Towards autonomic service-oriented applications

Davide Tosi* and Giovanni Denaro

Department of Informatics, Systems and Communication
University of Milano Bicocca
Milan, Italy
E-mail: tosi@disco.unimib.it
E-mail: denaro@disco.unimib.it
*Corresponding author

Mauro Pezzè

Department of Informatics, Systems and Communication
University of Milano Bicocca
Milan, Italy
and
Faculty of Informatics
University of Lugano
Lugano, Switzerland
E-mail: pezze@disco.unimib.it

Abstract: The integration of third-party web services can solve complex business problems and can reduce risks, costs and time-to-market. However, the task of the integrators is challenged by services that are maintained by different organisations, and that 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 taxonomy of integration faults that helps developers anticipate potential mismatches between discovered web services and applications, and design test cases and adaptors for each mismatch. A simple tool facilitates the deployment of a runtime infrastructure that embeds the test cases and the adaptors, automatically runs the test cases against newly discovered web services, uses the test results to diagnose service mismatches and executes the adaptors to overcome the revealed problems without user intervention.

Keywords: autonomic computing; self-adaptive service-oriented applications; integration mismatches; integration faults; web services.


Biographical notes: Davide Tosi received his BSc in Telecommunication Engineering (2000) at the Polytechnic of Milano, Italy, and an MSc in Computer Science (2002) and his PhD in Computer Science (2007) at the University of Milano Bicocca, Italy. Currently, Davide Tosi has a
1 Introduction

Web services and service-oriented architectures are emerging technologies for integrating enterprise applications, leveraging electronic Business-to-Business (B2B) and Business-to-Consumer (B2C) solutions and extending the life of legacy software. Web services are remote programs that are invoked over the internet, by means of standard protocols (e.g., Hyper Text Transfer Protocol (HTTP) and eXtensible Markup Language (XML)), for exchanging data between requesters and providers. Web services allow enterprises to export functionality outside the enterprise boundaries, thus enabling applications in different domains to rapidly and seamlessly integrate third-party expertise. For example, airlines and hotel chains can export services for online booking, so that travel agencies can combine these and other services to optimise travel plans. Service-oriented architectures solve complex business problems, decreasing risks, costs and time-to-market.

The integration of third-party web services is challenged by the difficulty of maintaining consistency between software systems that are maintained by different organisations and that may evolve dynamically and independently, because of either changes in the services or the dynamic discovery of new services. Service providers may change the implementation independently from clients, for example, to correct faults or meet new requirements. Clients can locate services dynamically, using service discovery mechanisms that allow them to discover and connect web services based on machine-readable descriptions. They may use different web services in different invocations: usually, a broker matches client requests with available services published by providers (according to common protocols, like Simple Object Access Protocol (SOAP), Web Service Description Language (WSDL) and Universal Description
Towards autonomic service-oriented applications

Discovery and Integration (UDDI), and every time applications reconnect to web services through brokers, they can get different matches and thus use different implementations of the requested services (Dabrowski and Mills, 2002).

Highly available requirements and dynamically discovered web services exclude the possibility of traditional stop-update-test-redoeploy-restart approaches to the integration of new or modified services. Autonomic and self-adaptive applications have been recognised as viable solutions for dealing with systems where size and complexity increase beyond the ability of humans to respond manually, coherently and in a timely manner to environment and system changes (Kephart and Chess, 2003). Autonomic and self-adaptive solutions are being experimented on in several application domains, but not yet extensively in service-oriented applications. The massive reuse of services and the frequent updates of implementations corresponding to compatible interfaces are typical of service-oriented applications, and allow one to define efficient domain-specific self-adaptive solutions for service-oriented applications.

In this paper, we focus on integration problems that derive from dynamic changes in the invoked services, and we propose a self-adaptive approach that uses test cases for revealing possible mismatches between requested and provided services, and dynamically selected adaptors for autonomously fixing integration problems accordingly.

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. 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, without indicating the measurement unit of the return temperature. This contract matches services that may return temperatures expressed in different measurement units, for instance Fahrenheit and Celsius, thus leading to client-side failures if the current measurement unit is wrongly interpreted.

Formal specification languages and semantic web technologies, for instance Semantic Web Service Description Language (WSDL-S) (W3C, 2005) and Ontology Web Language for Services (OWL-S) (DAML, 2004), can partially mitigate this problem, but suffer from the lack of generally agreed domain ontologies, and are of scarce help for services that come with incomplete or even missing specifications.

Our solution works for the client side without relying on additional server-side (meta-)information. We define a client-side runtime infrastructure that can be customised with test cases and adaptors: the infrastructure executes the test cases on services discovered at runtime, to reveal the presence of mismatches, and deploys suitable adaptors according to the test results; the deployed adaptors intercept calls to the target services and adjust the interactions as needed. A taxonomy of integration faults helps developers identify the weaknesses of interfaces that may lead to potential mismatches between discovered web services, and indicates suitable test and adaptation strategies for each type of mismatch. A simple tool facilitates customisation and deployment of the runtime infrastructure.
We draw on previous work in which we introduced the idea of using test cases and adaptors for solving integration mismatches in service-oriented applications (Denaro et al., 2005). Sections 2 and 3 exemplify the concerned integration mismatches and summarise the main aspects of our self-adaptive solution, respectively. In this paper, we instantiate the proposal in a tool-supported design methodology and report on the initial application experiences. The specific contributions of this paper are:

- a taxonomy of integration faults (Section 4) that helps service integrators identify possible integration mismatches, generate test cases for revealing integration problems and design suitable recovery actions. A taxonomy for mismatch identification was advocated but not developed in Denaro et al. (2005). The taxonomy reported in this paper classifies the sources of integration mismatches common in many web service applications.

- a prototype tool (Section 5) that helps integrators deploy test cases and adaptors in separate modules into the client applications

- an initial set of experience reports (Section 6) that shows how our approach supports the development of self-adaptive service-oriented applications in practice.

Finally, we survey related work in Section 7 and present our conclusions and final remarks in Section 8.

2 The service mismatch problem

In this section, we illustrate the problems that derive from dynamically evolving services with reference to a virtual store, an example that we use throughout the paper. VirtualStore interacts with brokers to dynamically discover remote stores and other services, for instance, credit card 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. Listing 1 shows an excerpt of the operations that characterise the specification of the request.

This WSDL description can match many WSDL specifications of services available from common UDDI repositories (e.g., Amazon.com), which differ in several small but important details. For instance, some providers offer a CartAdd operation that is able to manage multiple instances of the same item in a single add operation, while others do not support multiple instances. We used services that provide CartModify operations that remove an item by changing 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, that is, carts that lose their contents when sessions terminate (nonpersistent cart). All variants match the same WSDL interface.

Since VirtualStore can be connected to different shopping cart services at different times, we can experience failures due to the nonhomogeneous implementations of the target services.
Listing 1  Excerpt of the WSDL description of a shopping cart WS

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

  <!-- CartGet: Retrieves the list of items in a cart -->
  <operation name="CartGet">
    <input message="ws:CartGetRequest"/>
    <output message="ws:CartGetResponse"/>
  </operation>

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

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

  <!-- CartModify: Changes quantities, delete, or replaces items -->
  <operation name="CartModify">
    <input message="ws:CartModifyRequest"/>
    <output message="ws:CartModifyResponse"/>
  </operation>

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

  <!-- CartPurchase: Finalizes the purchase process -->
  <operation name="CartPurchase">
    <input message="ws:CartPurchaseRequest"/>
    <output message="ws:CartPurchaseResponse"/>
  </operation>
...
3 A self-adaptive approach

This paper investigates a self-adaptive approach for developing service-oriented applications. The approach combines novel techniques into a classic 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 1 illustrates how we instantiate a classic sense-plan-act control loop to deploy self-adaptive service-oriented applications. The invocation of a web service triggers monitoring mechanisms that check whether the invocation target is an updated or dynamically discovered service. Detecting a new service implementation triggers diagnosis mechanisms that run test cases on the target web service to reveal possible mismatches. If the diagnosis mechanism reveals mismatches, it triggers adaptation strategies that update the structure and the behaviour of the client application to solve the identified problems.

Figure 1 The self-adaptive control loop (see online version for colours)
Towards autonomic service-oriented applications

VirtualStore presented in Section 2, we can identify the set of services that comply with the WSDL cart specification in Listing 1. Different implementations of the WSDL specification may provide persistent and nonpersistent carts, that is, carts that keep or lose 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 2. The control loop customised with automatic test cases and adaptation strategies enhances all client invocations of the cart web service. Enhanced client applications intercept calls to the cart web service, dynamically check the persistence of the cart and adapt the local behaviour accordingly.

**Figure 2** The customised self-adaptive control loop for the cart persistency problem
(see online version for colours)

In line with test-based and self-healing approaches, we aim to solve some, but not all, problems. Solving all problems is not a realistic goal, but automatically solving even a few problems at runtime is an important improvement, since it reduces system malfunction and downtime.

4 Designing self-adaptive service-oriented applications

The customisation of the sense-plan-act control loops illustrated in the previous section is facilitated by a methodology and a design framework that help designers identify the customisation requirements for the control loops.

The methodology consists of three phases. In the first phase (setup), software architects analyse the specifications of the interfaces of the services selected for integration in the target application, and for each service specification identify ambiguities that may result in mismatches between different implementations. In the second phase (diagnosis customisation), software architects design test cases to reveal the occurrence of the potential mismatches identified in the setup phase. In the last phase
(adaptation customisation), software architects design adaptation strategies and associate them with the outcomes of the test cases of each service. Test cases and adaptation strategies are deployed in the application. When the application senses a change in a service, it automatically executes the associated test cases. If the test cases reveal service mismatches, the application locates and deploys the corresponding adaptation strategies.

Fault taxonomies facilitate the application of the methodology. A fault taxonomy serves as a checklist to focus potential mismatches, and indicates guidelines to define test and adaptation strategies for recurrent types of mismatches. Drawing on our experience with service-oriented applications, we have devised an initial taxonomy of service integration faults. Our fault taxonomy defines nine categories of mismatches, points to suitable test and adaptation strategies for each of them and provides examples.

The remainder of this section illustrates in detail the nine categories of our fault taxonomy.

4.1 C1 – Inconsistent interpretation of parameter or value

*Definition* – The semantics of input and output parameters or values may not be fully specified by the interface of the service, and this leaves the parameters or values open to inconsistent interpretation.

*Test suite* – The test suite can be defined by:
- identifying parameters or values that return distinguishable results for each possible interpretation
- invoking the service with the identified values
- checking the results.

*Adaptation suite* – The adaptation suite can be defined by designing a set of adaptors for each possible interpretation of the parameters or values. The adaptors intercept calls to and returns from the target service, and convert parameters and values as needed. When testing reveals an interpretation mismatch, the adaptation strategy dynamically selects the proper adaptor.

*Example* – Consider a web service that controls a camera based on three floating point parameters that indicate the position of the focus. Possible mismatches can derive from interpretations of the three values as Cartesian or polar coordinates. To reveal the use of polar instead of Cartesian coordinates, we can test the service by setting each parameter in turn to zero. No movement when executing one of the test cases indicates polar coordinates. Adaptors implement conversions between different types of coordinates, for example, from polar to Cartesian.

4.2 C2 – Inconsistent value domain or insufficient capacity/size

*Definition* – The syntax of service specifications does not provide information about either the domains of values that trigger different function modes or the capacity/size constraints, thus the structural constraints of the service are open to potential violation.
Towards autonomic service-oriented applications

Test suite – The test suite can be defined by:

- invoking the service with a representative value for each possible function mode and
capacity constraint
- checking the results.

Adaptation suite – The adaptation suite can be defined by designing a set of adaptors that intercept calls to the target service, identify the calls that violate value domains or capacities, and handle the violations by means of suitable priority policies. Nonviolating calls are simply forwarded. When testing reveals the existence of a domain or capacity violation, the adaptation strategy dynamically selects the adaptor that can handle the violation.

Example – Consider a web service that validates credit cards based on two strings that indicate credit card number and expiration date (a sample implementation can be found at www.trynt.com/trynt-credit-card-validation-api). Possible mismatches can derive from the domain of the credit card number, which may include either all possible types of credit card issuers or only a subset of them, and may or may not include a validation code at the end of the card number. To reveal limits in the supported credit card issuers, we can test the service with different credit card numbers, which represent different types of credit card issuers. Adaptors filter the non-supported credit card issuers to signal domain mismatches.

4.3 C3 – Inconsistent side effect on parameters or resources

Definition – The syntax of service specifications does not provide information about the side effects of operations, and this leaves the parameters or resources of the service open to unexpected side effects.

Test suite – The test suite may be defined by:

- invoking the service with values that may cause perceivable side effects
- checking the results.

Adaptation suite – The adaptation suite may be defined by designing a set of adaptors that either mask or fix the side effects. When testing detects side effects, the strategy dynamically selects the corresponding adaptors.

Example – Consider the web service that validates credit cards. It may or may not reduce the available credit as a side effect of the validation (this may lead to exhausting the credit without completing all of the requested transaction). To reveal whether the service reduces the available credit as a side effect of the credit card validation, a test case may try to validate a credit card twice with single values within the credit limit, and with the value of each request within the limit, and the sum of the two values that exceeds the limit. Adaptors can buffer the access to the credit validation and postpone the validation at the end of the transactions, to reduce the risk of credit overflow.
4.4 C4 – Missing or inconsistently implemented functionality

**Definition** – Operations exposed by service specifications may or may not be implemented by the service, or service functionalities may be inconsistently interpreted by software integrators.

**Test suite** – The test suite can be defined by:
- identifying cases that produce distinguishable results depending on the presence or interpretation of the functionality
- running the service for the identified cases
- checking the results.

**Adaptation suite** – The adaptation suite can be defined by designing a set of adaptors that implement the missing functionality or convert between different possible interpretations. When testing reveals missing or inconsistently interpreted functionality, the adaptation strategy dynamically selects the corresponding set of adaptors.

**Example** – Consider a web service that controls the access to a restricted area based on an access-identifier. The service may or may not keep track of the status of the access-identifier, and thus allow an access-identifier to re-enter the restricted area without leaving the area in between. To discover whether the service keeps track of the status of the access-identifier, we can try to enter the reserved area twice with the same access-identifier without leaving the area in between. An adaptor can keep track of the access-identifier status and warn the user about the risks of using different access-identifiers to enter and exit the area.

4.5 C5 – Underspecified nonfunctional characteristic

**Definition** – The syntax of service specifications does not fully specify nonfunctional requirements, thus leaving the nonfunctional requirements of the service open to potential violations.

**Test suite** – The test suite can be defined by:
- identifying cases that reveal different nonfunctional characteristics of the services
- running the service for the identified cases
- checking the results
- inferring the corresponding nonfunctional service profile.

**Adaptation suite** – The adaptation suite can be defined by designing ad hoc adaptors that relax the violated nonfunctional requirements to meet the nonfunctional profile of the target web service. When testing detects violations, the strategy dynamically selects the corresponding set of adaptors.

**Example** – Consider a composite service that invokes several elementary web services for booking a complete trip (flight, hotel, car, etc.; a sample implementation can be found at www.kayak.com/labs). Each elementary web service may be offered by different providers with different response times. The order of invocation of the
elementary web services may violate the performance requirements of the composite service. To discover whether the choice of the elementary web services violates the response time requirements of the composite service, we can infer the response time of the different providers with suitable test cases, and estimate the total average response time. Adaptors can attempt to dynamically discover other services while satisfying performance characteristics.

4.6 C6 – Inconsistent interaction protocol

**Definition** – Service specifications do not fully specify the sequence in which the operations of a target service can be invoked, thus leaving the interaction protocol of the service open to violation.

**Test suite** – The test suite can be defined by:
- identifying sets of operations that admit inconsistent sequences of calls, and generating a set of test cases accordingly
- running the service for the identified cases
- checking the results.

**Adaptation suite** – The adaptation suite can be defined by designing a set of adaptors that map the interaction protocol used by the client application to the interaction protocol of the target service. When testing reveals a violation of the interaction protocol, the adaptation strategy dynamically selects the adaptor that provides the correct mapping.

**Example** – Consider a web service that manages online orders, and stores items selected by the customer (a sample implementation can be found at http://aws.amazon.com). The service may or may not require the creation of an empty cart before adding the first item, thus resulting in potentially incorrect interactions. To discover whether carts must be explicitly created before adding any item, we can try to connect to the service and add an item without creating the cart. An adaptor can map the client invocations to the order of invocations required by the shopping cart web service.

4.7 C7 – Inconsistent cardinality of parameters in requests/responses

**Definition** – The service specification does not specify the mandatory characteristic of parameters, or the order of parameters for each operation, thus leaving the cardinality of parameters open to potential violations.

**Test suite** – The test suite may be defined by:
- identifying ambiguous alternatives of the number and order of parameters
- invoking the service for each identified alternative
- checking the results.

**Adaptation suite** – The adaptation suite may be defined by designing a set of adaptors that map client calls to target services with different numbers and orders of parameters. When testing reveals a violation in the number or order of the parameters, the adaptation strategy dynamically selects the adaptor that provides the correct mapping.
Example – The WSDL specification of the Google web search service defines the available queries with a generic set of parameters, resulting in possible incorrect service invocations. To discover the mandatory input parameters for a web search, we may try to use the service with the set of input parameters required by clients. An adaptor can complete the set of mandatory parameters with default values.

4.8 C8 - Missing or inconsistent failure propagation

Definition – Specifications do not always fully describe the failure handling properties of a service, thus leaving open the possibility of invoking exception handlers that may or may not be implemented, and of propagating exceptions and error codes inconsistently.

Test suite – The test suite may be defined by:
- identifying possible alternatives for exception-handling mechanisms, and exceptions and error codes
- invoking the service for each identified alternative
- checking the results.

Adaptation suite – The adaptation suite may be defined by designing a set of adaptors that handle the missing exceptional cases, or convert between different possible interpretations of exceptions and error codes. When testing reveals missing or inconsistent failure propagations, the adaptation strategy dynamically selects the adaptor that provides the correct interpretation.

Example – The Yahoo Marketing web service enables developing customised applications within Yahoo. The specification of the Yahoo web service does not define exceptions for the operation deleteReport in cases of incorrect invocation. To reveal the exception, we may call the service with well-known incorrect parameters. An adaptor can mask and handle the exception as needed.

4.9 C9 - System software incompatibility

Definition – Service specifications do not always define system software requirements, thus leaving systems, including operating systems, communication protocols and libraries, open to potential incompatibilities.

Test suite – The test suite may be defined by:
- identifying possible alternatives of system software characteristics
- invoking the service for operations that return distinguishable results for each possible alternative
- checking the results
- deriving the corresponding system software profile.

Adaptation suite – The adaptation suite can be defined by designing ad hoc adaptors that locally solve the detected incompatibilities. When testing reveals a system software incompatibility, the adaptation strategy dynamically selects the adaptor that fixes the problem.
Example – Consider a web service that accesses a remote file system for storage purposes (a sample implementation can be found at http://developer.nirvanix.com/). The format of the file paths depends on the remote operating system (for example, it is different for Windows and Linux), thus resulting in potentially incorrect interactions with the service. To discover the current path format, we can try to specify paths according to the formats of the different operating systems. An adaptor can rewrite the paths accordingly.

5 Prototype implementation

The implementation of self-adaptive service-oriented applications is facilitated by the Self-Healing Integrator for Web Services (SHIWS) framework. SHIWS automatically builds the runtime infrastructure that links the client application to the target web services, incorporating the diagnosis and adaptation strategies provided by designers. The runtime infrastructure handles the logics of the customised self-adaptive control loops at runtime, and supports dynamic selection and enactment of different adaptors.

We implemented a SHIWS prototype as a plug-in of the Eclipse open platform, an extensible development Integrated Development Environment (IDE) widely used in many domains. Since Eclipse is increasingly used to support software development throughout all phases of the development process, it is an excellent joint point for introducing a novel approach in the software process of service-oriented applications.

The current SHIWS prototype works for web applications written in Java.

5.1 The SHIWS Eclipse plug-in

At the Graphical User Interface (GUI) level, our prototype augments Eclipse with two new commands, Add new web service and Generate self-adaptation, and a new development perspective, the control loop customisation perspective.6

The command Add new web service facilitates the inclusion of new web services in a project. The developer is required to provide the URL of a WSDL specification that describes the characteristics of the included web service. SHIWS retrieves the WSDL description and parses it to identify the operations and parameters of the web service. The main effect of the command Add new web service is the initialisation of the artefacts for designing a customised self-adaptive control loop for the included web service. Such customisation artefacts consist of two files, which can be completed by the developer in the control loop customisation perspective.

The first customisation file contains the test cases, designed to reveal the occurrence of potential mismatches in the web service. Each test case is specified as a separate method, using the JUnit assertions library to signal the failure or success of the test. The second customisation file contains the adaptation suite, i.e., the set of modules designed to adapt different implementations of the web service. In the signature of test methods, we introduce the use of two new keywords, onSuccessSelects and onFailureSelects, which allow developers to specify the adaptors to be triggered when the test case succeeds or fails.

When the customisation is complete, the command Generate self-adaptation builds the corresponding runtime infrastructure. The runtime infrastructure consists of a set of automatically generated Java classes that, compiled with the application, provide the self-adaptive behaviour.
5.2 The runtime infrastructure

Figure 3 illustrates the high-level architecture of the typical runtime infrastructure generated by SHIWS for controlling the customised sense-plan-act control loop of a target web service.

**Figure 3** Architecture of the runtime infrastructure

![Architecture of the runtime infrastructure](image)

The *Wrapper* mediates the interactions between the application and the web service. It dynamically monitors the current service to check if it has changed since its last invocation. The *Wrapper* detects changes in the service provider by comparing the provider details between subsequent invocations of the service, and detects changes in the implementation of the services offered by the same provider by relying on version tags exported by the service. The *Wrapper* is built automatically starting from the WSDL description of the target web service.

The *Wrapper* delegates web service invocations to adaptation modules that are responsible for handling the adaptation of the web service. The proper adaptation module may vary for different web service implementations. The infrastructure allows the actual adaptation modules to be selected at runtime. Moreover, the infrastructure allows adaptation modules to be composed. The adaptation modules are provided by designers in the customisation phase.

When the *Wrapper* detects a change in the service implementation, it notifies the *TestDriver*. This component is built automatically based on the JUnit framework and embeds the test suite designed in the customisation phase. The *TestDriver* runs the test cases, and based on the embedded `onFailureSelects` and `onSuccessSelects` logic, selects the adaptation modules that best fit the characteristics of the current web service. Then, it reconfigures the infrastructure to use the selected adaptation modules.
6 Preliminary experimental results

In this section we illustrate the whole methodology with a simple example, and then we present the preliminary evaluation results on two industrial cases.

6.1 A tutorial example

Consider a simple application that provides statistical data about weather conditions, and uses web services to collect climate data in selected locations (we call it World-weather). The application uses a web service that provides the temperatures across the USA. This service is invoked with the zip code of a US location, and returns the corresponding temperature. For instance, at the time of writing, the portal xmethods.net provides a sample demo web service. Listing 2 reports the most relevant part of the WSDL description of the web service.

Listing 2 Excerpt of the WSDL description for a Weather-Temperature web service

```
...  
  <!-- getTempRequest message defines the input parameter zipcode -->
  <message name="getTempRequest">
    <part name="zipcode" type="xsd:string" />
  </message>

  <!-- getTempResponse message defines the output parameter return -->
  <message name="getTempResponse">
    <part name="return" type="xsd:float" />
  </message>

  <!-- getTemp operation -->
  <operation name="getTemp">
    <input message="tns:getTempRequest" />
    <output message="tns:getTempResponse" />
  </operation>

  ...  
```

The `message` tags specify two messages: the message `getTempRequest` that defines the parameter `zipcode` of the type string, and the message `getTempResponse` that defines the parameter `return` of the type float. The `operation` tag specifies the service interface that consists of the operation name `getTemp`, the input parameter as defined by the message `getTempRequest` and the return value as defined by the message `getTempResponse`.

We now illustrate how the methodology proposed in this paper facilitates designing the integration with the category of services that complies with this interface.

Based on our fault taxonomy, we identified a possible integration mismatch of the type ‘Inconsistent interpretation of parameter or value’ that may originate from the incompleteness of this service description. In particular, we identified the possibility of inconsistent interpretations of the return value: the service returns a
floating point value that can be interpreted according to different measurement units (in the example, we considered Fahrenheit and Celsius degrees), thus leading to possible inconsistent interpretations.

For diagnosis and adaptation, we customised the self-adaptive control loop. We designed a set of test cases that can reveal the reference measurement unit, and a set of adaptors that convert different measurement units. We distinguish the Fahrenheit from the Celsius degree by sampling the temperature of locations with seasonal temperature regularity. For example, values between 9 and 45 in San Diego in summer suggest temperatures expressed in Celsius, while values greater than 45 suggest Fahrenheit.

The top of Listing 3 (Lines 1–7) shows the test case that implements this test strategy (92101 is the zip code of San Diego). The stub object TemperatureWS is automatically generated by SHIWS based on the interface of the web service. It simplifies service invocations within the test cases, hiding the details of the SOAP protocol. The assertion verifies that the result is in the expected range.

**Listing 3** Test cases and adaptors designed for the temperature web service

```java
1 public void testTemp()
2 onSuccessSelects AdaptorCelsius
3 onFailureSelects AdaptorFahrenheit {
4    float temp;
5    String zipCode="92101";
6    temp = TemperatureWS.getTemp(zipCode);
7    assertTrue((temp>=9)&&(temp<=45));
}
8 public interface ITemperatureWSAdapt
9    extends ITemperatureWS {
10   public class AdaptorCelsius implements ITemperatureWSAdapt {
11      private static AdapteeWS adapteeWS;
12      public AdaptorCelsius(AdapteeWS adap) {this.adapteeWS = adap;}
13      public float getTemp(String zipCode) {
14         float temperature = adapteeWS.getTemp(zipCode);
15         return temperature;
16      }
17   }
18   public class AdaptorFahrenheit implements ITemperatureWSAdapt {
19      private static AdapteeWS adapteeWS;
20      public AdaptorFahrenheit(AdapteeWS adap) {this.adapteeWS = adap;}
21      public float getTemp(String zipCode) {
22         float temperature = adapteeWS.getTemp(zipCode);
23         return (float)(((temperature.floatValue())-32)/1.8);
24      }
25   }
```

The bottom of Listing 3 (Lines 8–23) shows the adaptors that are selected according to the result of the test. The adaptors are connected to the web service (AdapteeWS) and implement the interface ITemperatureWS. Drawing on the assumption that the client application works with Celsius degrees, the task of the two adaptors is straightforward:
AdaptorFahrenheit converts values coming from the web service from the Fahrenheit to the Celsius degree, while AdaptorCelsius forwards the unmodified result back to the application.

6.2 Experience report

We used the methodology proposed in this paper and the SHIWS prototype for designing two applications, VirtualStore and the Personal Mobility Manager (PMM).

6.2.1 VirtualStore

VirtualStore is a web portal for e-commerce applications (hereafter called backend e-stores). VirtualStore searches for best offers through the catalogues of backend e-stores, provides the abstraction of a shopping cart that can collect products from different backend e-stores and handles payments by means of credit cards to multiple backend e-stores in a single user transaction.

Self-adaptation requirements arise because VirtualStore manages the communication with the backend applications through a common set of web services for browsing product catalogues and handling remote shopping carts, and because VirtualStore uses third-party web services for credit card validation.

As for the product catalogue and the shopping cart web services, our implementation uses interfaces compatible with the Amazon web service (an excerpt of the referred shopping cart interface is described in Listing 1). As for credit card validation, we referred to the available web services published by Imagination Software,8 and TPISoftware.9 For the target categories of web services, we identified 11 possible mismatches based on the fault taxonomy described in Section 4 (five mismatches of type C2, one mismatch of type C3, three mismatches of type C4 and two mismatches of type C6). We then identified a total of 24 test cases that can identify the mismatches. Finally, we implemented 11 adaptors, one for each mismatch, to satisfy the adaptation requirements. Details on mismatches, test cases and adaptors have been omitted for space reasons and can be found in Tosi (2006).

6.2.2 PMM

The PMM is an advanced navigation system presented at the 2006 IEEE Service Computing Contest (Lorenzoli et al., 2006). PMM provides a highly configurable and customisable meta-application that can satisfy several user needs. PMM searches for optimal combinations of transportation according to local situations, such as traffic level, weather conditions and opening hours. PMM interacts with broker entities to customise the client application by dynamically discovering remote services. When final users need specific services to address specific contextual problems, PMM asks the broker for providers that satisfy the users’ needs and requirements.

We designed a self-adaptive PMM for seven categories of web services: global weather, weather forecast, environmental sensors, traffic alert, ticket purchase, map provider and authentication-authorisation-accounting services. For these categories of web services, we identified 24 possible mismatches, designed 66 test cases that could disambiguate the mismatches and implemented 24 adaptors, one for each mismatch, to satisfy the adaptation requirements (Lorenzoli et al., 2006).
To validate the correctness and performance impact of the self-adaptive behaviours, we experimented on the PMM with different implementations of the same service categories:

- two different implementations of the authentication-authorisation-accountingWS, one provided by ArcWeb and one developed in our lab
- two different implementations of the EnvironmentalSensorsWS, both developed in our lab
- two different implementations of the MapProviderWS, one released by ArcWeb and the second one from devnet.map24.com
- one implementation of the TrafficAlertWS, provided by ArcWeb
- two different implementations of the TicketPurchaseWS, both developed in our lab
- two different implementations of the WeatherWS, one provided by www.webservicex.net, and one implemented in our lab
- two different implementations of the WeatherForecastWS, one provided by ArcWeb and one implemented in our lab.

The experiments revealed that the self-adaptive PMM could successfully deal with all target web services with no significant performance degradation.

Table 1 summarises the statistics of our approach on the three baseline applications. The first column (Application) lists the three experimented applications. The second column (Service) lists the number of services that are integrated into each application. The remaining columns list the number of identified mismatches, designed test cases, and developed adaptors, respectively.

<table>
<thead>
<tr>
<th>Application</th>
<th>Service</th>
<th>Mismatch</th>
<th>Test case</th>
<th>Adaptor</th>
</tr>
</thead>
<tbody>
<tr>
<td>World-weather</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>VirtualStore</td>
<td>3</td>
<td>11</td>
<td>24</td>
<td>11</td>
</tr>
<tr>
<td>PMM</td>
<td>7</td>
<td>24</td>
<td>66</td>
<td>24</td>
</tr>
</tbody>
</table>

6.3 Lessons learned

The experiences with VirtualStore and PMM highlight some important characteristics of our approach:

- Separation of concerns – In our approach, the adaptation issues of each web service are handled in separate modules. They are designed and developed independently from both other services and the main functionality of the client application. From a design viewpoint, we found the approach extremely beneficial, since it allows one to focus on the main functionality and the adaptation problems at different times.
• **Automatic generated infrastructure** – Much of the infrastructure for controlling the dynamics of self-adaptive loops is automatically generated by SHIWS. This eliminates some technical difficulties in the integration, such as handling the communication via SOAP primitives, checking changes in the service provider or implementation, executing test cases and activating adaptation modules as needed.

• **Fault taxonomy support** – The taxonomy of integration faults guides designers through the analysis of the adaptation requirements. It works as a checklist that focuses on typical mismatches, and suggests candidate test cases and adaptation patterns. The fault taxonomy seems the natural joint point for integrating new domain knowledge in the approach.

• **Reusable adaptation modules** – The loose dependency between adaptation modules and client applications indicates that the same adaptation modules can be (at least partially) reused across applications that integrate similar web services. This suggests the possibility of designing libraries of standard adaptation modules.

• **Performance issues** – Checking for large amounts of potential mismatches can have a perceivable impact on the interaction between applications and services, especially if distributed over the network. In our approach, test cases are executed only when service implementations change, and may prevent runtime failures that impact negatively on the dependability of the application. Trading performance for dependability is often acceptable, especially if performance is only seldom affected, and dependability threats may prevent access to the desired functionality. To reduce the impact on performance, test cases may be executed offline. Test cases can be executed offline in at least two cases:
  1. when services are statically bound to the applications, and thus self-healing strategies can be executed at deployment time
  2. when providers or brokers are aware of the self-healing strategies and can thus execute test cases before the applications access the services. This last option requires changes to the infrastructure.

### 7 Related work

The feedback loops applied to monitoring the characteristics of a computer system and keeping application parameters under control relate to recent initiatives on autonomic computing and self-managed systems (Kephart and Chess, 2003). In the last few years, researchers have investigated this notion in many application domains with different goals and approaches. These included self-configuring software architectures (for example, Cheng et al., 2004; Oreizy et al., 1999; Valetto and Kaiser, 2003), self-adaptive use and allocation of computing resources (for example, Cobleigh et al., 2002; Poladian et al., 2004), self-healing mechanisms (for example, Dabrowski and Mills, 2002) and programming models inspired from biology (for example, George et al., 2002). Our work exploits the ideas of autonomic computing and self-managed systems to guarantee the interoperability of clients with remote web services that have many independent implementations.
Unexpected web service interactions can be monitored and diagnosed by means of assertions. In general, assertions and testing represent complementary detection mechanisms. Testing samples the service function with specific inputs that lead to distinct outcomes for each possible implementation, while assertions could fail to distinguish among different implementations for values that satisfy the contracts of the considered implementations. For example, testing can identify implementations that use either the Fahrenheit or Celsius degree with ad hoc samplings, while assertions might accept wrong interpretations for values that are valid for both measurement units. Assertions outperform testing for dynamically detecting inadmissible behaviours outside of the sampled cases, for example, an implementation that suddenly returns too high or too low temperatures because of failed sensors.

Baresi and Guinea use assertions (in combination with orchestration languages) to monitor whether or not a web service complies with a predefined functional contract. The developers of client applications specify the expected functional contracts with suitable assertions. The assertions are checked at runtime against the responses from web services. The identified contract violations trigger the adaptation strategies, such as the selection of alternative services (Baresi and Guinea, 2005). The approach of Baresi and Guinea is complementary to ours. Their approach aims to increase fault tolerance to contract violations, while our approach allows the application to adapt to the diverse valid implementations that may derive from underspecified elements in the service description.

Research in service verification and services composition aims to identify formal methods for checking and verifying the automatic integration of web services. For example, both Foster et al. (2004) and Beyer et al. (2005) propose approaches that use automata to describe the interactions among web services. These approaches do not consider the complete life cycle of the service, thus missing important problems deriving from the high degree of evolution that characterises Service-Oriented Architecture (SOA). For example, none of these approaches describes how to identify and solve interface mismatches deriving from implementation changes that are not directly observable in the service specification.

Other approaches to solve semantic ambiguity in web service descriptions focus on ontologies and semantic web technologies (for example, McGuinness and Van Harmelen, 2006). These approaches aim to build common reference frameworks to share and reuse data across applications, enterprises and community boundaries. The hindrance to using these approaches in practical applications is the difficulty of defining generally agreed domain ontologies. Nonetheless, based on these efforts, technologies for semantic web services have emerged in the last few years as an attempt to enrich web service languages with ontological annotations, both to facilitate discovery and to lower interoperability barriers (Roman et al., 2005; Haller et al., 2005; Solanki et al., 2004; Paolucci and Wagner, 2006). Here we discuss the Web Service Modelling Ontology (WSMO)/Web Service Modelling eXecution environment (WSMX) stack of technologies that explicitly address web services adaptation.

The WSMO (Roman et al., 2005) enables semantic descriptions of both web services and client goals. These descriptions can be matched to facilitate service discovery and interaction. The authors of WSMO recognise that heterogeneity naturally arises in open and distributed environments, and propose mediators to address mismatches that can derive from the heterogeneity of resources that ought to be interoperable. The typically foreseen tasks of mediators include translating reference ontologies, adapting communication protocols and reconfiguring service
workflows. WSMO tools include the web service execution environment, WSMX (Haller et al., 2005), a software system that executes WSMO-based web services and the related mediators.

Technologies such as WSMO and WSMX, albeit promising, are still in their infancy. We acknowledge in them the merit of pointing out the necessity of coping with interoperability mismatches. However, they provide no guidance to identify likely mismatches and automatically diagnose their occurrence when applications are connected to new implementations of a web service. Current mediators are mostly limited to well-standardised adaptations, such as applying predefined mappings between ontologies and protocols. It remains unclear if general-purpose adaptations can be designed with these technologies.

Our approach proposes guidelines to identify mismatches and devise test cases to reveal them. Our adaptors exploit the full power of a programming language, and can realise general-purpose adaptations, including partial recovery and failure masking. We are currently investigating synergies between semantic web technologies and our ideas, with the aim to use ontological reasoning to complement testing-based diagnoses, and mediators to serve as part of the adaptation tasks.

8 Conclusions

Web services support the rapid and seamless integration of enterprise applications both within and outside the enterprise boundaries. Service discovery mechanisms and autonomous maintenance by different providers increase flexibility and extensibility, but also introduce new problems.

In this paper, we propose a self-adapting mechanism that enables service-based applications to diagnose potentially critical mismatches and select suitable adaptation strategies. The approach is based on the identification of potential service mismatches and the definition of specific diagnosis and adaptation mechanisms that are automatically deployed in a general framework, thus releasing developers from annoying technical details. The methodology applies ideas from autonomic computing and self-managed systems, while the framework is inspired by Computer-Aided Software Engineering (CASE) tools that automatically instantiate general infrastructures, leaving only the creative task to the software developers.

To avoid loss of integrity, we assume either no side effects on the services or the availability of sandbox execution environments, as increasingly provided by service providers, like in the case of our experiments.

The preliminary experimental results indicate the efficiency of the self-adaptive approach to overcome nontrivial mismatches. The degree of self-adaptability of an application depends on the quality of the fault taxonomy that guides the mismatch identification process. We are currently conducting extensive experiments on industrial benchmarks to investigate in depth the benefits of the approach and improve the fault taxonomy proposed in this paper.
References


Towards autonomic service-oriented applications


Websites


Notes

1 Contracts are usually expressed in standard machine-readable languages, like the WSDL (Christensen et al., 2001).
2 In this as well as in several other cases, we assume the availability of sandbox implementations of the services that can be tested without effects on the actual service.
6 In Eclipse, a perspective is a customised view of the IDE, which is tailored around a specific programming language or functionality.
7 Current web services do not always provide version information, but it is reasonable to expect that the increasing use of web services will foster such good practice. Examples of services that provide version information can be found at www.arcwebservices.com.
8 http://webservices.imacination.com