a pervasive application requires the use of some sort of autonomic capabilities. Obviously, an application cannot be dynamically updated by a user who is supposed to be unaware of the surrounding infrastructure.

Autonomic systems are usually structured according to a simple reference architectural model introduced by IBM [10]. This reference architecture clearly defines two distinct entities: the managed artifacts and an autonomic manager. Managed artifacts are the software entities that are automatically administered in an autonomic element. The autonomic manager is the module in charge with the run time administration of the managed artifacts. Managed artifacts provide specific interfaces, called control points or touch points, for monitoring and adaptation.

Here the managed artifacts are clearly the mediation chains realizing the multimodal interactions. The purpose of the autonomic manager is to create and adapt the multimodal interactions, using the dynamic capabilities of the underlying component model (Cilia). It is driven in its decisions by high level goals set by the users (or by an initial administrator). A user can choose an interaction policy at runtime. For now, a user can choose between two predefined interaction policies: *simple* and *bind-all*. If the *simple policy* is selected, the autonomic manager tries to connect each task of the application to sensors of the devices. With the *bind-all* policy, the autonomic manager tries to connect all available sensors to tasks of the application. This policy likely implies that different modalities are defined for a given task (i.e., equivalence of modalities for a given task as defined by the CARE properties). The selection of the interaction policies is made through the DynaMo-settings graphical interface.

The autonomic manager is reactive: when a modification occurs in the environment, it computes a new mediation chain or adapts the current one. Similarly, if the user changes the interaction policy, the mediation chain is changed or adapted. Adaptations can be done globally, that is at the mediation chain, or at the component level. Impacts of a modification are actually well delimited from an architectural point of view. On the one hand, the multimodal process is decoupled from the autonomic manager and, on the other hand, devices and applications are simply not aware of the mere existence of the multimodal processes.

From the users' point of view, the impacts of modification are obviously more important. First, the time needed to completely change an interaction is about 3 seconds (for up to 15 mediation components). It is less than 1 second when a single component is removed or replaced. The biggest impact is obviously due to the change in the way interactions are conducted. A new device is used or the same device is used differently. Of course, the adaptation is supposed to better serve the user and her/his interaction preferences.

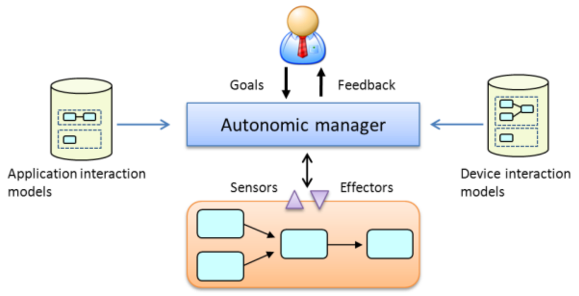

Figure 6. Model-driven autonomic management

The autonomic manager contains the domain-specific knowledge needed to appropriately create and update mediation chains. In most autonomic systems, this knowledge is encoded in rules and thus is not really explicit. Recently, several studies have used explicit models in order to specify the autonomic manager knowledge or constraints [8]. In particular, architectural models are used as a basis for system construction and update. The purpose of the autonomic manager is here to maintain consistency between a target architectural structure, generally made of collaborating components, and the actual structure of a running system. Models make explicit global and local properties. It is possible to use more generic autonomic managers that are easier to understand and maintain.

This latter approach is limited to domains where clear reference architecture can be defined. This is usually not the case for multimodal interfaces in very dynamic settings: we cannot define a reference architectural model of the mediation chain for multimodal process. Based on the architectural work in the Arch model [1], we have defined an alternative approach where the autonomic manager manipulates incomplete models, named interaction models. These interaction models are specific to devices and applications. The purpose of the autonomic manager is then, depending on the runtime context, to relate the interaction models of the devices and applications in order to define the architecture of the mediation chain. These interaction models are presented in detail in the following section.

We conclude the presentation of the autonomic manager by considering the users' involvement in the whole process. Users are involved at different stages. First, they select the high level goals (i.e., interaction policy) driving the autonomic manager. Users also decide on the applications to be controlled through multimodal interaction and, somehow, on the available devices (turning them on or off for instance). Moreover a major aspect of autonomic computing is the feedback provided by the system to the administrators or the users. In our case, appearance and disappearance of a service (i.e., an application or a device) are notified to the user by a pop-up window. Also, semantic alignment allows a device dedicated to a task to be bound to that task, if this task is present in the controlled application. The main limit so far concerns observability by the user of an interaction modality. We provide partial observability of the available modalities by graphically displaying which sensors of the devices are connected to the tasks of the current active application. Although this representation is updated each time a change occurs in the multimodal process, it is only useful for simple cases (e.g., a press button of a device to trigger a simple action of the application). For complex cases with complementary modalities and equivalent modalities for a given task, the representation is too partial to be useful. Information from the mediation chain is accessible and further work must be done to define the way to present the interaction modalities. We focused so far on designing and developing our DynaMo framework, but DynaMo clearly defines a good candidate platform for studying and experimenting this aspect of dynamic multimodal interfaces.

4. MODELS

As previously indicated, the autonomic manager is driven by partial models to manage multimodal processes. These models store and make explicit most of the information necessary to generate interactions. Two kinds of models are defined: proxy models and interaction models. This separation has been done in order to target the two different stakeholders: developers and interaction designers. Indeed for developing a multimodal application with DynaMo, developers first create proxies (in Java) for the applications and the interaction devices as well as their proxy models. Interaction designers can then define interaction models for these applications and devices. Interaction models are expressed in an xml language and we provide a graphical editor to define such interaction models.

Proxy models defined by developers are attached to applications and devices. A proxy model contains information about the process that is used by the discovery manager in the execution machine (see section 3.1). From this information, the discovery manager is able to track the corresponding applications or devices and start its corresponding proxy. The model also contains information about the protocols and ports to be used to communicate with other devices and about provided data types. Based on this information, the autonomic manager can connect the endpoints of a mediation chain to the proxies. If a data type is a number (float or integer), an interval has to be provided. An interval is composed of a lower bound and an upper bound. This information enables the autonomic manager to adapt intervals at runtime by inserting an adaptor between incompatible intervals. For example, if a device provides numbers in the interval [-180, +180], and a connected application task handles numbers in the interval [0, +100], the automatically inserted adaptor does a linear transformation for each value between these two intervals. Figure 7 shows the meta-model of the proxy models.

Interaction designers, now, specify one or several interaction models for each proxy. Such an interaction model defines the way an application or a device can be used from an interaction point of view. Because such a model relates to a single proxy, it only describes a partial interaction that has to be completed by the autonomic manager. Interaction models contain information about data semantics, data processing and data path. Semantics-related knowledge is important for the autonomic manager in order to go beyond type alignments. Through data processing, an interaction designer is able to enhance an interaction by adding tasks to an application, synchronizing data and so on. For example, if a media player application proposes a task to control sound volume through a number, then the interaction designer can add a task that mutes the volume. A data path is a series of functions that data will pass through. Figure 7 shows the meta-model of the interaction models.

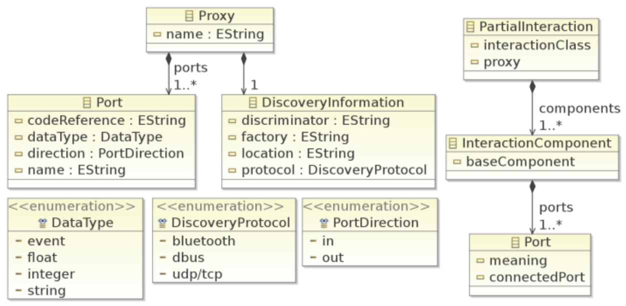

Figure 7. Proxy (left) and interaction (right) meta-models

It is to be noted that defining semantics without any guidance does not really make sense because the autonomic manager needs to match meanings defined in the different interaction models. Several interaction classes have been predefined. An interaction class defines several meanings that make sense together. An interaction model references one interaction class, so only the meanings of this class can be attached to data of this model. For instance in the example of the following section, we use the interaction class named MediaPlayer that defines meanings including *Pause* and *Mute* (figure 10) and the interaction class GamePad including the meanings *Up* and *Down*.

In order to ease data processing definition, a predefined library of processing functions is provided. The interaction designer declares which function has to be used and provides a configuration for the function. For example, a triggering function sends an event as soon as it receives a value greater than a configured maximum value. Moreover generic composition functions (e.g., temporal fusion) based on the CARE properties [19] are provided. These functions are specific to the interaction domain. They have been crafted with reuse concerns in mind. The functions are implemented by components. Thus, the interaction designer specifies a partial interaction by declaring which base components to use, configuring them and binding their ports together in the graphical editor. At this stage of the specification, data types can generally be ignored because the autonomic manager will be able at runtime to infer each port data type. This inference leads to a completion of the component configuration, and adds a data type converting component if necessary.

Specifying a partial interaction by assembling and configuring several domain-specific components facilitates the work of the interaction designer. This approach maximizes reusability by providing generic components, decreases technical difficulties by hiding implementation details. These models are conformed through instantiation to their corresponding meta-model. The autonomic manager relies on a general meta-model that integrates these two meta-models and notably includes relation between proxies, partial interactions and component library.

5. ILLUSTRATIVE EXAMPLE

The following example shows how the autonomic manager deals with two devices and two applications. The first application is a media player software, named *VLC* (www.videolan.org/vlc). The second application is a sudoku game (KSudoku, http://games.kde.org/game.php?game=ksudoku). *VLC* and *KSudoku* can receive commands through an inter-process communication system, named *D-Bus* (dbus.freedesktop.org). The first device is a Blu-ray remote control, namely *BD Remote Control* (or *BDRC*). The second device is a controller for a video game console, namely *Wii Remote* (or *Wiimote*). The two devices use the *Bluetooth* protocol to send data. Figure 8 shows an excerpt of the proxy models of *VLC* and the two devices.

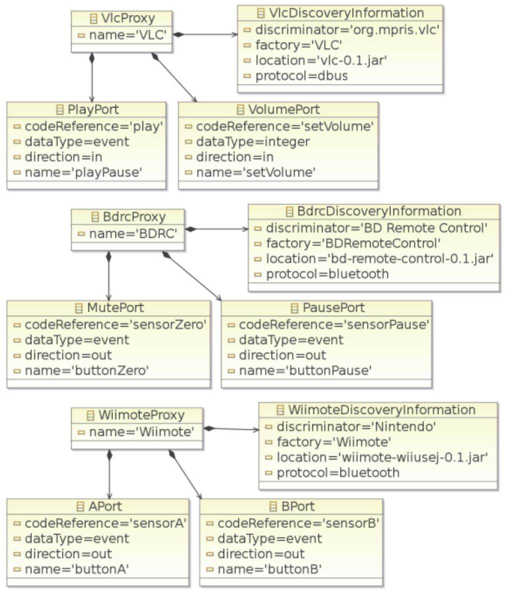

Figure 8. Excerpt of the proxy models

The example follows this scenario: "Alice has previously used DynaMo. She wants to watch a movie. She starts *VLC* and activates her *BDRC*. Bob comes and talks to Alice, so she pauses the movie. Later, when the *BDRC* runs out of power, she activates her *Wiimote*. Finally, she found the movie boring, so when Bob comes to ask something, she just mutes the sound volume to answer. She then finally decides to play Sudoku, with the *Wiimote* that she is holding".

From the autonomic manager point of view, it receives a discovery notification about *VLC*, hence it downloads the *VLC* binary proxy from the repository and starts it. Since no device is discovered, no mediation chain is generated. Then, it receives a discovery notification about the *BDRC*. It starts the *BDRC* proxy. Now, a mediation chain can be generated. Amongst the interaction models of *BDRC* and *VLC*, it selects the two that use the same interaction class, namely the *MediaPlayer* interaction class. It instantiates the components declared in the interaction models, and binds mediators of each interaction if their semantics match. When Alice pushes the pause button, a data is sent by the *BDRC*. The proxy gets the data and passes an event to a mediator. The event follows a path through the mediation chain and is received by the pause port of the *VLC* proxy. The proxy calls the pause task on *VLC*. As soon as the autonomic manager is notified of the *Wiimote* discovery, it starts the proxy. No interaction class of *VLC* and *Wiimote* interaction models matches each other. The autonomic manager generates the same mediation chain plus a part that connects the *Wimote* proxy to the *VLC* proxy. This new part is created only from information about data type, because it does not have any semantic information. Finally, when the *BDRC* runs out of energy, the autonomic manager is notified, and generates the same mediation chain without the *BDRC* part. Finally when Alice starts the *KSudoku* game, the corresponding binary proxy is started. Since both *KSudoku* and *Wiimote* have an interaction model of the same class, namely *GamePad*, the manager will instantiate the components declared in the interaction models of the *GamePad* class. For the Wiimote, the *GamePad* interaction model declares in particular that the accelerometer coupled with a gesture recognition has the meanings *up* and *down* of the *GamePad* class. Alice can then perfom vertical movements with the Wiimote to change the selected cell in the Sudoku.

Figure 9 presents excerpts of partial interaction models for *VLC* and *BDRC*. The matching interaction class is *MediaPlayer*. Its meaning set contains *pause* and *mute*. Since only components can be declared in the interaction models, attaching a meaning is done by declaring an *identity* component and by attaching a meaning to a port of the component. Since same meanings are employed in each model, the autonomic manager can directly bind these ports. The same approach is applied for *KSudoku* and *Wiimote*: the matching interaction class is *GamePad* and its meaning set contains *up* and *down*.

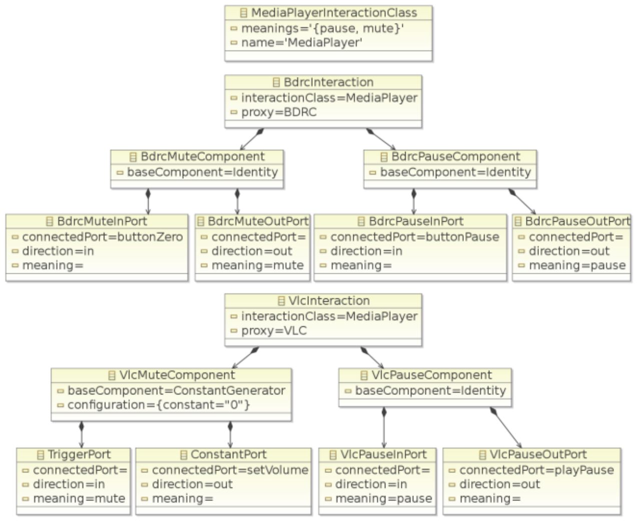

Figure 9. Excerpt of the partial interaction models

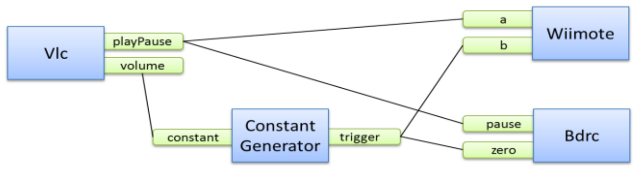

Figure 10. Excerpt of the generated mediation chain

The generated mediation chain for *VLC* is shown in figure 10. The identity components declared in the interaction models are not apparent in the mediation chain, since the *identity* component does not modify data that pass through it. They are removed at the end of the generation process. The *Wiimote* proxy model does not have an interaction model that uses the *MediaPlayer* interaction class. This lack of information results in random bindings between *Wiimote* proxy ports and *VLC* ports. Of course, the autonomic manager verifies the data type compatibility. Moreover, since the interaction policy is set to *simple,* it distributes the bindings between application tasks to prevent that a single task is bound to all sensors. With a policy set to *bind-all*, all the *Wiimote* sensors would be bound to the tasks: for instance two buttons could command the same task.

6. CONCLUSION

Pervasive computing implies dynamic and heterogeneous environments. Multimodal interfaces fit well in this pervasive landscape. In particular dynamic multimodality allows the users to use whatever devices to engage an interaction, depending on the context. In this paper we presented the autonomic DynaMo framework for the development and runtime management of pervasive multimodal interfaces. Our contribution is dedicated to software engineering of dynamic multimodal interfaces by providing a robust and extensible framework based on service-oriented and autonomic computing. Indeed the DynaMo architecture relies on three main parts: a domain-specific component model, an autonomic manager, and an execution machine. The component model enables developers and interaction designers to express their knowledge. The autonomic manager used these models to generate and maintain a multimodal interaction. The multimodal interaction data-flow from input devices to an interactive application is managed by a generated mediation chain. The execution machine is able to effectively realize the interaction provided by the autonomic manager.

DynaMo adopts a global approach for developing multimodal interfaces: On the one hand, DynaMo supports the cases where multimodal adaptable interaction is completely defined at design time as with the existing toolkits and platforms HephaisTK [4] ICARE [3], OpenInterface [17] and Squidy [11]. In these cases, the interaction models are complete and the autonomic manager does not have to complete partial interaction models. For instance using DynaMo we have developed the multimodal map navigator described in [18] for illustrating OpenInterface. On the other hand, DynaMo also defines an autonomic solution when partial models are defined and multimodal interaction is not fully defined at design time. The role of the autonomic manager is to build complete input multimodal interfaces based on runtime conditions and in conformance with the predicted partial interaction models.

The DynaMo framework being fully operational, as further work, we will first perform experimental evaluations with users. Such experiments will enable us to enrich the autonomic manager by identifying new policies. For example, the adaptation is currently realized without learning from the users' inputs. However, several cases would obviously leverage the learning. For example, if a button is never used by a user, DynaMo could propose to bind this button to another function. The usage of an autonomic architecture will ease the machine learning process because sensing and actuating are already done. Moreover based on our framework we will study an important aspect of dynamic multimodal interfaces that is how to make observable by the users the performed changes in multimodal interaction. Our general research direction is to make the autonomic manager observable and controllable by the users by defining different levels for tuning the autonomic capacity of the framework and therefore make the user in control of her/his pervasive environments.

7. REFERENCE

[1] 1992. A metamodel for the runtime architecture of an interactive system: the UIMS tool developers workshop. *SIGCHI Bull.* 24, 1 (Jan. 1992), 32-37.

[2] Bardin, J., Lalanda, P., Escoffier, C. 2010. Towards an Automatic Integration of Heterogeneous Services and Devices. *Proc.* of *APSCC'10*. IEEE, 171-178.

[3] Bouchet, J., Nigay, L., Ganille, T. 2004. ICARE software components for rapidly developing multimodal interfaces. *Proc. of ICMI'04*. ACM, 251-258.

[4] Dumas, B., Lalanne, D., Ingold R. 2009. HephaisTK: A Toolkit for Rapid Prototyping of Multimodal Interfaces. *Proc. of ICMI-MLMI 2009*. ACM 231-232.

[5] Escoffier, C., Hall, R. S., Lalanda, P. 2007. iPOJO: an Extensible Service-Oriented Component Framework. *Proc. of SCC'07*. IEEE, 474-481.

[6] France, R., Rumpe, B. 2007. Model-driven Development of Complex Software: A Research Roadmap. *Proc. of FOSE'07*. IEEE, 37-54.

[7] Garcia, I., Pedraza, G., Debbabi, B., Lalanda, P., Hamon, C. 2010. Towards a service mediation framework for dynamic applications. *Proc. of APSCC'10*. IEEE, 3-10.

[8] Garlan, D., Cheng, S. W., Huang, A. C., Schmerl, B., Steenkiste, P. 2004. Rainbow: Architecture-Based Self Adaptation with Reusable Infrastructure. *IEEE Computer* 37, 10 (Oct. 2004), 46-54.

[9] Huebscher, M. C., McCann, J. A. 2008. A survey of autonomic computing. *ACM Comput. Surv.* 40, 3, Article 7 (Aug. 2008).

[10] IBM Corporation. An Architectural Blueprint for Autonomic Computing. http://www-03.ibm.com/autonomic/pdfs/AC%20Blueprint%20White%20Paper%20V7.pdf

[11] König, W. A., Rädle, R., Reiterer, H. 2009. Squidy: a zoomable design environment for natural user interfaces. *Proc. of CHI EA'09*. ACM, 4561-4566.

[12] Lalanda, P., Bellissard, L., Balter, R. 2006. Asynchronous Mediation for Integrating Business and Operational Processes. *IEEE Internet Computing* 10, 1 (Jan. 2006), 56-64.

[13] Oviatt, S. 2003. Advances in Robust Multimodal Interface Design. *IEEE Comput. Graph. Appl.* 23, 5 (Sept. 2003), 62-68.

[14] Oviatt, S. 2007. Multimodal interfaces. *Human-Computer Interaction Handbook: Fundamentals, Evolving Technologies, and Emerging Applications*, 2nd edition. L. Erlbaum Assoc. Inc., Chap. 14, 286-304.

[15] Papazoglou, M. P., Georgakopoulos, D. 2003. Service-Oriented Computing: Introduction. *Commun. ACM* 46, 10 (Oct. 2003), 24-28.

[16] Satyanarayanan, M. 2001. Pervasive computing: vision and challenges. *IEEE Personal Communications*, 8, 4 (Aug. 2001), 10-17.

[17] Serrano, M., et al.. 2008. The openinterface framework: a tool for multimodal interaction. *Proc. of CHI EA'08*. ACM, 3501-3506. www.oi-project.org

[18] Serrano, M., Juras, D., Nigay, L. 2008. A three-dimensional characterization space of software components for rapidly developing multimodal interfaces. *Proc.of ICMI'08*. ACM, 149-156.

[19] Serrano, M., Nigay, L. 2009. Temporal aspects of CARE-based multimodal fusion. *Proc. of ICMI-MLMI'09*. ACM, 177-184.

[20] Weiser, M. 1991. The computer for the 21st century. *Scientific American*, 265, 3 (Sept. 1991), 66-75.