Multi-Dimensional Service Compositions

L. Baresi, E. Di Nitto. and S. Guinea
Politecnico di Milano
Dip. Elettronica e Informazione
I-20133 Milano, Italy
{baresi | dinitto | guinea}@elet.polimi.it

S. Dustdar
Vienna University of Technology
Distributed Systems Group
A-1040 Wien, Austria
dustdar@infosys.tuwien.ac.at

Abstract

The wide diffusion of reliable Internet is pushing two key novelties in the conception of modern software applications: the Software as a Service paradigm and the idea of the Internet of Things. Traditionally, services and things have been considered as separate entities, even addressing different needs and application domains. In contrast, we feel that services and things should be integrated and demand for proper design and programming paradigms that ease the task of system builders and enable reuse of components through various systems. Furthermore, we also see the need to take into account the many cross-cutting issues that are typical of any complex application (e.g., security, user interface, transactionality). We suggest multi-dimensional service assembly as the right abstraction for taking into account all these different aspects. In this paper we sketch our ideas, discuss the implications of multi-dimensional service assembly, and draft a research agenda that goes towards the development of a well established theory in this area.

1. Introduction

The evolution of communication technologies, with the *Internet* as aggregator, is pushing two key novelties in the conception of modern software applications: the *Software as a Service* paradigm and the idea of the *Internet of Things*.

Software as a Service enables consumers of software artifacts to use them remotely. Services are widely used as integration means for complex systems [4] in continuously evolving scenarios [5]. Moreover, they also provide the underpinnings of Web 2.0 applications [10] and act as the backbone of various domain-specific standards, like for example AMI-C [1] and AUTOSAR [9] in the automotive industry.

The *Internet of Things* includes a number of devices such as RFID tags, sensors, and GPS antennas, which allow us to conceive *live* contexts that can be exploited by various applications and services. RFID tags have already substituted more conventional management systems in logistics, while the GPS antennas embedded in many devices (e.g., our mobile phones) provide easily accessible information about current positions, and enable the most disparate context-sensitive applications: from simple routing systems to location-aware advisers (for restaurants, hotels, site seeing, etc.).

Traditionally, services and things have been considered as separated entities, even addressing different needs and application domains. However, in the last few years, we have observed the tendency of using them together to address problems within the context of pervasive computing. For example, think of a logistics application that, among the other functions, manages the delivery of containers to proper destinations. In such an application, we would like to track the position of each container, and maybe set its internal temperature, without dealing with the low-level devices directly. Indeed, we would like to be able to perform these operations on containers independently of whether they are traveling on a train or waiting in a parking area. Thus, we would like to see the containers as *abstract data types* simply accessible through a uniform service interface regardless of the context in which they are living or of the devices that are needed to trace and control them. When dealing with such kinds of applications, various *cross-cutting issues* need to be considered as well. These concern the kind of GUI they offer, the level of security they guarantee, the ability to manage identities, the contextual information they are able to deal with, and such.

We could build a software system that addresses the above scenario by implementing it in an ad-hoc manner and by hard-coding the interactions with low-level devices into the application logic. However, we feel that the increasing number of applications of this kind demands for proper design and programming paradigms that ease the task of system builders and enable reuse of components through various systems. In literature there are proposals like WS-

CIM [3] and DPWS [2] that hide the peculiarities of devices behind a service-like interface to enable their integration in traditional software applications, but also to ease their substitutability. These standards only mimic hardware characteristics; they do not provide suitable high-level abstractions to let designers seamlessly "compose" them with their business services and substitute them when needed.

In this paper we suggest that software components, devices, and cross cutting issues compose a *multi-dimensional service assembly*. They can be regarded as services themselves offering the same interaction paradigm, making them composable in more complex services. For instance, in the example of the logistic application the service abstracting the container as a whole could be built by composing the lower level services provided by a GPS antenna, temperature and pressure sensors, and RFID tags mounted on the container itself. Moreover, the container service could dynamically adapt, based on the situation, to exploit the GPS service associated with the vehicle that is transporting it, or any other devices offering position information.

The rest of this paper is organized as follows. Section 2 presents an integrated vision and the concept of multi-dimensional service composition. Section 3 introduces the more advanced features of our proposal, and Section 4 concludes the paper by drawing a research agenda.

2. Integrated Vision

The key assumption of this research effort is that the main elements we consider (devices, software, and cross-cutting issues) can be regarded as *services*, and that these are seamlessly composable to provide a complete application. We want to break the barrier that *service composition* is only about combining the operations offered by remote software components. We want to be able to treat every single application element as a service, and compose its features. Figure 1 renders the *multi-dimensionality* of the problem: each axis identifies a particular aspect, and a complete application is a proper blending of representatives of some (or all) of the axes. The implementation of this "vision" requires:

- Suitable wrappers to start conceiving applications from a set of homogeneous entities. The need for special-purpose wrappers is clear when we aim to abstract physical devices into software elements, but it is also mandatory when packaging different cross-cutting concerns, or different families of software services (e.g., Web vs. REST services);

- Composition means being able to accommodate these different needs. We propose that a single *high-level* mechanism be adopted when assembling operations, data, and non-functional characteristics. For example, we could consider the well-known domain of Web services, and BPEL (Business Process Execution Language), for assembling the different aspects. However, we believe its primitives do not provide the right level of abstraction, and do not foster a proper separation of concerns.

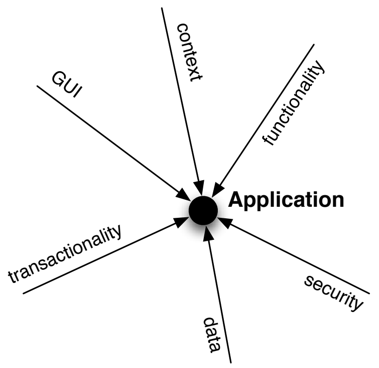

Figure 1. Some dimensions of modern applications.

A *model-driven* approach [8] is mandatory in this case. We propose models to provide designers with multi-faceted representations of their applications, but also to automatically produce the different parts of the application, that is, those needed to assemble the services and cope with their native technologies. To tackle these problems, and obtain a homogeneous view, we can start from the reference model of Figure 2. We consider that an Application can exploit either a set of Services or an Aggregate Service. Each Service is implemented by a Component, which is an abstract concept. A component can be: a Software Component –which further specializes in Business Element, whose meaning is clear, and Utility, used to provide a cross-cutting concern– a Physical Device, or an Aggregate Service itself. Each Component may provide a Management Interface. Through these interfaces, the composition designer gathers a partial insight of both the service's functionality, and the degree to which it can be customized during composition.

The aggregating element is the assembly of the operations provided by the different business elements. This core must be suitably complemented with the abstractions provided by the physical devices and the features supplied by the utilities. While it is meaningful that business elements and physical devices cooperate at the same abstraction level, utilities may work behind the scene to complement the services. For example, if a business element e wants to exchange secure messages, the proper security mechanism

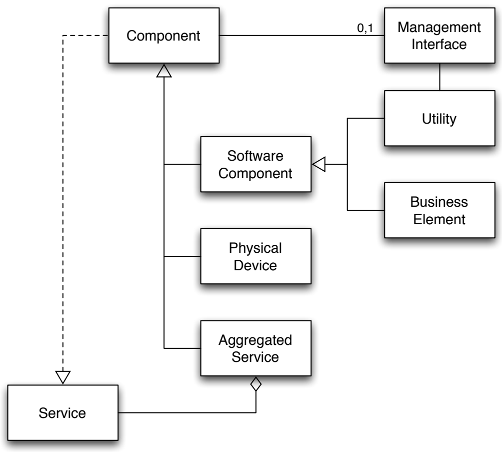

Figure 2. Simplified reference model.

must be assembled with e. This is why each component, besides its functional interface (Service), must also provide a suitable management interface.

We foresee three kinds of services:

- A **black-box service** is a service that only provides a syntactical specification of its capabilities, and does not provide a management interface.

- A **grey-box service** is a service that already provides a management interface. There are many ways to create a management interface. Among others, we cite instrumentation techniques, such as code annotation. Indeed, a sufficiently expressive declarative language could be used to describe a service's functional and non-functional composition hooks.

- A **white-box service** is a service for which we have knowledge of the internal design and code. This means that the functional dependencies it has are known, and that composition hooks can be found by looking at the code.

3. Advanced Features

As for composition paradigms, we are investigating both workflow-like assemblies (a-la BPEL) and also more loosely-coupled interactions (e.g., compositions based on a publish and subscribe infrastructure [7]). In this paper, however, we prefer to stress the servilization of physical devices, the blending of cross-cutting utilities, and the run-time adaptation features.

3.1. Physical devices

The abstractions of Figure 2 allow us to treat complex physical objects (made possible through different physical devices) as services. As such we can even aggregate them. For example, as previously stated, we could mask the sensors, GPS antennas, and RFID tags mounted on a container, and provide a single service *Container*. It would be capable of locating the container, getting and setting its temperature, and sensing its presence in a given area, allowing us to ignore low-level details.

More specifically, we foresee two separate abstraction levels. Application services (the highest level) mask application elements, which offer operations and raise events to the rest of the system. These services can be associated with different contexts during the evolution of the system, or even belong to different contexts at the same time.

On the other hand, application services take advantage of active physical devices for getting information and for executing operations. However, they do not deal with them directly. Instead, we propose a second intermediate abstraction level called *logical level*. These services can also be functional elements in the system, offer operations (used by application services), and raise events (for the application level). A logical service does not only abstract the single device, but it also abstracts the access gateway for it. This means that a logical service offers the functionality of both the device and the gateway it is associated with. For example in the case of RFID tags, it hides the actual reader: when a tag is seen by a short range reader, its abstraction can offer an approximation of the position of a tag given the position of a reader.

3.2. Cross-cutting utilities

Service compositions can be enriched by a number of cross-cutting utilities. Some may contribute to define a composition's quality of service (e.g., by adding *transactionality*, *security*, or *identity management*), others may enable context-aware behavior (e.g., by adding the notion of *context* or *user profiles*), while others may even provide rich run-time management features (e.g., *monitoring* and *adaptation* [6]). What this means is that a single service, or a composition, does not need to implement such aspects directly, but it can rely on out of the box utilities. This obviously has a big impact on a service's design and development. Indeed, a service can concentrate on its core business without having to cope with (or foresee) all the different ways it may be utilized.

As an example, depending on the scenario in which the service is to be used (i.e., depending on the Service Level Agreement established with its client), this service may setup different security policies. For instance, it could allow for insecure transactions, or enforce one or more security standards through composition. Also, if the service is context-aware, it could be designed to take advantage of context information leaving to a context-collector utility the task to manage how the information is retrieved at run-time. In this case, the service could be customizable as it could choose among various context-collector utilities, each of which will be more or less suitable to a specific application scenario. Finally, a service could follow the model-view-controller design pattern and provide the model and the controller logic, and delegate the related view to an appropriate utility. This means that the service may have different GUIs if it is run on a notebook or on a mobile device.

Which utility to use in a given composition can be a design issue, but we can also envision it occurring at run time. Any one of them might be modified in the wake of evolving requirements or situations, providing unprecedented levels of customization and adaptivity of the composition.

3.3. Run-time adaptation

To manage such complex service compositions, and harness the true advantages of customization and adaptation, we will need to develop an integrated information model for capturing requirements of complex and diverse nature, keeping in mind that the requirements may evolve. This calls for different requirements representations, and for a highly advanced management framework capable of monitoring them throughout the composition's execution cycle.

The ides is to extend our previous work in the field [11]. Indeed, we believe we can extend the integrated information model developed in SEMF (Service Evolution Management Framework) to cope with such complex compositions. Since it is impossible to enforce a single model for all types of requirements, we envision a model to which different sub-models are linked without enforcing any specific data representation. In SEMF we started by experimenting sub-models that capture SLAs, QoS, pre-conditions, licenses, interaction patterns, post-conditions, interfaces, taxonomies, folksonomies, and general documentation. A specific requirement type is described as an external source of information. Based on that, a particular requirement for a service, at any particular point it time, can easily be retrieved and processed. Utilizing this rich source of requirements information, we can check whether they are fulfilled and perform adaptation based on manageable requirements.

4. Conclusions

In this paper we have sketched our idea of regarding devices, software, and cross-cutting utilities under the unifying idea of multi-dimensional service assembly. We think this metaphor would simplify the design and implementation of pervasive systems where complex enterprise applications coexist with low-level software and specialized devices. Of course, in order to make our idea really effective, several issues need to be addressed, most of which are part of our future research agenda. We need to refine the conceptual model behind the idea of service assembly. In particular, we need to define a proper computational model that allows us to abstract from the peculiarities of specific devices and components. Moreover, the computational model should be connected to a proper run-time infrastructure that offers all the mechanisms needed to execute, combine, and adapt the various components of a service assembly. As we envisage our assemblies dynamically adapting to the execution context, the relationship with the autonomic computing field will have to be investigated, and in particular the techniques supporting self-configuration, self-healing, self-optimization, and self-protection of each single component and of the assembly as a whole.

References

[1] Automotive Multimedia Interface Collaboration. `www.ami-c.org/`.

[2] Devices Profile for Web Services. `specs.xmlsoap.org/ws/2006/02/devprof/devicesprofile.pdf`.

[3] WS-CIM Mapping Specification. `www.dmtf.org/standards/published_documents/DSP0230.pdf`.

[4] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services: Concepts, Architectures and Applications*. Springer, 2004.

[5] L. Baresi, E. Di Nitto, and C. Ghezzi. Toward Open-World Software: Issues and Challenges. *COMPUTER*, pages 36–43, 2006.

[6] L. Baresi and S. Guinea. A dynamic and reactive approach to the supervision of BPEL processes. In *Proceedings of the 1st India software engineering conference*, pages 39–48. ACM New York, NY, USA, 2008.

[7] P.-T. Eugster, P.-A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2), 2003.

[8] D. Frankel. *Model driven architecture*. Wiley New York, 2003.

[9] H. Heinecke, J. Bielefeld, K. Schnelle, N. Maldener, H. Fennel, O. Weis, T. Weber, J. Ruh, L. Lundh, T. Sandén, et al. AUTOSAR–Current results and preparations for exploitation. *7th EUROFORUM conference Software in the vehicle*, 2006.

[10] T. O'Reilly. What is Web 2.0: Design Patterns and Business Models for the Next Generation of Software.

[11] M. Treiber, H. Truong, and S. Dustdar. SEMF - Service Evolution Management Framework. In IEEE Computer Society, editor, *34th EUROMICRO Conference on Software Engineering and Advanced Applications, Special Session on Quality and Service-Oriented Applications*, 2008.