A01

084/04

university
of milano
bicocca

QD quaderni

department of informatics, systems and communication

Designing Self-Adaptive Service-Oriented Applications

G. Denaro, M. Pezzè, D. Tosi

research report n. 4
september 2006

www.aracneeditrice.it
info@aracneeditrice.it

via Raffaele Garofalo, 133 A/B
00173 Roma
(06) 93781065

ISBN 978-88-548-0946-8

ISSN 1828-3357

I edizione: gennaio 2007

Table of Contents

Designing Self-Adaptive Service-Oriented Applications

Giovanni Denaro[1], Mauro Pezzè[1] and Davide Tosi[1]

Università degli Studi di Milano - Bicocca
Via Bicocca degli Arcimboldi 8 20126 Milano, Italia

Abstract. The integration of third-party web services helps solve complex business problems and reduce risks, costs and time-to-market. However, the task of the integrators is challenged by services that are maintained by different organizations, and may evolve dynamically and autonomously. The impossibility of statically determining which service implementation will be bound at runtime may lead to unexpected failures.
This paper presents a novel approach for designing self-adaptive service-oriented applications, which autonomously react to changes in the implementation of the services, automatically detect possible integration mismatches, and dynamically execute suitable adaptation strategies. The solution proposed in this paper, is based on a runtime infrastructure that automatically tests remote web services, uses test results to diagnose service mismatches, and executes adaptation strategies to overcome the revealed problems without user intervention.

1 Introduction

Web services and service-oriented architectures are emerging technologies for integrating enterprise applications, leveraging electronic B2B and B2C solutions, and extending the life of legacy software. In a nutshell, web services are remote programs invoked over the Internet, using standard protocols (e.g., HTTP and XML) for exchanging data between requesters and providers. Web services allow enterprises to export functionality outside the enterprise bounds, thus enabling stakeholders of different domains to rapidly and seamlessly integrate third-party expertise into their applications. For example, airlines and hotel chains can export services for online booking; so that travel agencies can combine these and other services to optimize travel plans. Service-oriented architectures help solve complex business problems, decreasing risks, costs and time-to-market.

The integration of third-party web services is challenged by the difficulty of keeping consistency between software systems that are maintained by different organizations and may evolve dynamically and independently, because of both changes in the services and the dynamic discovery of new services. Service providers may change the implementation independently from clients, e.g., to correct faults or meet new requirements. For example, this is the case of old

Quaderni del Dipartimento
University of Milano – Bicocca ©,4-2006

DIPARTIMENTO
DI INFORMATICA
SISTEMISTICA
E COMUNICAZIONI

versions substituted with new ones. Moreover, in many service-oriented architectures, clients can locate services dynamically, using service discovery mechanisms that allow clients to discover and connect web services based on machine-readable descriptions. This means that clients may use different web services in different invocations depending on the choice of a service broker. As illustrated in Figure 1, a broker matches client requests with available services published by providers according to common protocols (e.g., SOAP, WSDL and UDDI), and each time applications reconnect to web services through brokers, they can get different matches and thus use different implementations of the requested services [5].

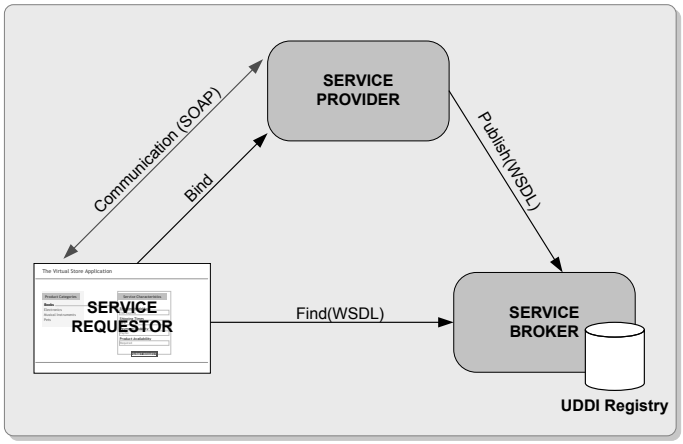

Fig. 1. Main entities and interactions in a service oriented architecture

In this paper, we focus on integration problems that derive from dynamic changes of the invoked services. Different services or service implementations that can be invoked to satisfy a given request must comply with a contract that indicates the characteristics of the required service [1]. In principle, services that comply with the same contract should be equivalent, but in practice contracts tend to specify little more than the service syntax and parameters, leaving many semantic details unspecified and thus implementation-dependent. For example, we have been using web services for obtaining the weather temperatures in US districts on the basis of a contract that required the target location to be indicated with the zip-code and the temperature to be returned as a floating point value, but did not indicate the measurement unit of the return temperature. This contract matched many services that returned temperatures expressed in different measurement units, e.g., Fahrenheit and Celsius, thus leading to client-side failures.

[1] Contracts are usually expressed in standard machine-readable languages, e.g. the Web Service Description Language (WSDL) [3].

We propose a solution that exploits a self-adaptive approach based on a mechanism for revealing possible mismatches between requested and provided services, and for dynamically adapting the client application accordingly. An integration fault taxonomy helps service integrators identify possible integration mismatches, generate test cases for revealing integration problems, and design recovery actions. Integrators can code test cases and adaptation strategies in separate modules, which we refer to as adaptation aspects. We automatically weave adaptation aspects into the client applications, so that the modified client application executes test cases whenever a new service implementation is detected to reveal possible mismatches, and triggers suitable adaptation mechanisms accordingly. We refer to the proposed approach as adaptive integration of web services.

This paper is organized as follows. Section 2 describes mismatch problems that may derive from the integration of web services, referring to a working example that will be used throughout the paper. Section 3 presents an overview of our approach to adaptive integration of web services. Section 4 and Section 5 respectively introduce details of the approach discussing the methodology and the framework for developing self-adaptive service oriented applications. In Section 6 we summarize the results of our experiments. We discuss related work in Section 7, and finally we conclude in Section 8.

2 The Service Mismatch Problem

As discussed in the introduction, the services invoked by an application may change due both to autonomous modifications of the service implementation by the provider and to bindings to new services dynamically discovered by a broker. In this section, we illustrate the problems that derive from dynamically evolving services through a virtual store, an example that we use throughout the paper. Figure 2 illustrates the interactions between the virtual store and the broker to dynamically discover remote stores and other services. When customers select product categories (e.g. electronics, books, musical instruments, etc...) and required purchase characteristics (e.g. price limit, delivery time, shipping and payment modalities, etc...), VirtualStore asks the broker for providers that satisfy the requirements. If the selected providers do not offer all the required services, VirtualStore integrates the services by asking the broker for additional services, e.g., credit cards validations, or shopping carts.

The dynamic integration of different services may lead to several problems. Consider for example, the request for a shopping cart web service. Figure 3 shows an excerpt of the operations that characterize the specification of the request.

This WSDL description can match many WSDL specifications of services available from common UDDI repositories e.g. Amazon [www.amazon.com], that differ in several small but important details. For instance, some providers offer a `CartAdd` operation able to manage multiple instances of the same item in a single add operation, while others may not support multiple instances; we used services that provide `CartModify` operations that remove an item by changing

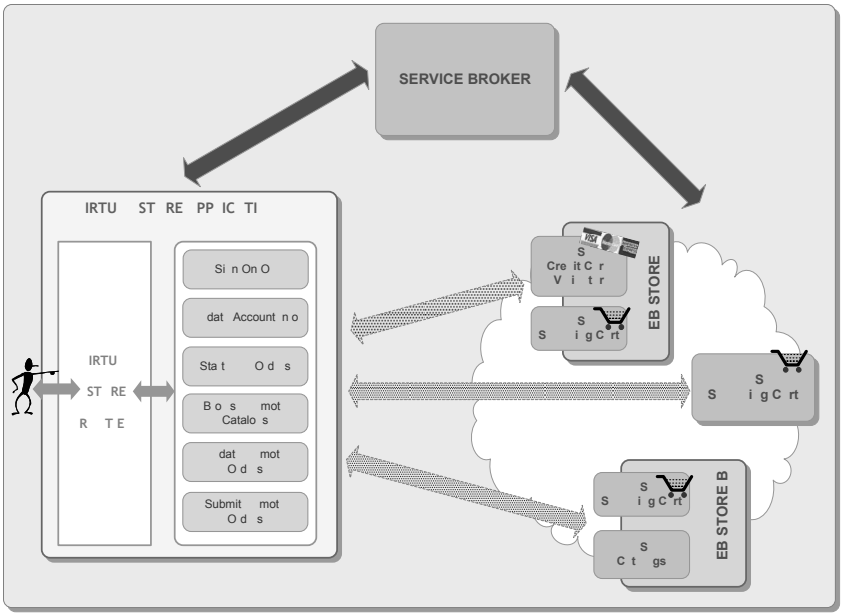

Fig. 2. The overall architecture of the VirtualStore application

its quantity to zero and others that remove an item by decreasing the quantity by one; we discovered services that can create empty carts (`CartCreate` operation) and others that require a selected item to create a cart. Finally, we used services that can manage orders and purchases processed in different user sessions (*persistent cart*), and others that do not, i.e., that loose their contents when sessions terminate (*non-persistent cart*). All variants match the WSDL interface.

Since VirtualStore can be connected to different shopping cart services at different times, we can have several relevant failures due to the inhomogeneous implementations of the target services.

3 A Self-Adaptive Approach

High availability requirements and dynamically discovered web services exclude the possibility of traditional stop-update-test-redeploy-restart approaches to the integration of new or modified services. Self-adaptive applications have been recognized as viable solutions for dealing with systems where size and complexity increase beyond the ability of humans to respond manually, coherently and timely to environmental and system changes [7]. Self-adaptive solutions are being experimented in several application domains, but not in service-oriented applications yet. The massive reuse of services and the frequent updates of implementations corresponding to compatible interfaces are typical of service-oriented

```
...
<!-- ItemSearch: Searches and selects an item through a product catalog --
    >
- <operation name="ItemSearch">
          <input message="tns:ItemSearchRequest"/>
          <output message="tns:ItemSearchResponse"/>
       </operation>


<!-- CartGet: Retrieves the list of items in a cart that you have built
    but have not yet submitted to store for a customer to purchase-->
- <operation name="CartGet">
          <input message="tns:CartGetRequest"/>
          <output message="tns:CartGetResponse"/>
       </operation>

<!-- CartAdd: Adds selections to the cart -->
- <operation name="CartAdd">
          <input message="tns:CartAddRequest"/>
          <output message="tns:CartAddResponse"/>
       </operation>

<!-- CartCreate: Creates a remote shopping cart for a single customer -->
- <operation name="CartCreate">
          <input message="tns:CartCreateRequest"/>
          <output message="tns:CartCreateResponse"/>
       </operation>

<!-- CartModify: Changes quantities of items, delete items, replaces items
    with new ones -->
- <operation name="CartModify">
          <input message="tns:CartModifyRequest"/>
          <output message="tns:CartModifyResponse"/>
       </operation>

<!-- CartClear: Removes all selections from a cart, emptying the cart -->
- <operation name="CartClear">
          <input message="tns:CartClearRequest"/>
          <output message="tns:CartClearResponse"/>
       </operation>

<!-- CartPurchase: Finalizes the purchase process -->
- <operation name="CartPurchase">
          <input message="tns:CartPurchaseRequest"/>
          <output message="tns:CartPurchaseResponse"/>
       </operation>

...
```

Fig. 3. Excerpt of the WSDL description of a shopping cart WS

applications, and allow for defining efficient domain specific self-adaptive solutions for the service-oriented domain.

In this paper we propose a self-adaptive approach to the development of service-oriented applications that combines novel techniques into a traditional sense-plan-act control loop, where the subject system is connected to a controller that in turn feeds commands back into the subject system. Figure 4 illustrates how to instantiate a classic sense-plan-act control loop to deploy self-adaptive service-oriented applications. The invocation of a web service triggers `monitoring` mechanisms. Such mechanisms identify changes in the invoked services that may depend on server-side implementation updates or dynamically discovered services. The detection of a new service implementation triggers `diagnosis` mechanisms that run test cases on the target web service to reveal possible mismatches. If the diagnosis mechanisms reveal any mismatches, associated `adaptation` strategies update the structure and the behavior of the client application to solve the identified problems and optimize the interaction with the target service.

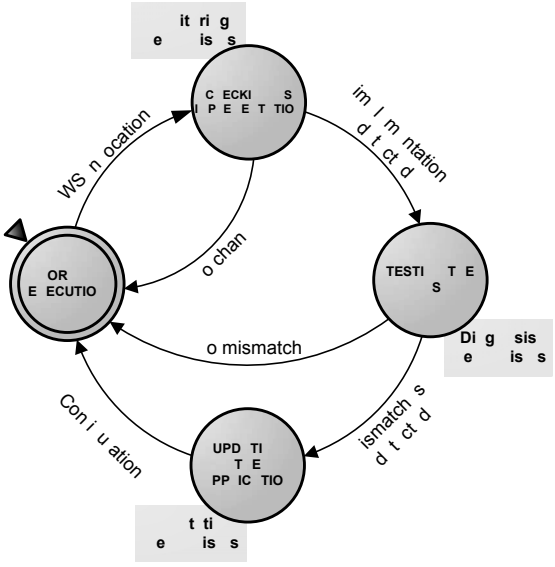

Fig. 4. The generic adaptation control loop

The sense-plan-act control loop must be instantiated for each set of services that comply with a specific contract. The customization consists of defining a set of test cases that can reveal mismatches between different implementations of the same contract and a set of adaptation strategies for the classes of possible mismatches. For example, in the VirtualStore presented in Section 2, we can identify the set of services that comply with the WSDL cart specification in Figure 3.

Different implementations of the WSDL specification may provide persistent and non-persistent carts, i.e., carts that keep or loose their contents through different sessions. A self-adaptive mechanism for these kinds of mismatches can include a set of test cases to verify the persistency of the cart, and adaptation strategies to overcome persistency problems, as illustrated in Figure 5.

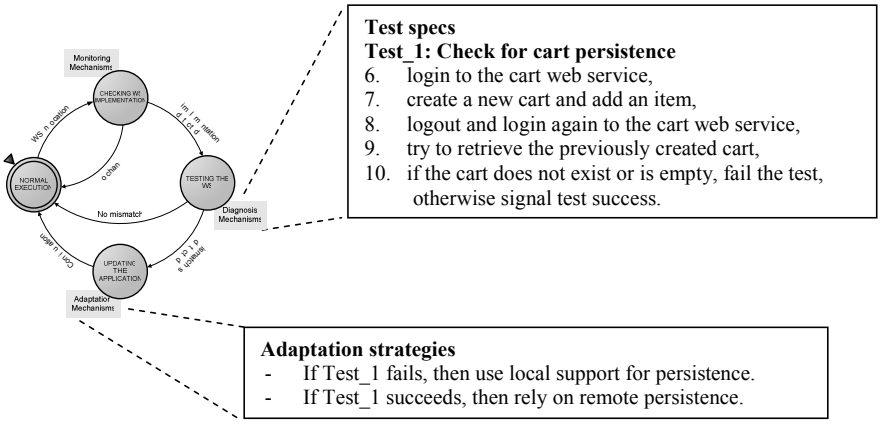

Fig. 5. The instantiated adaptation control loop for the persistency problem

The control loop instantiated with automatic test cases and adaptation strategies will enhance all client invocations of the cart web service. Enhanced client applications intercept all calls to the cart web service, dynamically check the persistence of the cart, and adapt the local behavior accordingly.

4 Design Time Support

The customization of the sense-plan-act control loop is based on a methodology and a design framework that help designers identify the customization requirements for control loops. The methodology is composed of three steps illustrated in Figure 6 that are executed when a new web service category is selected for integration in the target application.

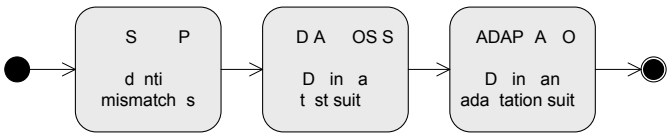

Fig. 6. The methodology for designing customized control loops