klase

Platform for the Composition of Web Services: Requirements and Architecture

ITC-irst
Università di Trento
Delta Dator
Università di Genova
Università di Roma

Abstract. The growth of service-oriented computing calls for powerful automated service composition, monitoring and verification capabilities. The availability of a variety of specification languages for web services introduce a further degree of complexity to the problem, requiring flexible platforms. In this deliverable, we discuss in detail the above requirements, which are at the basis of the design of the Astro platform, and the consequent architectural choices. This provides the foundation for the detailed software architecture presentation in D7.2.

Document Identifier	Deliverable D7.1
Project	MIUR-FIRB project RBAU01P5SS "Knowledge Level Automated Software Engineering"
Version	v1.0
Date	Dec. 12, 2006
State	Draft
Distribution	Public

Acknowledgements.

This document is part of a research project funded by the FIRB 2001 Programme of the “Ministero dell’Istruzione, dell’Università e della Ricerca” as project number RBNE0195K5.

The partners in this project are: Istituto Trentino di Cultura (Coordinator), Università degli Studi di Trento, Università degli Studi di Genova, Università degli Studi di Roma “La Sapienza”, DeltaDator S.p.A..

Chapter 1

Executive Summary

Web services are, in essence, internet-accessible functionalities that can be published, searched for, and executed remotely. Although they realize a simple concept, the adoption of web services is a powerful paradigm for a very large and relevant set of applications, such as e-Commerce, e-Government, e-Health, telecommunications. In all these areas, the availability of services on the web make it possible to realize prospects formerly impossible, e.g. to combine at real-time geographically dislocated businesses, or to remotely interact with government offices to obtain some authorizations.

However, the enactment of web services presents some difficult issues. First, in order to make use of different available services to exploit them in combination, it is necessary to study their protocols in detail, and interact with them accordingly, or to program an ad-hoc protocol that does that. In both cases, this requires a considerable effort for a human actor, and errors are likely to happen while interacting with the services or programming a novel one that orchestrates them. In particular, the latter choice allows reusing the effort for future interactions with those protocols, but requires a deep knowledge of the programming languages for web services, and even for skilled programmers, is a time-consuming and error-prone task. Therefore, automated support for *web service composition* is in order.

Moreover, in a distributed environment, where actors may be independently added, removed or reprogrammed, it may become necessary to verify that the available services obey some properties, to validate the persistency of their validity w.r.t. certain desiderata. This also strongly motivates the need to (automatically) pursue ad-hoc adaptations of existing services, which can therefore be submitted to an exhaustive testing a priori form their deployment.

Finally, even in case all services taking part to an interaction may have been previously verified, it is extremely important to be able to monitor their runtime interaction, so to check that the runtime behavior is adherent to the formal service specifications, and satisfies some QoS measures.

The requirements above hold no matter what the tools used to represent web services and properties to be verified and monitored. Indeed, a large variety of such tools are available, e.g. workflow languages, semantic languages, and logical languages.

In this deliverable, we spell out in detail the requirements for the Astro platform, which is intended to support the development and enactment of services in all its phases; this leads to an architectural design which is modular and flexible with respect to the language issues. In Deliverable D7.2, we describe in detail the way the architectural principles are instantiated within a software architecture.

The deliverable is structured as follows. In Chapter 2 we discuss the benefits and issues in web service architectures, and set up the background discussing the desiderata in this area. In Chapter 3 we spell out in detail such desiderata, which we take as the requirements for the Astro architecture. In Chapter 4, we introduce a reference scenario, which enables us to introduce a development process which our tool suite is meant to support in all phases In Section 5, we describe our architecture, under different points of view, and discuss how the choices taken for it s design impact in the realization of the aforementioned requirements. In Section 6 we wrap up by putting the development of the architecture within the context of the Astro project history.

Contents

Chapter 2

Introduction

Web services provide the basis for the development and execution of business processes that are distributed over the network and available via standard interfaces and protocols. The enactment of web services can provide huge benefits in major areas, in terms of economical gain, increased speed and quality of service, and broadened functionalities. For instance, in e-government, different (regional, national) customer services can be represented as service protocols, which can be combined to perform in real time otherwise lengthy authorization procedures. Similarly, in e-Commerce web services can be used to perform complex transactions that involve a variety of partners, to obtain a composed 'packet' for the user.

In all these cases, the success of the web service paradigm stands in the possibility of combining independently developed, deployed and published services, reusing an existing, and possibly very large, wealth of information. However, handcrafting web services that interact with existing components is a hard task, time-consuming and error-prone, requiring a deep knowledge of both the complex languages used to describe web services, and of the domain (i.e. of the service components into play). Moreover, in a distributed setting such as this, services may evolve independently, making it necessary to perform costly adaptation and maintenance of the combined processes.

For these reasons, the success of this approach relies in a crucial way on the existence of support tools that perform the *automated composition of web services* on the basis of some formally stated user requirements. Web service composition can be thought of as a particular case of software synthesis, and is therefore a very demanding task that requires the adoption of advanced techniques.

However, supporting web service composition is not enough to support the life-cycle of web services. *Automated verification* is another fundamental functionality for the development and maintenance of web services. Only by automated verification, indeed, it is possible to formally check that some independently developed service obeys some desiderata, or that a formerly composed service can obey to some requirements additional to the ones it was designed for. In a setting where partner services may (and often do)

evolve, this continuous checking/adaptation process must be thoroughly supported by effective verification techniques.

Finally, *automated monitoring* must be supported during the execution of web services. There are two main reasons behind this. The first is that the possibility of low-level communication problems, as well as the presence of malware or sniffer agents, may severely perturb the interactions between partner services. Therefore it is crucial to be able to readily detect such interferences, so to trigger appropriate recovery actions. Second, it is important to be able to obtain quantitative quality measures which are related to the actual data exchanged between services, and to the responses of partners during the interactions. For instance, if a business relies on requesting loan approvals to a bank service partner, and these requests turns out to fail 99 times out of 100, this may signal that there is something wrong in the way the request procedure is carried out, or that the guarantees required by such bank are overly strict; then, different actions can be carried out (formal verification of the procedure, or search of a better bank).

The three functionalities above are required independently of the tools used to develop and combine services, and of the possible languages and tools for describing and realizing monitoring and verification. Different standards and languages have been proposed to develop and combine web services. WS-BPEL(Business Process Execution Language for Web Services), WS-CDL (Web Service Choreography Description Language) and OWL-S are some of the most promising emerging standards for describing the behavior of the services. Monitoring and verification is usually performed on the basis of logically specified requirements; several logics have been proposed, in most cases based on describing behaviors as formulas of a temporal logic such as e.g. LTL or CTL [Eme90].

For each of these, a variety of automated tools and techniques have been deployed, mostly revolving around the idea of 'search': modeling a search space corresponding to a system's behaviors, and visit it to either build a new system, verify that some behavior is never met, or check that an observed behavior is compliant with some definition.

Our aim is to define an open and flexible architecture enabling us to be unbounded from both the specific definition languages used for composition, verification and monitoring, and from the (search) techniques used to implement them.

On the basis of such a general architecture, we deliver a reference implementation of the architecture for a given setting of languages and techniques. In particular, we will rely on WS-BPELas the language for specifying the services part of the composition, so to ease integration of our architecture with a vast set of standard tools such as Active BPEL, one of the most quoted suites, providing an open source WS-BPELengine and a freeware and extendable WS-BPELDesigner.

In this way, our instantiation of the architecture supports the business analyst in the whole design and maintenance process of complex services, defining a structured development process supported via a tool suite.

In the following, we first discuss the functional and non-functional requirements for

our architecture, then we ground them by making use of a reference scenario, before discussing the general architecture proper, and one particular instantiation related to a choice of languages for representing services and complementary components (e.g. monitors).

Chapter 3

Requirements

In this chapter we spell out in detail the functional and non-functional requirements underlying the architecture, which will motivate our architectural choices. More in detail, our architecture should support the following functionalities:

1. *Automated web service Composition*
   Intuitively, the Web Service Composition problem amounts to the extraction of an executable process, which interacts with some existing processes to realize a protocol that satisfies some user requirements. More specifically, we define Web Service Composition under the following assumptions:

   - the existence of a (fixed) set of services, which from here on will be referred to as the 'components' or 'partners' in the composition. We abstract away from the issue of how these services have been discovered, and from the related issue of advertising them over the network.
   - the publication of services in a way which may may abstract away from their actual realization, and in particular from the implementation of (portions of) their internal logics. This is an important factor to take into account: in many cases, service publishers may not be willing to disclose the logics that drive their decisions; e.g., the reasons that can lead a bank to refuse a payment may be related to some bank policy that the bank has no interest in disclosing. In this case, an external observer should understand that 'a choice will be taken by the service provider', with no further detail.
   - the existence of requirements that pose constraints over the behavior of the desired composed service w.r.t. to its user. In general, two different aspects are represented by separate components. A first portion of the requirement describes the expected service protocol in an abstract way, limited to defining the interactions between the composed service and its user. The properties that should be obeyed by such a protocol make up the second portion of the requirement, and need to be declared formally. We remark that the abstract

definition of the protocol can be obtained directly be a specular abstract definition of the expected user behavior, which can be simply understood as one additional partner in the composition.

- all the partner services communicate, during the execution of the composed service, exclusively with it. That is, the composed service acts as an 'orchestrator' in a star configuration. Notice that of course this does not preclude the usage of different instances of the partners within other orchestrations, nor of the same partner in another orchestration at a different time.

The assumptions above can be combined to provide a compact and more formal description of the problem of web service composition: *given a set of partner protocols $W_U, W_1, \ldots, W_n$ (of which W_U represents the user), which may include nondeterministic behaviors, and a formal requirement ρ over the behavior of their orchestration, web service composition must produce a deterministic protocol description which, acting as an orchestrator between $W_U, W_1, \ldots, W_n$, satisfies ρ.*

2. *Automated web service Adaptation*
In many cases, an existing orchestration of component services (proposed either by a human operator, or by an automated composition procedure) is rendered invalid by some modifications in one or more of the many elements taking part to it: some composition requirement, or the protocol established by some partners. While web service composition refers to a static situation, in general, the appearance of new services or the modification/removal of existing ones should trigger novel composition instances so that a novel service is built to realize the user requirement. In these cases, performing a novel composition from scratch is neither effective nor desirable, since the whole wealth of design effort spent for the previous orchestration is wasted away. It is preferable, instead, to take the existing orchestration as a reference point, and to adapt it to suit the new requirement and/or partner protocols by minimal changes. This, in turn, requires defining a notion of "minimal distance" between different orchestrations, and the adoption of specific adaptation techniques that may greatly differ from those used in composition. The setting of adaptation can therefore be given as follows: *given a set of partner protocols $W_U, W_1, \ldots, W_n$ (of which W_U represents the user), which may include nondeterministic behaviors, an existing orchestrator W and a formal requirement ρ over the behavior of the new orchestration, web service adaptation must produce a deterministic protocol description W' which, acting as an orchestrator between $W_U, W_1, \ldots, W_n$, satisfies ρ, and minimally differs from W amongst all services obeying the same constraint.*

3. *Automated web service Verification*
Intuitively, verification consists of exhaustively checking that the behaviors of some service or set of services obey some requirements. This can be spelled out in several variants, depending on whether services are considered in isolation, embedded in some environment, or as an interacting group. Here, we take the following view of the problem: *given a set of web services and a set of formal requirements, web*

service verification consists in checking each requirement against the set of services, considering every possible behavior of the group within some external environment, and either (a) confirming that the requirement is valid, or (b) produce a counterexample for the requirement, in the form of an admissible service behavior that contradicts it. We remark the importance of (b) for the purposes of debugging; in turn, this restricts the focus to a set of verification techniques, discarding others (e.g. those based on automated theorem proving).

4. *Automated Generation of web service monitors*
   Monitoring, in general, is concerned about the displacement of elements that intercept communications amongst different component services, and that analyze them to provide continuous measures to an external observer, or to signal that some formal requirements has been contradicted by the observed protocol. Different conceptions of monitoring are possible, based on the mechanisms for intercepting communication, and on the ability to state properties over single services or sets of them. We take the following view: *given a set of web services, a set of formal requirements and a set of measures, monitoring should automatically produce and deploy one or more monitor components that intercept communications so that (a) every measure is provided in a continuous way and (b) every violation of a formal requirement is signaled. Monitors must be independent from services, i.e. their presence should not affect the behavior of services nor their representation.* Therefore, we allow for complex requirements that may involve more than one service; moreover, we rule out all of those monitoring mechanisms that are based on modifying (annotating) the existing services, so to fully decouple the handling of services form that of their monitors.

The Astro platform aims not just at supporting all the activities above, but - perhaps even more important - to do it in the framework of a structured development process which includes design time and run-time activities. Typically, the starting situation is one where some description for the component services of an orchestration are available, having been located e.g. through some service registry, and downloaded from the web. Given these, and some user requirements for the composition, monitoring and verification, the platform must automatically, in order, (a) perform the required composition tasks, (b) perform every required verification over the elements of the orchestration(s), (c) generate every required monitor for the components of the orchestration(s), (d) deploy the generated code, intercept monitoring exceptions, and if needed, perform new orchestration/adaptation tasks, iterating the tasks. We call this iterative process the *development and runtime* life cycle of web service orchestration, and we add as an explicit requirement for our architecture to support it. Such a requirement hints at the necessity for a sort of scripting language that allows a user to describe the required tasks, and the actions to be Of course ease of development is a crucial keyword, and the actions to be undertaken depending on the possible results and exceptions, both during the off-line phase of development, and during run-time.

On top of this, the following non-functional requirements should be satisfied:

1. *Independence from languages*
   All the requirements above have been stated in a way which is independent from the language chosen to represent services, as well as from the languages chosen for composition, verification, and monitoring requirements. The architecture should be fully general and not commit to any specific choice in this sense.

2. *Scalability*
   A standard non-functional requirement is that the architecture should support the above functionalities in real situations, where several complex service components enter into play, and complex (composition, verification and monitoring) properties are used.

Chapter 4

A reference scenario

In this chapter, we introduce a reference scenario that, albeit simple, will ease our further discussions on the requirements and the way the architecture satisfies them. Also, we will use this scenario to discuss the development process which our architecture is meant to support.

Our reference 'virtual on-line shop' (VOS) scenario assumes the existence of a set of existing item selling services, and of a set of payment services. An item selling service can be invoked to buy an item, and first establishes a dialog with the user to communicate the availability of the requested item(s) and the price. An electronic payment receipt can be obtained by using one of the payment services; notice that also in this case, the protocol that needs to be established with them is not a simple query-and-response one. In particular, for our purposes, we consider a simple version of this scenario where only one Bank and a Shop services are available for payment and acquisition of items respectively. The Shop protocol goes as follows: the service is activated by a user request for an item; the service then checks for the availability of the requested item, and in case of unavailability, sends back a negative answer and terminates in a state logically associated to a failure of the transaction. If the item is available, the service decides on a price, and proposes it to the partner, giving also details on its bank account for the payment, suspending for either an acceptation or a refusal of the proposal. In case of refusal, the service simply acknowledges this, and terminates - again, in a failure state. Otherwise, prior to assume the acquisition has been successfully concluded, the service awaits for a receipt; if, vice versa, the service is signaled that no receipt is ready, the transaction terminates unsuccessfully. The Bank service goes as follows: it receives an amount of money, and two bank account numbers; it checks that the bank account numbers are correct (i.e. they are actually bank account numbers, and the first pertains to this bank), and it checks that moving the amount of money form the first account number to the second is viable. If not, it responds with a refusal, and restarts; otherwise, the money is moved, and a receipt is produced and given back as a result. Retrial is allowed only twice, after which the refusal is definitive and the service ends signaling that the transaction has been unsuccessful.

In such a setting, a useful provision consists of a Virtual On-line Shop, that is, a service that combines a sell and payment service to clients, offering them the ability to directly require an item, agree on the proposed price, and let the composed service take care of the payment and receipt handling, as well as of handling the various failures that may happen at each stage. Such a VOS would act as an orchestrator between some selling services, and some payment services, see Figure 4.1.

The handcrafted design and realization of a VOS is far from trivial, even in our case where its partner services are not too complex. It requires a full comprehension of their functioning, and a detailed implementation that e.g. specifies the correct types at the interfaces, and establishes communications using the right mechanisms (e.g. asynchronous vs. synchronous calls), correctly correlating different messages to form a consistent flow of information.

For this reason, we want to use automated web service composition; the problem instantiates as follows: *given (a) the abstract protocols for the Bank, the Shop, and the User and (b) a formal requirement ρ modeling that "if possible, the required item must bought from the User; otherwise, withdraw the item acquisition; and in any case, all involved partners must be consistently informed", produce an executable orchestrator that may be executed to interact with (any possible instantiation of) Bank, Shop and User and such that all executions satisfy ρ.*

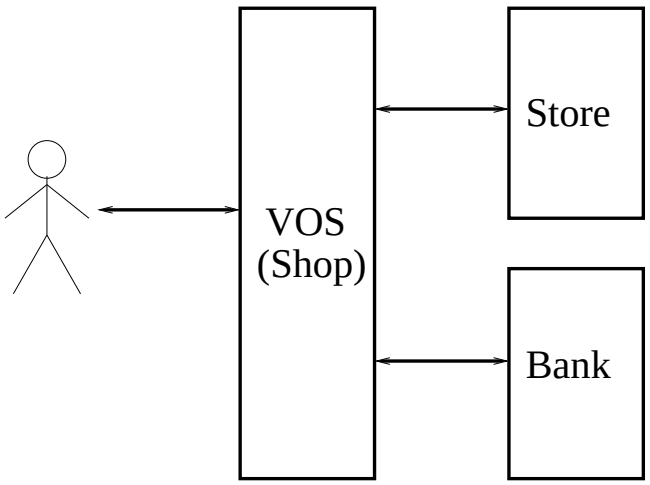

Figure 4.1: Virtual On-line Shop scenario

Once the VOS is built, it can be validated against several properties which are not immediate to detect; this is even more crucial if the VOS has been subject to some handcrafted modifications, e.g. to link it to a different Bank or Shop protocol. For instance, one important property that can be check is that the VOS is deadlock-free. Also, we want to be sure that the VOS does satisfy the intended requirement in that either all components succeed, or the transaction is rolled back uniformly for all components. Or again, we may check that the bank may both refuse and accept some User or Shop credentials.

It is also important that these properties can be monitored on line, as well as other properties that involve more than one of the interacting components. For instance, we may monitor that the credentials given by the a Store are always accepted by the a Bank. Or, we may monitor the QoS of (every instance of) a Store, counting how many times an item is unavailable or computing the average of refusals on the side of the customer.

Supporting all the activities above in the framework of the development and runtime process, for this scenario, means that the starting point is a situation where the abstract description of the available component services (Bank and Shop) are available; e.g., they have been located through some service registry, and downloaded from the web. At this stage, it is the task of the designer to provide an abstract description of the User protocol, that models the expected behavior of the User of a VOS. Once all partner descriptions are in place, the designer must formulate requirements for the composition, monitoring and verification, and insert them into the requirement script that triggers the various platform functionalities in turn: service composition, verification over the produced orchestrator (and the component services), generation of the monitors, and deployment of orchestrator and monitors.

Then, the platform must support the runtime part of the life cycle by acting upon the signaling of an exception by some monitor: the user should be able to recover the execution trace and either simulate it, or use it within the context of verification (e.g. using it as an assumed behavior of the services) or automated adaptation.

Chapter 5

Architecture

In this section, we describe our architecture, and discuss the way it satisfies the requirements and supports the development process we stated in the previous chapter. We will start with a high-level description of the architecture, based on a distinction between two different abstraction layers (views). We will then discuss each view in turn. Finally, we will discuss the way this architecture can be instantiated to support a set of chosen languages and methods, and to be integrated within a standard development suite, results in a usable and convenient implementation of our development process.

5.1 Top-level structure

The architecture we propose relies on a logical and an integration views as shown in Figure 5.1.

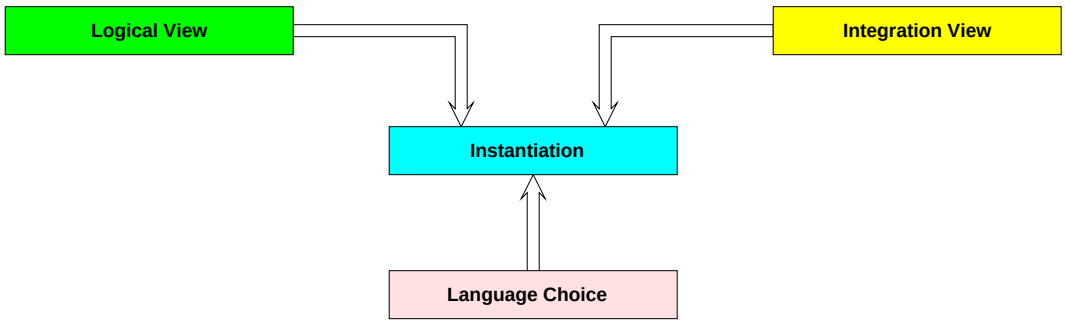

Figure 5.1: Architecture Levels

The logical view describes the architecture from an high level standpoint, simply in terms of inputs, outputs, and transformation relations. Therefore, this view completely abstracts away from language-specific issues, from the techniques chosen to handle them, and from the way these techniques are integrated in a sensible flow of development and

maintenance. The integration view focuses on the dynamics of the development process, situating the components of the architecture within a structured process that covers the whole life cycle of web services. The instantiation of the architecture relies on a specific choice of languages and techniques to deal with the issues presented in the two views above.

In the following, we detail each view in turn.

5.2 Logical View

Figure 5.2 shows the logical view of our architecture, which revolves around the crucial concept of using an intermediate language (IL from now on) to represent and manipulate stateful entities. We remark that the notion of stateful entity refers to several actors in our requirements: the component services in a composition scenario, the produced service orchestrator, and also the monitors in case monitoring is required. Notice that these entities are often represented using different concrete languages, which may support different features. For instance, the most widespread service languages support asynchronous communication and message correlations; for monitors, the key required ability is that of intercepting and interpreting messages. Composition and verification requirements, often expressed through some logics, can also be thought of a stateful entities, namely (non-communicating) automata. It is therefore crucial that IL is general and powerful enough to represent all the features of such diverse families of languages.

Given this, the logical view relies on two families of translation functionalities. In input, a translation phase is used to transform the inputs of (monitoring, composition, verification) functionalities in IL format. In output, a translation phase is used to emit a result in the proper format, be it a composed orchestrator service, or a monitor.

Once elements are represented into IL format, all of the three functionalities can be thought of as transformations that extract a set of IL entities from a set of input IL entities. In particular:

- Automated web service Composition starts from a set of n IL entities that represent (abstract) service components, from an IL that represents the (abstract) desired user interface to the orchestrator, and from another IL that represents the behavioral requirements for the orchestration. As a result, it produces one IL that represents the (executable) orchestrator.

- Automated web service Verification starts from a set of n IL entities that represent (abstract and/or concrete) services that are meant to work together as an orchestration, and another IL that represents the logical requirement to be verified. It produces another IL that represents the counterexample for the property; conventionally, an 'empty' IL is produced to signal that no counterexample exists, i.e. that the property holds.

- Automated Generation of Monitors starts from a set of n IL entities that represent (abstract and/or concrete) services that are meant to work together as an orchestration, and from a set of m ILs that represent properties that need to be monitored. As a result, a set of p ILs are produced that represent monitors. Notice that p may be different both from m and n: a monitor may be built to monitor several properties at the same time, if these are associated to the same service(s).

We remark that in all cases, the transformations are based on a general idea of combinatorially visiting a *search space* which describes all the possible behaviors of the considered set of services. This allows reusing a wealth of techniques developed in different areas of AI, such as planning techniques, and symbolic power-set construction of automata.

Figure 5.2 shows the transformation process in the case of the composition: the set of (WS-BPEL, WS-CDL, OWL-S) component services that constitute the Business Domain is translated to a set of automata defined in IL; search techniques are applied resulting in a transformed STS, which is finally emitted in the target language.

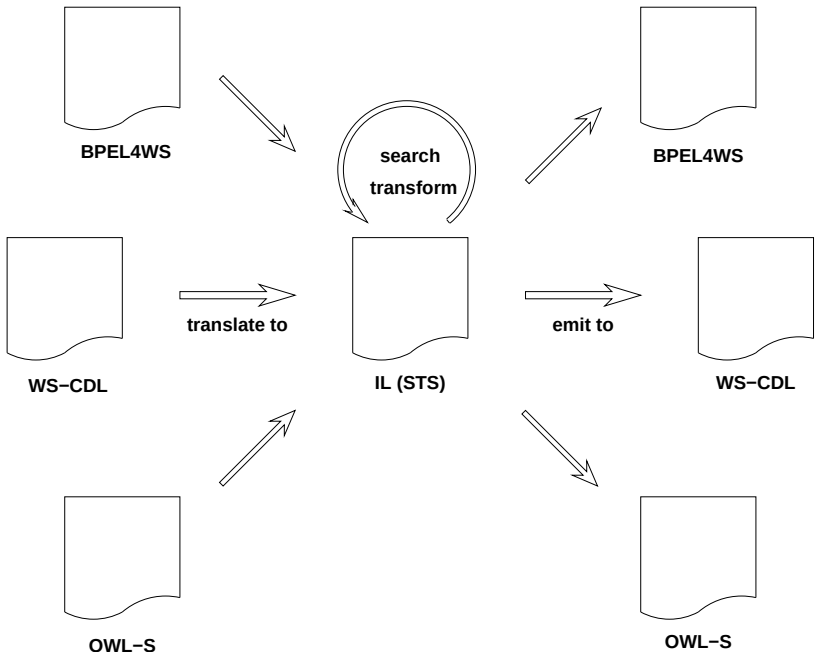

Figure 5.2: Logical level

Therefore, the Intermediate Language acts as a sort of LINGUA FRANCA for the application of a variety of search and transformation techniques, and to support multiple input and target languages.

5.3 Integration View

To obey the requirement to support a structured web service development process, the proposed architecture must be thought of as integrated within a tool suite which supports a designer/engineer in the definition and governance of the service composition, during all phases of the life cycle of the service. In particular, such a suite must comprise at least three components: a *designer*, allowing a user to describe web services using some standard language; a *deployer*, which is in charge of converting the web service descriptions in a format amenable to execution by some standard engine, and setting them up for execution; and finally an *execution engine*, which executes web services, handling communication between different services, catching exceptions and so on. Of course, the elements of the suite must be designed to work together in a consistent way; e.g., the deployer must refer to the specific execution engine, all components must refer to a unique web service description language, and each component must be able to share informations (e.g. on errors) with the others. Each of the components is a complex piece of code; e.g., usually, designers implement powerful GUIs to allow a user to describe services using a graphical description, and to help him/her to incrementally refine the description by filling up low-level details.

It should be evident by now that the development process which we refer to, where initially no orchestration exists, and finally an orchestrated service has been automatically composed, verified and deployed, together with monitors, needs to interact with all components of the suite in a variety of ways. In particular:

- The designer must be linked to the module implementing the composition functionality, to allow a user to design an abstract protocol of a desired orchestrator, and to edit some automatically generated one. Moreover, the designer must be enriched to allow the description of composition, verification and monitoring requirements, based on the specific languages chosen for these purposes. Notice that the integration should not consist solely in taking the designer's output as the input to composition (or verification or monitoring), but also in having the designer recognize feedback from such components and present it appropriately.

- The deployer must be integrated with the composition and monitoring functionalities, so that once the orchestrator has been automatically generated, it is appropriately set up to run and interact with the orchestrated components, and at the same time, monitors are made active.

- The execution engine must be extended to deal with the presence of the (automatically generated and deployes) monitors. This requires both handling the way communication is intercepted by the monitors, and presenting monitor-generated messages either to a user, or to the other parts of the architecture (e.g. to trigger adaptation and redeployment of an orchestration).

Such a complex integration of different tasks and techniques with different suite modules, needs to be suitably coordinated. This can be conveniently performed by a CONTROLLER component, which calls and coordinates the required tools. By programming the controller by means of a PROBLEM DESCRIPTION LANGUAGE, we can allow a user to flexibly describe not just which tasks should be performed, but also in which (partial) order, and which actions should be triggered by some results of runtime monitoring.

Figure 5.3 shows how the various components fit into this integration scheme; ideally, all of the components could be developed as separate plugins of a generic development platform, so that common interfaces and functionalities are factored out. One such emerging platform is Eclipse [Hol04].

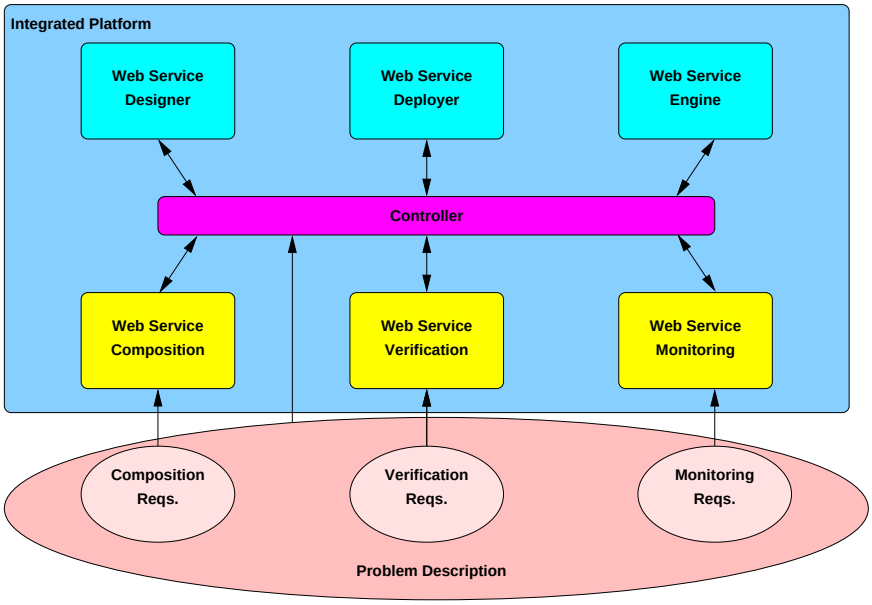

Figure 5.3: Integration View

5.4 An instantiation

Several different instantiations are possible for our architecture, depending on the concrete languages chosen to represent web services, to deploy monitors, to provide the intermediate representation used by the search mechanisms, and to allow a user to specify requirements. We here describe a reference implementation, that instantiates the architectural views discussed above for one such choice of languages. In particular, our reference implementation operates upon services expressed in WS-BPEL, a standard language for Web services that expresses the business logics in terms of procedural service behavior. WS-BPELis based on imperative style constructs combined with an asynchronous communication model, and relies on WSDL specifications for data and message types. WS-BPELsupports powerful constructs to correlate messages pertaining to a distributed flow of

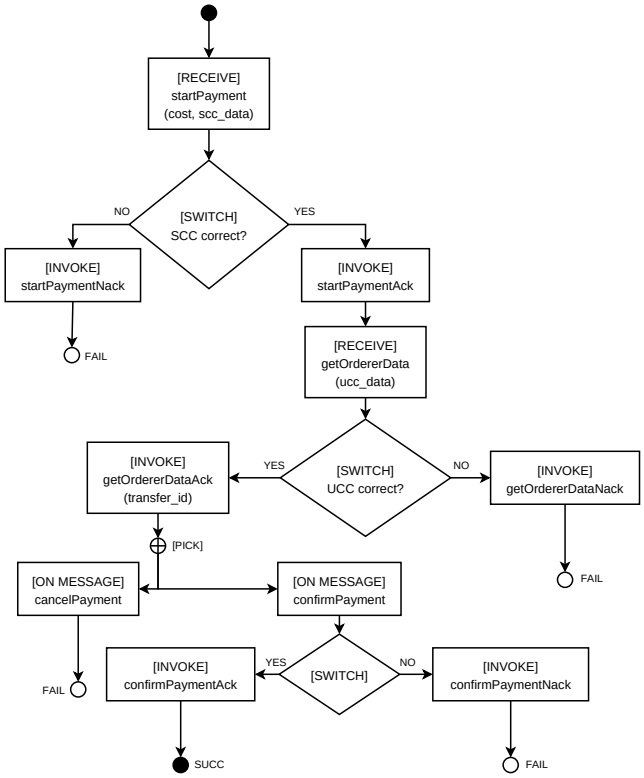

Figure 5.4: State Transition System of Bank Process

information, and includes mechanisms to deal with exceptions. One very relevant feature of WS-BPELis the ability to represent "abstract" as well as "executable" processes, therefore supporting the different levels of abstraction required in web service composition. Namely, in its "abstract" flavor, services are allowed to expose opaque choices that, by modeling nondeterministic behaviors, hide internal logics that the service provider may not be willing to disclose. Vice versa, "executable" WS-BPELdetails every computational aspect and can therefore be run using standard execution engines.

This means that, for instance, our vision of composition is based on having abstract WS-BPELprocesses as input, and aims at obtaining an executable WS-BPELas output.

For what concerns monitors, our choice in this instantiation is to have them implemented as Java objects, which can be then easily deployed in distributed architectures in such a way that they intercept messages between services without interfering with them. This choice is particularly convenient since most existing WS-BPELengines are also written in Java, therefore making the integration of Java monitors for the sake of message interception an easy task.

The counterexamples generated by the verification process, instead, are generated as XML-described sequences of messages, so that they can be easily browsed and analyzed using standard tools.

Finally, the problem description language is based on an XML description of the composition, verification and monitoring tasks, which all refer to component services. Each task is described as a logical formula, in XML format; in our architecture instantiation, we support the branching time CTL logics for all tasks, plus some task-specific logics for composition and monitoring. In particular, we support preference-based composition via the EaGLe logic, and we enrich CTL monitoring with the ability to provide quantitative statistics.

To represent all the features of the above imperative languages and logics, we adopt an intermediate language which is based on a rather general notion of State Transition System. In particular, our IL entities are annotated asynchronous automata which feature a blocking semantics for what concerns message receiving. This allows (a) a direct mapping of (a significant subset of) WS-BPEL, (b) a simple representation of monitors, and (c) easily evaluating the logical status of services against the specifications.

As an example, Figure 5.4 represents the STS of the WS-BPELBank of the WS-BPELprocess in our reference scenario. The transformation functions for the various functionalities, given the language choices depicted above, lead to the picture of Figure 5.5:

- a BPEL2IL transformation function is used to convert WS-BPELcode into IL entities;
- a IL2BPEL transformation function that emits an IL as a WS-BPELservice, associated with a WSDL definition of the interfaces;
- a IL2JAVA transformation function that emits an IL as a piece of Java code, which can readily be deployed as a monitor over a standard web service engine such as Active BPEL [Act].
- accessory transformation functions IL2XML, IL2DOT, which serve to emit IL-represented automata in formats better amenable to visualizing and browsing. The XML standard and the DOT graph representation are particularly suitable int that respect.

On top of these, the specific search techniques used in this instantiation of the architecture, rely on two base transformation functionalities. ILSET2ILSYNCH transformation is used to transform a set of IL entities, which represent independently evolving services, into an equivalent unique IL. The new IL, which in essence represents the synchronous product of the services, can be taken as the search space to more easily apply search techniques for composition; IL2ILPOWERSET is used to transform an IL-represented automata into its power-set, an automata that represents the knowledge that can be attained by an external observer of the automata based on observing its behavior. This standard functionality is at the core of monitor generation. Figure 5.5 gives an overview of the component required to defined the reference implementation.

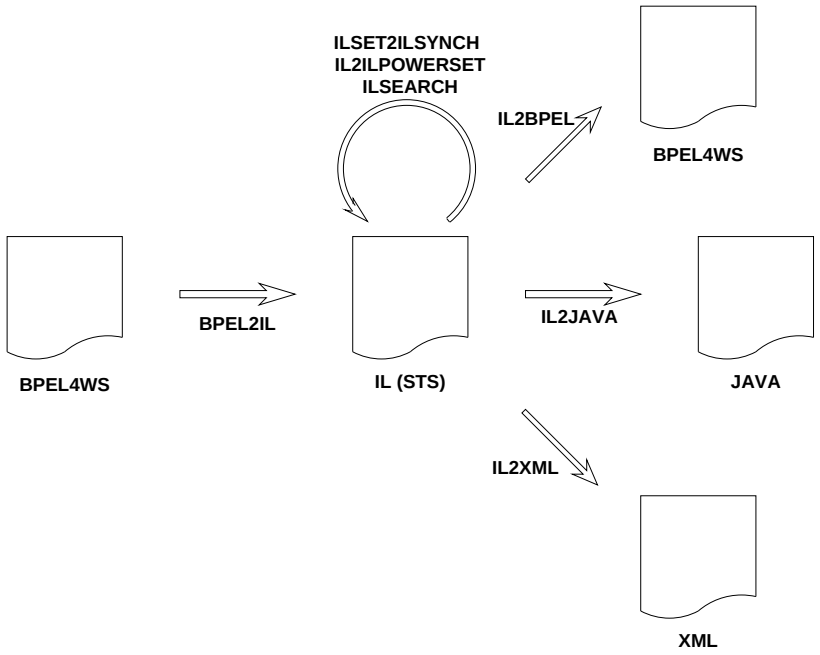

Figure 5.5: Physical level overview

Chapter 6

History of the Deliverable

Here, we describe how the work described in the deliverable has evolved along the 4 years of the project.

6.1 1st year

The first year of the project was mostly devolved to establishing the feasibility of adapting state-of-the-art search techniques to pursue automated composition and verification of web services. These first tests revolved on a semi-automated recasting of generic protocols directly into the native language NuSMV featured by the adopted search tools. Nevertheless, these tests highlighted already the need for an intermediate language that would ease the analysis tasks to debug and optimize the service representation against the search procedures.

6.2 2nd Year

At this stage, the ongoing research activity on verification and monitoring, as well as on composition, made it evident the possibility and advantages of designing a structured development process that would cover all phases in the life-cycle of web services. A technology acquisition phase was started to select which, amongst the possible service development platforms, could have been eligible as a basis for designing the ASTRO platform. Three platforms were evaluated: BPWS4J [BPW02], Collaxa, and ActiveBPEL. The first tests were performed using the Collaxa platform.

6.3 3rd year

In the 3rd year, we switched to the ActiveBPEL web service platform, mostly due to better copyright format under which this platform is distributed, to its integration within the Eclipse development platform, and to the availability of technical support. Moreover, we completed the design of our task description language, based on an XML format. This achievement made it possible to integrate our platform as an Eclipse plugin, therefore providing an integrated tool where all of the activities related to development and test of services can be exploited. This platform and its successive developments has been, since then, the core of our demos at international workshops and conferences such as ICAPS'05, ICAPS'06, ECAI'06 [MMG+05, MMF+06a, MMF+06b].

6.4 4th Year

In the 4th year, we thoroughly tested our platform over different scenarios, as witnessed in several publications [PTB05, PATB05, BTPT06, MPT06]. This led us to refine the capabilities of our platform and to ease its user interaction. For instance, our current instantiation allows for a variety of verbosity levels during the composition and verification phases, and we introduced a way to graphically analyze intermediate code. This activity is ongoing, as well as the refining of the underlying techniques, by considering more and more elaborate testbeds.

Bibliography

[Act] ActiveBPEL. The Open Source BPEL Engine - http://www.activebpel.org.

[BPW02] alphaWorks: BPWS4J Overview. Web page at http://www.alphaworks.ibm.com/tech/bpws4j, 2002.

[BTPT06] Fabio Barbon, Paolo Traverso, Marco Pistore, and Michele Trainotti. Run-Time Monitoring of Instances and Classes of Web Service Compositions. In *Proceedings of ICWS'06*, 2006.

[Eme90] E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*. Elsevier, 1990.

[Hol04] Steve Holzner. *Eclipse*. O'Reilly, 2004.

[MMF+06a] M.Trainotti, M.Pistore, F.Barbon, P.Bertoli, A. Marconi, P.Traverso, and G. Zacco. ASTRO: Supporting Web Service Development by Automated Composition, Monitoring and Verification. In *Proceedings of Demos, ICAPS'06*, 2006.

[MMF+06b] M.Trainotti, M.Pistore, F.Barbon, P.Bertoli, A. Marconi, P.Traverso, and G. Zacco. ASTRO: Supporting Web Service Development by Automated Composition, Monitoring and Verification. In *Proceedings of Demos, ECAI'06*, 2006.

[MMG+05] M.Trainotti, M.Pistore, G.Calabrese, G.Zacco, G.Lucchese, F.Barbon, P.Bertoli, and P.Traverso. ASTRO: Supporting Composition and Execution of Web Services. In *Proceedings of Demos, ICAPS'05*, 2005.

[MPT06] A. Marconi, M. Pistore, and P. Traverso. Specifying Data-Flow Requirements for the Automated Composition of Web Services. In *Proc. SEFM'06*, 2006.

[PATB05] M. Pistore, A.Marconi, P. Traverso, and P. Bertoli. Automated Composition of Web Services by Planning at the Knowledge Level. In *Proc. IJCAI'05*, 2005.

[PTB05] M. Pistore, P. Traverso, and P. Bertoli. Automated Composition of Web Services by Planning in Asynchronous Domains. In *Proc. ICAPS'05*, 2005.