

Verification and Validation of Web Service Compositions using Formal Methods



DISSERTATION FOR THE DEGREE OF DOCTOR OF COMPUTER SCIENCE TO BE
PRESENTED WITH DUE PERMISSION OF THE DEPARTMENT OF COMPUTER
SCIENCE, FOR PUBLIC EXAMINATION AND DEBATE

Author: *José Antonio Mateo Cortés*
Supervisors: *Dr. Valentín Valero Ruiz*
Dr. Jiří Srba

Acknowledgements

Abstract

Nowadays, most of the computing systems are based on service-oriented computing (SOC). This paradigm aims at replacing complex monolithic systems by a composition of interacting systems called services. A service encapsulates self-contained functionality and offers it over a well-defined and standardized interface. It allows cross-organizational collaborations in which each participant is in charge of a particular tasks leading to the development of scalable, flexible and low-cost distributed applications. Each service works as an autonomous component, performing only the tasks for which it has been implemented. As the development of such services is independent, companies can reuse a considerable amount of components, thus saving money and time. Moreover, these technologies are widely used due to their ability to provide interoperability among services from different companies since all the participants know the services offered by the others as well as how to access them.

Due to privacy concerns or commercial policy, entities participating in one of these architectures have no access to complete information, that is, the code implementing the services the participants consume is hidden, thus being impossible to examine or verify the implementation of the consumed services. Another issue is that web services are usually *stateless*, which means that no state is stored from the clients viewpoint. However, some new applications and services have emerged, which require to capture the state of some resources. Thus, new standards to manage the state of a web service have appeared. For instance, Open Grid Services Infrastructure (OGSI) was conceived to allow designers to manage resources when using web services, and this standard became Web Services Resource Framework (WSRF) when new improvements were introduced.

Obviously, in this scenario the probability of making errors is higher than working in a monolithic scenario. Therefore, there is a clear need of apply-

ing any kind of technique to ensure the correctness of each participant and their composition. In this Thesis, we present a formal language called BPELRF and its semantics. The aim of this language is to model a set of bussiness processess implemented in the de-facto standard modelling language, WS-BPEL, but enriched with the ability to manage distributed resources. These distributed resources are managed according to the guidelines provided by the standard WSRF. Moreover, we provide a visual model of this language in terms of coloured Petri nets in order to ease uninitiated people to deal with it, and we use the well-known toolbox, CPNTools, to verify the composition of web services with distributed resources expressed in BPELRF. As usual, the process of building manually the Petri nets model of large scenarios is time-consuming and error-prone. Therefore, we have implemented a tool to support web designers that, given a BPELRF specification, it extracts automatically the coloured Petri nets of the scenario. Finally, this model can be verified using CPNTools.

On the second part of the Thesis, we extend the classical definition of Workflow nets with time features. Workflow nets were introduced by Wil van der Aalst as a formalism for the modelling, analysis and verification of business workflow processes. The formalism is based on Petri nets abstracting away most of the data while focusing on the possible flow in the system. Its intended use is in finding design errors such as the presence of deadlocks, livelocks and other anomalies in workflow processes. Such correctness criteria can be described via the notion of *soundness* that requires the option to complete the workflow, guarantees proper termination and optionally also the absence of redundant tasks.

After the seminal work on workflow nets, researchers have invested much effort in defining new soundness criteria and/or improving the expressive power of the original model by adding new features and studying the related decidability and complexity questions. In this Thesis, we define a quantitative extension of workflow nets with timing features called timed-arc Workflow nets. It allows us to argue, among others, about the execution intervals of tasks, deadlines and urgent behaviour of workflow processes. Our workflow model is based on timed-arc Petri nets where tokens carry timing information and arcs are labelled with time intervals restricting the available ages of tokens used for transition firing. Here, we consider both discrete and continuous time semantics, thus conforming a whole theory of workflow nets. Finally, all the theory presented in this

thesis have been introduced in the tool Tapaal, offering researchers a mean to model timed-arc workflow nets and to automatically verify (strong) soundness.

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Objectives	2
1.3	Dissertation Structure	4
2	State of the Art	7
2.1	Motivation	7
2.2	Web Services modelling	12
2.2.1	WS-BPEL	17
2.3	Cloud Computing/Grid Computing	24
2.4	Introduccin	25
2.5	Comparacin entre servicios web y Grid computing/Cloud computing	28
2.6	Web Services Resource Framework(WSRF)	31
2.6.1	Introduccin	33
2.6.2	WS-ResourceProperties	34
2.6.3	WS-Base Faults	38
2.6.4	WS-ServiceGroup	39
2.6.5	WS-ResourceLifetime	39
2.7	WS-Notification	41
2.7.1	WS-BrokeredNotification	44
2.8	Service-Oriented Computing (SOC)	45
2.8.1	Service-Oriented Architecture (SOA)	46
2.8.2	Web Services (WS)	46
2.8.3	Web Service Composition	47
2.9	Formal Analysis of Web Service Compositions	49
2.9.1	Specification Formalisms	51

2.10 Summary	51
3 BPELRF	53
3.1 Introduction	53
3.2 Background and Related Work	55
3.3 Syntax and semantics of BPEL+RF	59
3.4 Case Study: Automatic management system for stock market in- vestments	66
3.4.1 Related Work	67
3.5 Prioritised-Timed Coloured Petri Nets	68
3.6 Prioritised-Timed Coloured Petri Net Semantics for WS-BPEL+WSRF	69
3.6.1 Prioritised-Timed Coloured Petri Nets	69
3.6.2 PTCPN Semantics for WSRF/BPEL	71
3.6.3 Basic activities	73
3.6.4 Ordering structures	74
3.6.5 WSRF-compliant	76
3.7 Case study: Online auction service	80
3.8 Summary	82
4 Timed-arc workflow nets	89
5 Conclusions, Contributions and Future Works	91
Bibliography	94

List of Figures

2.1	Example of systems where formal methods are (can be) used	8
2.2	Cost evolution of fixing a bug.	11
2.3	Client-server web architecture	13
2.4	Web Service architecture stack.	15
2.5	Integration of Web Services using WS-CDL.	16
2.6	Actores en un sistema en la nube.	29
2.7	Ejemplo de uso de WS-Notification sin broker.	42
2.8	Web Service architecture stack.	47
2.9	Part of a shipping service specification in BPEL	50
3.1	Orchestration Translation	72
3.2	Sequence Translation	75
3.3	Parallel Activity Translation.	76
3.4	Pick Activity Translation.	77
3.5	While Activity Translation.	77
3.6	CreateResource Activity Translation.	78
3.7	Subscribe Activity Translation.	78
3.8	GetProperty Activity Translation.	79
3.9	SetProperty Activity Translation.	79
3.10	SetTimeout Activity Translation.	79
3.11	A simplified PTCN for the online auction system.	82

List of Tables

3.1	Conversion table	84
3.2	Action transition rules.	85
3.3	Delay transition rules without notifications.	85
3.4	Action and delay transition rules for orchestrators.	86
3.5	Choreography transition rules.	86
3.6	Conversion table	87

Chapter 1

Introduction

1.1 Motivation

The development of software systems is becoming more complex with the appearance of new computational paradigms such as Service-Oriented Computing (SOC), Grid Computing and Cloud Computing. In these systems, the service provider needs to ensure some levels of quality and privacy to the final user in a way that had never been raised. It is therefore necessary to develop new techniques to benefit from the advantages of recent approaches, as Web service compositions. Formal models of concurrency have been widely used for the description and analysis of concurrent and distributed systems. Grid/Cloud environments are characterized by a dynamic environment due to the heterogeneity and volatility of resources. To composite web services, there are two complementary views: Choreography and Orchestration. The choreography view describes the observable interactions among services and can be defined by using specific languages such as Web Services Choreography Description Language (WS-CDL) or by using more general languages like UML Messages Sequence Charts (MSC). On the other hand, orchestration concerns the internal behaviour of a Web service in terms of invocations to other services. Web Services Business Process Execution Language (WS-BPEL) [1] is normally used to describe Web service orchestrations, so this is considered the de-facto standard language for describing Web services workflows in terms of web service compositions. Later on, a brief

introduction of WS-CDL is provided, and a deep description of WS-BPEL is introduced since it will be used as part of this work.

To facilitate additional interoperability among services, more standardization is required to deal with distributed resources. In January of 2004, several members of the *Globus Alliance* organization and the computer multinational *IBM* with the help of experts from companies such as *HP*, *SAP*, *Akamai*, etc. defined the basis architecture and the initial specification documents of a new standard for that purpose, Web Services Resource Framework (WSRF) [?]. Although the Web service definition does not consider the notion of state, interfaces frequently provide the user with the ability to access and manipulate states, that is, data values that persist across, and evolve as a result of Web service interactions. The messages that the services send and receive imply (or encourage programmers to infer) the existence of an associated stateful resource. It is then desirable to define Web service conventions to enable the discovery of, introspection on, and interaction with stateful resources in standard and interoperable ways [?].

The main motivation of the first part of the Thesis is to provide a formal semantics for WS-BPEL+WSRF to manage stateful Web services workflows by using the existing machinery in distributed systems, and specifically a well-known formalism, such as prioritised-timed coloured Petri nets, which apart of being a graphical model, but they also provide us an easier way to simulate and analyse the modelled system. Thus, our aim is not to provide just another WS-BPEL semantics. In order to deal with the integration of BPEL plus WSRF in a proper way, we have realized that it is more convenient to introduce a specific semantic model, which covers properly all the relevant aspects of WSRF such as notifications and resource time-outs. The integration of WS-BPEL and WSRF is not new; in the literature, there are a bundle of works defining this integration, but none of these works define a formal semantics in terms of Petri nets.

1.2 Objectives

Next, we describe the general and specific objectives of the Thesis.

Main Objective

The main objective of the Thesis can be splitted into two main objectives. The first one is the definition of a formal language to model Grid/Cloud Computing environments, reusing some of the web technologies presented to date. The second one is to propose a new extension of workflow nets in terms of timed-arc Petri nets, thus providing two formal models in which the time semantics is discrete or continuous. Obviously, these objectives are too general and, therefore, we list below a set of subobjectives that are required to achieve these overall objectives.

Specific Objectives

To meet this overall objectives the following specific objectives should be achieved:

- **Objective 1: State-of-the-art**

1. Study of different formalisms for the modelling and analysis of Grid/- Cloud Computing applications using web services.
2. Summarise the current definitions of soundness and the different extensions of workflow nets presented to date.

- **Objective 2: Technological framework definition**

1. Study of the current techniques for modelling and implementing Web service composition and Grid/Cloud Computing applications.
2. Analyse the different tools for modelling workflow nets as well as possible target applications of the theory presented in this Thesis.

- **Objective 3: Development of the proposal**

1. Define the specific models, either the operational semantics or the PTCPNs semantics of the language BPELRF.
2. Extend the current definition of workflow nets with a time semantics.
3. Adapt the definition of soundness to this timed scenario.
4. Develop tools within the theory presented here.

5. Analyse and evaluate both proposals.

• **Objective 4: Examples and Case studies**

1. Propose a set of simple examples where the main features are displayed.
2. Study a set of theoretical examples where the power of both proposals and its main aspects is characterized .
3. Demonstrate the applicability of this work applying it to real (industry-based) case studies.

1.3 Dissertation Structure

This Thesis is organised in five different chapters as follows.

Chapter 1 makes a brief introduction of the Thesis, showing the motivation, the main objectives and the scope of it.

Chapter 2 shows the state of art of the contents included here. This chapter includes a brief description of Service-Oriented Computing (SOC) and distributed computing, e.g. Grid and Cloud computing and the use of formal methods for the analysis of web service compositions and the benefits of using formal techniques in the development of software and hardware. Moreover, a deep introduction of the standards used in this work is presented. Finally, we get into workflow nets and the possible extension of this formal model as well as its main properties.

Chapter 3 presents a formal specification language called BPELRF, which takes two well-known standards (WS-BPEL and WSRF) as basis, to model synchronous and asynchronous stateful interactions. This language is enriched with a publish-subscribe architecture, service discovery, event and fault handling and time-outs. As usual, an operational semantics for this language is defined. Moreover, we define a visual model of it in terms of coloured Petri nets and a tool to verify some properties of the specifications written in BPELRF.

In **Chapter 4** we suggest a workflow model based on timed-arc Petri nets and study the foundational problems of soundness and strong (time-bounded) soundness. We explore the decidability of these problems and compare the discrete and continuous semantics of timed-arc workflow nets.

Chapter 5 shows the main conclusions, contributions and future works of this Thesis.

Chapter 2

State of the Art

In this chapter, it will be introduced the state-of-the-art related to the specification, formalization and verification of stateful web services and their composition as well as the use of formal methods in this topic. The aim of this chapter is to provide the reader with the basic notions about formal methods and stateful web service compositions in order to help he/she in the understanding of the Thesis. To begin with, a brief introduction of formal methods and why they are needed is presented. Second, a survey about the different technologies used to model web services and the different approaches to compose them are introduced and, next, the different mechanisms available to improve these web services with distributed resources. Finally, the different formal models used here are defined. On the other hand, we introduce workflow nets and why they are useful to model business processes. Some informal definition about the properties can be studied with this formal model is also provided.

2.1 Motivation

Throughout the history of computing, engineers and researchers have used different formal methods to improve the quality of hardware and software. These systems with continuous technological progress in integration techniques and programming methodologies inevitably grow in scale and complexity. Because of this complexity, the probability of error is higher and, in addition, some of



Figure 2.1: Example of systems where formal methods are (can be) used .

these errors can cause incalculable economic losses, time or even the loss of human lives. Therefore, the main aim of designers should be to provide developers with the required tools to build systems with a negligible error rate and with the lowest cost. However, this task is far from trivial since one needs to ensure the correctness of the specifications and needs to provide techniques that ease error detection and the verification of the developed models without consuming so much time of the development process. One of the ways that engineers have been used to achieve this goal is the use of formal techniques to ensure the correctness of the development process as well as the product under construction. These formal methods can be defined as the set of procedures and tools based on mathematical languages that virtually ensure the correctness of a system [3] since they increase the level of knowledge that the participants have about the system, revealing inconsistencies and ambiguities that could not be detected using other techniques, i.e., the use of formal methods provides a greater degree of refinement of the model than other methods.

In the past, the use of formal techniques in practice seemed to be utopian and unrealizable. Among other causes, the notations used to require a high mathematical background in mathematics and, therefore, they were too complicated for the uninitiated in the topic. The techniques did not allow the system

to be scalable and the existing tools were too difficult to use or understand or even there were no tools for a particular technique or formalism. In addition, case studies were not convincing enough and, therefore, developers could not appreciate the usefulness of formalization. However, in the early 90s, it started to glimpse a new way in this area. For the specification of software, the industry began to use the language Z [6] in order to obtain rigorous specifications. For hardware verification, major companies such as Intel and AMD started to use formal techniques such as *model checking* or *theorem proving* to supplement tests on simulators. This led to the description of larger case studies, which was beneficial for the advance of this area since other developers started to consider the possibility of introducing the use of formal techniques into their development processes. In Figure 2.1, one can observe different systems in which these techniques are currently used to ensure proper operation. For instance, big companies (e.g Boeing and Airbus) use formal languages to specify the requirements of the equipment as well as they use formal methods to verify the most critical systems in the aircrafts. Moreover, automotive companies verify the most critical systems (e.g. brake or airbag systems) using *model checking*.

The main advantages of using formal methods are:

- The use of mathematics as a base gives this approach a certain rigour.
- Identify ambiguity and inconsistencies.
- Facilitates the construction of consistent and *deadlock-free* systems.
- Provides customer confidence in the system.
- There are many tools that support the existing techniques.
- Find bugs early should save money.

The main disadvantages (or beliefs) that slow the progress of this area are:

- It is believed that the use of formal methods slows the development process.
- Many developers think it is difficult to work with formal specifications.
- It does not guarantee the correctness of the implemented code (only the model it is based).

- Increasing system complexity causes an exponential increase the complexity of the verification.

As commented previously, companies can use formal methods along the entire development lifecycle of a system, both hardware and software. Here, we will focus on software since this Thesis studies different standards for building software components. Next, we describe the different phases in which designers can apply any formal technique.

One of the most important part in the development of a system is the requirements specification. A specification can be seen as a technical document where the features and services needed to build a product are stated. Nevertheless, it can also include information on subsequent steps such as verification, validation, testing, etc. Therefore, this should be the first part in which the participants should apply formal methods, taking the required time to correctly specify the system since a neat and correct specification will influence the rest of the process. Anyway, make a proper specification does not guarantee the absence of errors because the presence of faults is an intrinsic characteristic of the systems. In this sense, the simple act of writing the document helps engineers to find errors in the early stages of the development process, helping the company to save money and time. In Figure 2.2, one can observe what is the effect (in money) of finding a bug in the different phases. As can be observed, the cost of fixing a bug increases as we advance in the lifecycle and, therefore, it is recommended to find these bugs as soon as possible. In this Thesis, we propose a formal language and its visual model to specify web service compositions with distributed resources, but this will be presented in Chapter 3.

In the classic life cycle, the verification and validation phases are performed after the implementation phase, but as we have seen in Figure 2.2, it is advisable to detect these errors as soon as possible. As expected, it is practically impossible to verify completely all the behaviour of a complex system so that the goal of researchers in this area is to check whether certain properties hold in the model. The properties of interest will be related to the classical problems of concurrency (*deadlock*, *mutual exclusion*,...) and some aspects directly related to the system itself such as check the adherence of it to certain time constraints. For example, in a banking system, it is mandatory to ensure that transactions



Figure 2.2: Cost evolution of fixing a bug.

meet the stipulated time for completion because if you exceed these restrictions some security issues could come out.

In this sense, one can follow two different ways to perform the verification of a system: *Human-directed proof* or *Automated proof*. The first one is used when you want to strengthen the knowledge of the system rather than completely ensure the correctness of it, and, therefore, it is a person who check the properties manually. This variant improves the knowledge of the system, but it is time-consuming and error-prone due to the entire process is conducted for a human being. In the second approach (*automated proof*) there are also two variants: *automated theorem proving* and *model checking*. The *automated theorem proving* is conducted by a program that tries to produce a formal proof of a system from scratch, giving a description of it, a set of logical axioms and a set of inference rules. On the other hand, model checking [4] is a technique for verifying finite state concurrent systems. It has a number of advantages over traditional approaches that are based on simulation, testing, and deductive reasoning. In particular, model checking is normally automatic and usually quite fast. Also, if the design contains an error, model checking will produce a counterexample that can be used to pinpoint the source of the error. Here, the specification can be expressed in propositional temporal logic propositional normally LTL [8] or CTL [10] or some of its variants, and the system is represented as a graph

of transitions between states. The main challenge in model checking is dealing with the state space explosion problem. When dealing with web systems, this problem occurs in systems with many components that can interact with each other or systems with data structures that have many different values. In such cases the number of global states can be enormous. Researchers have made considerable progress on this problem over the last ten years.

2.2 Web Services modelling

Although the Web was initially intended for the exclusive use of human beings, many experts believe that it needs to evolve (probably through modular design and construction services) to better support for the automation of many tasks. The concept of *service* provides a higher level of abstraction to organize large-scale applications and build more open environments, helping to develop applications with improved productivity and quality with respect to other approaches. As services are only a mean for building distributed applications, it is required to evaluate the different existing approaches in this area. Figure 2.3 shows an example of service-based architecture, where there are three main parts: a consumer, a provider (the servers) and a set of records, where the services are stored. The role of the providers is to publish and/or advertise the services offered in the records, where consumers can find and invoke them. Current standards that support interactions between web services provide a solid foundation for service-oriented architecture. The web architecture is a framework that can be reinforced with more powerful representations and techniques inherited from other approaches.

In this way, Service-Oriented Computing (SOC) paradigm promotes the use of services for the development of massively distributed applications, trying to achieve the creation of fast, low-cost, flexible and scalable applications [?]. Services are the main building block of this paradigm, being these services self-describing and platform-independent. Thanks to the use of standards for the description, publication, discovery and invocation, the services can be integrated without taking care of the low-level implementation details of each service. The aim of SOC is to make possible the creation of dynamic business processes and



Figure 2.3: Client-server web architecture

agile applications by providing an easy way to assemble application components into a loosely coupled network of services.

To reach the goals of SOC, a Service-Oriented Architecture (SOA) is defined. SOA is a software architecture based on the utilization of services, being these services provided to the user of the application or to other services in the network. This is possible by the use of service interfaces that can be published and discovered. SOA is based on a model of roles where every service can play multiple roles. For example, a service can offer certain functionality to a user and, at the same time, being the consumer of the functionality provided by some other services. Such model reduces the complexity of applications and increases their flexibility. Although at the beginning of SOA there were several architectures aspiring to become SOA standards [?, ?], the most successful one was the architecture based on Web Services.

W3C defines a Web Service (WS) in the following way:

“A Web Service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an inter-

face described in a machine-processable format (specifically WSDL). Other systems interact with the Web Service in a manner prescribed by its description using SOAP-messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.”

We can see in this definition that there are two basic standards related to Web Services: Web Service Description Language (WSDL) for the definition of the service functionality and its properties [?], and Simple Object Access Protocol (SOAP) for the exchange of XML messages between services [?]. There is also an additional standard called Universal Description, Discovery and Integration (UDDI) used to create Web Service directories and to search for services in the network [?], but this is a bit out of date. The use of these standard protocols is the key point to improve the integration between different parties in a web service architecture.

In Figure 2.8 a possible representation of the web service architecture stack is shown. One can see that the three standards described above are only a small part of the stack. One also need protocols to define security aspects (ensuring that exchanges of information are not modified or forgotten in a verifiable manner and that parties can be authenticated), to provide reliable messaging for the exchange of information between parties, to specify the collaboration between services when we compose them, to individually describe the behaviour of each service in a business process, etc. The problem is that whereas the standards for basic services (WSDL and SOAP) are widely adopted for their respective purposes, the situation is not very clear when we talk about composing services, having multiple protocols aspiring to become a standard in this layer.

Two different approaches can be followed when we designing web service compositions. They are called *orchestration* and *choreography*. The former describes the individual business process followed by each one of the participants in the composition, while the latter describes the composition from a global viewpoint, defining the interactions (exchange of messages) happening between the parties, that is, how they collaborate in the composition. However, the ideal solution would be fusing both approaches in a single language and environment [?].

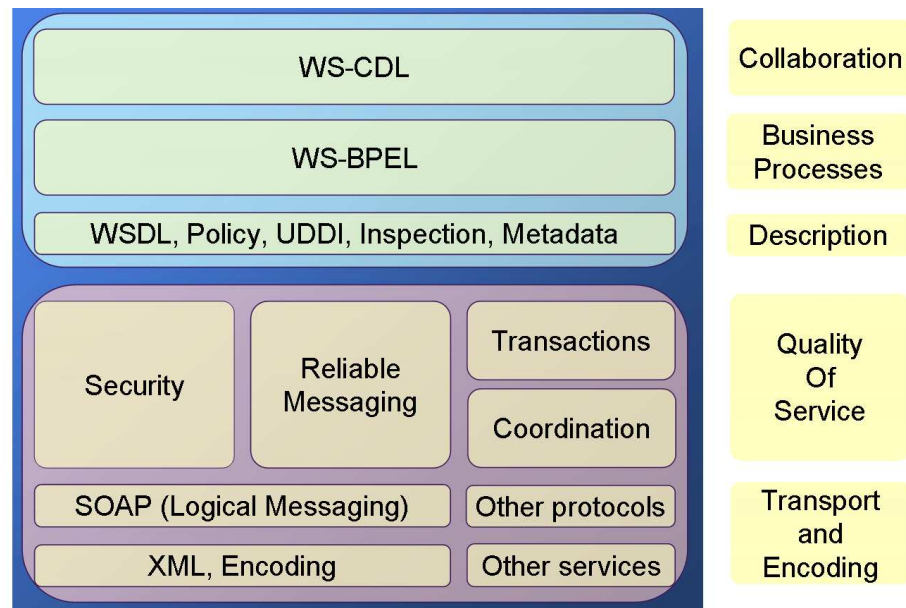


Figure 2.4: Web Service architecture stack.

Anyway, the languages we can use in both cases should accomplish some common goals: (i) the capacity of modelling service interactions, including control flow and data constraints, (ii) the possibility of specifying exceptional behaviour, indicating which errors can happen in the execution of the composition and the way of handling these errors, and (iii) the ability to model web service compositions at a high level, without taking care of the implementation details of each one of the services.

Regarding the choreography approach, there are several languages that have been designed for that purpose. One of the most popular languages is Web Services Choreography Description Language (WS-CDL), which specifies the common and complementary observable behaviour of all participants in a composition [?]. It is based on XML and describes the peer-to-peer collaborations between the composite web services from a global point of view, that is, the exchange of messages to achieve a common business goal. The aim of this language is allowing the composition of any kind of web services, regardless of the platform hosting the service or the implementation language. Figure 2.5 is an example of how WS-CDL can be useful for the integration of different kinds of web services.

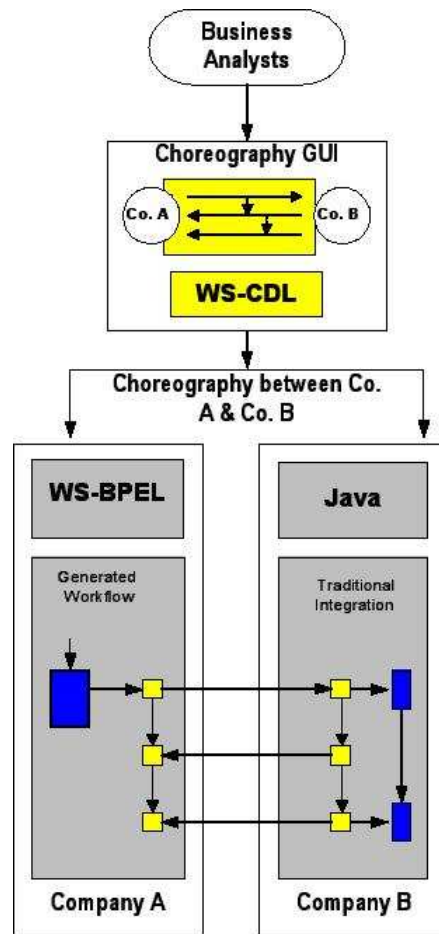


Figure 2.5: Integration of Web Services using WS-CDL.

A WS-CDL document defines a hierarchy of choreographies, where there is only one top-level choreography, marked explicitly as the *root choreography*. The basic building block of a choreography is the *interaction* element. It indicates information exchanges between participants, possibly including the synchronization of some information values. These interactions are performed when one participant sends a message to another participant in the choreography. When the message exchanges complete successfully, the interaction completes normally.

We can distinguish two different kinds of *complex activities* inside a choreography: the *workunit* element and the ordering structures. The *workunit* element specifies a condition that must be fulfilled in order to perform some work and/or the repetition of some work. It completes successfully when the set of activities

inside completes successfully. *Ordering structures* are used to combine basic activities and other complex activities in a nested way, expressing the order in which actions are performed within the choreography. There are three ordering structures: The *sequence* ordering structure expresses that the set of activities inside must be executed sequentially. The *parallel* ordering structure indicates that the set of activities inside must be executed concurrently. It completes successfully when all the concurrent activities complete successfully. And the *choice* ordering structure specifies that only one of multiple activities can be executed. If the choice have workunits inside, only the first one in lexical order with a “true” guard condition is selected. If there are other activities, there is no way to know which one is selected; it is considered as a non-observable decision.

Different types of exceptions are considered in WS-CDL. Exception workunits can be defined to handle all these exceptions. They may also be used as the mechanism to recover from the exceptions. At least one exception workunit must be defined. The guard of the workunit can be used to specify the particular type of exception we want to handle. Only one exception workunit can match each exception. If multiple exception workunits are defined, the order of evaluating them is based on the order in which the workunits have been defined. When the matching happens, the actions of the matched workunit are executed. If no matching happens and a default exception workunit exists, then the actions of this workunit are executed. Otherwise, the exception is raised in the parent choreography. WS-CDL also allows us to define finalization actions within a choreography that can confirm or cancel the effects of this choreography, so we can use this actions for compensation.

2.2.1 WS-BPEL

Traditional methods for integration and business process automation typically involve embedded logic inside of applications designed to meet a specific business need such as ERP, supply chain, or CRM. The development, testing, and deployment efforts required to change these applications make integration and process changes both costly and complex.

To address these issues, proprietary products emerged to abstract integration and process automation into a new layer of software tools. These software

products liberated integration and process tasks from the underlying business systems so they could be more effectively changed, managed, and optimized.

The Business Process Execution Language for Web Services (BPEL4WS) was first conceived in July, 2002 with the release of the BPEL4WS 1.0 specification, a joint effort by IBM, Microsoft, and BEA. This document proposed an orchestration language inspired by previous variations, such as IBM's Web Services Flow Language (WSFL) and Microsoft's XLANG specification.

Joined by other contributors from SAP and Siebel Systems, version 1.1 of the BPEL4WS specification was released less than a year later, in May of 2003. This version received more attention and vendor support, leading to a number of commercially available BPEL4WS-compliant orchestration engines. Just prior to this release, the BPEL4WS specification was submitted to an OASIS technical committee so that the specification could be developed into an official, open standard.

The OASIS WS-BPEL Technical Committee was active from April 2003 to May 2007. It was co-chaired by Diane Jordan of IBM and John Evdemon of Microsoft. The Committee's email archives remain publicly accessible.

In April 2007, WS-BPEL version 2.0 was approved as an OASIS Standard. More than 37 organizations collaborated to develop WS-BPEL, including representatives of Active Endpoints, Adobe Systems, BEA Systems, Booz Allen Hamilton, EDS, HP, Hitachi, IBM, IONA, Microsoft, NEC, Nortel, Oracle, Red Hat, Rogue Wave, SAP, Sun Microsystems, TIBCO, webMethods, and other members of OASIS.

In January 2008, OASIS issued a Call for Participation in the BPEL4People Technical Committee. The group works to define a WS-BPEL extension enabling the definition of human interactions ("human tasks") as part of a WS-BPEL process.

There were ten original design goals associated with BPEL:

Define business processes that interact with external entities through web service operations defined using WSDL 1.1, and that manifest themselves as Web services defined using WSDL 1.1. The interactions are "abstract" in the sense that the dependence is on portType definitions, not on port definitions. Define

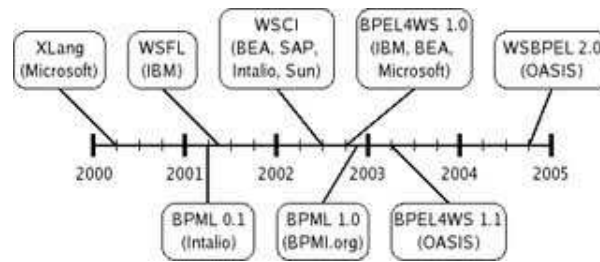


Figure 2.6: Integration of Web Services using WS-CDL.

business processes using an XML-based language. Do not define a graphical representation of processes or provide any particular design methodology for processes. Define a set of Web service orchestration concepts that are meant to be used by both the external (abstract) and internal (executable) views of a business process. Such a business process defines the behavior of a single autonomous entity, typically operating in interaction with other similar peer entities. It is recognized that each usage pattern (i.e., abstract view and executable view) will require a few specialized extensions, but these extensions are to be kept to a minimum and tested against requirements such as import/export and conformance checking that link the two usage patterns. Provide both hierarchical and graph-like control regimes, and allow their use to be blended as seamlessly as possible. This should reduce the fragmentation of the process modeling space. Provide data manipulation functions for the simple manipulation of data needed to define process data and control flow. Support an identification mechanism for process instances that allows the definition of instance identifiers at the application message level. Instance identifiers should be defined by partners and may change. Support the implicit creation and termination of process instances as the basic lifecycle mechanism. Advanced lifecycle operations such as "suspend" and "resume" may be added in future releases for enhanced lifecycle management. Define a long-running transaction model that is based on proven techniques like compensation actions and scoping to support failure recovery for parts of long-running business processes. Use Web Services as the model for process decomposition and assembly. Build on Web services standards (approved and proposed) as much as possible in a composable and modular manner.

BPEL and Web services now provide a standardized integration interface and a standardized language for integration and process automation. BPEL, in effect, has the potential to commoditize the capabilities provided by proprietary EAI and BPM solutions. As often occurs in a commodity market, the resulting prices for products and services are certain to fall. Developing the web services and exposing the functionalities is not sufficient. We also need a way to compose these functionalities in the right order ? a way to define business processes which will make use of the exposed functionalities. We would obviously prefer a relatively simple and straightforward way to define such processes, particularly because we know that business processes change often, therefore we would like to modify them easily. BPEL is thus comparable to general purpose programming language such as Java, but it is not as powerful as Java. On the other hand it is simpler and better suited for business process definition. Therefore BPEL is not a replacement but rather a supplement to modern languages. Choreography has not gained support from the industry which would be comparable to BPEL. This is where the BPEL (Business Process Execution Language for Web Services, also WS-BPEL or BPEL4WS) becomes important. BPEL allows composition of web services and is thus the top-down approach to SOA ? the process oriented approach to SOA. Let us have a closer look at a typical BPEL process. First, the BPEL business process receives a request. To fulfill it, the process then invokes the involved web services and finally responds to the original caller. Because the BPEL process communicates with other web services, it relies heavily on the WSDL description of the web services invoked by the composite web service.

The process-oriented approach to SOA requires a language for relatively simple description of how web services should be composed into business processes. Of course it would be great if such descriptions could also be executed, which would allow us not only to define abstract process definitions, but to write exact executable specifications of processes. BPEL is such a language. Actually it is the first language which:

BPEL Language 1. Allows us to define abstract and executable processes 2. Is supported by the majority of companies 3. Software exists (from several vendors) on which such processes can be executed (BPEL servers) and developed (BPEL designers).

The idea and motivation behind almost each new technology and platform for enterprise application development is to provide an environment where better business applications can be developed with less effort ?business applications which should closely align to the business processes, which should not be too complex, and which can be adapted to the changing nature of business processes without too much effort. Java has provided an excellent platform for developing such applications, but business applications today cannot be isolated. Within companies, business applications have to interoperate and integrate - so they have between companies. Integrating different applications has always been a difficult task for various functional and technology related reasons. BPEL builds on top of XML and web services. It is an XML-based language which supports the web services technology stack, including SOAP, WSDL, UDDI, WS-Reliable Messaging, WS-Addressing, WS-Coordination and WS-Transaction. BPEL represents a convergence of two early workflow languages, WSFL (Web Services Flow Language) and XLANG. WSFL was designed by IBM and is based on the concept of directed graphs. XLANG was designed by Microsoft and is a block-structured language. BPEL combines both approaches and provides a rich vocabulary for description of business processes. A BPEL process specifies the exact order in which participating web services should be invoked. This can be done sequentially or in parallel. With BPEL, we can express conditional behavior, for example, a web service invocation can depend on the value of a previous invocation. We can also construct loops, declare variables, copy and assign values, define fault handlers, and so on. By combining all these constructs, we can define complex business processes in an algorithmic manner. From the perspective of composing web services to execute business processes, orchestration is the more flexible approach compared to choreography: We know exactly who is responsible for the execution of the whole business process. We can incorporate web services, even those that are not aware that they are a part of a business process. We can also provide alternative scenarios when faults occur. Developing the web services and exposing the functionalities is not sufficient. We also need a way to compose these functionalities in the right order ? a way to define business processes which will make use of the exposed functionalities. We would obviously prefer a relatively simple and straightforward way to define such processes, particularly because we know that business processes change often, therefore we would like to modify them easily. BPEL is

thus comparable to general purpose programming language such as Java, but it is not as powerful as Java. On the other hand it is simpler and better suited for business process definition. Therefore BPEL is not a replacement but rather a supplement to modern languages such as Java. BPEL follows the orchestration paradigm. Choreography is covered by other standards, such as WSCI (Web Services choreography Interface) and WS-CDL (Web Services Choreography Description Language). Choreography has not gained support from the industry which would be comparable to BPEL. This is where the BPEL (Business Process Execution Language for Web Services, also WS-BPEL or BPEL4WS) becomes important. BPEL allows composition of web services and is thus the top-down approach to SOA ? the process oriented approach to SOA. Let us have a closer look at a typical BPEL process. First, the BPEL business process receives a request. To fulfill it, the process then invokes the involved web services and finally responds to the original caller. Because the BPEL process communicates with other web services, it relies heavily on the WSDL description of the web services invoked by the composite web service.

Executable processes allow us to specify the exact details of business processes. They can be executed by an orchestration engine. In most cases BPEL is used for executable processes. Abstract business protocols allow us to specify the public message exchange between parties only. They do not include the internal details of process flows and are not executable. Choreography on the other hand does not rely on a central coordinator. Rather, each web service involved in the choreography knows exactly when to execute its operations and whom to interact with. Choreography is a collaborative effort focused on exchange of messages. All participants of the choreography need to be aware of the business process, operations to execute, messages to exchange, and the timing of message exchanges. The most recent answer to the integration challenge is the Service Oriented Architecture (SOA) and the web services technologies. The bottom-up view of the SOA sees different business applications exposing their functionalities through web services. Thus we can now access different functionalities of different legacy and new developed applications in a standard way (through web services). Such access to functionalities is important because typical companies have a large number of existing applications which have to be integrated.

WS-BPEL [?], for short BPEL, is an OASIS orchestration language for specifying actions within Web service business processes. BPEL is therefore an orchestration language in the sense that it is used to define the composition of services from a local viewpoint, describing the individual behaviour of each participant. BPEL is designed to support the description of both behavioural service interfaces and executable service-based processes [7]. A behavioural interface (known as abstract process) is a specification of the behaviour of a class of services, capturing constraints on the ordering of messages to be sent to and received from a service. An executable process defines the execution order of a set of activities (mostly communication activities), the partners involved in the process, the messages exchanged between partners, and the events and exception handling specifying the behaviour when specific events or faults occur.

Thus, a WS-BPEL abstract process is a partially specified process that is not intended to be executed and it must be explicitly declared as “abstract”. As its name indicates, an abstract process may hide some of the required operational details expressed by an executable artifact. All the constructs of executables processes are made available to abstract processes and, consequently, they share the same expressive power []. Although a particular Abstract Process definition might contain complete information that would render it Executable, its Abstract status states that any concrete realizations of it are permitted to perform additional processing steps that are not relevant to the audience to which it has been given.

Abstract processes serve a descriptive role, with more than one use case. One such use case might be to describe the observable behavior of some or all of the services offered by an executable process. Another use case would be to define a process template that embodies domain-specific best practices. Such a process template would capture essential process logic in a manner compatible with a design-time representation, while excluding execution details to be completed when mapping to an executable Process.

As mentioned above it is possible to use WS-BPEL to define an executable business process, that is, the language effectively defines a portable execution format for business processes that rely exclusively on web Services and XML data. Moreover, such processes execute and interact with their partners in a consistent way regardless of the supporting platform or programming model

used by the implementation of the hosting environment. The continuity of the basic conceptual model between Abstract and Executable Processes in WS-BPEL makes it possible to export and import the public aspects embodied in Abstract Processes as process or role templates while maintaining the intent and structure of the observable behavior. This applies even where private implementation aspects use platform dependent functionality. This is a key feature for the use of WS-BPEL from the viewpoint of unlocking the potential of web services because it allows the development of tools and other technologies that greatly increase the level of automation and thereby lower the cost in establishing cross enterprise automated business processes.

In detail, WS-BPEL defines a model and a grammar for describing the behavior of a business process based on interactions between the process and its partners. The interaction with each partner occurs through web service interfaces, and the structure of the relationship at the interface level is encapsulated in what is called a *partnerLink*. The WS-BPEL process defines how multiple service interactions with these partners are coordinated to achieve a business goal, as well as the state and the logic necessary for this coordination. WS-BPEL also introduces systematic mechanisms for dealing with business exceptions and processing faults. Moreover, WS-BPEL introduces a mechanism to define how individual or composite activities within a unit of work are to be compensated in cases where exceptions occur or a partner requests reversal.

A WS-BPEL process is a reusable definition that can be deployed in different ways and in different scenarios, while maintaining a uniform application-level behavior across all of them.

BPEL processes use *variables* to temporarily store data. Variables are therefore declared on a process or on a scope within that process. I

BPEL activities can be *basic* or *structured*. *Basic activities* are those which describe the elemental steps of the process behaviour, such as the assignment of variables (*assign*), empty action (*empty*), time delay (*wait*), invoke a service (*invoke*) and receive a message (*receive*), reply to a client (*reply*), and throw an exception (*throw*). We also have an action to *terminate* the process execution at any moment (*exit*). For technical reasons we have also included a barred form of

reply action, which is used when a service invocation expects a reply, in order to implement the synchronization with the *reply* action from the server.

Structured activities encode control-flow logic in a nested way. The considered structured activities are the following: a *sequence* of activities, separated by a semicolon, the parallel composition, represented by two parallel bars (`||`), the conditional repetitive behaviour (*while*), and a timed extension of the receive activity, which allows to receive different types of messages with a time-out associated (*pick*).

2.3 Cloud Computing/Grid Computing

Gracias a la rpida evolucin que ha tenido la sociedad, servicios bsicos para el desarrollo de la vida cotidiana son comnmente suministrados a los ciudadanos, de tal manera, que cualquier persona puede tener acceso inmediato a ellos de forma fcil. Hoy en da, estos servicios, conocidos en el mundo anglosajn como “utility services”, engloban el suministro de agua, electricidad, gas y telfono, pero en los ltimos tiempos est cobrando fuerza una vieja idea que se intent llevar a cabo sin xito a finales de los aos 60 y principios de los 70, el “Utility Computing”. Este nuevo paradigma de computacin es normalmente confundido con Cloud o Grid Computing, pero hay ciertos matices que los diferencian. “Utility Computing” se puede entender como el modelo de negocio que subyace en una infraestructura Cloud o Grid, es decir, puede ser entendido como el medio de cobro de servicios computacionales similar al que se hace con la electricidad, por lo que el usuario pagar slo por su consumo, mientras que los costes asociados a la produccin y distribucin de potencia de cmputo sern sufragados por las compaas suministradoras. As, las pequeas y medianas empresas podrn competir en igualdad de condiciones con las grandes empresas, ya que no sera necesario hacer una gran inversin en datacenters para poder ofrecer un determinado servicio y se fomentara la creacin de empresas, ya que estos datacenters suponen una fuerte inversin inicial que muchos emprendedores no pueden acometer. Adems, el usuario final de las aplicaciones, servicios o infraestructuras tambin se beneficiara porque al reducir costes de produccin se reduce el precio de los productos. Por seguir con el smil de la energa, podemos ver el cloud computing como una gran central generadora de energa que da suministro a millones de usuarios y

que evita que dichos usuarios tengan que tener su propia central en casa para poder encender sus aparatos eléctricos, mientras que el “utility computing” es la forma de tarificar el gasto de los usuarios o, de una forma más abstracta, podemos verlo como el contador que muestra el consumo energético. Al igual que pasa con el software, los protocolos o cualquier paradigma relacionado con la informática, Cloud Computing debe atravesar una serie de etapas para poder comprobar si toda esta publicidad que le están dando las empresas sirve de veras para ahorrar costes y favorecer la competitividad o solo es más una forma de aumentar ingresos o, en el caso de la investigación, obtener nuevos fondos. En este sentido, algunos autores consideran que Cloud Computing no es más que una nueva forma de nombrar lo que toda la vida se ha llamado Grid o Web Services y que realmente no supone ningún avance en el campo de la informática. Este artículo tratará de presentar más ampliamente la arquitectura y conceptos para comprender un sistema de computación basado en la nube y mostrar las diferencias entre dos enfoques clásicos de computación (Grid y Web Services) y Cloud. Finalmente, se propondrá una serie de ideas que pueden llegar a convertirse en trabajos de investigación en un futuro.

2.4 Introducción

En 1943, el presidente de IBM, Thomas J. Watson, predijo:

“I think there is a world market for about five computers”

Esta frase, tan comentada en el mundo de la informática en los últimos tiempos, ha pasado de ser una predicción con poco fundamento a ser una realidad en la actualidad.

Cloud Computing, el viejo sueño de ofrecer servicios de computación como utilidad, tiene el potencial de transformar gran parte de la industria informática, haciendo el software más atractivo al ofrecerlo como servicio y moldeando la forma en que se diseña y compra el hardware. Con este nuevo enfoque, cualquier emprendedor con buenas ideas para ofrecer servicios a través de Internet no necesitar realizar grandes inversiones en equipamiento para llevar a cabo su proyecto

ni necesitar contratar inicialmente mucho personal que gestione y mantenga dicho equipamiento. Adems, no tiene que realizar complicados estudios previos para calcular el nmero de usuarios potenciales y evitar, as, unos de los principales quebraderos de cabeza de los jefes de proyecto: el sobre-aprovisionamiento o el infra-aprovisionamiento. Estos dos conceptos junto con la elasticidad de recursos pueden ser considerados como claves en computacin en la nube, ya que el objetivo de reducir costes es directamente proporcional a la correcta estimacin de recursos en “tiempo real” y esta correcta estimacin slo se puede proporcionar si el sistema cumple la propiedad de elasticidad, es decir, que en un intervalo de tiempo relativamente corto aumentas y disminuyes los recursos dedicados a una tarea con un coste econmico bajo. Este enfoque puede hacer que, a primera vista, no se perciba la posibilidad de utilizar mtodos formales con este tipo de sistemas, puesto que normalmente se utilizan tcnicas formales en el diseo de sistemas con tiempos de respuesta crticos como sistemas de navegacin de un avin, transporte de materiales peligrosos, etc. De esta manera, se hace inviable el uso de la computacin en la nube cuando se exijan tiempos de respuesta muy bajos, ya que, por ejemplo, un sistema de navegacin de un avin no puede esperar varios minutos a que se le asignen nuevos recursos para tomar una decisin. Sin embargo, si que hay otro tipo de sistemas en los que los mtodos formales y el cloud computing pueden converger, los sistemas de alta disponibilidad.

As, utilizando tcnicas formales se pueden disear este tipo de sistemas y verificar la ausencia de fallos en su construccin. Por ejemplo, una tienda de venta online podra pasar de tener cientos de usuarios simultneamente a miles de ellos en periodos como las vacaciones de Navidad, de manera que necesitara mucha mayor potencia de computacin si quiere satisfacer a todos los clientes y no perder ingresos ni nuevos clientes por no poder atender esa demanda. Para satisfacer esta necesidad, hara un estudio preliminar de cuantas visitas como mximo puede tener en ese periodo y comprara los datacenters necesarios para no tener problemas de congestin, lo que supone una inversin grande en infraestructura por parte de la compaa, sin embargo, una vez acaba la campaa navidea la demanda de usuarios vuelve a ser de unos cientos y la empresa se encuentra con que tiene una potencia de cmputo que no va a necesitar y, por tanto, no est amortizando econmicamente la inversin realizada. En este sentido, si la compaa

en lugar de comprar los datacenters hubiese comprado capacidad de cómputo a un proveedor entonces habría amortizado en mayor medida el dinero invertido y no tendría máquinas en sus oficinas que ocupan bastante espacio y que tienen unos niveles de carga de trabajo muy bajos.

Cloud Computing se refiere tanto a las aplicaciones que se ofrecen como servicios a través de Internet como al hardware y software que está presente en los datacenters que proveen dichos servicios. Estos servicios se han referido normalmente como Software as a Service (SaaS). El datacenter en sí es lo que se considera la nube (o cloud). Desde el punto de vista del hardware, hay tres aspectos novedosos en Cloud Computing:

1. La ilusión de tener recursos ilimitados bajo demanda eliminando a los usuarios la necesidad de aprovisionarse antes de acometer una tarea.
2. La eliminación de la inversión inicial en equipamiento permitiendo a las compañías empezar con pocos recursos e ir aumentando cuando las necesidades aumenten.
3. La posibilidad de pagar por el uso de recursos de computación a corto plazo según se necesiten (por ejemplo, procesadores por hora o capacidad de almacenamiento de datos por día) y poder liberarlos cuando no sean necesarios.

Cloud Computing podrá tener el mismo impacto en la producción de software que el que tuvieron las fundiciones de metal en la industria del hardware. En principio, las compañías fabricantes de hardware necesitaban tener sus propias instalaciones donde fabricar los componentes que componían sus productos, lo que les suponía un gran esfuerzo económico para construir y operar estas instalaciones y, por consiguiente, hacía que el precio de los equipos se doblase en cada nueva generación. Sin embargo, la aparición de compañías que fabricasen componentes favoreció que empresas más pequeñas pudiesen entrar en el mercado del hardware, copado hasta aquel entonces por Intel o Samsung, que eran las únicas que podían hacer frente a este gran esfuerzo económico. De manera similar, la computación en la nube podrá jugar el papel que hicieron las fundiciones, favoreciendo la competencia y evitando monopolios de grandes empresas.

Debido a que muchas empresas usan servicios software como base para su modelo de negocio, se presentan, a continuacin, los actores que formarn parte de este escenario. Los *Services Providers*(SPs) hacen accesibles los servicios a los *Service Users* por medio de interfaces que se comunican a travs de Internet. Dado que uno de los objetivos de la nube es externalizar la provisin de servicios, se necesita la aparicin de otro actor que ofrezca esta infraestructura como “servicio” llamado *Infrastructure Provider*, migrando los recursos desde los SPs al IPs y, as, los SPs pueden ganar flexibilidad y reducir costes como se puede ver en la figura 2.6.

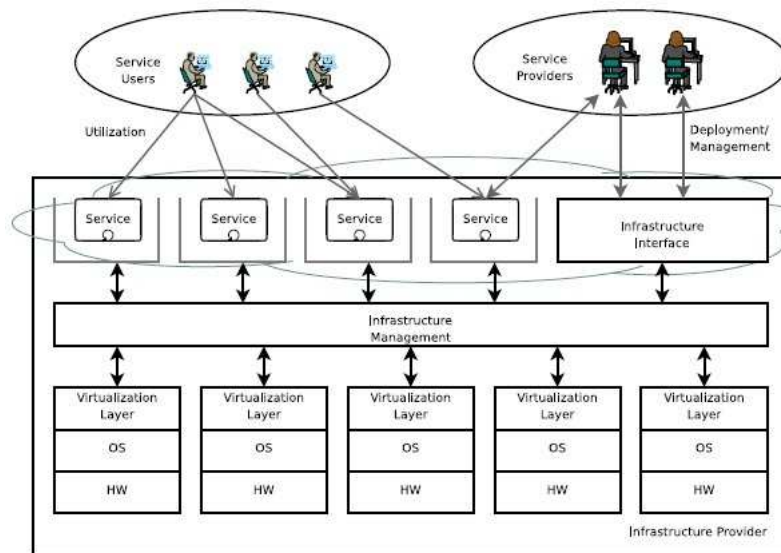


Figure 2.7: Actores en un sistema en la nube.

2.5 Comparacin entre servicios web y Grid computing/Cloud computing

Como es sabido, nuestro grupo de investigacin ha centrado su investigacin en el desarrollo de una metodologa que permita construir y verificar sistemas con restricciones temporales mediante el uso de tcnicas formales. En los ltimos aos, se ha aplicado esta metodologa en el rea de los servicios web, ms concretamente, en que estos servicios cumplan la tarea que se les encomienda y que se coordinen automticamente para conseguir llevar a cabo un trabajo ms general. El problema que estn teniendo los servicios web es que como tuvieron

un gran auge hace pocos años, muchos grupos de investigación centraron sus estudios en este campo y, por tanto, hay muchos investigadores proponiendo nuevas aproximaciones y esto ha llevado a que existen ciertas partes como BPEL o WS-CDL que están bastante estudiadas. De esta manera, está surgiendo un sistema donde los métodos formales pueden jugar un papel muy importante y donde nuestro grupo puede beneficiarse de su amplia experiencia tanto en formalización como en servicios web, el cloud computing. Este nuevo paradigma, como se ha expuesto anteriormente, está viviendo su época de plenitud en este momento y grandes empresas como Google, IBM, Microsoft han decidido dar un paso al frente y apostar fuertemente por la computación en la nube. Además, muchos gobiernos están interesados en migrar sus servicios a la nube para abaratar costes y permitirles escalabilidad cuando les sea necesaria. Por ejemplo, hay que preguntarse si es necesario para la Agencia Tributaria tener grandes centros de datos cuando la demanda de servicios por parte de los ciudadanos sólo crece en la época de la declaración de la renta. Probablemente la respuesta sea afirmativa porque se necesita almacenar todos esos datos y dar cierta confianza de que tus datos fiscales no van a caer en manos de gente con no muy buenas intenciones, pero toda la necesidad de cálculo si que se puede externalizar para ahorrar costes en equipamiento o incluso podrán crear una nube privada entre todos los organismos que colaboren con la agencia pública para compartir recursos e información. En este sentido, este tipo de sistemas donde la seguridad, privacidad y la disponibilidad son un requisito innegociable es donde podemos centrar parte de nuestras investigaciones e intentar mejorar alguno de los componentes de la arquitectura cloud expuesta en el apartado anterior. Por ejemplo, la mayoría de grupos de investigación desarrollan herramientas, pero la fase de pruebas o no existe o se le dedica poco tiempo. La semana pasada nos reunimos con uno de los grandes investigadores en el campo del Grid/Cloud Computing, Karim Djemame, y nos contó que el principal problema que tenía era ese que no sabían concretamente porque funcionaba bien su herramienta y que estaba bastante interesado en la verificación de su herramienta.

Por otro lado, a continuación se enumeran algunas diferencias entre servicios web y grid/cloud computing para ver donde es posible aplicar nuestra experiencia en este sistema. En primer lugar, podemos considerar que los servicios

web son en s software que se ofrece como servicio (SaaS), aunque existan ciertas diferencias entre ambos enfoques, por ejemplo, la estandarizacin. Por tanto, este software podra estar compuesto de un conjunto de servicios, probablemente comunicados a travs de Internet, y que se coordinan para realizar una determinada tarea. Hasta el momento, nada nuevo, pero la principal diferencia reside en la virtualizacin, ya que los diferentes servicios que ofrece la nube se realizan en mquinas virtuales en lugar de directamente sobre un servidor como puede ser el caso del servicio web, de manera que la concurrencia en el sistema es mayor.

Otra diferencia es la persistencia de los datos. Si queremos coordinar varios servicios web para que realicen sumas la nica posibilidad de que stos puedan almacenar el resultado es guardndolo en la base de datos, sin embargo, existe una aproximacin llamada WSRF (Web Services Resources Framework) que ha sido estandarizada y que resuelve este problema. En este framework cada servicio web lleva asociado un recurso o varios del sistema de manera que puedes interactuar con el servicio y decidir a que recurso acceder. La principal ventaja que tiene es que todos los servicios se definen con WSDL (Web Services Description Language) y que la comunicacin, direccionamiento, etc. est estandarizado, de manera que la colaboracin entre sistemas de este tipo es sencilla. Otra ventaja es que el usuario tiene la posibilidad de decidir con que recursos interacta. As, podremos aadir una capa inferior en nuestra metodologa que permitiese la definicin de servicios web con recursos y una vez verificado que el sistema es correcto, desplegar estos servicios web en las mquinas fsicas. Este enfoque encajara perfectamente con nuestra investigacin, ya que utiliza servicios web con recursos y estos recursos tienen restricciones temporales para evitar que un usuario abarque todo el sistema.

Tambin, podemos observar que cloud computing podra verse como una capa que se colocara debajo de los servicios web, ya que se puede utilizar stos para acceder a los recursos, pero hay que resaltar que la nube no es solo ofrecer software como servicio, sino que tambin hay infraestructura y plataforma como servicio, cosa que los servicios web no pueden abarcar. Es decir, una parte del cloud computing (SaaS) puede compararse directamente con los servicios web,

pero las otras dos partes no tienen nada que ver, por lo que sera como comparar el protocolo TCP/IP con la arquitectura de un PC, aunque es necesario que ambas aproximaciones (servicios web y cloud computing) converjan para el crecimiento de ambos paradigmas, igual que grid computing y servicios web convergieron en WSRF.

Por ltimo, a modo de curiosidad la principal diferencia entre un sistema grid y uno cloud reside en la virtualizacin, ya que en grid el usuario no comparte en tiempo real los recursos que tiene asignados, mientras que en cloud es indispensable la virtualizacin de recursos para conseguir dar servicio a ms clientes y conseguir ese ahorro que prometen los proveedores.

2.6 Web Services Resource Framework (WSRF)

La arquitectura que presentan los servicios web ha sido ampliamente aceptada como medio para estructurar las interacciones existentes entre los servicios que forman parte de un sistema distribuido y que colaborar para conseguir un objetivo comn. En la actualidad, los desarrolladores requieren a los entornos una mayor estandarizacin para facilitar interoperatividad adicional entre dichos servicios, pero hasta mediados de 2004 ningn grupo de investigacin o grupo de expertos se haba planteado seriamente la idea de proponer un estndar para modelar la comunicacin entre servicios web que poseen recursos persistentes asociados. As, en Enero de ese ao, varios miembros de la organizacin *Globus Alliance* y de la multinacional informtica IBM definieron, con la ayuda de expertos de empresas como HP, SAP, Akamai, etc., la especificacin de los documentos que deberan producirse en este modelo y la base de una arquitectura inicial. Estos documentos fueron enviados a la organizacin encargada de su estandarizacin, OASIS, en Marzo de 2004. En un principio, se formaron dos comits que se encargaran del estudio y desarrollo de ciertas partes de este nuevo estndar. Por un lado, estaba el *WSRF Technical Committee* que gestionaba cuatro especificaciones: *WS-ResourceProperties*, *WS-ResourceLifetime*, *WS-ServiceGroup*, y *WS-BaseFaults*. Por otro lado, el *WSN Technical Committee* se encargaba de las especificaciones: *WS-BaseNotification*, *WS-Topics*, y *WS-BrokeredNotification*.

WS-Resource Framework est inspirado en el trabajo realizado previamente por el *Global Grid Forum's Open Grid Services Infrastructure (OGSI) Working Group* [9]. Ms concretamente, puede ser visto como una sencilla refactorizacin de los conceptos e interfaces desarrollados en la especificacin *OGSI V1.0*, de manera que explota los recientes desarrollos en el rea de los servicios web (por ejemplo, WS-Addressing).

El objetivo de este trabajo es introducir los conceptos fundamentales para la gestin y destruccin de servicios web persistentes, es decir, servicios web que lleven asociados recursos donde guardar los estados de los mismos, ya que hasta la aparicin de esta aproximacin, los servicios web eran considerados “*stateless*” y, por tanto, no podan almacenar temporalmente datos o resultados de sus operaciones de una manera sencilla para el usuario, ya que era necesario almacenarlos en una base de datos ajena al servicio. En este enfoque, es necesario codificar la relacin entre el servicio y el recurso en trminos de patrones utilizando una serie de tecnologas ampliamente estudiadas, como, por ejemplo, el WS-Addressing y, tambin, ser necesario hacer sus propiedades accesibles desde el exterior a travs de un interfaz. En este sentido, llamaremos *WS-Resource* a la asociacin entre un servicio web y un recurso persistente.

2.6.1 Introduccin

WS-Resource Framework [2] es una especificacin, desarrollada por OASIS y algunas de las empresas informticas ms pioneras, cuyo propsito es definir un marco genrico para el modelado y acceso a recursos asociados a servicios web, as como las relaciones entre dichos recursos en un entorno Grid/Cloud. Esta aproximacin est compuesta por un conjunto de especificaciones que definen la representacin del WS-Resource en los trminos que especifican los mensajes intercambiados y los documentos XML relacionados. Asimismo, incluye mecanismos que describen el medio para consultar el estado de un recurso y la descripcin del servicio, que forman conjuntamente la definicin de un WS-Resource. Adems, definen los pasos necesarios para hacer el estado de un servicio web accesible a travs de su interfaz (descrita en WSDL).

Normalmente, las interfaces de los servicios web proporcionan al usuario la posibilidad de acceder y manipular el estado del mismo, como, por ejemplo, valores de datos que evolucionan por la interacción entre varios servicios. En otras palabras, los intercambios de mensajes que se implementan en el comportamiento de los servicios tienen como objetivo permitir el acceso a estos recursos persistentes. Sin embargo, la noción de recursos persistentes que subyace en la implementación de los servicios no es tan evidente en la definición de la interfaz [5]. Los mensajes que estos servicios envían y reciben implican (o animan al programador a inferir) la existencia de un tipo de recurso asociado. Por tanto, es deseable que se definan estándares que permitan el descubrimiento, creación, introspección, interacción y destrucción de dichos recursos y que la forma elegida para llevar a cabo esta misión sea lo más interoperable posible. Estas observaciones han motivado la aparición de la propuesta comentada anteriormente, WS-Resource, para modelar estados en el contexto de los servicios web. Un WS-Resource se define como la composición de un servicio web y sus recursos persistentes asociados, esto es, (i) expresado como una asociación de un documento XML con un tipo definido con uno o varios *portTypes* (un servicio podrá jugar un determinado rol si implementa todos los *portTypes* que comprenden ese rol) y (ii) direccionado y accedido de acuerdo al patrón del recurso implícito, una derivación de las *Endpoint References* del WS-Addressing. Una *Endpoint Reference* estará compuesta por: Uniform Resource Identifier (URI), parámetros del mensaje que se envíe para solicitar el envío de la *Endpoint Reference* y datos relativos a la interfaz que se usa. En este intercambio, el identificador del recurso persistente es encapsulado en una *Endpoint Reference* y usado para identificar al recurso en cualquier intercambio de mensajes entre los servicios que formen la coreografía. Así, WSRF permite declarar, acceder, monitorizar y destruir WS-Resources mediante mecanismos convencionales, lo que facilita la tarea de gestión, ya que no es necesario hacer más difícil la lógica de decisión del servicio propietario del recurso para procesar los mensajes de gestión. Estos mecanismos convencionales componen cinco especificaciones técnicas que definen los medios por los cuales:

- Se destruye un WS-Resource, ya sea de manera síncrona con respecto a una petición explícita de destrucción o, a través de un mecanismo basado en tiempos (scheduled). Además, es posible declarar unas características específicas de los recursos (WS-ResourceProperties) que podrán ser utilizadas

para inspeccionar y monitorizar el tiempo de vida de dicho WS-Resource (WS-ResourceLifetime).

- Se definen los tipos de WS-Resource, que estn compuestos por la interfaz de la descripcin del servicio web (WSDL) y por un documento XML de propiedades del recurso. Por otro lado, el estado del WS-Resource puede ser consultado y modificado a travs del intercambio de mensajes (WS-ResourceProperties)
- Un Endpoint Reference (WS-Addressing) puede ser renovado cuando su informacin de direccionamiento ha caducado o ha dejado de ser vlida por algn error (WS-RenewableReferences).
- Adems, se define la capacidad de implementar entornos heterogneos como colecciones de servicios web, sean o no WS-Resources (WS-ServiceGroups).
- La notificacin de errores puede ser ms estandarizada al usar tipos XML Schema para definir los fallos base y definir reglas que muestren cmo esos fallos son usados y extendidos (WS-BaseFaults).

2.6.2 WS-ResourceProperties

Como se ha comentado anteriormente, WSRF utiliza una especificacin concreta para definir las propiedades del WS-Resource. Este recurso estar compuesto por la definicin de la interfaz en WSDL y un documento XML (Resource Properties Document) que especifica las propiedades del mismo, por ejemplo, el tamao de disco, la capacidad del procesador, etc., de tal manera que si queremos acceder, modificar o actualizar este documento debemos utilizar una serie de mensajes preestablecidos en la especificacin. Las operaciones que se pueden hacer son las siguientes:

GetResourceProperty

Esta operacin como su propio nombre indica permite al servicio web que realiza la peticin recuperar el valor de una **nica** propiedad del documento de propiedades. Para aclarar ms los conceptos se define el siguiente ejemplo.

Dado el documento de propiedades:

```
...
<GenericDiskDriveProperties
xmlns: tns = ''http://example.com/diskDrive'' >
  <tns:NumberOfBlocks>22</tns:NumberOfBlocks>
  <tns:BlockSize>1024</tns:BlockSize>
  <tns:Manufacturer>DrivesRUs</tns:Manufacturer>
</GenericDiskDriveProperties>
...
```

Una posible petición puede ser:

```
...
<s12:Body>
  <wsrp:GetResourceProperty
    xmlns:tns = ''http://example.com/diskDrive''>
      tns:NumberOfBlocks
  </wsrp: GetResourceProperty>
</s12:Body>...
```

GetMultipleResourceProperties

Este método es equivalente al anterior, pero para acceder a más de una propiedad del documento en el mismo mensaje, es decir, se utiliza para evitar congestionar la red. El mensaje enviado será:

```
...
<wsrp:GetMultipleResourceProperties
xmlns:tns = ''http://example.com/diskdrive'' >
  <wsrp:ResourceProperty>tns:NumberOfBlock</wsrp:ResourceProperty>
  <wsrp:ResourceProperty>tns:BlockSize</wsrp:ResourceProperty>
</wsrp:GetMultipleResourceProperties>
...
```

SetResourceProperties

Este método se utiliza para realizar cambios en el documento de propiedades. Existen 3 tipos de cambios:

- Insert: Permite aadir nuevas propiedades en el documento.
- Update: Se utiliza para actualizar el valor de alguna propiedad.
- Delete: Elimina propiedades del documento.

Un posible ejemplo de peticin sera:

```
...
<s12:Body>
  <wsrpw:SetResourceProperties
    xmlns:tns='http://example.com/diskdrive '>
    <wsrp:Update>
      <tns:NumberOfBlocks>143</tns:NumberOfBlocks>
    </wsrp:Update>

    <wsrp>Delete resourceProperty='tns:Manufacturer '/>

    <wsrp:Insert>
      <tns:someElement>42</tns:someElement>
    </wsrp:Insert>

  </wsrpw:SetResourceProperties>
</s12:Body>
...
```

El documento de propiedades quedara con el siguiente formato:

```
...
<GenericDiskDriveProperties
  xmlns:tns='http://example.com/diskDrive '>

  <tns:NumberOfBlocks>143</tns:NumberOfBlocks>
  <tns:BlockSize>1024</tns:BlockSize>
  <tns:someElement>42</tns:someElement>

</GenericDiskDriveProperties>
...
```

QueryResourceProperties

Como su propio nombre indica, este mtodo se utiliza para realizar consultas sobre propiedades del recurso. Por ejemplo si queremos saber si el nmero de bloques es mayor que 20 y el tamao de bloque es 1024 realizaramos la siguiente consulta:

```
...
<s12:Body>
  <wsrp:QueryResourceProperties>
    <wsrp:QueryExpression
      Dialect = ''http://www.w3.org/REC-xpath-19991116''>
      boolean(/*/NumberOfBlocks>20 and */BlockSize=1024)
    </wsrp:QueryExpression>
  </wsrp:QueryResourceProperties>
</s12:Body>
...
```

La respuesta que envía el otro servicio es:

```
...
<s12:Body>
  <wsrp:QueryResourcePropertiesResponse>
    true
  </wsrp:QueryResourcePropertiesResponse>
</s12:Body>
...
```

2.6.3 WS-Base Faults

El diseñador de un servicio web normalmente utiliza interfaces definidas por otros, por lo que un método que estandarizase el formato de los mensajes de notificación de errores facilitara la labor de los desarrolladores. Este es el objetivo de WS-BaseFaults. Los mensajes de fallos en WSRF tienen el siguiente formato:

```
...
<BaseFault>
  <Timestamp>xsd:dateTime</Timestamp>
  <OriginatorReference>
    wsa:EndpointReferenceType
  </OriginatorReference> ?
  <ErrorCode dialect='anyURI'>xsd:string</ErrorCode>?
  <Description>xsd:string</Description> *
  <FaultCause>wsbf:BaseFault</FaultCause> *
</BaseFault>
...
```

donde:

- **Timestamp:** Hora exacta cuando el fallo ha ocurrido.
- **OriginatorReference:** Dirección en formato WS-Addressing del servicio que ha generado el fallo.
- **ErrorCode:** Código de error para ser utilizado por sistemas de información de fallos, por ejemplo, POSIX errno.

- Description: Explicacin de la causa del fallo (en lenguaje natural).
- FaultCause: Causa tcnica del fallo.

2.6.4 WS-ServiceGroup

Esta especificacin permite crear grupos que comparten una serie de propiedades en comn, es decir, agrupar diferentes servicios web que tienen comportamientos similares.

2.6.5 WS-ResourceLifetime

El tiempo de vida de un WS-Resource se define como el periodo que transcurre entre su instanciacin y su destruccin. La misin de esta especificacin es estandarizar el proceso de destruccin de un recurso y definir mecanismos para monitorizar este ciclo de vida, pero lo que no se define es cmo crear el WS-Resource. Generalmente, en los sistemas distribuidos, los clientes slo quieren tener un recurso por un determinado intervalo de tiempo, aunque en muchos escenarios es ms apropiado para el cliente que se produzca la inmediata destruccin del recurso. Otro ejemplo claro de uso se presenta cuando el cliente quiere suscribirse a un servicio por un cierto tiempo y quiere que despus de este tiempo se destruya dicha unin. Como se coment en la introduccin, existen dos formas de destruir un recurso: inmediata, mediante un mensaje explcito o temporizada, mediante un mensaje que activa o gestiona un timer.

Destruccin inmediata

Para la destruccin inmediata slo hace falta poner `< wsrl : Destroy / >` dentro del cuerpo (`< Body >`) del mensaje SOAP que se enva al servicio que gestiona el recurso y dicho servicio responder con `< wsrl : DestroyResponse / >` dentro del cuerpo (`< Body >`) del mensaje SOAP de respuesta.

Destrucción temporizada

En este caso, el WS-Resource tiene asociado un tiempo de terminación que define el tiempo después del cual se espera que el recurso haya sido destruido y, razonablemente, se espera que antes del mismo el recurso esté disponible. A continuación se muestra un ejemplo de cómo determinar el tiempo de terminación de un recurso:

```
...
<s12:Envelope
  <ex:ResourceDisambiguator>
    uuid:ba32-8680cace43f9
  </ex:ResourceDisambiguator>
  <s12:Body>
    <wsrl:SetTerminationTime>
      <wsrl:RequestedTerminationTime>
        2001-12-31T12:00:00
      </wsrl:RequestedTerminationTime>
    </wsrl:SetTerminationTime>
  </s12:Body>
</s12:Envelope>
...
```

Como podemos observar el servicio que solicita la destrucción puede indicar la hora de destrucción y la hora actual (para evitar desajustes por la forma de representar la zona horaria). Una vez que `CurrentTime` alcanza el valor `TerminationTime`, el recurso se destruye sin ninguna intervención `ms` y se notifica al emisor del mensaje de destrucción que el recurso deja de estar disponible. Existe otro mensaje que se manda desde el receptor al emisor para comunicarle que ha recibido la petición de cambio.

Sin embargo, puede darse la situación de que haya `ms` de un servicio utilizando el recurso que vaya a destruirse por lo que el propietario del recurso puede decidir o no (se deja a libre elección del programador) implementar los mensajes WS-Notification para informar a los interesados que el recurso deja de estar disponible. Para llevar a cabo esta tarea debe crear este Topic:

```

...
<wstop:TopicSpace name='ResourceLifetime'
  targetNamespace=
    http://docs.oasis-open.org/wsr/2004/06/
    wsrf-WS-ResourceLifetime-1.2-draft-01.xsd

  <wstop:Topic name='ResourceTermination'>
    <wstop:MessagePattern>
      <wsrp:QueryExpression
        dialect= http://www.w3.org/REC-xpath-19991116 >
        boolean(/*/TerminationNotification)
      </wsrp:QueryExpression>
    </wstop:MessagePattern>
  </wstop:TopicSpace>
...

```

Adems, el mensaje de notificacin asociado debe contener los siguientes campos:

```

...
<wsrl:TerminationNotification>
  <wsrl:TerminationTime>xsd:dateTime</wsrl:TerminationTime>
  <wsrl:TerminationReason>xsd:any</wsrl:TerminationReason>?
</wsrl:TerminationNotification>
...

```

donde *TerminationTime* informa de la fecha de destruccin y *TerminationReason* contiene la explicacin de la destruccin.

2.7 WS-Notification

Esta especificacin permite a un *NotificationProducer* enviar un mensaje de notificacin a un *NotificationConsumer* de dos maneras diferentes:

1. El *NotificationProducer* enva un mensaje de notificacin al *NotificationConsumer* sin seguir ningn formalismo.

2. El *NotificationProducer* utiliza el formalismo que se describe a continuacin para enviar las notificaciones.

La opcin a utilizar la elegir el suscriptor cuando mande la peticin de suscripcin. En este sentido, la segunda opcin permite al usuario recibir un amplio rango de mensajes de notificacin, ya que la informacin que se enva en estos mensajes se obtiene de un rbol de Topics (temas) y, por tanto, se permite enviar subrboles en un mismo mensaje para informar de diferentes Topics. En la Figura 2.7 vemos un ejemplo:

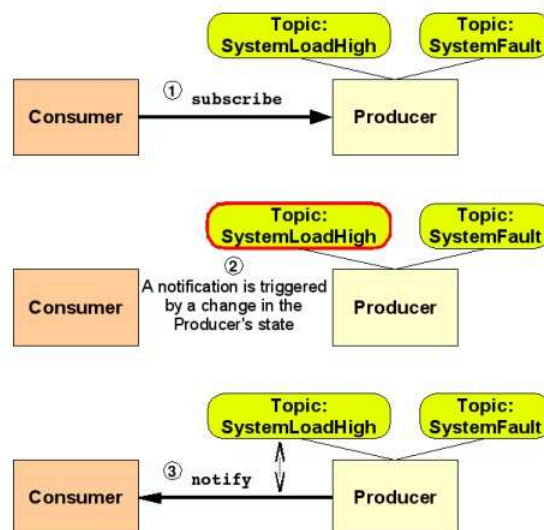


Figure 2.8: Ejemplo de uso de WS-Notification sin broker.

Este caso muestra un ejemplo de interaccin entre un consumidor y un productor de notificaciones, en el caso de que el suscriptor y el consumidor sean la misma entidad. El sistema es simple ya que tenemos un consumidor y un productor que publica 2 topics: *SystemLoadHigh* y *SystemFault*. Los pasos necesarios son:

1. En primer lugar, el consumidor se suscribe al topic *SystemLoadHigh*, por lo que internamente se crea un *Subscription resource* con la informacin de la suscripcin. El productor debe implementar un mtodo *Subscribe* y el consumidor un mtodo *Notify*.

2. Despues, el productor debe enviar una notificacin cuando el sistema sobrepase una determinada carga de trabajo. Por ejemplo, nuestro sistema enviar notificaciones cuando la carga de trabajo sea mayor de 50%.
3. Por ltimo, el productor enva la notificacin invocando la operacin *Notify* en el consumidor.

Un ejemplo de mensaje *Notify* es:

```
...
<wsnt:Notify>
  <wsntw:NotificationMessage>
    <wsnt:Topic Dialect= xsd:anyURI >
      {any}
    </wsnt:Topic>
    <wsnt:ProducerReference?>
      wsa:EndpointReference
    </wsnt:ProducerReference>
    <wsnt:Message>xsd:any</wsnt:Message>
  <wsnt:NotificationMessage>+
</wsnt:Notify>
...
```

Como podemos observar el mensaje *Notify* contiene uno o varios mensajes de notificacin (*NotificationMessages*). Los campos dentro de stos son:

- Topic: La informacin del topic que se enva.
- Dialect: El dialecto usado para expresar el topic anterior, es decir, el lenguaje utilizado para expresarlo.
- ProducerReference: Direccin del productor.
- Message: Una copia de la carga til (payload) del mensaje actual.

A continuacin, se muestra el mensaje que manda el suscriptor para registrar su inters en uno o ms topics:

```
...
<wsnt:Subscribe>
```

```

<wsnt:ConsumerReference>
  wsa:endpointReference
</wsnt:ConsumerReference>
<wsnt:TopicExpression Dialect = xsd:anyURI >
  {any}
</wsnt:TopicExpression>
<wsnt:UseNotify>xsd:boolean</wsnt:UseNotify>?
<wsnt:Precondition>wsrp:QueryExpression</Precondition>?
<wsnt:Selector>wsrp:QueryExpression</wsnt:Selector>?
<wsnt:SubscriptionPolicy>{any}</wsnt:SubscriptionPolicy>?
<wsnt:InitialTerminationTime>
  xsd:dateTime
</wsnt:InitialTerminationTime>?
</wsnt:Subscribe>

...

```

Los conceptos importantes en este mensaje son *UseNotify* que se utiliza para decidir si el mensaje de notificación sigue el formalismo WS-Notification o se manda sin formato, *Precondition* que es la condición que genera mensajes de notificación, es decir, si se cumple esta condición se generan mensajes, pero debe cumplirse también la condición *selector* para enviarlos a los destinatarios que es la que se usa para decidir si se transmiten o no los mensajes generados. Además, *SubscriptionPolicy* se podrá utilizar para controlar el ratio de envío de mensajes (por ejemplo, no más de 3 por segundo) y *InitialTerminationTime* contiene una sugerencia del tiempo de vida de la suscripción. WSRF también incluye mensajes para detener la suscripción, reanudarla o para que un servicio que acaba de unirse a una suscripción pueda obtener un historial de notificaciones sobre un determinado topic.

2.7.1 WS-BrokeredNotification

Un *NotificationBroker* es un intermediario que, entre otras cosas, permite el envío de mensajes entre uno o varios *Publishers* y uno o varios *NotificationConsumers*. La misión del *Publisher* es observar ciertas situaciones y crear mensajes de notificación para informar de esas situaciones, mientras que el broker es el encargado

de distribuir estos mensajes.

En este caso, se pueden dar tres relaciones entre las partes: *simple publishing*, *composable publishing* y *demand-based publishing*. En el primer caso, el *Publisher* es el encargado de observar las situaciones y notificarlas al broker que ser el encargado de transmitirlos a los interesados. En el segundo caso, el papel del *Publisher* lo realiza una entidad que implementa una serie de servicios especificados en WS-Notification (*NotificationProducer*). En este caso, el mensaje de notificación puede llegar a otros consumidores que estuviesen suscritos al productor. En ambos casos, el broker puede pedir al *Publisher* que se registre para poder publicar mensajes sobre un topic determinado. El último enfoque (*demand-based publishing*) requiere que el *Publisher* sea un *NotificationProducer* y, as, acepte mensajes de suscripción. El objetivo es reducir el número de mensajes de notificación haciendo que estos solo se manden cuando se soliciten expresamente.

This chapter is about the state of the art of Service-Oriented Computing, the use of formalisms for the analysis of Web Service compositions, and the specification of electronic contracts (e-contract). We specially focus on the application of e-contracts to Web Service compositions, describing some of the existing approaches for that purpose. We also review the related work in the field of formal specification and verification of e-contracts.

2.8 Service-Oriented Computing (SOC)

The Service-Oriented Computing (SOC) paradigm promotes the use of services for the development of massively distributed applications, trying to achieve the creation of fast, low-cost, flexible and scalable applications [?]. Services are the main building block of this paradigm, being these services self-describing and platform-independent. Thanks to the use of standards for description, publication, discovery and invocation, the services can be integrated without taking care of the low-level implementation details of each service. The aim of SOC is to make it possible the creation of dynamic business processes and agile applications by providing an easy way to assemble application components into a loosely coupled network of services.

2.8.1 Service-Oriented Architecture (SOA)

To reach the goals of SOC, a Service-Oriented Architecture (SOA) is defined. SOA is a software architecture based on the utilization of services, being these services provided to the user of the application or to other services in the network. This is possible by the use of service interfaces that can be published and discovered.

SOA is based on a model of roles where every service can play multiple roles. For example, a service can offer a certain functionality to a user and, at the same time, being the consumer of the functionality provided by some other services. Such model reduces the complexity of applications and increases their flexibility.

Although at the beginning of SOA there were several architectures aspiring to become SOA standards [?, ?], the most successful one was the architecture based on Web Services.

2.8.2 Web Services (WS)

W3C defines a Web Service (WS) in the following way [?]:

“A Web Service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web Service in a manner prescribed by its description using SOAP-messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.”

We can see in this definition that there are two basic standards related to Web Services: Web Service Description Language (WSDL) for the definition of the service functionality and its properties [?], and Simple Object Access Protocol (SOAP) for the exchange of XML messages between services [?]. There is also an additional standard called Universal Description, Discovery and Integration (UDDI) used to create Web Service directories and to search for services in the network [?]. The use of these standard protocols is the key point to improve the integration between different parties in a Web Service architecture.

In Figure 2.8 a possible representation of the Web Service architecture stack is shown. We can see that the three standards described above are only a small part of the stack. We also need protocols to define security aspects (ensuring that exchanges of information are not modified or forged in a verifiable manner and that parties can be authenticated), to provide reliable messaging for the exchange of information between parties, to specify the collaboration between services when we compose them, to individually describe the behaviour of each service in a business process, etc. The problem is that whereas the standards for basic services (WSDL and SOAP) are widely adopted for their respective purposes, the situation is not so clear when we talk about composing services, having multiple protocols aspiring to become a standard in this layer.

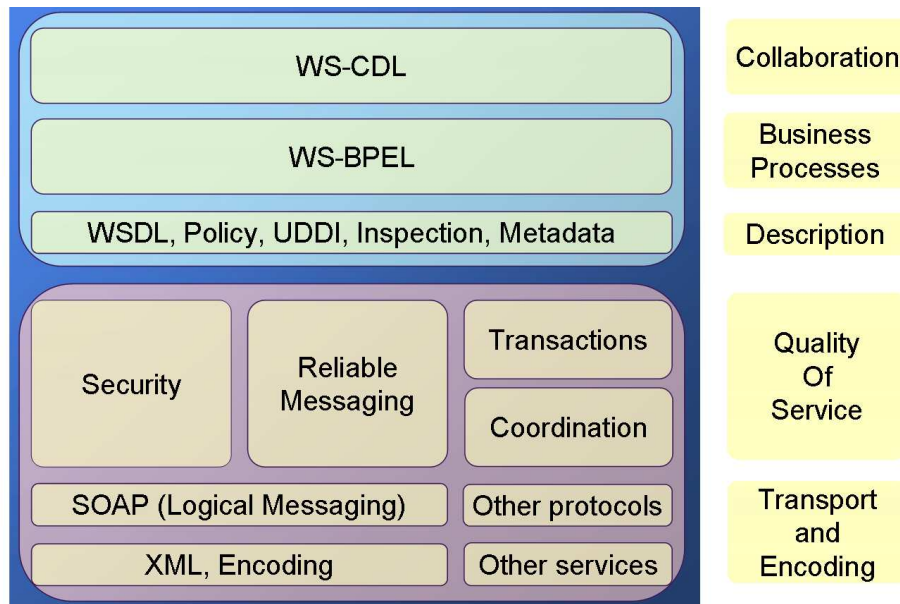


Figure 2.9: Web Service architecture stack.

2.8.3 Web Service Composition

Two different approaches can be followed when we talk about composing Web Services. They are called *orchestration* and *choreography*. The former describes the individual business process followed by each one of the participants in the composition, while the latter describes the composition from a global viewpoint, defining the interactions (exchange of messages) happening between the parties,

that is, how they collaborate in the composition. However, the ideal solution would be fusing both approaches in a single language and environment [?].

Anyway, the languages we can use in both cases should accomplish some common goals: (i) the capacity of modelling service interactions, including control flow and data constraints, (ii) the possibility of specifying exceptional behaviour, indicating which errors can happen in the execution of the composition and the way of handling these errors, and (iii) the ability to model Web Service compositions at a high level, without taking care of the implementation details of each one of the services.

Business Process Execution Language (BPEL)

Considering the orchestration approach, one of the most used languages for this purpose is Business Process Execution Language (BPEL), also known as BPEL4WS or WSBPEL [?]. A BPEL specification models the behaviour of the different services taking part in a business process, defining a BPEL source file for each service. BPEL provides an XML syntax to describe the control logic necessary to coordinate all the services participating in the process. An orchestration engine is responsible for the execution of the processes, coordinating all the activities and compensating any error that could happen. BPEL can be seen as a layer over WSDL, providing this latter language the available functionality while BPEL defines how to sequence the operations [?].

The components of a BPEL specification are the following:

- **Events**, which describe the flow execution in an event-driven way.
- **Variables**, which are defined by using WSDL schemes for internal or external purposes, and are used in the message flow.
- **Correlations**, which identify the processes interacting by means of messages.
- **Fault handling**, defining the behaviour when an exception is thrown.
- **Event handling**, defining the behaviour when an event happens.
- **Activities**, which are the basic unit of behaviour of a Web Service.

For example, in figure 2.9 we can see part of a BPEL specification corresponding to shipping service.

2.9 Formal Analysis of Web Service Compositions

In last years the formal analysis of Web Service compositions has been recognized as an important problem that needs to be solved [?, ?, ?, ?]. A strict analysis is necessary to guarantee the correct composition of the services for multiple reasons: the integration of several independent applications, the possibility of having incomplete specifications of some services, the impossibility of using traditional evaluation techniques such as software testing, ...

The analysis of Web Service compositions usually consists of checking its specification correctness by analysing if a set of functional and non-functional requirements are fulfilled. The behavioural requirements are especially interesting, as they specify the properties required to reach a concrete business goal. These requirements can include general properties such as deadlock freeness or correct termination, but they can also be defined explicitly, for example defining the conformance to a concrete model. This formal analysis is usually done by means of formal methods.

Formal methods [?] are a collection of notations and techniques for describing and analyzing systems [?]. They are called *formal* because they are based on mathematical theories (logics, automata, sets, ...). In formal methods we can distinguish between the *formal specification*, which describes unambiguously a system or its properties, and the *formal analysis* or *formal verification*, which serves to verify if a system satisfies its specification.

Formal methods usually cannot guarantee the correctness of the implementation and, if the specification is not correct, the verification results will also be wrong. Despite these and some other limitations, formal methods are still needed because they increase the confidence on system reliability, minimize the number of errors in the implementation and can find out errors impossible to find with other techniques such as testing.

When we are going to apply a formal method it is very important to choose the right level of abstraction in the specification of the system depending on what we


```
<process name="shippingService"
...
  <sequence>
    ...
    <if>
      <condition>
        bpel:getVariableProperty('shipRequest',
          'props:shipComplete')
      </condition>
      <sequence>
        ...
        <invoke partnerLink="customer"
          operation="shippingNotice"
          inputVariable="shipNotice">
          <correlations>
            <correlation set="shipOrder"
              pattern="request" />
          </correlations>
        </invoke>
      </sequence>
    <else>
      <sequence>
        ...
        <while>
          <condition>
            $itemsShipped
            <lt;
            bpel:getVariableProperty(
              'shipRequest', 'props:itemsTotal')
          </condition>
          <sequence>
            ...
            <invoke partnerLink="customer"
              operation="shippingNotice"
              inputVariable="shipNotice">
              <correlations>
                <correlation set="shipOrder"
                  pattern="request" />
              </correlations>
            </invoke>
            ...
          </sequence>
        </while>
      </sequence>
    </else>
  </if>
</sequence>
</process>
```

Figure 2.10: Part of a shipping service specification in BPEL

want to analyze or verify. An underspecification can lead to wrong verification results because the specification is incomplete, whereas an overspecification can lead to the *state explosion problem*, making the intended analysis of the system infeasible.

Formal methods can be applied in different stages of the computer system development process, providing different information about the system in each one of these stages. The application of these methods is usually complex, being impossible without some kind of tool support. For this purpose, the language syntax of the specification must be explicit and the language semantics of the specification must be restricted. In this way, formal specifications are amenable to automate analysis and verification.

The choice of using one formal method or another depends on many factors: the problem we want to solve, the type of system, the properties we want to check, and so on. In the case of complex systems, such as distributed concurrent systems, it is needed a combination of several techniques for the analysis.

2.9.1 Specification Formalisms

2.10 Summary

This chapter has described the state of the art of Service-Oriented Computing (SOC), the formalization of Web Service compositions, and the specification of electronic contracts (e-contracts).

The development of systems based on Web Services allows the creation of fast, low-cost, flexible and scalable applications, where the integration is possible thanks to the definition of multiple standard protocols (WSDL, SOAP, UDDI). However, the correct composition of Web Services is still an open problem, where different approaches can be followed, such as orchestration (BPEL) and choreography (WS-CDL, WSCI, OWL-S). The formal analysis of Web Service compositions by means of formal methods is therefore necessary to guarantee the correct composition.

Several formal techniques can be used for the analysis of Web Service compositions, being model checking one of the most popular. This is an automated

technique consisting of the construction of a finite-state model of the system to check if some properties are satisfied. Different specification formalisms can be used in model checking (process algebras, Petri nets, automata) and there are several tools supporting each one of these formalisms (CWB-NC, CPN Tools, UPPAAL).

Chapter 3

BPELRF

3.1 Introduction

The development of software systems is becoming more complex with the appearance of new computational paradigms such as Service-Oriented Computing (SOC), Grid Computing and Cloud Computing. Grid/Cloud environments are characterized by a dynamic environment due to the heterogeneity and volatility of resources. In these systems, the service provider needs to ensure some levels of quality and privacy to the final user in a way that had never been raised. It is therefore necessary to develop new techniques to benefit from the advantages of recent approaches, such as Web service compositions. There are two complementary views to composite Web services: Choreography and Orchestration. The choreography view describes the observable interactions among services and it can be defined by using specific languages such as Web Services Choreography Description Language (WS-CDL) or by using more general languages like UML Messages Sequence Charts (MSC). On the other hand, orchestration concerns the internal behaviour of a Web service in terms of invocations to other services. Web Services Business Process Execution Language (WS-BPEL) [?] is in general used to describe Web service orchestrations, and, consequently, this is considered the de-facto standard language for describing Web services in terms of Web service compositions.

In addition, complex Web systems are composed by smaller services where each one carries out different tasks. Thus, to model such systems, specification languages should offer designers the constructions required to compose and discovery those basic services. Web service compositions could be modelled by using one of the two standards previously presented (WS-CDL or WS-BPEL). Service discovery is the process of finding a suitable Web service for a given task, and, normally, its definition is implementation-dependent.

While Web service implementations are typically stateless, their interfaces frequently provide users with the ability to access and manipulate states, i.e., data values that persist across, and evolve as a result of Web service interactions. In other words, the message exchanges that Web services implement are frequently intended to enable access to stateful resources. Besides, the messages that the services send and receive imply (or programmers infer) the existence of an associated resource type. Therefore, it is desirable to define Web service conventions to enable the discovery of, introspection on, and interaction with stateful resources in standard and interoperable ways. Most important, such an approach improves the robustness of design time selection of services during application assembly, and runtime binding to specific resource instances [?]. To facilitate additional interoperability among services, more standardization is required to deal with distributed resources. In January of 2004, several members of the *Globus Alliance* organization and the computer multinational *IBM* with the help of experts from companies such as *HP*, *SAP*, *Akamai*, etc. defined the basis architecture and the initial specification documents of a new standard for that purpose, Web Services Resource Framework (WSRF) [?].

Likewise, it is important to define a mechanism to state how the users are notified when important situations occur. To this end, a publish/subscribe architecture might be implemented, which is based on information preferences expressed in advance. Thus, whenever new content is available on one resource, the publisher would send that information to the subscriber. As far as this work is concerned, services might express its disposition to receive notifications when certain conditions hold. As WSRF specifications encourage, the present work utilizes WS-Notification as basis to cope with publish-subscribe architecture.

After introducing the relevant concepts, let us remark the main contributions of this work. The integration of WS-BPEL and WSRF is not new, since, in the

literature, one can find some works defining this integration from a technical point of view. Some of these works will be presented in the Related Work section. Surprisingly, to the best of our knowledge, this is the first work, which taking as a starting point WSRF and WS-BPEL, defines a complete and formal language to model and analyse stateful Web service compositions. In addition to this, the necessary formal machinery to build a publish-subscribe architecture is also provided here improving the expressiveness of the language, and a formal primitive to discover new services is defined. It is worthwhile to mention that the aim of this paper is not to provide yet another WS-BPEL semantics since WS-BPEL has received much attention in recent years when many operational semantics for it have arisen. As opposed to this, the main aim here is to gather the benefits of putting together WS-BPEL and WSRF to manage stateful Web services workflows by using existing formalisms in distributed systems. Additionally, in order to deal with WSRF in a proper way, we have realised that it would be better to consider a semantic model with the appropriate “tools” to cope with all the relevant aspects of WSRF such as notifications and resources timeouts. To particularise and motivate our approach, we introduce now an ideal scenario. The scenario, where our approach might be applicable, is a *group buying*, also known as collective buying. This scenario consists of a group of sellers and buyers, where offers are interpreted as resources created by sellers. Buyers subscribe to those offers, and they are notified when the offers/resources meet their expectations. This simple scenario will allow us to show readers how the situation can be managed with what we consider a very simple and succinct language. The rest of the paper is organised as follows. In Section 3.2, we present the basic concepts for a better understanding of this paper and some related works. In Section 3.3, we define the language itself and its operational semantics, whereas, in Section 3.7, we illustrate by means of a case study how it works. Finally, Section ?? presents some conclusions and possible future directions to improve the present work.

3.2 Background and Related Work

Overview of BPEL/WSRF. WSRF [?] is a resource specification language developed by OASIS and some of the most pioneering computer companies, whose

purpose is to define a generic framework for modelling Web services with stateful resources (WS-Resource), as well as the relationships among these services in a Grid/Cloud environment. This approach consists of a set of specifications that define the representation of the WS-Resource in the terms that specify the messages exchanged and the related XML documents. These specifications allow the programmer to declare and implement the association between a service and one or more resources. It also includes mechanisms to describe the means to check the status and the service description of a resource, which together form the definition of a WS-Resource. This WS-Resource is accessible through its unique identifier, *EndPoint Reference (EPR)*, which is defined by using WS-Addressing.

On the other hand, Web services are becoming more and more important as a platform for Business-to-Business integration. Web service compositions have appeared as a natural and elegant way to provide new value-added services as a combination of several established Web services. Services provided by different suppliers can act together to provide another service; in fact, they can be written in different languages and can be executed on different platforms. As we noticed in the introduction, we can use Web service compositions as a way to construct Web service systems where each service is an autonomous entity which can offer a series of operations to the other services conforming a whole system. In this way, it is fairly necessary to establish a consistent manner to coordinate the system participants since each of them may use a different approach, and, consequently, it is common to use specific languages such as WS-BPEL to manage the system workflow. WS-BPEL, for short BPEL, is an OASIS orchestration language to specify actions within Web service business processes. These actions are represented by the execution of two types of activities (*basic* and *structured*) that perform the process logic. *Basic activities* are those which describe elemental steps of the process behaviour and *structured activities* encode control-flow logic, and can therefore contain other basic and/or structured activities recursively [?].

The WSRF elements considered in our language are¹:

- **WS-ResourceProperties:** There is a explicit specification to define WS-Resource properties, based on a Resource Properties Document (RPD), which

¹In WSRF there are some additional technical elements to increase the modelling power that due to its technical nature are not considered in our framework.

defines the properties of the associated resource (disk size, processor capacity, etc.). Nevertheless, for simplicity, we only consider a single property for each resource, which is an integer value. Among the operations allowed by the standard are *GetResourceProperty* and *SetResourceProperty*, which are used to manipulate the resource property values.

- **WS-ResourceLifetime:** The WSRF specification does not provide a standard way to create resources. However, resources have an associated lifetime, which means that once this time has elapsed, the resource is considered to be destroyed, and the subscribers are correspondingly notified. We have then included, for completeness, an operation to publish resources, *publishResource*, in which the initial value of the resource, its lifetime, a textual identifier in order to allow users to discover it, and the activity that must be launched upon its destruction are indicated. We also have an operation in order to modify the current resource lifetime, *setTimeout*.
- **WS-Notification:** Clients can subscribe to WSRF resources in order to be notified about some topics (resource conditions). We therefore include the *subscribe* operator for a customer to subscribe to a resource, indicating the condition under which it must be notified, and the activity that must be executed upon that event.

Related Work. WS-BPEL has been extensively studied with many formalisms, such as Petri nets, Finite State Machines and process algebras, but there are only a few works considering WS-BPEL enriched with WSRF, and they only show a description of this union, without a formalization of the model. In [?] Slomiski uses BPEL4WS in Grid environments and discusses the benefits and challenges of extensibility in the particular case of OGSI workflows combined with WSRF-based Grids. Other two works centered around Grid environments are [?] and [?]. The first justifies the use of BPEL extensibility to allow the combination of different GRIDs, whereas Ezenwoye et al. [?] share their experience on BPEL to integrate, create and manage WS-Resources that implement the factory/instance pattern.

On the other hand, Ouyang et al. [?] define the necessary elements for translating BPEL processes into Petri nets. Thus, they cover all the important aspects

in the standard such as exception handling, dead path elimination and so on. The model they consider differs from ours in that we formalize the whole system as a composition of orchestrators with resources associated, whereas they describe the system as a general scope with nested sub-scopes leaving aside the possibility of administering resources. Furthermore, we have also formalized the event handling and notification mechanisms. Another extensive semantics for BPEL 2.0 is presented in [?] by Dumas et al, which introduces two new interesting improvements. They define several patterns to simplify some huge nets and introduce the semantics for the WS-BPEL 2.0 new patterns. Related to π -calculus semantics, Dragoni and Mazzara [?] propose a theoretical scheme focused on dependable composition for the WS-BPEL recovery framework. In this approach, the recovery framework is simplified and analysed via a conservative extension of π -calculus. The aim of this approach clearly differs from ours, but it helps us to have a better understanding of the WS-BPEL recovery framework. Other work focused on the BPEL recovery framework is [?]. Although this is more focused in the compensation handler, they describe the corresponding rules that manage a Web service composition. Our work is therefore quite complete as we define rules for nearly all possible activities. In addition, we also consider time constraints. Finally, we would like to highlight the works of Farahbod et al. [?] and Busi et al. [?]. In the first one, the authors extract an abstract operational semantics for BPEL based on abstract state machines (ASM) defining the framework $BPEL_{AM}$ to manage the agents who perform the workflow activities. In this approach time constraints are considered, but they do not formalize the timed model. On the other hand, the goal of the latter one is fairly similar to ours. They also define a π -calculus operational semantics for BPEL and describe a conformance notion. They present all the machinery to model Web service compositions (choreographies and orchestrations). The main difference with our work is that we deal with distributed resources. In a similar fashion Luchi and Mazzara in [?] presents other π -calculus operational semantic, $web\pi_\infty$, which is centred on the idea of event notification as the unique error handling mechanism. It is clear that this proposal differs from ours since they focus their attention in the error handling mechanism, however their claiming of simplifying the error handling using only the notification mechanism can be performed in our proposal since this is the mechanism used in the resource framework and therefore a technique shared by WS-BPEL and WS-RF. For fur-

ther details about the formalization of service oriented languages we would like to encourage the reader to review the works presented at the SENSORIA project in [?]. Here, an extensive work is presented from different international research groups aimed by the common goal of providing a rigorous software engineering view point for service-oriented system using as a cornerstone the formal specification of Web Services and WS-BPEL in particular. Works such as SOCK [?], CaSPiS [?], COWS [?], B-lite [?] or Orc [?] are either presented or reviewed. The first one, SOCK (Service Oriented Computing Kernel [?]), is a formal calculus which aims at characterizing the basic features of Service Oriented Computing and takes its inspiration from WS-BPEL, considered by the authors as the “de facto” standard for Web Service technology. The second one, CaSPiS (Calculus of Services with Pipelines and Sessions [?]) uses the Java framework IMC. Authors take advantage of the already built-in IMC features such as session oriented and pattern matching communication mechanisms easing the task of implementing in Java all CaSPiS abstractions. Other one, COWS (Calculus for Orchestration of Web Services [?]), is a new foundational language for SOC whose design has been influenced by WS-BPEL. COWS combines a number of elements from process calculi, e.g. asynchronous communication, polyadic synchronization, pattern matching, protection, delimited receiving and killing activities.

3.3 Syntax and semantics of BPEL+RF

We use the following notation: *ORCH* is the set of orchestrators in the system, *VAR* is the set of integer variable names, *PL* is the set of necessary partnerlinks, *OPS* is the set of operations names that can be performed, *EPRS* is the set of resource identifiers ($EPRS \subseteq \mathbb{N}$), and *A* is the set of basic or structured activities that can form the body of a process. The specific algebraic language that we use for the activities is defined by the following BNF-notation:

$$\begin{aligned}
 A ::= & \text{throw} \mid \text{receive}(pl, op, v) \mid \text{invoke}(pl, op, v_1) \mid \text{exit} \mid \text{reply}(pl, op, v) \mid \\
 & \overline{\text{reply}}(pl, op, v_2) \mid \text{assign}(expr, v_1) \mid \text{empty} \mid A; A \mid A \parallel A \mid \text{while}(cond, A) \mid \\
 & \text{wait}(timeout) \mid \text{pick}(\{(pl_i, op_i, v_i, A_i)\}_{i=1}^n, A, timeout) \mid \text{getProp}(vEPR, v_1) \mid \\
 & \text{getTimeout}(vEPR, v_1) \mid \text{publishResource}(O, val, timeout, tag, vEPR, A) \mid \text{discover}(tag, vEPR) \mid \\
 & \text{setProp}(vEPR, expr) \mid \text{setTimeout}(vEPR, timeout) \mid \text{subscribe}(O, vEPR, cond', A)
 \end{aligned}$$

where $O \in ORCH$, $pl, pl_i \in PL$, $op, op_i \in OPS$, $timeout \in \mathbb{N}$, $expr$ is an arithmetic expression constructed by using the variables in VAR and integers; v, v_1, v_2, v_i range over VAR , tag is a string used to identify and discover resources that match a certain pattern, and $val \in \mathbb{Z}$. A variable $vEPR$ is used to store temporarily the resource identifier (EPR). A condition $cond$ is a predicate constructed by using conjunctions, disjunctions, and negations over the set of variables VAR and integers, whereas $cond'$ is a predicate constructed by using the variable $vEPR$, as representative of the resource value, and integers.

BPEL basic activities leveraged in our model are: *invoke* to request services offered by service providers, *receive* and *reply* to provide services to partners, *throw* to signal an internal fault explicitly, *wait* to specify a delay, *empty* to do nothing, *exit* to end the business process and *assign*, which is used to assign a variable value. The *structured activities* are: *sequence* (represented here as $;$), which contains two activities that are performed sequentially, *while* to provide a (conditional) repeated execution of one activity, *pick* that waits for the occurrence of exactly one event from a set of events (including an alarm event) executing then the activity associated with that event, and, finally, *flow* (\parallel operator in our syntax) to express concurrency. Another family of control flow constructs in BPEL includes event, fault and compensation handlers. An event handler is enabled when its associated event occurs, being executed concurrently with the main orchestrator activity. Fault handlers are performed when some failure has occurred, so the control is transferred to them. In this work, we only cover the fault and event handling, leaving compensation as a matter of future research. Besides, we do not take into consideration other advanced constructions such as correlation sets, dynamic partnerlinks or instance creation. However, an important aspect in current services technology is that of publishing and discovering of resources, which is considered in our framework. The correspondence among the syntax of WS-BPEL, WSRF, WS-Notification and our model is shown in Table 3.6. An orchestration is here defined as a pair $O = (A, A_f)$, where A and A_f are activities defined by the previous syntax. Specifically, A represents the normal workflow (and possible event handling activities which run in parallel with it), and A_f is the orchestrator fault handling activity.

Before we begin, we introduce some notations and definitions needed to describe the operational semantics.

Definition 1 State

We define a state as a pair $s=(\sigma, \rho)$, where σ represents the variable values in the system and ρ captures the global resource state. We characterise σ as a global function in \mathbb{Z}^{VAR} , but, in practice, each orchestrator will manage its own local variables. Furthermore, $\rho = \{(O_i, EPR_i, v_i, Subs_i, t_i, tag_i, A_{e_i})\}_{i=1}^r$, where r is the number of resources in the system. Each resource has an owner (publisher), O_i , a unique identifier, EPR_i , and, at each state, a particular value, v_i , and a lifetime, t_i , initialized with the activity *publishResource*, which can be changed by using the function *setTimeout*. A_{e_i} is the activity that must be run when it expires, whereas tag_i is used as a textual description for discovery purposes. The resources in ρ are therefore published by means of the *publishResource* activity, and potential subscribers must discover the resource identifier (EPR) by using the *discover* activity. Moreover, $Subs_i = \{(O_{ij}, cond'_{ij}, A_{e_{ij}})\}_{j=1}^{s_i}$, $i \in \{1, \dots, r\}$, is the set of resource subscribers, their associated delivery conditions and the event handling activity $A_{e_{ij}}$ that must be thrown in the case that $cond'_{ij}$ holds; s_i is the number of orchestrators currently subscribed to this resource and $O_{ij} \in ORCH$ are the subscriber identifiers.

Notation. Along the following lines, we introduce some notation used in the operational semantics. Given a state $s = (\sigma, \rho)$, a variable v and an expression e , we denote by $s' = (\sigma[e/v], \rho)$ the state obtained from s by changing the value of v for the evaluation of e , and $s^+ = (\sigma, \rho')$, where $\rho' = \{(O_i, EPR_i, v_i, Subs_i, t_i - 1, tag_i, A_{e_i}) \mid t_i > 1\}_{i=1}^r$. Let the function $Subs(s)$, which return the state s removing from each $Subs_i$ those subscriptions whose associated condition has held at s . We omit its formal definition since it is straightforward.

A partnerlink is here considered as a pair (O_i, O_j) representing the two roles in communication: sender and receiver. Furthermore, $\sigma(vEPR_i) \in \rho$ and $tag_i \in \rho$ will denote that there is a tuple $(O_i, EPR_i, v_i, Subs_i, t_i, tag_i, A_{e_i}) \in \rho$, where $\sigma(vEPR_i) = EPR_i$. Given a predicate $cond$, we use the function $cond(s)$ to mean the resulting value of this predicate at the current state s , $sel(\rho, tag)$ to return a randomly selected $EPR \in \rho$ among those whose tag attribute is tag , $val(\rho, tag)$ to return the current value of the resource, $time(\rho, vEPR)$ to return its lifetime and, finally, $getEPR()$ to generate non-repeated resource identifiers. Let $\sigma(vEPR) = EPR$, $\rho[w/vEPR]_1$ is used to denote that the new value in ρ of the resource EPR is w , $\rho[t/vEPR]_2$ denotes a change of the resource lifetime, and the function $Add_subs(\rho, vEPR_i, O_{ij}, cond'_{ij}, A_{e_{ij}})$

denotes that $(O_{i_j}, cond'_{i_j}, A_{e_{i_j}})$ is added to the subscribers of the resource $EPR_i \in \rho$ or $cond' = cond'_{i_j}$ in the case that O_{i_j} was already in $Subs_i$. At the same time, we need an additional function to launch the corresponding activities when the subscriber condition holds at the current state s . Let $s=(\sigma, \rho)$ with $\rho = \{(O, EPR_i, v_i, Subs_i, t_i, tag_i, A_{e_i})\}_{i=1}^r$, we define the function $N(O, s) = ||\{A_{e_{i_j}} | (O_{i_j}, cond'_{i_j}, A_{e_{i_j}}) \in Subs_i, O_i = O, cond'_{i_j}(s) = true\}_{i=1}^r$, with $j \in \{1, \dots, s_i\}$.

The operational semantics for this language is defined at three levels, the internal one corresponds to the evolution of one activity as a single entity. In the second one, we define the transition rules which establish the orchestrator evolution, whereas the third level corresponds to the composition of different orchestrators and resources to conform a choreography.

Definition 2 Activity Operational semantics

We define the activity operational semantics by using two types of transition:

1. $(A, s) \xrightarrow{a} (A', s'), a \in \text{Act}$ (Action transitions).
2. $(A, s) \rightarrow_1 (A', s^+)$ (Delay transitions).

where Act is the set of actions that can be performed. This set can be easily deduced from the rules in Table 3.2.

Notice that we have included a τ -action that represents an empty movement in order to represent the unobservable behaviour. *Action transitions* capture a state change by the execution of an action $a \in \text{Act}$, which can be empty (τ). *Delay transitions* capture how the system state changes when one time unit has elapsed. In Tables 3.2, 3.3 we show the rules for these transitions.

Next, we only introduce a short explanation of some rules of Table 3.2. As can be observed, for the basic activities (*throw*, *exit*, *invoke*, *receive*, *reply*, ...), when the corresponding action is performed we reach the *empty* activity. With regard to the communication among services, our language is endowed with five activities to carry out this task. The model we use here is based on the *invoke* and *receive* (or *pick*) operations, as well as the *reply* activity that uses a server to reply to a client. We have also added a barred version of the *reply* activity to synchronise with the response from the client.

We have therefore introduced this last activity in our semantics to deal with the synchronous or asynchronous nature of the *invoke* activity (one-way or request-response operation, respectively), so the *reply* activity is optional in the syntax depicted in Table 3.6. Below, a toy example is depicted to explain how it works.

Example 3.3.1

In this example, there are two actors: a customer and a seller. The customer contacts a seller in order to gather information about a specific product identified by *id1*. The seller checks the stock and send the requested information to the customer. Let the orchestrations $O_c = (A_c, \text{empty})$ and $O_s = (A_s, \text{empty})$, the BPEL-RF code for the primary activity of both participants is:

$$A_c = \text{invoke}(pl_1, \text{info}, id1); \overline{\text{reply}}(pl_1, \text{info}, id3)$$

$$A_s = \text{receive}(pl_2, \text{info}, id2); \text{reply}(pl_1, \text{info}, id4)$$

According to rules **invoke**, **receive**, **reply** and $\overline{\text{reply}}$ in Table 3.2, the customer sends $\sigma(id1)$, which represents the product identifier, and starts the activity $\overline{\text{reply}}$ to receive the response in $id3$. Seller stores product identifier in its variable $id2$, and, therefore, $s' = (\sigma[m/id2], \rho)$, with $m = \sigma(id1)$. Finally, seller sends the product information $\sigma(id4)$ to the customer, which stores it in $id3$, leading to $s'' = (\sigma[m/id3], \rho)$.

Rules for the sequence and parallel operators are straightforward, but notice that when one of the arguments performs either the *throw* action or the *exit* action, the composite activity also performs this action conducting the workflow to the empty activity (rules **Seq3**, **Par3** and **Par4**). As regards resource, rule **publishResource** states that the resource and its information is added to the resource set, whereas the resource identifier (e) for this newly created resource is stored in variable $vEPR$, resulting in the new state $s' = (\sigma[e/vEPR], \rho \cup \{O, e, val, \emptyset, t, tag, A\})$. Time elapsing is captured by rules in Table 3.3, notice that the activities for which the passage of time is allowed are *wait*, *empty*, *receive*, *invoke* and *pick*.

At the orchestrator level (Table 3.4), we will require to identify the orchestrator that executes the activity as well as its mode of operation, which can be normal or failure. A superscript m is used to indicate the current operation mode, which can be either empty (normal) or 'f' (failure). An orchestrator enters into the failure mode when an exception has occurred. In the case of a double exception, the orchestrator terminates its execution as rule **ORCH3** in Table 3.4 captures.

Definition 3 Orchestration Operational semantics

The transitions in this level have the following form:

1. $(O : (A, s)^m) \xrightarrow{a} (O : (A', s')^{m'}), a \in \text{Act}.$
2. $(O : (A, s)^m) \rightarrow_1 (O : (A', s^+)^m).$

where $O = (A, A_f)$.

Finally, the outermost semantic level corresponds to the choreography level, which is defined upon the two previously levels. In Table 3.5, we define the corresponding transition rules for the choreography semantics. Observe that Table 3.5 includes rules with negatives premises, which, in principle, could pose decidability problems. Nevertheless, these premises are all observable from the syntax of the involved terms.

Definition 4 Choreography operational semantics

A choreography is defined as a set of orchestrators that run in parallel exchanging messages: $C = \{O_i\}_{i=1}^c$, where c is the number of orchestrators presented in the choreography. A *choreography state* is then defined as follows: $S_c = \{(O_i : (A_i, s)^{m_i})\}_{i=1}^c$, where A_i is the activity being performed by O_i , m_i its *mode*, and s is the current global state.

Definition 5 Labelled transition system

For a choreography C , we define the semantics of C as the labelled transition system obtained by the application of rules in Table 3.5, starting at the state s_{0_c} :

$$lts(C) = (\mathcal{Q}, s_{0_c}, \rightarrow)$$

where \mathcal{Q} is the set of reachable choreography states, s_{0_c} is the initial state where all the variables in σ are initialised to 0 and ρ is empty, and $\rightarrow = \rightarrow_1 \cup \{\xrightarrow{a} \mid \text{for all basic activity } a, \text{ or } a = \tau\}$.

3.4 Case Study: Automatic management system for stock market investments

The case study concerns a typical automatic management system for stock market investments, which consists of 3 participants: the online stock market system and 2 interested investors, A_1 and A_2 . Here, the resource will be the stocks of a company (a bank, for instance) that the investors want to buy just in case the price falls below an established limit. This threshold is fixed previously by means of subscriptions, i.e., an investor subscribes to the resource (the stocks) with a certain guard (the value of the stocks he/she want to pay for it). The lifetime lft will be determined by the stock market system and the resource price will be fluctuating to simulate the rises/drops of the stocks. We do not take into account the stock buy process since our aim is to model an investors' information system. Thus, the participants will be notified either when their bids hold or the resource lifetime expires. Let us consider the choreography $C = (O_{sys}, O_1, O_2)$, where $O_k = (PL_k, Var_k, A_k, A_{f_k}, \mathcal{A}_{e_k})$, $A_{f_k} = exit$, $k=sys, 1, 2$; $Var_{sys} = \{at, vEPR, epr\}$, $Var_i = \{epr_i, vEPR_i, at_i\}$, $i = 1, 2$. Hence, variables epr, epr_1, epr_2 are used to store the identifier of the resource in each of the participants, whereas variables $vEPR, vEPR_1, vEPR_2$ contain its value. Obviously, epr is initialized internally when the resource is *published*, and epr_1 and epr_2 are *discovered* by the buyers. Finally, at, at_1, at_2 control the period of time in which the auction is active. Suppose that all the variables are initially 0:

$Asys = assign(24, at); publishResource(1000, 24, "bank", empty); while(actualTime() \leq at, Abid)$

```
Abid = getProp(epr, "value", vEPR); assign(vEPR + bid(), vEPR); setProp(epr, vEPR); wait(1, 2)
A1 = discover("bank", epr1); getProp(epr1, "timeout", at1); subscribe(O1, epr1, value < 980,
    invoke(pl1, buy, vEPR)); pick((pl1, buy, vEPR1, empty), empty, at1)
A2 = discover("bank", epr2); getProp(epr2, "timeout", at2); subscribe(O2, epr2, value < 990,
    invoke(pl2, buy, vEPR)); pick((pl2, buy, vEPR2, empty), empty, at2)
```

Here, the function *bid* is used to increase/decrease the stocks value simulating the fluctuation of the stocks price.

3.4.1 Related Work

WS-BPEL has been extensively studied with many formalisms, such as Petri nets, Finite State Machines and process algebras, but there are only a few works considering WS-BPEL enriched with WSRF, and they only show a description of this union, without a formalization of the model. In [?] Slomiski uses BPEL4WS in Grid environments and discusses the benefits and challenges of extensibility in the particular case of OGSi workflows combined with WSRF-based Grids. Other two works centred around Grid environments are [?] and [?]. The first justifies the use of BPEL extensibility to allow the combination of different GRIDs, whereas Ezenwoye et al. [?] share their experience on BPEL to integrate, create and manage WS-Resources that implement the factory/instance pattern.

On the other hand, Ouyang et al. [?] define the necessary elements for translating BPEL processes into Petri nets. Thus, they cover all the important aspects in the standard such as exception handling, dead path elimination and so on. The model they consider differs from ours in that we formalize the whole system as a composition of orchestrators with resources associated, whereas they describe the system as a general scope with nested sub-scopes leaving aside the possibility of administering resources. Furthermore, we have also formalized the event handling and notification mechanisms. Another extensive semantics for BPEL 2.0 is presented in [?] by Dumas et al, which introduces two new interesting improvements. They define several patterns to simplify some huge nets and introduce the semantics for the WS-BPEL 2.0 new patterns. Related to π -calculus semantics, Dragoni and Mazzara [?] propose a theoretical scheme focused on dependable composition for the WS-BPEL recovery framework. In this approach, the recovery framework is simplified and analysed via a conserva-

tive extension of π -calculus. The aim of this approach clearly differs from ours, but it helps us to have a better understanding of the WS-BPEL recovery framework. Other work focused on the BPEL recovery framework is [?]. Although this is more focused in the compensation handler, they describe the corresponding rules that manage a Web service composition. Our work is therefore quite complete as we define rules for nearly all possible activities. In addition, we also consider time constraints. Finally, we would like to highlight the works of Farahbod et al. [?] and Busi et al. [?]. In the first one, the authors extract an abstract operational semantics for BPEL based on abstract state machines (ASM) defining the framework BPEL_{AM} to manage the agents who perform the workflow activities. In this approach time constraints are considered, but they do not formalize the timed model. On the other hand, the goal of the latter one is fairly similar to ours. They also define a π -calculus operational semantics for BPEL and describe a conformance notion. They present all the machinery to model Web service compositions (choreographies and orchestrations). The main difference with our work is that we deal with distributed resources.

For further details about the formalization of service oriented languages we would like to encourage the reader to review the works presented at the SENSORIA project in [?]. Here, an extensive work is presented from different international research groups aimed by the common goal of providing a rigorous software engineering view point for service-oriented system using as a cornerstone the formal specification of Web Services and WS-BPEL in particular.

3.5 Prioritised-Timed Coloured Petri Nets

In this section we introduce the specific model of prioritised-timed coloured Petri net that we consider for the translation.

3.6 Prioritised-Timed Coloured Petri Net Semantics for WS-BPEL+WSRF

3.6.1 Prioritised-Timed Coloured Petri Nets

We use prioritised-timed coloured Petri nets, which are a prioritised-timed extension of coloured Petri nets [?], the well-known model supported by CPN Tools².

Definition 6

(Prioritised-Timed Coloured Petri Nets)

We define a prioritised-timed coloured Petri net (PTCPN) as a tuple $(P, T, A, V, G, E, \lambda, D, \pi)$, where³:

- P is a finite set of *coloured places*. Colours used in this semantics will be introduced progressively, as we define the PTCPNs corresponding to each activity. We will use timed and untimed coloured tokens, so timed tokens will have associated a time stamp, according to the CPNTools interpretation [?].
- T is a finite set of *transitions* ($P \cap T = \emptyset$).
- $A \subseteq (P \times T) \cup (T \times P)$ is a set of directed *arcs*.
- V is a finite set of *integer variables* i.e. $Type(v)$, for all $v \in V$. We will assume that all variables have 0 as initial value.
- $G : T \rightarrow EXPR_V$ is the *guard function*, which assigns a Boolean expression to each transition, i.e. $Type(G(t)) = Bool$. $EXPR_V$ denotes the expressions constructed using the variables in V , with the same syntax admitted by CPN Tools.
- $E : A \rightarrow EXPR_V$ is the *arc expression function*, which assigns an expression to each arc.

²Official web page: <http://cpntools.org/>

³ We use the classical notation on Petri nets to denote the precondition and postcondition of both places and transitions:

$$\forall x \in P \cup T : \bullet x = \{y \mid (y, x) \in A\} \quad x \bullet = \{y \mid (x, y) \in A\}$$

- λ is the *labelling function*, defined both on places and transitions. Transitions can be labelled with either activity names or \emptyset . Places are labelled as *entry places*, *output places*, *error places*, *exit places*, *internal places*, *variable places* and *resource places*, which, respectively, correspond to the following labels: $\{in, ok, er, ex, i, v, r\}$. In our specific model, a PTCPN will have an only *entry place* p_{in} , such that $\bullet p_{in} = \emptyset$, which will be initially marked with a single token, whose colour value will be 0. According to WS-BPEL and WSRF standards, we can distinguish between two kind of termination: *normal* and *abnormal*. On the one hand, the *normal* mode corresponds to the execution of a workflow without faults or without executing any *exit* activity. Thus, in our net model, there is an *output place* p_{ok} , such that $p_{ok}^\bullet = \emptyset$, which will be marked with one token of colour 0 when the workflow ends normally. On the other hand, a workflow can finish abnormally by means of the execution of an explicit activity (exit or throw) as well as the occurrence of an internal fault in the system. Each PTCPN has also a single *error place* p_{er} , which will become marked with one token of colour 0 in the event of a failure, then starting the fault handling activity. In a similar way, the *exit place* will be marked when the *exit* is executed by an orchestrator.

Variable places are denoted by p_v , to mean that they capture the value of variable v . They contain a single token, whose colour is the variable value. For any resource r in the system we will have two complementary resource places, p_{r_i} , p_{r_a} . The first one will be marked with one token when the resource has not been instantiated or has been released (due to a time-out expiration), whereas the second one becomes marked when the resource is created, its token colour being a tuple representing the resource identifier (EPR), lifetime, value, list of subscribers and activity to be executed upon the time-out expiration. All the remaining places will be considered as *internal*.

- $D : T \rightarrow \mathbb{N} \times \mathbb{N}$ (*delay function*), which associates a time interval to each transition. For $D(t) = [d_1, d_2]$, this means that the time delay associated to t can be any value in this interval, all of them with the same probability.
- $\pi : T \rightarrow \mathbb{N}$ is the *priority function*, which assigns a priority level to each transition.

Markings of PTCPNs are defined in the same way supported by CPNTools, as well as the semantics of PTCPNs, so due to the lack of space we omit the formal definitions. The interested reader may see for instance [?].

3.6.2 PTCPN Semantics for WSRF/BPEL

Before introducing the PTCPN semantics, we define the formal model that captures the integration of BPEL and WSRF.

A system for our purposes consists of a set of orchestrators that run in parallel using a set of distributed resources. Orchestrators relate with one another by invoking the services they respectively provide. This set of orchestrators and resources is here called a *choreography*. We use the following notation: $ORCH$ is the set of orchestrators in the system, Var is the set of integer variable names, PL is the set of partnerlinks, OPS is the set of operation names that can be performed, $EPRS$ is the set of resource identifiers, and A is the set of basic or structured activities that can form the body of a process.

An orchestrator O is defined as a tuple $O = (PL, Vars, A, A_f, \mathcal{A}_e)$, where PL are the partnerlinks this orchestrator uses to communicate with others, $Vars$ is the set of local variables of this orchestrator, A and A_f are activities of WS-BPEL and WSRF, and \mathcal{A}_e is a set of activities. Specifically, A represents the normal workflow, A_f is the orchestrator fault handling activity and $\mathcal{A}_e = \{A_{e_i}\}_{i=1}^m$ are the event handling activities.

Activities in BPEL-WSRF follow the syntax defined by the following BNF expression (see Table 3.6 for the equivalence with the XML syntax of BPEL and WSRF):

$$\begin{aligned} A ::= & \text{throw} \mid \text{receive}(pl, op, v) \mid \text{invoke}(pl, op, v_1) \mid \text{exit} \mid \\ & \text{reply}(pl, v) \mid \overline{\text{reply}}(pl, op, v_2) \mid \text{assign}(expr, v_1) \mid \text{empty} \mid \\ & A; A \mid A \parallel A \mid \text{while}(cond, A) \mid \text{wait}(a, b) \mid \\ & \text{pick}(\{(pl_i, op_i, v_i, A_i)\}_{i=1}^n, A, \text{timeout}) \mid \text{getProp}(EPR, v) \mid \\ & \text{createResource}(EPR, val, \text{timeout}, A) \mid \\ & \text{setProp}(EPR, expr) \mid \text{setTimeout}(EPR, \text{timeout}) \mid \\ & \text{subscribe}(EPR, cond', A) \end{aligned}$$

where $O \in ORCH$, $EPR \in EPRS$, $pl, pl_i \in PL$, $op, op_i \in OPS$, $a, b \in \mathbb{N}$, $a \leq b$, $expr$ is an arithmetic expression constructed by using the variables in Var and

integers; v, v_1, v_2, v_i range over Var , and $val \in \mathbb{Z}$. A condition $cond$ is a predicate constructed by using conjunctions, disjunctions, and negations over the set of variables Var and integers, whereas $cond'$ is a predicate constructed by using the corresponding EPR (as the resource value) and integers. Notice that *setProp* and *getProp* do not contain the property name since, for simplicity, we are only considering a single property for each resource. We therefore use its EPR as representative of this property. It is worth noting that we have previously presented an operational semantics for this language in the previous work [?].

Let us call N_A , N_f and N_{e_i} the PTCPNs that are obtained by applying the translation to each one of these activities A , A_f , A_{e_i} , with $i \in \{1, m\}$:

$$\begin{aligned} N_A &= (P_a, T_a, A_a, V_a, G_a, E_a, \lambda_a, D_a) && \text{(PTCPN for } A) \\ N_f &= (P_f, T_f, A_f, V_f, G_f, E_f, \lambda_f, D_f) && \text{(PTCPN for } A_f) \\ N_{e_i} &= (P_{e_i}, T_{e_i}, A_{e_i}, V_{e_i}, G_{e_i}, E_{e_i}, \lambda_{e_i}, D_{e_i}) && \text{(PTCPN for } A_{e_i}) \end{aligned}$$

Let $p_{a_{in}}$, $p_{f_{in}}$ and $p_{e_{i_{in}}}$ be the initial places of N_A , N_f and N_{e_i} respectively; $p_{a_{ok}}$, $p_{f_{ok}}$ and $p_{e_{i_{ok}}}$ their *correct* output places, $p_{a_{er}}$, $p_{f_{er}}$ and $p_{e_{i_{er}}}$ their *error* places and, finally, $p_{a_{ex}}$, $p_{f_{ex}}$ and $p_{e_{i_{ex}}}$ their *exit* places. The PTCPN for the orchestrator is then constructed as indicated in Fig. 3.1. This PTCPN is then activated by putting one token 0 on $p_{a_{in}}$. However, we can have other marked places, for instance, those associated with integer variables or resources. The other places are initially unmarked. ~~The other places are initially unmarked.~~

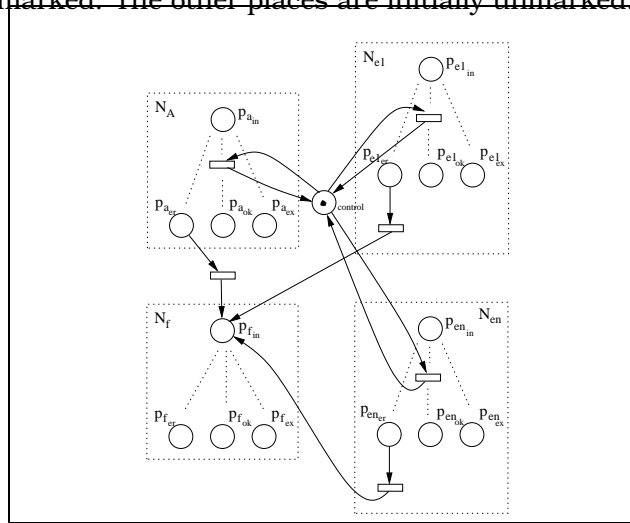


Figure 3.1: Orchestration Translation

Variables and resources: There is one place for each variable, whose token value is the current variable value. As regards resources, there are two places

associated to each resource, p_{r_i} , p_{r_a} . For any resource r , p_{r_a} becomes marked when the orchestrator executes the *createResource* activity, whereas the second one, p_{r_i} , is marked as far as the orchestrator does not execute the *createResource* activity. When the resource lifetime terminates, the resource is released, passing the token from p_{r_a} to p_{r_i} . Observe that we can know in advance the number of resources in the system by reading the WS-BPEL/WSRF document.

3.6.3 Basic activities

- *Throw, Empty, Assign, Exit and Wait* activities:

These are translated as indicated in Fig. ??, by means of a single transition labelled with the name of the corresponding activity linked with the corresponding terminating place. The time required to execute *assign*, *empty*, *throw* and *exit* is negligible, so that the corresponding transitions have a null delay associated. Notice that for the *assign* activity translation we use a self loop between the transition and the place associated with the variable (p_v) in order to replace its previous value by the new one, being this new value obtained from an expression (exp) consisting of variables p_{v1}, \dots, p_{vn} and integers. For the *wait* activity, we have a time interval $[a, b]$ associated, so the delay is randomly selected inside this interval.

Notice the use of a “control” place, to arrest all possible remaining activities in the system when either throw or exit are executed. Thus, the idea is that all transitions in the net must be connected with this place, as the different illustrations show.

- *Communication activities*: The model we use is based on the invoke and receive operations, as well as the reply activity that uses a server to reply to a client. We have also added a barred version of reply to synchronise with the response from the client. We have therefore introduced this last activity in our semantics to deal with the synchronous or asynchronous nature of invoke activity (one-way or request-response operation, respectively), so the *reply* activity is optional in the syntax depicted in Table 3.6. Fig. ?? shows the translation for both the invoke/receive and the reply/*reply* pairs of activities. Part ?? of the figure corresponds to the invoke/receive translation, in which the net of the invoke activity is depicted on the left-hand-side part, whereas the receive activity is depicted on the right-hand-side part. There are two shared places, PL_{ij_s} and PL_{ij_r} , which are used to implement the synchronisation between the invocation and reception of

services. Both places are associated to the partnerlink used for this communication, denoted here by (i, j) , where i and j are the orchestrator identifiers performing those activities. Notice that the value of a single variable is transmitted, which is obtained from the corresponding variable place, p_v . In the same way, the receive activity stores this value in its own variable. The interpretation of Fig. ?? is analogous.

3.6.4 Ordering structures

WS-BPEL defines structured activities for various control-flow patterns:

- Sequential control between activities is provided by `<sequence>`, `<if>`, `<while>`, `<repeatUntil>`, and the serial variant of `<forEach>`.
- Concurrency and synchronization between activities is provided by `<flow>` and the parallel variant of `<forEach>`.
- Deferred choice controlled by external and internal events is provided by `<pick>`.

The set of structured activities in WS-BPEL is not intended to be minimal [?], so there are cases where the semantics of one activity can be represented using another activity. Nevertheless, in order to reduce the complexity of our translation, our approach omits many derived activities only dealing with the most important ones from the modelling viewpoint, such as sequence, parallel and choice. For all these cases we provide the translation by only considering two activities. However, the generalization to a greater number of activities is straightforward in all of them.

- *Sequence*: A sequence of two activities $A_1; A_2$ (with PTCPNs N_{A_1} and N_{A_2} , respectively) is translated in a simple way (Fig. 3.2), by just collapsing in a single place (this will be an internal place of the new PTCPN) the *output* place P_{ok} of N_{A_1} , and the *entry* place of N_{A_2} . The *entry* place of the new PTCPN will be the *entry* place of N_{A_1} . The *output* place of the new PTCPN will be the *output* place of N_{A_2} , and we also collapse the *exit*, *error* and *control* places of both PTCPNs.

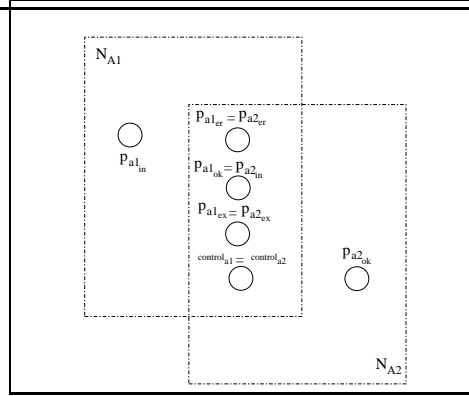


Figure 3.2: Sequence Translation

- *Parallel*: The translation for a parallel activity is depicted in Fig. 3.3, which includes two new transitions $t1$ and $t2$. The first to fork both parallel activities and the second to join them when correctly terminated. Transition $t1$ thus puts one token on the initial places of both PTCPNs, N_{A_1} and N_{A_2} , in order to activate them, and also puts one token on a new place, p_c , which is used to stop the execution of one branch when the other has failed or the exit activity is explicitly executed in one of them. This place is therefore a precondition of every transition in both PTCPNs, and it is also a postcondition of the non-failing transitions. However, in the event of a failure or an exit activity, the corresponding *throw* or *exit* transition will not put the token back on p_c , thus arresting the other parallel activity.

Notice also that the *error* places of N_{A_1} and N_{A_2} have been joined in a single error place (p_{er}), which becomes marked with one token on the firing of one *throw* transition. In this case, the other activity cannot execute any more actions (p_c is empty), so some dead tokens would remain permanently on some places in the PTCPN. However, these tokens cannot cause any damage, since the control flow has been transferred either to the fault handling activity of the PTCPN, once the place p_{er} has become marked, or the whole system has terminated once the place p_{ex} is marked.

- *Pick* ($\{(pl_i, op_i, v_i, A_i)\}_{i=1}^n, A, timeout$): The $\langle pick \rangle$ activity waits for the occurrence of exactly one event from a set of events, also establishing a timeout for this selection. The translation is depicted in Fig. 3.4 where a timer is implemented on the place p_a in order to enforce the firing of transition ta when the timeout has elapsed, thus activating N_A . Notice also the use of both timed and untimed places in this figure, respectively called *INT* and *UINT*.

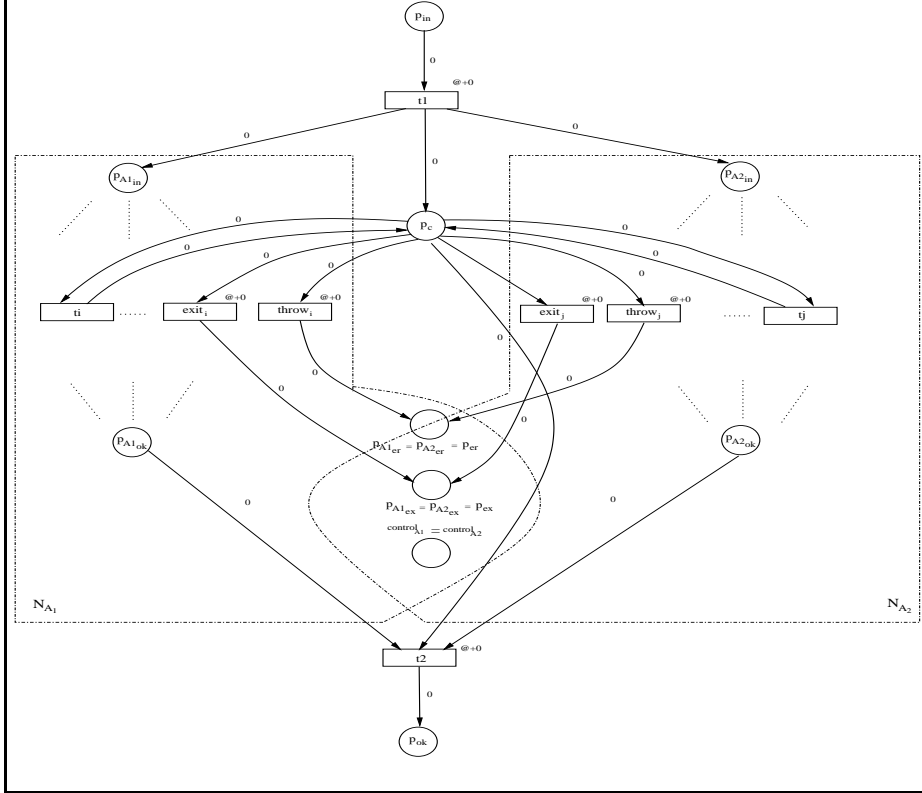


Figure 3.3: Parallel Activity Translation.

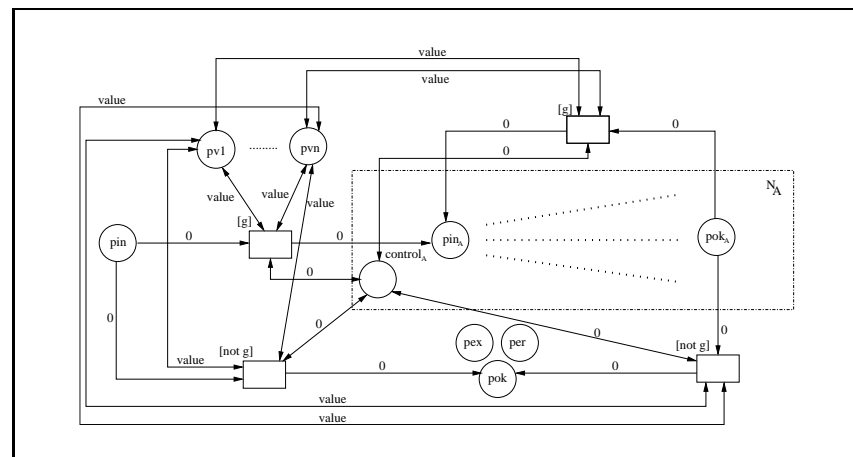
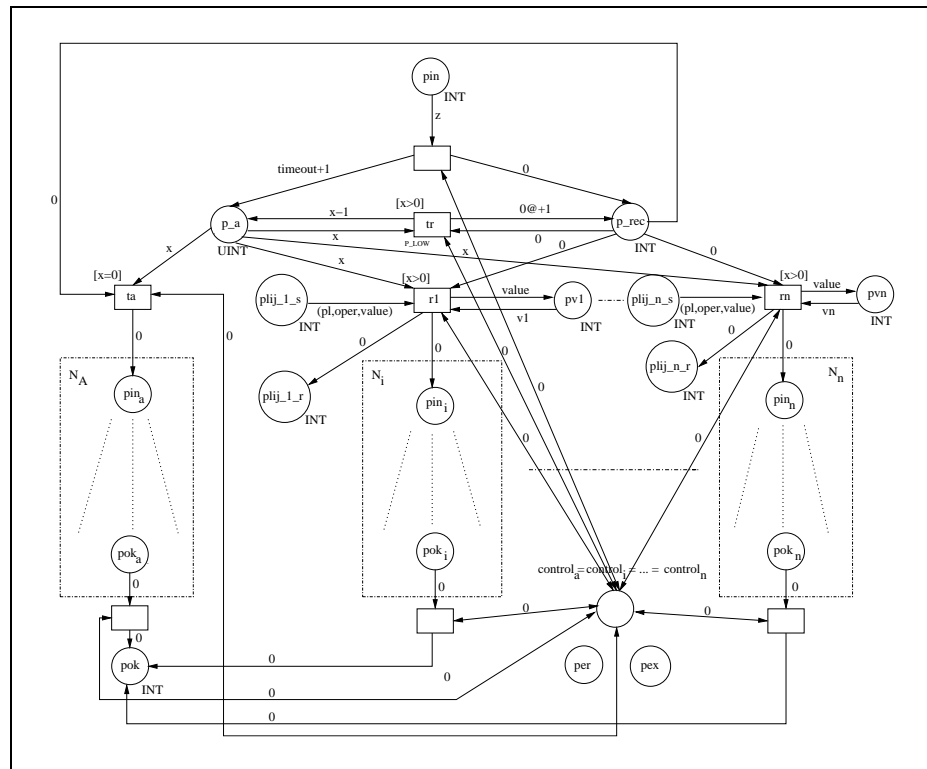
- *While* (cond,A): The machinery needed to model this construction is fairly straightforward since we only must check if the repetition condition holds or not in order to execute the contained activity or skip it. Fig. 3.5 shows this translation.

3.6.5 WSRF-compliant

Let us now see the WSRF activities, and their corresponding translations.

- *CreateResource* ($EPR, val, timeout, A$): EPR is the resource identifier, for which we have two complementary places in Fig. 3.6, p_{r_i} and p_{r_a} , where the sub-index represents the state of the resource: i when it is inactive and a when it is active. The initial value is val , and A is the activity that must be executed when the time-out indicated as third parameter has elapsed.

We can see in Fig. 3.6 how the transition *createResource* removes the token from the *inactive* place, and puts a new token on the active place, whose colour contains the following information: resource identifier (EPR), its lifetime (max), and its value (val). Transition $t0$ is executed when the lifetime



of the resource has expired, thus removing the token from the *active* place, marking again the *inactive* place, and activating N_A . We can also see that the *active* place is linked with a number of transitions, which correspond to the subscribers (we know in advance these possible subscribers from the WS-BPEL/WSRF document). These transitions can only become enabled if the corresponding places $subs_i$ are marked by performing the correspond-

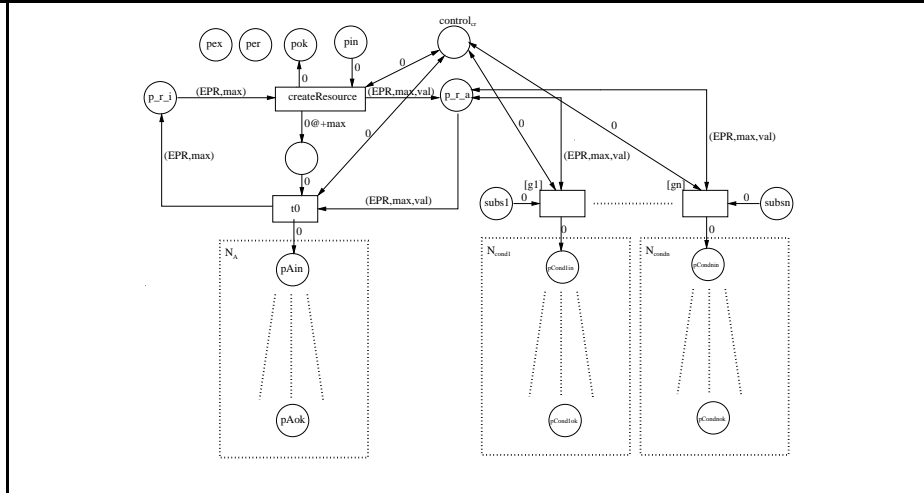


Figure 3.6: CreateResource Activity Translation.

ing activity *subscribe*. The PTCPNs N_{cond_i} are the nets for the activities passed as parameter in the invocation of a subscribe activity.

- **Subscribe (EPR,cond',A):** In this case, an orchestrator subscribes to the resource *EPR*, with the associated condition *cond'*, upon which the activity *A* must be performed. Fig. 3.7 shows this translation, where we can observe that the associated place *subs_i* is marked in order to allow the execution of the PTCPN for the activity *A* if the condition *g_i* holds. On the contrary, if the resource is not active, we will throw the fault handling activity.

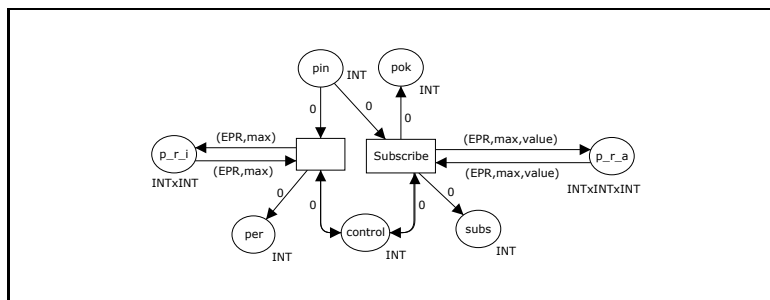


Figure 3.7: Subscribe Activity Translation.

- **GetProp (EPR,v)** and **SetProp (EPR,expr)**: These are easily translated, as shown in Figs. 3.8 and 3.9, where the resource value is obtained and assigned to variable v (GetProp), or a new value is assigned to the resource (SetProp)

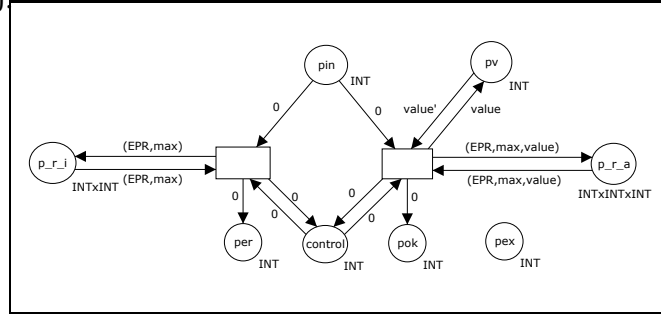


Figure 3.8: GetProperty Activity Translation.

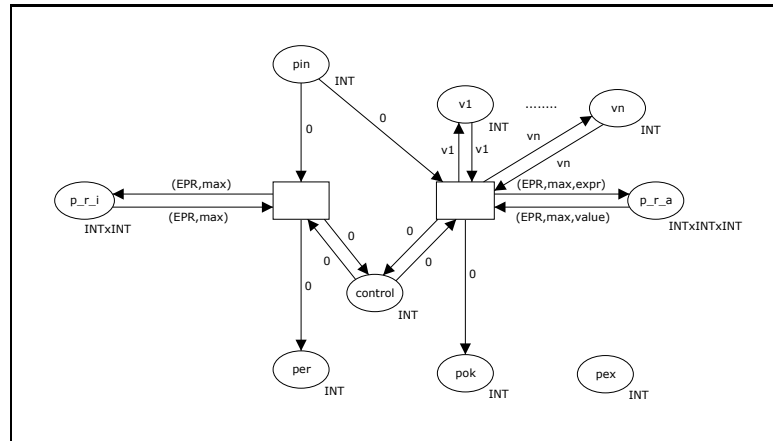


Figure 3.9: SetProperty Activity Translation.

- **SetTimeout (EPR,timeout)**: This activity is analogous to *SetProp* activity. In this case, the resource lifetime is updated with a new value. Fig. 3.9 shows this translation.

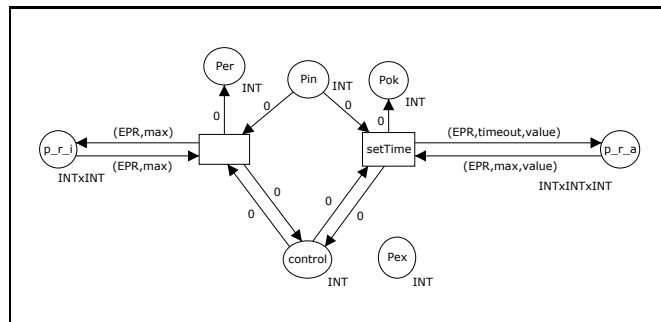


Figure 3.10: SetTimeout Activity Translation.

3.7 Case study: Online auction service

The case study concerns a typical online auction process, which consists of three participants: the online auction system and two buyers, A_1 and A_2 . A seller owes a good that wants to sell to the highest possible price. Therefore, he introduces the product in an auction system for a certain time. Then, buyers (or bidders) may place bids for the product and, when time runs out, the highest bid wins. In our case, we suppose the resource is the product for auction, the value of the resource property is the current price (only the auction system can modify it), the resource subscribers will be the buyers, their subscription conditions hold when the current product value is higher than their bid, and the resource lifetime will be the time in which the auction is active. Finally, when the lifetime has expired, the auction system sends a notification to the buyers with the result of the process (the identifier of the winner, v_w) and, after that, all the processes finish. Let us consider the choreography $C = (O_{sys}, O_1, O_2)$, where $O_i = (PL_i, Var_i, A_i, A_{f_i}, A_{e_i})$, $i=1,2$, $Var_{sys} = \{v_w, v_1, v_2, v_{EPR}, at, t\}$, $Var_1 = \{at_1, v_1, v_{w_1}\}$, $Var_2 = \{at_2, v_2, v_{w_2}\}$, $A_{f_1} = exit$, and $A_{f_2} = exit$. Variable v_{EPR} serves to temporarily store the value of the resource property before being sent; $v_1, v_2, v_w, v_{w_1}, v_{w_2}$ are variables used for the interaction among participants, and, finally, at, at_1 and at_2 are used to control the period of time in which the auction is active. In this example, we consider a period of 10 time units. Suppose $s_{O_{sys}}, s_{O_1}$ and s_{O_2} are the initial states of O_{sys}, O_1 and O_2 , respectively, and all the variables are initially 0:

```

 $A_{sys} = assign(10, at); createResource(EPR, 25, 11, A_{not});$ 
            $while(actualTime() \leq at, A_{bid})$ 
 $A_1 = wait(1, 1); subscribe(O_1, EPR, EPR \geq 0, A_{cond_1});$ 
            $invoke(pl1, auction\_time_1, at1); \overline{reply}(pl1, auction\_time_1, at1);$ 
            $while(actualTime() \leq at_1, A_{bid_1}); receive(pl3, bid\_finish_1, v_{w_1}, empty)$ 
 $A_2 = wait(1, 1); subscribe(O_2, EPR, EPR \geq 0, A_{cond_2});$ 
            $invoke(pl2, auction\_time_2, at2); \overline{reply}(pl2, auction\_time_2, at2);$ 
            $while(actualTime() \leq at_2, A_{bid_2}); receive(pl4, bid\_finish_2, v_{w_2}, empty)$ 
 $A_{not} = ((invoke(pl_3, bid\_finish_1, v_w) || invoke(pl_4, bid\_finish_2, v_w))$ 
 $A_{bid} = getprop(EPR, v_{EPR}); pick($ 
            $(pl1, auction\_time_1, t, reply(pl1, auction\_time_1, at)),$ 

```

```

    (pl2, auction_time2, t, reply(pl2, auction_time2, at)),
    (pl1, cmp, v1, while(v1 > vEPR, assign(v1, vEPR);
      setProp(EPR, vEPR); assign(1, vw))),
    (pl2, cmp, v2, while(v2 > vEPR, assign(v2, vEPR);
      setProp(EPR, vEPR); assign(2, vw))), empty, 1)
A_cond1 = getProp(EPR, vEPR); invoke(pl1, bid_up1, vEPR)
A_cond2 = getProp(EPR, vEPR); invoke(pl2, bid_up2, vEPR)
A_bid1 = receive(pl1, bid_up1, v1); assign(v1 + random(), v1);
    invoke(pl1, cmp, v1); subscribe(O1, EPR, EPR > v1, A_cond1); wait(1, 1)
A_bid2 = receive(pl2, bid_up2, v2); assign(v2 + random(), v2);
    invoke(pl2, cmp, v2); subscribe(O2, EPR, EPR > v2, A_cond2); wait(1, 1)

```

Regarding to the operations *auction_time1* and *auction_time2* inform buyers about the period of time in which the auction is active via variables *at*, *at1* and *at2*, which are used in the while structures to control this period. The operations *bid_up1* and *bid_up2* are used to increase the current bid by adding a random amount to the corresponding variable *v_i*. The operation *cmp* is an auction system operation that receives as parameter a variable of the buyers, *v_i*. If the value of this variable is greater than the current value of *v_{EPR}*, then *v_{EPR}* is modified with this new value, that is, the new bid exceeds the current bid. After that, by means of the activity *setProp(EPR, v_{EPR})*, we can update the value of the resource property with the new bid. Finally, the operations *bid_finish1*, *bid_finish2* update the value of *v_w* to inform the buyers who is the winner once the auction has expired.

In Fig. 3.11, we depict a simplified version of the PTCPN for the online auction system. The complete model can be accessed at the following web address: <http://www.dsi.uclm.es/retics/bpelrf/>. Here, we have constructed a hierarchical net relying on the notions of substitution transitions, sockets and ports offered by CPNTools [?]. We have then simulated and analysed the system, and we have concluded that the system finalizes successfully, that is, the output place of the system (*ok*) is reached in all the simulations. To check the consistency of the model, we have simulated the possibility of reaching an error place. For instance, if we delete the *wait(1,1)* sentences from activities *A₁* and *A₂*, then it would imply that the buyers could access to the resource, that is the

bid, even before the resource has been created. This possibility would trigger the expected error. Furthermore, we have analysed the data output from an experiment consisting of 5000 simulations. From the analysis of these data, we observe that the system is fair, from the point of view of the buyers, since they have equal right to place a bid. Indeed, the average of placed bids from each buyer is similar. Other information gathered from these data shows that buyers can evenly place higher bids than their competitors.

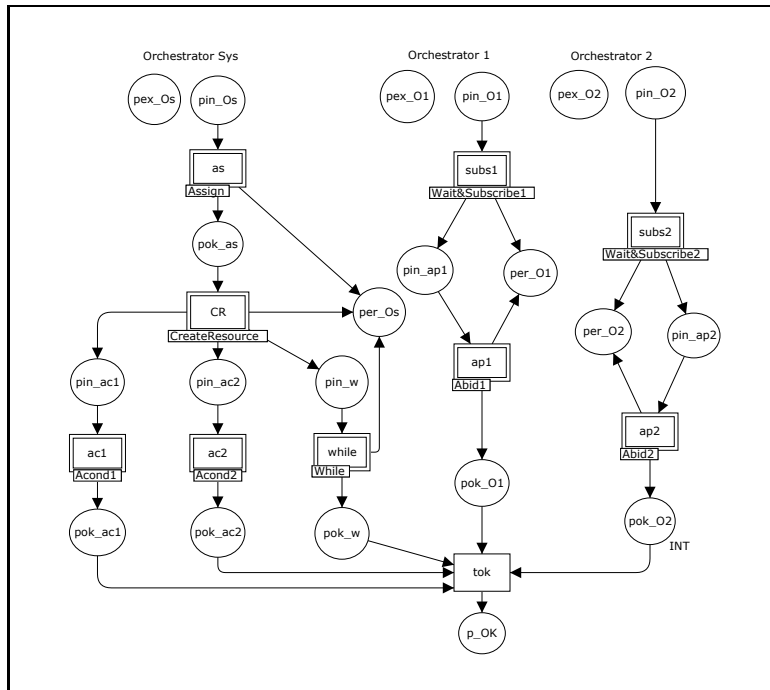


Figure 3.11: A simplified PTCPN for the online auction system.

3.8 Summary

This chapter has described a methodology called *Correct-WS* for the design and verification of Web Service compositions and a tool called WST supporting several phases of this methodology. As a proof of concept the tool has been apply to an Internet purchase process case study. The chapter ends with a brief description of some parallel works which aim is also the development of correct Web Service compositions.

Although in this chapter sequence diagrams have been used to design the service composition behaviour in a proper way, sometimes we just have an elec-

tronic contract defining all the contractually correct behaviours of the system in a more general way. In these cases we would like to have a user-friendly representation of the e-contracts but having at the same time a formal background allowing us to verify the correctness of the e-contracts. That is what Chapter ?? is focused on.

WS-BPEL/WSRF/WS-Notification Syntax		Model
<pre> <process ...> <partnerLinks> ... </partnerLinks>? <Variables> ... </Variables>? <faultHandlers> ... </faultHandlers>? <eventHandlers> ... </eventHandlers>? (activities)* </process> </pre>		(A, A_f)
<pre><throw/> /any fault</pre>		throw
<pre><receive partnerLink="pl" operation="op" variable="v" createInstance="no"> </receive></pre>		receive(pl, op, v)
<pre><reply partnerLink="pl" operation="op" variable="v"> </reply></pre>		reply(pl, op, v)
<pre> <invoke partnerLink="pl" operation="op" inputVariable="v₁" outputVariable="v₂"?> </invoke> </pre>		invoke(pl, op, v ₁): $\overline{reply}(pl, op, v_2)$
<pre><empty> ... </empty></pre>		empty
<pre><exit> ... </exit></pre>		exit
<pre><assign> <copy> <from> expr </from> <to> v₁ </to> </copy> </assign></pre>		assign(expr, v ₁)
<pre><wait> <for> timeout </for> </wait></pre>		wait(timeout)
<pre> <sequence> activity₁ activity₂ </sequence> </pre>	<pre> <flow> activity₁ activity₂ </flow> </pre>	$\frac{A_1 ; A_2}{A_1 \parallel A_2}$
<pre><while> <condition> cond </condition> activity₁ </while></pre>		while(cond, A)
<pre> <pick createInstances="no"> <onMessage partnerLink="pl" operation="op" variable="v"> activity₁ </onMessage> <onAlarm> <for> timeout </for> activity₁ </onAlarm> </pick> </pre>		pick({(pl _i , op _i , v _i , A _i) _{i=1} ⁿ , A, timeout)
<pre> <invoke partnerLink="Factory" operation="CreateResource" inputVariable="O, val, timeout, tag" outputVariable="vEPR"> </invoke> <assign> <copy> <from variable="EPR"> part="ref" query="/test:CreateOut/wsa:endpointreference" </from> <to> partnerLink="Factory" </to> </copy> </assign> </pre>		publishResource(O, val, timeout, tag, vEPR, A)
<pre> <wsrp:GetResourceProperty> <wsa:Address> vEPR </wsa:Address> tns:value variable:Identifier1 </wsrp:GetResourceProperty> </pre>		getProp(vEPR, v ₁)
<pre> <wsrp:GetResourceProperty> <wsa:Address> vEPR </wsa:Address> tns:timeout variable:Identifier1 </wsrp:GetResourceProperty> </pre>		getTimeout(vEPR, v ₁)
<pre> <wsrp:SetResourceProperties> <wsa:Address> vEPR </wsa:Address> <wsrp:Update> expression </wsrp:Update> </wsrp:SetResourceProperties> </pre>		setProp(vEPR, expr)
<pre> <wsrl:SetTerminationTime> <wsa:Address> vEPR </wsa:Address> <wsrl:RequestedTerminationTime> timeout </wsrl:RequestedTerminationTime> </wsrl:SetTerminationTime> </pre>		setTimeout(vEPR, timeout)
<pre> <wsnt:Subscribe> <wsnt:ConsumerReference> O </wsnt:ConsumerReference> <wsnt:ProducerReference> vEPR </wsnt:ProducerReference> <wsnt:Precondition> cond' </Precondition> </wsnt:Subscribe> </pre>		subscribe(O, vEPR, cond', A)
<pre> <wsnt:Notify> <wsnt:NotificationMessage> <wsnt:SubscriptionReference> O </wsnt:SubscriptionReference> <wsnt:ProducerReference> EPR </wsnt:ProducerReference> <wsnt:Message> ... </wsnt:Message> </wsnt:NotificationMessage> </wsnt:Notify> </pre>		Spawn the associated event handler activity
<pre><invoke partnerLink="pl" operation="discover" inputVariable="tag1" outputVariable="vEPR"> </invoke></pre>		discover(tag, vEPR)

Table 3.1: Conversion table

<p>(Throw) $(throw, s) \xrightarrow{throw} (empty, s)$</p> <p>(Invoke) $(invoke(pl, op, v_1), s) \xrightarrow{invoke(pl, op, \sigma(v_1))} (empty, s)$</p> <p>(Receive) $(receive(pl, op, v), s) \xrightarrow{receive(pl, op, m)} (empty, s')$ where $v \in VAR, m \in \mathbb{Z}, op \in OPS, pl \in PL$, and $s' = (\sigma[m/v], \rho)$</p> <p>(Assign) $(assign(expr, v_1), s) \xrightarrow{assign(expr, v_1)} (empty, s')$ where $v_1 \in VAR, expr$ is an arithmetic expression, and $s' = (\sigma[expr/v_1], \rho)$.</p> <p>(Seq2) $\frac{(A_1, s) \xrightarrow{a} (empty, s'), a \neq exit, a \neq throw}{(A_1; A_2, s) \xrightarrow{a} (A_2, s')}$</p> <p>(Par1) $\frac{(A_1, s) \xrightarrow{a} (A'_1, s'), a \neq exit, a \neq throw}{(A_1 A_2, s) \xrightarrow{a} (A'_1 A_2, s')}$</p> <p>(Par3) $\frac{(A_1, s) \xrightarrow{a} (empty, s), (a = throw \vee a = exit)}{(A_1 A_2, s) \xrightarrow{a} (empty, s)}$</p> <p>(Par5) $(empty empty, s) \xrightarrow{\tau} (empty, s)$</p> <p>(While2) $\frac{\neg cond(s)}{(while(cond, A), s) \xrightarrow{\tau} (empty, s)}$</p> <p>(PublishResource) $(publishResource(O, val, t, tag, vEPR, A), s) \xrightarrow{publishResource(O, val, t, tag, e, A)} (empty, s')$ where $t \geq 1, O \in ORCH, val \in \mathbb{Z}, vEPR \in VAR, e = getEPR()$ and $s' = (\sigma[e/vEPR], \rho \cup \{O, e, val, \emptyset, t, tag, A\})$.</p> <p>(GetP) $\frac{s = (\sigma, \rho), \sigma(vEPR) \in \rho}{(getProp(vEPR, v_1), s) \xrightarrow{getProp(\sigma(vEPR), val(\rho, vEPR))} (empty, s')}$ where $vEPR, v_1 \in VAR$ and $s' = (\sigma[val(\rho, vEPR)/v_1], \rho)$.</p> <p>(GetT) $\frac{s = (\sigma, \rho), \sigma(vEPR) \in \rho}{(getTimeout(vEPR, v_1), s) \xrightarrow{getTimeout(\sigma(vEPR), time(\rho, vEPR))} (empty, s')}$ where $vEPR, v_1 \in VAR$ and $s' = (\sigma[time(\rho, vEPR)/v_1], \rho)$.</p> <p>(SetP) $\frac{s = (\sigma, \rho), \sigma(vEPR) \in \rho}{(setProp(vEPR, expr), s) \xrightarrow{setProp(\sigma(vEPR), expr)} (empty, s')}$ where $vEPR \in VAR$ and $s' = (\sigma, \rho[expr/vEPR]_1)$.</p> <p>(SetT) $\frac{s = (\sigma, \rho), \sigma(vEPR) \in \rho}{(setTimeout(vEPR, t), s) \xrightarrow{setTimeout(\sigma(vEPR), t)} (empty, s')}$ where $t \geq 1, s' = (\sigma, \rho[t/vEPR]_2)$.</p> <p>(Subs) $\frac{s = (\sigma, \rho), \sigma(vEPR) \in \rho}{(subscribe(O, vEPR, cond', A), s) \xrightarrow{subscribe(O, \sigma(vEPR), cond', A)} (empty, s')}$ where $s' = (\sigma, Add_subs(\rho, vEPR, O, cond', A))$.</p> <p>(D1) $\frac{s = (\sigma, \rho), tag \in \rho}{(discover(tag, vEPR), s) \xrightarrow{discover(tag, sel(\rho, tag))} (empty, s')}$ where $vEPR \in VAR, s' = (\sigma[sel(\rho, tag)/vEPR], \rho)$.</p>	<p>(Exit) $(exit, s) \xrightarrow{exit} (empty, s)$</p> <p>(Reply) $(reply(pl, op, v_2), s) \xrightarrow{reply(pl, op, m)} (empty, s')$ where $v_2 \in VAR, m \in \mathbb{Z}, pl \in PL, op \in OPS$, and $s' = (\sigma[m/v_2], \rho)$.</p> <p>(Reply) $(reply(pl, op, v), s) \xrightarrow{reply(pl, op, \sigma(v))} (empty, s)$</p> <p>(Seq1) $\frac{(A_1, s) \xrightarrow{a} (A'_1, s'), a \neq exit, a \neq throw}{(A_1; A_2, s) \xrightarrow{a} (A'_1; A_2, s')}$</p> <p>(Seq3) $\frac{(A_1, s) \xrightarrow{a} (empty, s), (a = throw \vee a = exit)}{(A_1; A_2, s) \xrightarrow{a} (empty, s)}$</p> <p>(Par2) $\frac{(A_2, s) \xrightarrow{a} (A'_2, s'), a \neq exit, a \neq throw}{(A_1 A_2, s) \xrightarrow{a} (A_1 A'_2, s')}$</p> <p>(Par4) $\frac{(A_2, s) \xrightarrow{a} (empty, s), (a = throw \vee a = exit)}{(A_1 A_2, s) \xrightarrow{a} (empty, s)}$</p> <p>(While1) $\frac{cond(s)}{(while(cond, A), s) \xrightarrow{\tau} (A; while(cond, A), s)}$</p> <p>(Pick) $(pick(\{(pl_i, op_i, v_i, A_i)\}_{i=1}^n, A, t), s) \xrightarrow{pick(pl_i, op_i, m, A_i)} (A_i, s')$ where $t \geq 1, v_i \in VAR, m \in \mathbb{Z}, pl_i \in PL$, and $s' = (\sigma[m/v_i], \rho), \forall i \in \{1..n\}$.</p> <p>(GetT2) $\frac{s = (\sigma, \rho), \sigma(vEPR) \notin \rho}{(getTimeout(vEPR, v_1), s) \xrightarrow{throw} (empty, s)}$</p> <p>(GetP2) $\frac{s = (\sigma, \rho), \sigma(vEPR) \notin \rho}{(getProp(vEPR, v_1), s) \xrightarrow{throw} (empty, s)}$</p> <p>(SetP2) $\frac{s = (\sigma, \rho), \sigma(vEPR) \notin \rho}{(setProp(vEPR, expr), s) \xrightarrow{throw} (empty, s)}$</p> <p>(Subs2) $\frac{s = (\sigma, \rho), \sigma(vEPR) \notin \rho}{(subscribe(O, vEPR, cond', A), s) \xrightarrow{throw} (empty, s')}$</p> <p>(SetT2) $\frac{s = (\sigma, \rho), \sigma(vEPR) \notin \rho}{(setTimeout(vEPR, t), s) \xrightarrow{throw} (empty, s)}$</p> <p>(D2) $\frac{s = (\sigma, \rho), tag \notin \rho}{(discover(tag, vEPR), s) \xrightarrow{discover(tag, -1)} (empty, s')}$ where $vEPR \in VAR, s' = (\sigma[-1/vEPR], \rho)$.</p>
---	--

Table 3.2: Action transition rules.

<p>(Wait1D) $\frac{t > 1}{(wait(t), s) \rightarrow_1 (wait(t-1), s^+)}$</p> <p>(SequenceD) $\frac{(A_1, s) \rightarrow_1 (A'_1, s^+)}{(A_1; A_2, s) \rightarrow_1 (A'_1; A_2, s^+)}$</p> <p>(ParallelD) $\frac{(A_1, s) \rightarrow_1 (A'_1, s^+) \wedge (A_2, s) \rightarrow_1 (A'_2, s^+)}{(A_1 A_2, s) \rightarrow_1 (A'_1 A'_2, s^+)}$</p> <p>(ReceiveD) $(receive(pl, op, v), s) \rightarrow_1 (receive(pl, op, v), s^+)$</p> <p>(Pick2D) $\frac{t > 1}{(pick(\{(pl_i, op_i, v_i, A_i)\}_{i=1}^n, A, t), s) \rightarrow_1 (pick(\{(pl_i, op_i, v_i, A_i)\}_{i=1}^n, A, t-1), s^+)}$</p>	<p>(Wait2D) $(wait(1), s) \rightarrow_1 (empty, s^+)$</p> <p>(EmptyD) $(empty, s) \rightarrow_1 (empty, s^+)$</p> <p>(InvokeD) $(invoke(pl, op, v_1), s) \rightarrow_1 (invoke(pl, op, v_1), s^+)$</p> <p>(PickD) $(pick(\{(pl_i, op_i, v_i, A_i)\}_{i=1}^n, A, 1), s) \rightarrow_1 (A, s^+)$</p>
--	--

Table 3.3: Delay transition rules without notifications.

(ORCH1) $\frac{(A, s) \xrightarrow{a} (A', s'), a \neq \text{exit}, a \neq \text{throw}}{(O : (A, s)^m) \xrightarrow{a} (O : (A', s')^m)}$	(ORCH2) $\frac{(A, s) \xrightarrow{\text{throw}} (\text{empty}, s)}{(O : (A, s)) \xrightarrow{\text{throw}} (O : (A_f, s)^f)}$
(ORCH3) $\frac{(A, s) \xrightarrow{\text{throw}} (\text{empty}, s)}{(O : (A, s)^f) \xrightarrow{\text{throw}} (O : (\text{empty}, s)^f)}$	(ORCH4) $\frac{(A, s) \xrightarrow{\text{exit}} (\text{empty}, s)}{(O : (A, s)^m) \xrightarrow{\text{exit}} (O : (\text{empty}, s)^m)}$
(ORCH5) $\frac{(A, s) \rightarrow_1 (A', s^+)}{(O : (A, s)^m) \rightarrow_1 (O : (A', s^+)^m)}$	

Table 3.4: Action and delay transition rules for orchestrators.

(Chor1) $\frac{(O_i : (A_i, s)^{m_i}) \xrightarrow{\text{exit}} (O_i : (\text{empty}, s)^{m_i})}{\{(O_k : (A_k, s)^{m_k})\}_{k=1}^c \xrightarrow{\text{exit}} \{(O_k : (A'_k, s)^{m'_k})\}_{k=1}^c}$ where if $k \neq i, A'_k = A_k$.	
(Chor2) $\frac{(O_i : (A_i, s)^{m_i}) \xrightarrow{\text{throw}} (O_i : (A'_i, s)^{m'_i})}{\{(O_k : (A_k, s)^{m_k})\}_{k=1}^c \xrightarrow{\text{throw}} \{(O_k : (A'_k, s)^{m'_k})\}_{k=1}^c}$ where if $k \neq i, A'_k = A_k$, and $m'_k = m_k$.	
(Chor3) $\frac{(O_i : (A_i, s)^{m_i}) \xrightarrow{a} (O_i : (A'_i, s')^{m'_i}), a \neq \text{exit}, a \neq \text{throw}, a \neq \text{receive}, a \neq \text{invoke}, a \neq \text{reply}, a \neq \text{reply}, a \neq \text{pick}}{\{(O_k : (A_k, s)^{m_k})\}_{k=1}^c \xrightarrow{a} \{(O_k : (A'_k N(O_k, s'), s'')^{m'_k})\}_{k=1}^c}$ where if $k \neq i, A'_k = A_k, m'_k = m_k$ and $s'' = \text{Subs}(s')$.	
(Chor4) $\frac{(O_i : (A_i, s)^{m_i}) \rightarrow_1 (O_i : (A'_i, s^+)^{m_i}), \forall i \in \{1, \dots, c\}, \text{and rules chor5, chor6, chor7 are not applicable}}{\{(O_k : (A_k, s)^{m_k})\}_{k=1}^c \rightarrow_1 \{(O_k : (A'_k, s^+)^{m_k})\}_{k=1}^c}$	
(Chor5) $\frac{(O_i : (A_i, s)^{m_i}) \xrightarrow{\text{invoke}(pl, op, n)} (O_i : (A'_i, s)^{m_i}), pl = (O_i, O_j), (O_j : (A_j, s)^{m_j}) \xrightarrow{\text{receive}(pl, op, n)} (O_j : (A'_j, s')^{m'_j})}{\{(O_k : (A_k, s)^{m_k})\}_{k=1}^c \xrightarrow{\text{invoke}(pl, op, n)} \{(O_k : (A'_k, s')^{m'_k})\}_{k=1}^c}$ where if $k \neq i, k \neq j, A'_k = A_k, m'_k = m_k$.	
(Chor6) $\frac{(O_i : (A_i, s)^{m_i}) \xrightarrow{\text{reply}(pl, op, n)} (O_i : (A'_i, s)^{m_i}), pl = (O_i, O_j), (O_j : (A_j, s)^{m_j}) \xrightarrow{\text{reply}(pl, op, n)} (O_j : (A'_j, s')^{m'_j})}{\{(O_k : (A_k, s)^{m_k})\}_{k=1}^c \xrightarrow{\text{reply}(pl, op, n)} \{(O_k : (A'_k, s')^{m'_k})\}_{k=1}^c}$ where if $k \neq i, k \neq j, A'_k = A_k, m'_k = m_k$.	
(Chor7) $\frac{(O_i : (A_i, s)^{m_i}) \xrightarrow{\text{invoke}(pl, op, n)} (O_i : (A'_i, s)^{m_i}), pl = (O_i, O_j), (O_j : (A_j, s)^{m_j}) \xrightarrow{\text{pick}(pl, op, n, A)} (O_j : (A'_j, s')^{m'_j})}{\{(O_k : (A_k, s)^{m_k})\}_{k=1}^c \xrightarrow{\text{invoke}(pl, op, n)} \{(O_k : (A'_k, s')^{m'_k})\}_{k=1}^c}$ where if $k \neq i, k \neq j, A'_k = A_k, m'_k = m_k$.	

Table 3.5: Choreography transition rules.

WS-BPEL/WSRF Syntax		Model
<code><process ...></code> <code><partnerLinks> ... </partnerLinks>?</code> <code><Variables> ... </Variables>?</code> <code><faultHandlers> ... </faultHandlers>?</code> <code><eventHandlers> ... </eventHandlers>?</code> <code>(activities)*</code> <code></process></code>		(PL, Var, A, A_f, A_e)
<code><throw/> /any fault</code>		throw
<code><receive partnerLink="pl" operation="op" variable="v" createInstance="no"></code> <code></receive></code>		receive(pl,op,v)
<code><reply partnerLink="pl" variable="v"> </reply></code>		reply(pl,v)
<code><invoke partnerLink="pl"</code> <code>operation="op"</code> <code>inputVariable="v₁"</code> <code>outputVariable="v₂"?> </invoke></code>		invoke(pl,op,v ₁); $\overline{reply}(pl,op,v_2)$
<code><empty> ... </empty></code>		empty
<code><exit> ... </exit></code>		exit
<code><assign> <copy> <from>expr</from> <to>v₁</to> </copy> </assign></code>		assign(expr,v ₁)
<code><wait> <from>a</from> <to>b</to> </wait></code>		wait(a,b)
<code><sequence></code> <code>activity₁</code> <code>activity₂</code> <code></sequence></code>	<code><flow></code> <code>activity₁</code> <code>activity₂</code> <code></flow></code>	$\frac{A_1 ; A_2}{A_1 \parallel A_2}$
<code><while> <condition>cond</condition> activity₁ </while></code>		while(cond,A)
<code><pick createInstance="no"></code> <code><onMessage partnerLink="pl" operation="op" variable="v"></code> <code>activity₁</code> <code></onMessage></code> <code><onAlarm> <for>timeout</for> activity₁ </onAlarm></code> <code></pick></code>		$\text{pick}(\{(p_i, op_i, v_i, A_i)\}_{i=1}^n, A, \text{timeout})$
<code><invoke partnerLink="Factory" operation="CreateResource"</code> <code>inputVariable="val,timeout" outputVariable="EPR"></code> <code></invoke> <assign> <copy> <from variable="EPR"> part="ref"</code> <code>query="/test:CreateOut/wsa:endpointreference" </from></code> <code><to> partnerLink="Factory" </to> </copy> </assign></code>		createResource(EPR,val,timeout,A)
<code><wsrp:GetResourceProperty></code> <code><wsa:Address>EPR</wsa:Address></code> <code>variable_identifier</code> <code></wsrp:GetResourceProperty></code>		getProp(EPR,v)
<code><wsrp:SetResourceProperties></code> <code><wsa:Address>EPR</wsa:Address></code> <code><wsrp:Update> expression</wsrp:Update></code> <code></wsrp:SetResourceProperties></code>		setProp(EPR,expr)
<code><wsrl:SetTerminationTime></code> <code><wsa:Address>EPR</wsa:Address></code> <code><wsrl:RequestedTerminationTime></code> <code>timeout</code> <code></wsrl:RequestedTerminationTime></code> <code></wsrl:SetTerminationTime></code>		setTimeout(EPR,timeout)
<code><wsnt:Subscribe></code> <code><wsnt:ProducerReference> EPR</wsnt:ProducerReference></code> <code><wsnt:Precondition> cond' </Precondition></code> <code></wsnt:Subscribe></code>		subscribe(EPR,cond',A)
<code><wsnt:Notify></code> <code><wsnt:NotificationMessage></code> <code><wsnt:ProducerReference> EPR</wsnt:ProducerReference></code> <code><wsnt:Message> ... </wsnt:Message></code> <code></wsnt:NotificationMessage></code> <code></wsnt:Notify></code>		Executes the associated event handler activity

Table 3.6: Conversion table

Chapter 4

Timed-arc workflow nets

Chapter 5

Conclusions, Contributions and Future Works

This chapter presents the conclusions of this Thesis, reviews the contributions of this work, and suggests some possible future lines of research. It also includes a list of the publications obtained as a result of this work.

Bibliography

- [1] S. Askary C. Barreto B. Bloch F. Curbera M. Ford Y. Goland A. Guzar N. Kartha C. K. Liu R. Khalaf D. Knig M. Marin V. Mehta S. Thatte D. van der Rijn P. Yendluri A. Yiu A. Alves, A. Arkin. Business Process Execution Language for Web Services (version 2.0). 2007.
- [2] T. Banks. Web Services Resource Framework (WSRF) - Primer. OASIS, 2006.
- [3] E. Clarke and J. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 1996.
- [4] O. Grumberg E. Clarke and D. Peled. *Model Checking*. MITPress, Cambridge, MA, 1999.
- [5] S. Graham S. Tuecke K. Czajkowski D. Ferguson F. Leymann M. Nally T. Storey W. Vambenepe I. Foster, J. Frey and S. Weerawarana. Modeling stateful resources with web services. <http://www-106.ibm.com/developerworks/library/wsresource/ws-modelingresources.pdf>, *Tech. Rep.*, 2004.
- [6] S. Schuman J. R. Abrial and B. Meyer. A Specification Language, in On the Construction of Programs. *Cambridge University Press*, eds. A. M. Macnaghten and R. M. McKeag, 1980.
- [7] Chun Ouyang, Eric Verbeek, Wil M. P. van der Aalst, Stephan Breutel, Marlon Dumas, and Arthur H. M. ter Hofstede. Formal semantics and analysis of control flow in ws-bpel. *Sci. Comput. Program.*, 67(2-3):162–198, 2007.
- [8] A. Pnueli. The temporal logic of programs. *Proc. 18th IEEE Symp. on Foundations of Computer Science*, 1977.

-
- [9] I. Foster J. Frey S. Graham C. Kesselman T. Maguire T. Sandholm P. Vanderbilt S. Tuecke, K. Czajkowsk and D. Snelling. Open Grid Services Infrastructure (OGSI) Version 1.0. Global Grid Forum Recommendation. , *Global Grid Forum*, 2003.
- [10] J. Sifakis T. Henzinger, X. Nicollin and S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111(2):193–244, 1994.

Bibliography

- [1] S. Askary C. Barreto B. Bloch F. Curbera M. Ford Y. Goland A. Guzar N. Kartha C. K. Liu R. Khalaf D. Knig M. Marin V. Mehta S. Thatte D. van der Rijn P. Yendluri A. Yiu A. Alves, A. Arkin. Business Process Execution Language for Web Services (version 2.0). 2007.
- [2] T. Banks. Web Services Resource Framework (WSRF) - Primer. OASIS, 2006.
- [3] E. Clarke and J. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 1996.
- [4] O. Grumberg E. Clarke and D. Peled. *Model Checking*. MITPress, Cambridge, MA, 1999.
- [5] S. Graham S. Tuecke K. Czajkowski D. Ferguson F. Leymann M. Nally T. Storey W. Vambenepe I. Foster, J. Frey and S. Weerawarana. Modeling stateful resources with web services. <http://www-106.ibm.com/developerworks/library/wsresource/ws-modelingresources.pdf>, *Tech. Rep.*, 2004.
- [6] S. Schuman J. R. Abrial and B. Meyer. A Specification Language, in On the Construction of Programs. *Cambridge University Press*, eds. A. M. Macnaghten and R. M. McKeag, 1980.
- [7] Chun Ouyang, Eric Verbeek, Wil M. P. van der Aalst, Stephan Breutel, Marlon Dumas, and Arthur H. M. ter Hofstede. Formal semantics and analysis of control flow in ws-bpel. *Sci. Comput. Program.*, 67(2-3):162–198, 2007.
- [8] A. Pnueli. The temporal logic of programs. *Proc. 18th IEEE Symp. on Foundations of Computer Science*, 1977.

-
- [9] I. Foster J. Frey S. Graham C. Kesselman T. Maguire T. Sandholm P. Vanderbilt S. Tuecke, K. Czajkowsk and D. Snelling. Open Grid Services Infrastructure (OGSI) Version 1.0. Global Grid Forum Recommendation. , *Global Grid Forum*, 2003.
- [10] J. Sifakis T. Henzinger, X. Nicollin and S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111(2):193–244, 1994.