Autonomic Pervasive Applications Driven by Abstract Specifications

Ozan Gunalp, Levent Gurgen, Vincent Lestideau, Philippe Lalanda

Autonomic Pervasive Applications Driven by Abstract Specifications

Ozan Günalp
CEA-LETI
Minatec Campus
Grenoble France
ozan.gunalp@cea.fr

Levent Gürgen
CEA-LETI
Minatec Campus
Grenoble France
levent.gurgen@cea.fr

Vincent Lestideau
Grenoble University
220, rue de la Chimie
38041 Grenoble France
vincent.lestideau@imag.fr

Philippe Lalanda
Grenoble University
220, rue de la Chimie
38041 Grenoble France
philippe.lalanda@imag.fr

ABSTRACT

Pervasive application architectures present stringent requirements that make their development especially hard. In particular, they need to be flexible in order to cope with dynamism in different forms (e.g. service and data providers and consumers). The current trend to build applications out of remote services makes the availability of constituent application components inherently dynamic. Developers can no longer assume that applications are static after development or at run time. Unfortunately, developing applications that are able to cope with dynamism is very complex. Existing development approaches do not provide explicit support for managing dynamism. In this paper we describe Rondo, a tool suite for designing pervasive applications. More specifically, we present our propositions in pervasive application specification, which borrows concepts from service-oriented component assembly, model-driven engineering (MDE) and continuous deployment, resulting in a more flexible approach than traditional application definitions. Then the capabilities of our application model are demonstrated with an example application scenario designed using our approach.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures – Domain-specific architectures, D.2.13 [**Software Engineering**]: Reusable Software – Domain engineering, Reuse models.

General Terms

Management, Design, Experimentation, Languages.

Keywords

Pervasive Computing, Internet of Things, Service-oriented Computing, Autonomic Computing

1. INTRODUCTION

Pervasive computing aims to remove the barrier between users and computing systems by blending the computers into the users' environment [14]. This vision is becoming possible in the near future thanks to recent evolution in mobile, wireless and sensor technologies. Computing and communication power of these devices continuously increase, allowing them to run execution platforms (e.g. mobile phones, embedded and tiny operating systems on sensor nodes) and communicate wirelessly with low cost and specialized protocols, e.g. IEEE 802.15.4, 6LoWPAN, RPL, CoAP. This recent evolution now allows devices to be (re-) used in various application domains such as environmental monitoring, surveillance, city infrastructures or home/office automation are being imaginable, providing thus user-centric applications integrating seamlessly with physical environments.

The development of pervasive applications is a difficult challenge because the developer needs to manage contextual changes, device and application dynamism, and business logic. As a consequence, current pervasive applications are generally insufficient in terms of software engineering: they are difficult to design, code, test and maintain; most existing solutions are proprietary, limited in terms of provided services and executed in a closed world.

In order to ease the design, development and management of applications, a widely used approach is to delegate certain common aspects (e.g. distribution, resource access, security) to an execution platform, also called "a middleware" [8]. Middleware forms an abstraction layer for applications to be built using these common services and let them focus mainly on applicative concerns. As a result, application designs and development models depend strictly on the capabilities of the middleware they are running on. Middleware designs in pervasive computing are today very much influenced by service-oriented computing principles. Service-Oriented Computing (SOC) is a programming paradigm, which aims to promote loose coupling between components using services [11]. As so, applications can be built upon loosely coupled service providers and consumers based on these contracts. Late binding and sustainability becomes possible opening the way of dynamically adaptable software architectures. This way, service-oriented middleware is able to provide run-time support for making changes in software architectures.

However, adapting the software architecture, notably an application, requires more than just APIs for making the change but also decision-making systems. Autonomic Computing [9] offers such mechanisms to monitor and manage systems and more recently there are many works [13] in self-adaptive software systems that apply these principles to software architectures. Knowledge about the managed system is thus very important in order to make decisions about adapting software architectures. Models@run.time focuses on using MDE techniques to represent information about runtime phenomena [3, 16]. Pervasive applications should leverage this information alongside with dynamic capabilities of underlying middleware in order to adapt to their execution context and continue providing value added services.

Self-IoT'12, September 17, 2012, San Jose, CA, USA
Copyright © 2012 ACM 978-1-4503-1753-5... $15.00.

Traditional application definitions fall short on responding to challenges brought by pervasive computing. This paper presents the ongoing work for creating a specification model for pervasive applications and corresponding execution environment. For this, we propose a tool suite, Rondo, which enables design, development and execution of dynamic pervasive applications using runtime models and autonomic computing principles. In the section 2, a background on pervasive applications and their requirements is given. Section 3 presents the Rondo framework and section 4 gives its application definition model. Section 5 gives an example application to demonstrate the capabilities of our tool. Section 6 gives some implementation details. Section 7 presents the related work and Section 8 concludes the paper.

2. CHARACTERISTICS OF PERVASIVE APPLICATIONS

Designing and developing pervasive applications is a difficult challenge because the developer needs to manage a wide range of issues such as device contingency, device heterogeneity, changing user context and security of sensitive information. Moreover, developers of high-level services need to work closely with other experts in different domains like embedded systems and mobile networks. As a result, developers end up developing one-time solutions for these issues rather than concentrating on business logic of their application. Middlewares using software engineering paradigms like Component-based Software Engineering (CBSE) and SOC proved to be very efficient addressing some of the issues encountered while developing pervasive applications. These two approaches provide complementary solutions: On one hand service orientation enables defining dynamic software architectures without depending on actual service implementations. On the other hand, component models facilitate the development of service implementations by modularity and reuse principles; and also enable life cycle management of resulting applications. Lately, service-oriented component models emerge, combining principles of these two approaches. Service-oriented component models provide runtime composition of loosely coupled components with automatic handling of dynamism, non-functional aspects (such as distribution, persistence, security) and life cycle management.

Even though, CBSE and SOC address several general problems, there are many open challenges to be handled for facilitating development of pervasive applications:

2.1 Heterogeneity

Services available for an application are heterogeneous and may be provided by some third-party software or remote device discovered over network. Aside from the vast diversity of communication protocols on different application domains cited above; applicative protocols tend to have different natures like RPC-based, event based or stream based. Thereby, applications need not only cover different communication protocols but also be able to cope with different natures of application level protocols. Furthermore, discovering and managing devices in a pervasive environment is a difficult task, as one should take care of device dynamicity and metadata information like service identification, physical location of the device or Quality of Service metrics.

2.2 Reactivity to data

Pervasive applications offer services with added value by leveraging the data coming from different sources, including sensor devices. So it is only natural to expect that in a pervasive application, a service depend not only on other service specifications but also to well-defined data types, where meta-information of the data is more important than its origin. Also, this data-orientation imposes a programming scheme where the consumer reacts to an event containing data produced by the provider. Therefore, a pervasive middleware should enable defining dependencies over data types and assure that these dependencies are satisfied with the data produced by data provider services.

2.3 Context-awareness

Pervasive applications need to transform raw data sensed from devices into more meaningful state indicators called 'context'. The content of context is very subjective to a particular application. Therefore, one can expect from an application to incorporate "*context provider services*" that are responsible of transforming raw information from different, possibly heterogeneous sources to context state. Examples of applicative context may include user's current behavior, location, mood or general routine as well as environmental information like temperature and luminosity level, etc. Another kind of context information represents the state of the running computing system from different non-functional aspects. Performance metrics are good examples for execution context: Memory and CPU consumption, response time, bandwidth, etc.

2.4 Dynamic Adaptability

In a pervasive computing environment applications should adapt constantly to the changing context. As exemplified above, the execution context of a pervasive application involves software modules, as well as users and devices. Pervasive application should continue to satisfy user requirements in face of contingent devices, failing software modules and changing context. In order to do this, it should be aware of its context and flexible enough to be able to apply necessary configurations and change its behavior. In addition, all this adaptation should take place autonomously to reassure user acceptance and fulfill the pervasive computing vision.

However, in pervasive environments, due to changing context, there are many aspects that are not known at design time. Thus, hardcoded and fully specified applications are not a good match for pervasive context. On the other hand, some level of specification is needed to guide runtime adaptations and autonomic actions. This work focuses on defining an application definition model that will enable context-aware pervasive applications. Our model allows developers to specify different configurations of the application regarding changes in the context and available services in an abstract way. Then our application manager is capable of assembling and executing this model on a service-oriented environment, which is capable of handling service dynamism and event based communication. Lastly, our application manager with autonomous capabilities allows performing runtime adaptations on the application.

3. RONDO: A TOOL SUITE FOR PERVASIVE APPLICATION DESIGN

Rondo is a tool suite for designing, deploying and executing pervasive applications. Rondo framework uses a model-driven approach and has three main goals: designing dynamic applications, specifying pervasive environment and enabling application adaptations for context-awareness. This approach aims managing the life cycle of pervasive applications, from development until runtime changes. Rondo provides a domain-specific language based on the notion of components to define the architecture of pervasive applications. Then at runtime, the application manager takes this description and configures the service-oriented execution environment in order to deploy and start the application, all taking into account current state of the environment, represented by several runtime models (Figure 1). And while the application is running, this manager continues to monitor and manage the created application. The following section briefly introduces the building blocks of this approach.

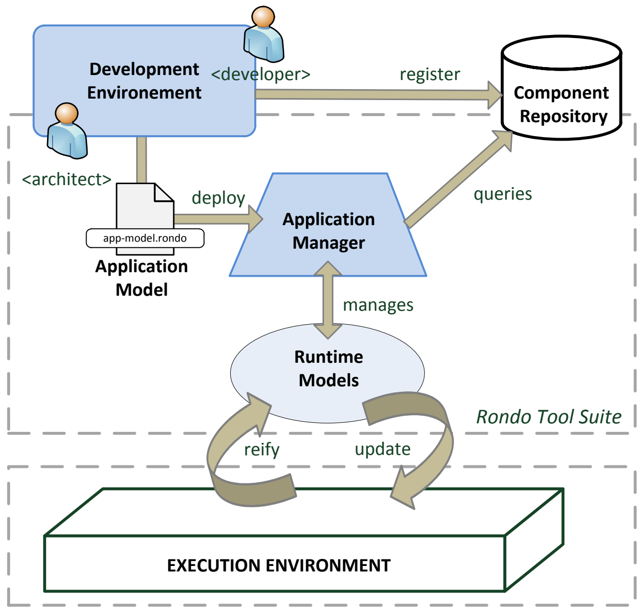

Figure 1: Rondo Global Approach

3.1 Execution Environment

In Rondo tool suite, we define an extended service-oriented component model in order to take into account event-based interactions (Figure 2). Service interactions between components are subject to service contracts, whereas event-based interactions are based on consumed data type and publish-subscribe principles. Like usual component models, our model includes properties that can be used to describe a component (related to the general aspects of a component type) and also those that are used to configure an executed component (component instance). Overall, a service-oriented approach and event-based interactions are necessary to tackle dynamism and reactivity issues of programming business logic for pervasive applications.

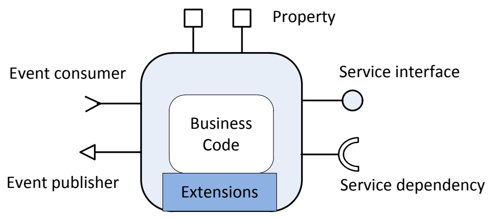

Figure 2: Component Model

Our dynamic execution framework provides an execution environment for service-oriented components. It is capable of dynamically installing, starting, stopping, uninstalling, and reconfiguring component instances, thus managing life cycle of executed services in the platform. It includes a local service registry and mechanisms for discovery and importation of distant services such as sensing devices. It also incorporates event delivery systems between components in order to respond to data reactivity requirement.

This general-purpose component model is a good candidate to represent devices (sensors and actuators) present in pervasive environments. A component instance reassuring the communication with the actual device would expose device functionalities as services and send notifications about changes in its state as events. Abstracting devices as components facilitates the access to device resources from application components, facilitating the task of pervasive service developer.

3.2 Runtime Models

Runtime models serve as abstraction between our application manager and the underlying execution platform. They represent relevant information about the current state of the execution environment. Different aspects of the system can be represented in this way; like component architecture, which represents running components, available services and bindings or still component repository; which represents deployed component declarations, dependencies of deployment units containing these declarations.

Runtime models provide a causal link between the application manager and the execution environment. They reify changes occurred in the environment and allow the application manager to make changes on the environment. They facilitate gathering information about the execution details that would otherwise be intrusive and platform dependent. This information would contribute to the construction of the knowledge about the system, used by autonomic decision making system.

3.3 Application Manager

The application manager is capable of taking an application model as entry, resolving, installing and maintaining it as the application evolves through changes in the context including different life cycle phenomena of constituting components and services. For this, application manager applies different strategies associated with each architectural block defined in the application model. These strategies are called ***policies***. A policy implements installation, activation and adaptation logic for the associated block. Policies use runtime models to make changes on the execution environment and get notified of the changes. For example, a component policy implements how to resolve the dependencies of a component declaration and how to instantiate it. At runtime, these policies work by monitoring the instances to make sure they are using required services and running without errors.

As an ongoing work, in the actual framework, default implementations for each of these policies are proposed. They can be extended for implementing specialized self-management policies, customized per application. However, in the following sections, this paper concentrates on the definition of the application model.

4. APPLICATION DEFINITION MODEL

Rondo application definition model allows developers defining applications by composing components, service discovery instructions and binding instructions. Unlike traditional service composition and architecture description languages (ADL), our application model allows including components into an application without specifying all the interactions between components and neither being tied to any specific component implementation. This enables selection, configuration and activation of different implementations at runtime. Optional binding instructions allow specifying explicitly which services to bind for some of the component dependencies. However, other dependencies would be resolved at runtime from registry of available services. Figure 3 presents the model of our application description. This chapter presents important entities constituting our model.

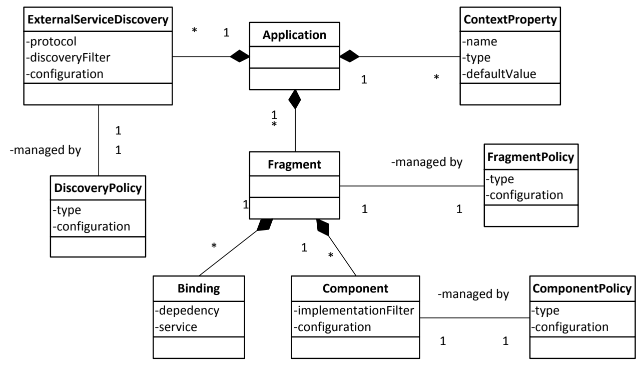

Figure 3: Rondo Application Model (simplified)

4.1 Components

In our model, each ***component*** represents an abstract block of the application, which satisfies a business function. Components include a filter property to enable selection at runtime of the implementation to execute. The filter allows drilling down among available component definitions to choose an implementation according to its identifier, version or other declarative properties. Also, resulted implementations should be checked towards the given instance configuration. This configuration is a set of properties serving to configure the instance of selected component with reference to its declaration, including:

- ***Dependency filter***, serving to filter available services at runtime to resolve a service dependency,
- *Service properties***,** which will be published with the provided service,
- ***Event consumption***, topic subscribed to consume events and filter applied to those events for delivery to the component instance,
- ***Event publishing*** properties, meta information added to published events
- And lastly ***property values*** to configure the instance.

Component configurations are completed by ***bindings,*** which express explicit wires between component dependencies and services. At the moment of instantiation, bindings are taken into account and they override default filter configurations.

4.2 Fragments

In Rondo application model, component and binding are regrouped in a concept called ***fragment***. A fragment is a set of components that will share the same life cycle during the execution of an application. Fragments represent different features included in applications. This concept is similar to the notion of variability in software architectures [15]. Fragments serve to attribute instantiation policies to a group of components. Expressing different features and managing their life cycle according to the context information enables modeling applications with dynamically changing behavior according to the context. To give an example, the following fragment is constituted of two components and a binding that wires service and dependency of these to components:

```
<fragment name="demo-fragment">
   <policy type="service-listener">
       <activates on="service" filter="
       (interface=org.rondo.demo.httpservice)"/>
   </policy>
   <component name="BarComp" impl="MyImpl">
       <policy type="simple"/>
       <configuration>
         <provides id="provide-resource"/>
       </configuration>
   </component>
   <component name="FooComp" implFilter="
   (specification=FooSpecification)" >
       <configuration>
          <requires id="require-res"
filter="(resource.name=*)"/>
          <provides id="http-servlet"/>
       </configuration>
   </component>
   <binding dependency="FooComp.require-res"
   service="BarComp.provide-resource">
</fragment>
```

Policy statement denotes that the fragment installs and activates constituting two components when there is a service available on the platform matching the given filter.

4.3 Context

There are numerous ways of modeling ***context*** information in pervasive environments [1]. Representing applicative context is highly dependent to the application logic and subjective for each application. Rondo framework employs a mechanism to centralize and distribute context information that represents the state of a relevant system property that will affect application architecture, such as memory consumption of the platform, free disk space or still more applicative properties like default user interaction language. To define these properties, an application definition employs a key-value list with default initiation values. Then, once at runtime, different components can write into this context, and others can get notified of changes or get the value of a context entry. Note that this mechanism is not for handling huge amounts of data, nor capable of complex event processing but is comparable to the environment variables of operating systems or properties in BPEL (Business Process Execution Language).

4.4 Discovery of External Services

Moreover, our application model includes instructions for ***discovering external services*,** notably for services provided by devices discovered at runtime. Instead of instantiating application components, the application manager configures the execution framework to activate discovery mechanisms. A discovery mechanism is in charge of discovering and representing external services as regular component instances. These instances serve as proxies that handle communication with the remote service provider. In most of the cases, each application customizes the way it interacts with external services in its environment. Having these application specific proxies allow natural access to these services, describing the interface between the pervasive application and its environment.

5. APPLICATION EXAMPLE

The utility of presented approach and application model is validated, demonstrating it in a smart home application. In this application scenario, user is in a house equipped with a home media server and several media renderers in different rooms that is capable of streaming content from the server. The aim of this application is to change the streaming place of currently played media so that the user can continue to watch and/or listen the media while he/she moves through the house. Figure 4 presents the architecture of this application scenario.

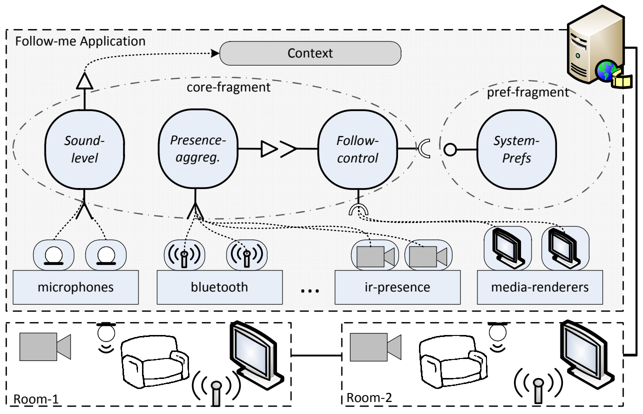

Figure 4: Scenario Follow-me

The application works as follows; user activates the application by interacting with a media renderer, indicating that he/she wants that the currently playing media follow him/her. This activates the *follow-controller* component, which gathers presence detection notifications and manages renderer devices to stop and resume the media streaming. Obviously detecting the location of the user is a major issue. The application combines different approaches in order to detect if a room is occupied by a user: Infra-red presence sensors, pressure sensors to detect if user sits on a couch in a room or still Bluetooth availability of a device associated with a user. So information coming from these detection mechanisms is aggregated and delivered to the controller component.

In addition to this, it is expected from the application to adapt to the following changes in the environment:

- If a room is not equipped with a media renderer or a media renderer is not active (e.g. turned off), the application reacts as slightly increasing the volume of the media streaming in the previously occupied room.
- The application keeps the volume of the diffused media in the same level across different media renderers, to assure the user comfort. However if the volume level exceeds a certain threshold, application overrides user-set volume level with system settings.

This application is described in Rondo application definition language shown in Figure 5. The application defines two fragments. First fragment is composed of static application components such as *follow-controller*, *presence-aggregator* and *sound-level*. As an example, the *follow-controller* component interacts with other components and devices via service and event interactions. This enables the first adaptation scenario where the *follow-controller* logic can react to changes in availability of services as well as to presence events.

The second fragment employs an activation policy, which is linked to the context definition *high-decibel*. This means that changes in this context state will activate or deactivate this fragment, which contains a component that provides a preference service. When available, this service is consumed by the controller to override user volume preferences. The context information *high-decibel*, is changed by the component *Sound-level* when microphones detect a high ambient sound level.

```
<rondo:rondo xmlns:rondo="fr.liglab.adele.rondo.application">
  <application name="follow-me">
<!-- Application context -->
    <context>
       <property name="high-decibel" type="boolean"
       defaultValue="false"/>
    </context>
<!-- Fragment 1 -->
    <fragment name="core-fragment">
       <policy type="core"/>
       <component name="follow-controller"/>
       <component name="presence-aggregator"/>
       <component name="sound-level"/>
    </fragment>
<!-- Fragment 2 -->
    <fragment name="pref-fragment">
       <policy type="context-listener">
         <activate on="context"
              value="(context/high-decibel=true)">
       </policy>
       <component name="system-preferences">
         <provides id="pref" />
       </component>
    </fragment>
<!-- Import devices -->
    <import name="media-renderers"
       protocol="upnp-renderer"/>
    <import name="ir-presence-sensors"
       protocol="zwave-ir-presence"/>
    <import name="pressure-sensors" protocol="coap-pressure"/>
    <import name="bluetooth-presence-sensors"
       protocol="xml-rpc-bluetooth"/>
    <import name="microphones" protocol="ip-micro"/>
  </application>
</rondo:rondo>
```

Figure 5: Description of the Follow-me application

Lastly, various *import* statements represent different devices and services to be discovered in the pervasive environment.

6. IMPLEMENTATION

We have implemented a prototype of the Rondo tool suite using iPOJO [6] as our execution platform, which is an extensible service-oriented component model running on top of OSGi [10], which simplifies the development of component implementations. For discovery and importation of external services into our execution platform, Rose [2], an ecosystem for exporting and importing distributed services is used.

As the central element of the tool suite, the application manager profits from dynamic architecture capabilities of iPOJO. When a new application description file is deployed to the platform, the application manager parses the XML-based language (see Figure 5) and creates the abstract specification model of the application. Then different portions of this model (e.g. fragment, component, import) are transferred to application manager policies referenced in the model. Default implementation of these policies focus on providing late deployment and implementation selection capabilities. They can be extended match application specific requirements to make adaptations on the application architecture at runtime.

We use runtime models to represent component types, running instances, available services and OSGi bundles. Runtime deployment of applications is being possible thanks to dynamic deployment features of our platform. We are able to download and deploy deployment units from the component repository, where it is possible to query deployment units by component descriptions, exposed services and required services.

7. RELATED WORK

In this section, we discuss some of the relevant works addressing pervasive and autonomic application development. There are several different approaches to define applications as a composition of services.

In [12], the MUSIC planning-based middleware combines SOA and component-based software development, and creates adaptation plans according to changes in service descriptions and agreements. In [5], authors mix component-based static deployment descriptors with ECA (Event-Condition-Action) rules to design, deploy and reconfigure applications using runtime models. These two approaches concentrate on implementing autonomic managers and does not fully leverage dynamism brought by services in their application definitions. Unlike our proposition, where explicitly unsatisfied dependencies are resolved at runtime from available services; strict ADL definitions impose specifying all bindings between components.

Dia Suite [4] defines a development life cycle similar to ours to define the pervasive environment and applications. However, passing to execution, it generates the whole execution environment, leaving no room for variability, late deployment or dynamism. Finally, in [7] authors propose an opportunistic service composition language that not only uses available services in the platform but also includes running instances into applications. While this adds new frontiers to the application resolution, our approach aims to give more deterministic results.

8. CONCLUSION AND FUTURE WORK

In this paper, we have presented our vision on pervasive application development, where abstract specifications of applications are deployed and resolved in a service-oriented execution platform by the help of an autonomic application manager. Our application definition language allows specifying variability, component-service interactions and dynamism with varying level of determinism. Also service discovery instructions are included in our model, to enable defining external service discovery at runtime. Our vision leads to open pervasive execution platforms, where different stakeholders are able to deploy and execute their applications.

As an ongoing work, there are several points that we would like to improve in our proposition. We think that the key to build such an execution platform, where each application is dynamically variable, is to at least provide some mechanisms to constrain the visibility of services provided by different applications. Moreover, deploying and executing abstract application specifications requires dependency resolution mechanisms before execution. We are investigating different resolution strategies that we can employ in deployment and diagnostic tools.

9. ACKNOWLEDGEMENTS

This research work was supported by the European FP7 project BUTLER, under contract no. 287901.

10. REFERENCES

[1] M. Baldauf, S. Dustdar, and F. Rosenberg. A survey on context-aware systems. Int. J. Ad Hoc Ubiquitous Computing, 2(4):263– 277, June 2007.

[2] J. Bardin, C. Escoffier, and P. Lalanda. Towards an Automatic Integration of Heterogeneous Services and Devices. In Proceedings of the Services Computing Conference (APSCC), pages 171–178. IEEE Computer Society, December 2010.

[3] N. Bencomo. On the use of models during software execution. In Proceedings of the 2009 ICSE Workshop on Modeling in Software Engineering, MISE '09, pages 62–67, Washington, DC, USA, 2009. IEEE Computer Society.

[4] D. Cassou, J. Bruneau, and C. Consel. A tool suite to prototype pervasive computing applications. In PerCom Workshops, pages 820–822, 2010.

[5] J. Dubus and P. Merle. Applying OMG D&C specification and ECA rules for Autonomous Distributed Component-Based Systems. In Proceedings of the 2006 International Conference on Models in software engineering, MoDELS'06, pages 242–251, 2006. Springer-Verlag.

[6] C. Escoffier, R. S. Hall, and P. Lalanda. iPojo: an Extensible Service-Oriented Component Framework. In IEEE SCC, pages 474–481, 2007.

[7] J. Estublier, I. A. Dieng, E. Simon, and D. Moreno-Garcia. Opportunistic computing experience with the SAM platform. In Proceedings of the 2nd International Workshop on Principles of Engineering Service-Oriented Systems, PESOS '10, pages 1–7, New York, NY, USA, 2010. ACM.

[8] V. Issarny, M. Caporuscio, and N. Georgantas. A perspective on the future of middleware-based software engineering. In 2007 Future of Software Engineering, FOSE '07, pages 244–258, Washington, DC, USA, 2007. IEEE Computer Society.

[9] J. O. Kephart and D. M. Chess. The vision of autonomic computing. IEEE Computer Society, 36(1):41–50, Jan. 2003.

[10] OSGi Alliance. OSGi service platform release 4. [Online]. Available: http://www.osgi.org/Main/HomePage.

[11] M. P. Papazoglou, P. Traverso, S. Dustdar, F. Leymann, and B. J. Kramer. Service-oriented computing: A research roadmap. In Service Oriented Computing (SOC), number 05462 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2006.

[12] R. Rouvoy, P. Barone, Y. Ding, F. Eliassen, S. Hallsteinsen, J. Lorenzo, A. Mamelli, and U. Scholz. MUSIC: Middleware Support for Self-Adaptation in Ubiquitous and Service-Oriented Environments, Software Engineering for Self-adaptive Systems. pages 164–182. Springer-Verlag, Berlin, Heidelberg, 2009.

[13] M. Salehie and L. Tahvildari. Self-adaptive software: Landscape and research challenges. ACM Trans. Auton. Adapt. Syst., 4(2):14:1–14:42, May 2009.

[14] M. Satyanarayanan. Pervasive computing: Vision and challenges. IEEE Personal Communications, 8:10–17, 2001.

[15] M. Svahnberg, J. van Gurp, and J. Bosch. A taxonomy of variability realization techniques: Research articles. Software Practice and Experience, 35(8):705– 754, July 2005.

[16] T. Vogel, A. Seibel, and H. Giese. Toward megamodels at runtime. In Proceedings of the 5th International Workshop on Models@run.time at the 13th IEEE/ACM International Conference on Model Driven Engineering Languages and Systems (MoDELS 2010), Oslo, Norway, volume 641 of CEUR Workshop Proceedings, pages 13–24.