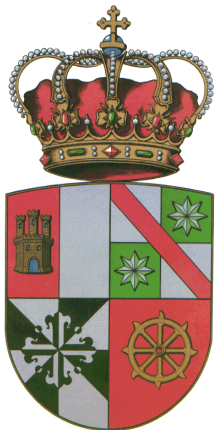
Verification and Validation of Web Service Compositions using Formal Methods



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, he code implementing the services the particpants 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

List of Figures

List of Tables

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) [2] 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 4 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 5** 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 6 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 verfication 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 diffent 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 processess. 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 [6] 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 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 benefitial 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 proccess. 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 4.

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

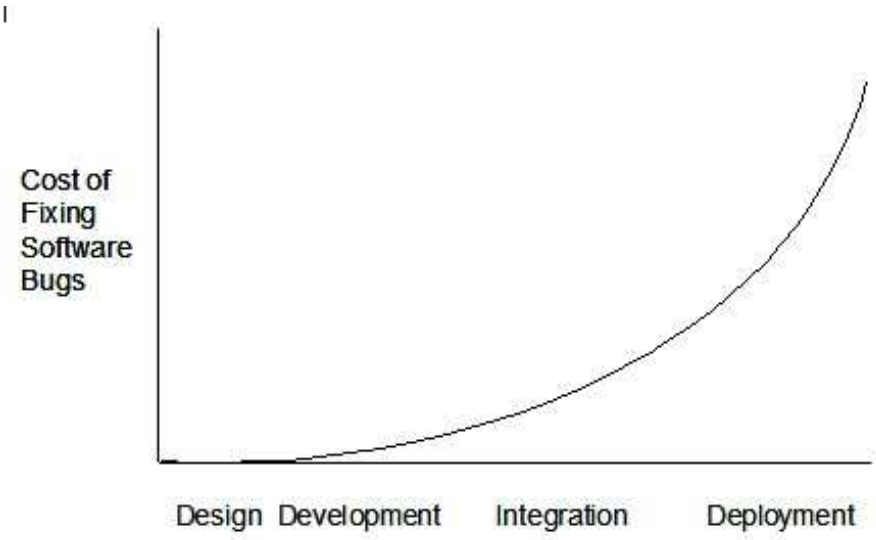


Figure 2.2: Cost evolution of fixing a bug.

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 [7] 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 [11] or CTL [13] 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

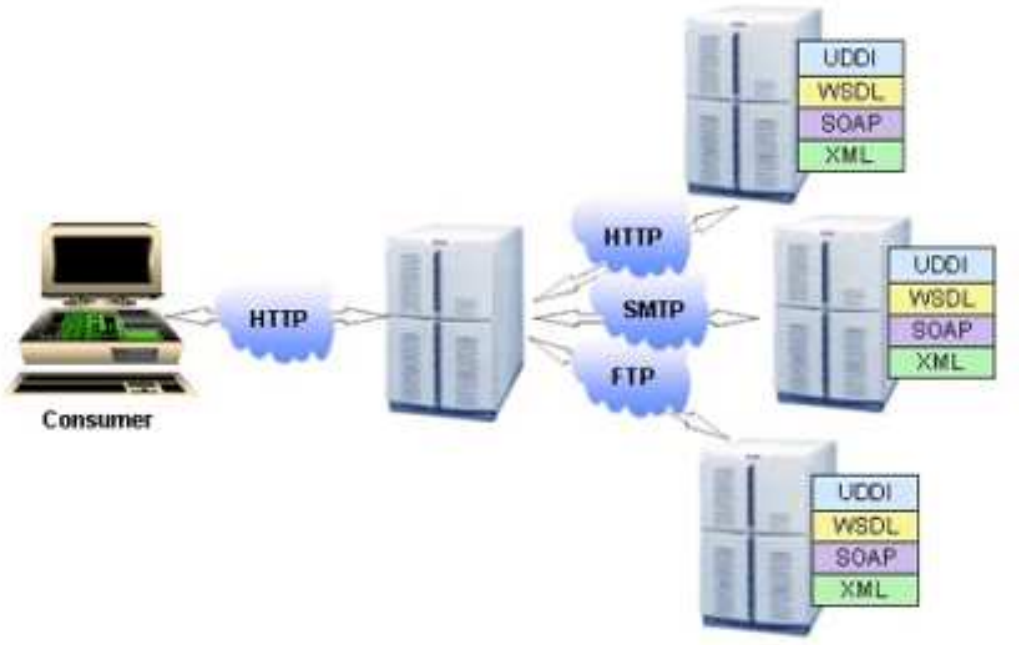


Figure 2.3: Client-server web architecture

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 2.4 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. In Figure 2.5, it is depicted graphically what it the role of each of them if they are compared with the

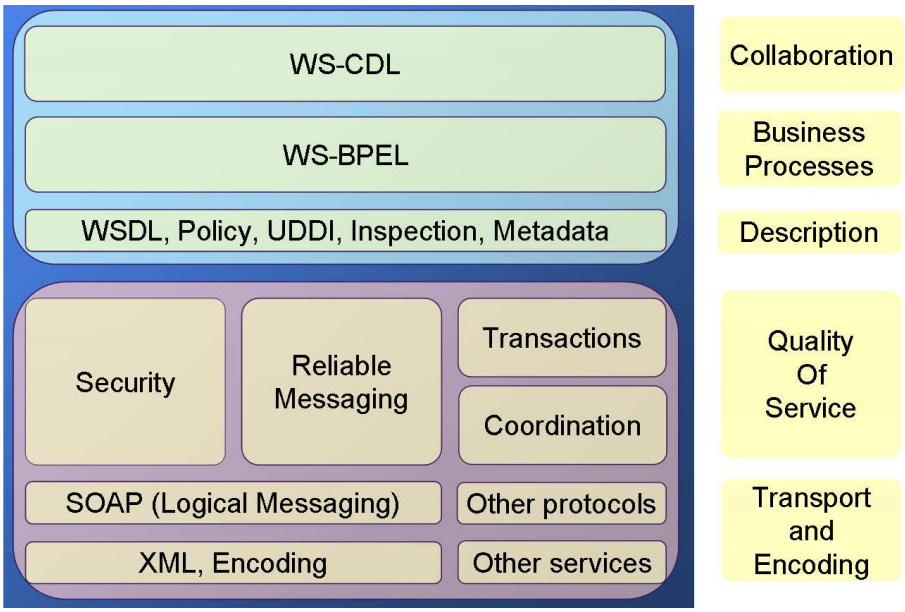


Figure 2.4: Web Service architecture stack.

musicians in an orchestra. Despite these differences, 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.

Comment: Concluir ventajas y desventajas de cada uno. 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

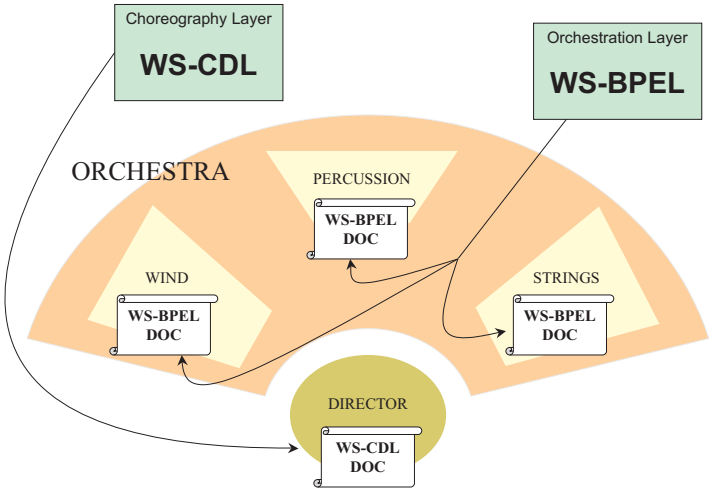


Figure 2.5: Choreography vs. Orchestration

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. 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 2.6 is an example of how WS-CDL can be useful for the integration of different kinds of web services.

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.

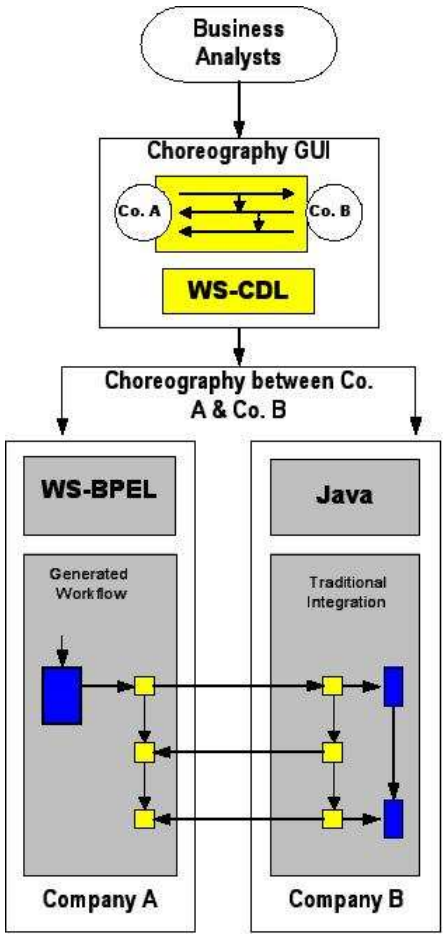


Figure 2.6: Integration of Web Services using WS-CDL.

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.

Next, we introduce the orchestration approach used in this Thesis.

2.2.1 WS-BPEL

In 2002, researchers and engineers from the main companies of the world (IBM, Microsoft, etc.) realised that the new and rapidly emerging process-oriented approach required the definition of a neat and precise language for describing how a set of intecting web services can be included in a business process. Traditional methods for integration and business process automation typically imply to embed the logic inside of the applications, complicating unnecessarily these applications. designed Moreover, this makes these applications really difficult to migrate to other platforms. The development, testing, and deployment efforts required to change these applications make integration and process changes both costly and complex [1]. 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 under-

lying business systems so that they could be more effectively changed, managed, and optimised. The idea and motivation behind almost each new technology for enterprise application development is to provide an environment where better business applications can be developed, requiring less effor. These business applications should closely align to the business processes, which should not be too complex and which should be adaptable to the changing nature of business processes without too much work. Within companies, business applications have to interoperate and integrate. Integrating different applications has always been a difficult task for various functional and technology related reasons [].

The Business Process Execution Language for Web Services (BPEL4WS), for short BPEL, was first conceived in July, 2002, with the release of the BPEL4WS 1.0 specification. This first draft was initially developed by just three companies, IBM, Microsoft, and BEA. This document proposed an orchestration language inspired by previous languages such as Web Services Flow Language (WSFL), developed by IBM and XLANG specification language developed by Microsoft. WSFL is based on the concept of directed graphs. XLANG is a block-structured language. BPEL combines both approaches and provides a rich vocabulary for the description of business processes. After this first attempt, other major companies such as SAP and Siebel Systems joined the former ones to write the version 1.1 of the BPEL4WS specification. It was released less than a year later, in May of 2003. Fortunately, this brand new version received much more attention and vendor support, leading to a number of commercially available BPEL4WS-compliant orchestration engines [9]. Before publishing it, the BPEL4WS specification was submitted to an OASIS technical committee in order to be evaluated so that the specification could evolve into an official and open standard. This technical committee was active from April 2003 to May 2007, and, during this time, a lot of contributions and improvements were received. In April 2007, WS-BPEL version 2.0 was approved as an OASIS standard.

As a proof of maturity, 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 [9]. Finally, in January 2008, another OASIS technical committee started to define a WS-BPEL extension to emcompass the

definition of human interactions ("human tasks") as part of WS-BPEL processes. Figure 2.7 summarises the evolution of WS-BPEL:

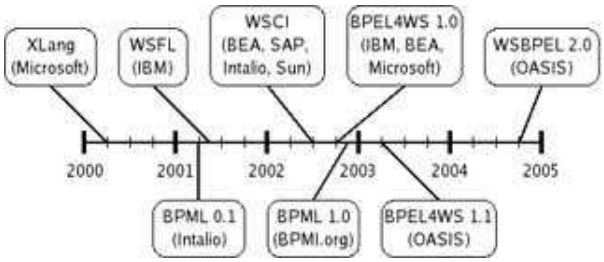


Figure 2.7: WS-BPEL evolution.

Furthermore, there were established ten original design goals associated with the definition of WS-BPEL [9]:

- Define business processes that interact with external entities through web service operations defined using WSDL, and that manifest themselves as web services using WSDL.
- Define 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 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 entities.
- 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 modelling space.
- Provide data manipulation functions for process data and the 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 "sus-

pend" and "resume" may be added in future releases for enhanced lifecycle management.

- Define a long-running transaction model, based on mature techniques such as compensation 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.

As a result, WS-BPEL along with web services technologies provide now a standardised integration interface and language for the composition of different services as well as for the automation of some tasks. Nevertheless, web scenarios are becoming more and more complex since they highly heterogeneous, that is, a lot of different services from different companies interact jointly to perfom a particular task. In particular, it is a folklore that business processes change relatively often due to this heterogeneity. Therefore, designers do not require only a way to compose a set of services, rather they also need a way to compose and modify them in the right order and in a relatively uncomplicated and straightforward way. Due to this, BPEL is sometimes compared to general purpose programming language, but it is not as powerful as one of the well-known programming language []. However, it is simpler and better suited for business process definition and, therefore, BPEL must be considered a supplement to modern languages rather a replacement.

After briefly introduce its history and design goals, we discuss next its technical details. 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. Choreography is covered by other standards, such as WS-CDL (commented previously). BPEL is designed to support the description of both behavioural service interfaces and executable service-based processes [10]. 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. In Figure 2.8, we can observe an example of the typical business process for a travel agency.

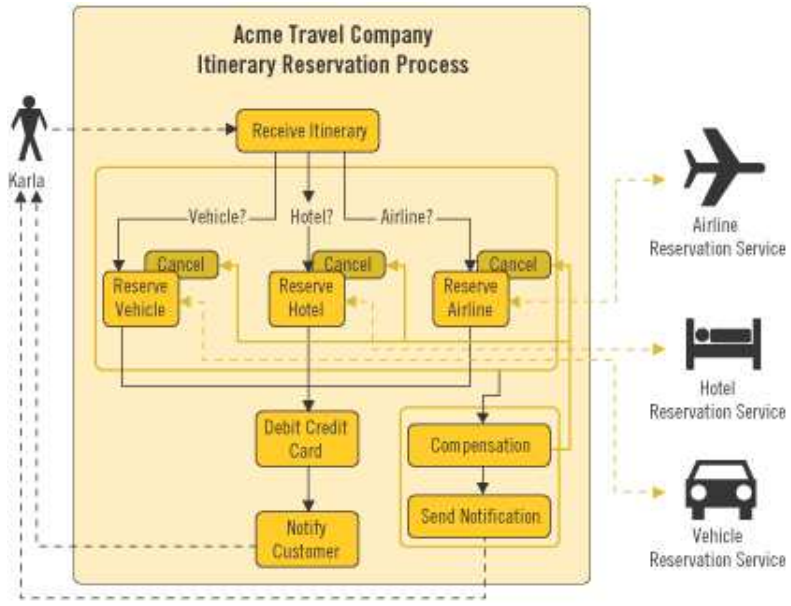


Figure 2.8: Example of a business process workflow.

According to the WS-BPEL standard, an 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 [9]. Therefore, the main different between an abstract and a executable processes is that the second one contains the exact details of business processes and, consquently, it is intended to be executed in an orchestration engine, whereas the first one offers a descriptive role, defining the message exchange between the parties involved. Specifically, an abstract process is usually used to describe the observable behavior of some or all of the services offered by an executable process and/or to define a process template that contains domain-specific best practices. Such a template can be seen as a design-time representation of the process logic, excluding execution details to be completed when mapping to an executable process. In most

cases BPEL is used for executable processes []. Moreover, the definition of a conceptual model in which one can define an abstract or an executable process is a key feature of WS-BPEL since the processes execute and interact with their partners in a consistent way regardless of the supporting platform or programming model used by the hosting environment, unlocking the potential of web services. This feature allows the development of tools and other technologies that greatly increase the level of automation, decreasing the cost in establishing cross enterprise business processes. Other benefit of using abstract processes is that they ensure the level of privacy required by some companies since the service implementation is hidden to the other participants.

Furthermore, WS-BPEL is an XML-based language which supports the web services technology stack, including SOAP, WSDL, UDDI and so on. It defines a model and a grammar for describing the behavior of a business process based on interactions between the process and its partners as well as the order of these interactions. The interaction with each partner is performed through web service interfaces, and the structure of the relationship at the interface level is encapsulated in what is called a partnerLink. WS-BPEL also introduces mechanisms for dealing with business exceptions and faults. Moreover, WS-BPEL introduces a mechanism to define how activities have to be compensated in those 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.

In Figure 2.9, we can observe a piece of the BPEL code for a booking process. BPEL processes use *variables* to temporarily store data. Variables are therefore declared on a process or on a scope within that process. Also, it provides *basic* or *structured* to declare the process logic. *Basic activities* are those which describe the elemental steps of the process behaviour [9]:

- The activity *assign* is used to store data into the process variables. This activity can be used to copy data from one variable to another as well as to populate new data in a variable using expressions. As usual, expressions are constructed using variables and constants.

```
<process name="TripReservation" targetNamespace="http://jbpm.org/examples/trip"
  xmlns:tns="http://jbpm.org/examples/trip" xmlns:tic="http://jbpm.org/examples
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:bpel="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
  xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/">

  <partnerLinks>
    <partnerLink name="traveler" partnerLinkType="tns:Traveler-Agent"
      myRole="Agent" />
    <partnerLink name="ticket" partnerLinkType="tns:Agent-Ticket"
      partnerRole="TicketIssuer" />
  </partnerLinks>

  <variables>
    <variable name="purchaseRequest" messageType="tns:purchaseRequest" />
    <variable name="cost" type="xsd:double" />
    <variable name="items" type="tns:ItemSet" />
    <variable name="cancelRequest" messageType="tns:cancelRequest" />
    <variable name="cancelResponse" messageType="tns:cancelResponse" />
    <variable name="detailRequest" messageType="tns:detailRequest" />
    <variable name="detailResponse" messageType="tns:detailResponse" />
    <variable name="dateReached" type="xsd:boolean" />
  </variables>

  <correlationSets>
    <correlationSet name="trip" properties="tns:tripLocator" />
  </correlationSets>
```

Figure 2.9: WS-BPEL code.

- The activity *empty* does nothing. For instance, one can decide to capture an exception and do nothing to handle it. Another use of *empty* is to provide a synchronization point in a parallel activity.

- The activity *wait* specifies a particular delay or deadline.

- To invoke a service, WS-BPEL offers the activity *invoke*. Normally, this activity is used to request an operation in a service. This operation is usually a basic activity in the provider. Operations can be of two types: request-response or one-way. One-way activities consist of sending a message (some variables can be enclosed) so that no response is expected as part of the operation, whereas a request-response invocation requires a message back. Evidently, this response message can be used to notify the sender about a fault during the operation. A more detailed explanation will be provided in Chapter 4.

- A *receive* activity is necesary to receive the message sent in the invoke activity. In many cases, this activity is the first part of the process.

- The *reply* activity is used to respond to a request previously accepted through an inbound message activity. For instance, it can be used in conjunction

with the receive activity to respond to the invocation of a service. Clearly, it is only meaningful for request-response interactions, but a one-way "response" can be sent by invoking the corresponding one-way operation on the sender.

- The activity *throw* is used to signal an internal fault explicitly.
- The activity *exit* is used to immediately end the process instance.
- WS-BPEL provides the user with the ability to declare new activities that are not contemplated in the specification. This is done using the *extension activities*. This extension is not explicitly contemplated in the theory of this Thesis.
- Finally, using the activity *rethrow* in a fault handler, it is possible to rethrow a fault. For instance, this activity is useful when the situation that causes the fault is not solved after the completion of the fault handler and, therefore, it is needed to rerun this handler to check if the situation has been solved afterwards.

On the other hand, *Structured activities* encode the control-flow logic of the process. The set of structured activities defined in the standard are the following:

- The activity *sequence* includes a set of activities that are performed sequentially in the order in which they appear in the structure. It ends when the last activity has finished.
- The activity *flow* provides concurrency and synchronization, creating a set of concurrent activities directly nested within the executing process and it enables synchronization dependencies between activities that are nested to it. A more detailed explanation of how this activity works is given in Chapter 4.
- The activity *if* specifies conditional behavior. As usual, the activity consists of an ordered list of one or more conditional branches defined by the "if" and optional "elseif" elements, followed by an optional "else" element.
- The activity *while* provides conditional repetitive behaviour.

- *RepeatUntil* provides the repeated execution of a contained activity. The difference with the activity *while* is that the inner activity is executed at least once.
- The activity *pick* waits for the occurrence of exactly one event from a set of events, and then executes the activity associated with that event. After an event has been selected, the other events are no longer accepted by that "pick". Moreover, a deadline for the occurrence of such events can be established in such a way if this deadline expires the activity ends. This structure has some similarity the choice operator in a process algebra althoudh with a predefined timeout. In WS-BPEL, it can be compared with a set of receive activities that run in parallel, where just only one can be executed, and a common deadline is set (to this end, the wait activity can be used).
- Lastly, the standard offers an activity (forEach) to execute the contained activity a predefined number of times. This number is expressed in the definition of the activity.

2.3 Heterogeneous Distributed Systems: Grid/Cloud Computing

In 1943, the president of IBM, Thomas J. Watson, predicted:

"I think there is a world market for about five computers"

In recent times, this phrase has been widely discussed since some authors consider that it is a clear example of failed prediction. Nevertheless, with the advent of new computational paradigms such as Grid and Cloud Computing, some authors argue that it will become a reality soon. In addition, other authors consider that this phrase is completely true nowadays since five companies are monopolising the world market [3].

Thanks to the fast development of society, daily basic services such as water, electricity, gas and telephone services are commonly supplied to citizens so that everybody can have immediate access to them. Today, these services

are known as "utility" services since customers are charged according to the consumption. In 1969, Leonard Kleinrock, one of the leading scientists of the American ARPANET agency, said: "Today, computer networks are in their infancy, but as they grow and become more sophisticated we will see the rise of the *Utility computing*". It is amazing how in 1969 a scientist could already see the usefulness of computers and the advent of a distributed computing model based on providing services and paying for them. What makes this statement more fascinating is that this year is when the Internet was born. The first version only connected 2 computers worldwide, but this person was already thinking that someday the Internet could connect millions of computers into a single network. This vision of computing (based on a model of on demand service provisioning) anticipated the massive transformation of the computer industry in the XXI century. Thus, major companies such as Google, Amazon or Microsoft are introducing it in their business model.

Unfortunately, Utility Computing is often confused with Cloud and Grid Computing. It is the underlying business model for a Grid or Cloud infrastructure, i.e. it can be seen as a mean of charging customers for computing services so that users pay only for the consumption, whereas the costs associated with the production and distribution of computing services will be undertaken by the provider. As happens with revolutionary software, protocols or any computer-related paradigm, Cloud Computing must undergo a series of steps to check if all the benefits promised by service providers really help companies to save costs and enhance the competitiveness. In this sense, Larry Ellison, founder and CEO of Oracle, believes that Cloud Computing is nothing more than a new way of naming what companies have been doing so far [3]:

> "The interesting thing about Cloud Computing is that we have redefined Cloud Computing to include everything that we already do. . . I do not understand what we would do differently in the light of Cloud Computing other than change the wording of some of our ads."
> *Larry Ellison, quoted in the Wall Street Journal, September 26, 2008.*

Many researchers have tried to define the term "Cloud Computing" without reaching a standard definition. For instance, Buyya et al. [5] define a cloud system as:

> "A cloud is a parallel and distributed system consisting of a collection of virtualized and interconnected computers that have been provisioned dynamically and they are presented as a single computational resource based on service level agreements (SLAs) established by negotiation between the service provider and the consumer."

In [14] one can find up to 21 different definitions of Cloud computing. For instance, Luis M. Vaquero et al. defines the Cloud as:

"The Cloud is a large and easy to use container of virtualized resources (such as hardware, services, development platforms . . .). These resources can be dynamically reconfigured to fit into a variable load (scale), allowing also the optimal use of these resources. This service is exploited through a pay-per-use model that is guaranteed by agreements."

Meter un par de parrafos de Grid

Finally, the main difference between a Cloud-oriented and a Grid-oriented system relies in the virtualisation of resources. In a Grid infrastructure, users do not share in real-time the resources allocated to them, whereas in a Cloud infrastructure the virtualisation is essential to serve more users, thus getting the savings promised by suppliers [].

2.4 Web services vs. Grid/Cloud Computing

In this section it is presented a summary of the main differences and sinergies between web services and Grid/Cloud Computing since a formal language to mix both approaches is one of the parts of this Thesis. First, one can consider that web services are themselves software offered as a service (SaaS) and a system is a composition of services coordinated via the Internet cooperating to perform a certain task. Nevertheless, there are still some differences between both approaches such as standardization. Above, we presented two (WS-CDL and WS-BPEL) of the standards to model web services compositions, but it is impossible to present a standard that describes the main concepts of Cloud Computing and, to some extent, of Grid Computing. One of the reasons is that Cloud and Grid are relatively new and, therefore, there has not been time to

agree a standard for them. The other reason is related to commercial policies since many big companies are competing to impose its services.

Another difference is data persistence. Web services are usually "stateless", which means that no state is saved in the system after performing an operation. The only way to save this state is to store it in a database. The main disadvantage of this approach is again the absence of agreement about a standard to do it and, therefore, this operation is completely platform dependent and it depends on the application scope. In this Thesis, we use a standard called Web Services Resources Framework (WSRF) that it is intended to solve this problem. The main advantage of it is that all the steps are standardised so that the cooperation between such systems is simple. Another advantage is that the user can decide which resources can take part in the interaction. WSRF is described in the next section.

In addition, Cloud/Grid computing could be considered as a layer to be placed on the bottom of the web services, and use them as a mean to access the resources. Thus, new standards must be defined in a similar fashion as in WSRF, but taking into account the particularities of the Cloud infrastructure. Here, we must emphasise that Cloud/Grid Computing is not only the act of offering software as a service since companies can provide infrastructure and platform as a service, which web services cannot cover.

2.5 Web Services Resource Framework

The aim of this section is to introduce the basic concepts for the management and destruction of stateful web services, i.e., web services with a set of resources associated to store the state after an operation. In this sense, we call a *WS-Resource* to the association between a web service and a persistent resource. To manage stateful web services, it is required to define the patterns used to create the relationship between the service and the resource. These patterns will reuse in most of the cases a series of widely studied technologies, e.g. WS-Addressing. Moreover, it is important to define how the properties of these resources can be accessible from outside. This is usually done through an interface.

The architecture provided by web services has been widely accepted as a means of structuring the interactions between services that are part of a distributed system. Currently, developers require a higher degree of standardization to provide additional interoperability between such services, but until mid-2004 no research group or group of experts had seriously considered the idea of proposing a standard for modelling the communication between stateful services. Thus, in January of 2004, several members of the organization Globus Alliance and the multinational IBM defined, with the help of experts from companies such as HP, SAP, Akamai, etc., the first specification and the basis of an initial architecture of WSRF. In March of 2004, these documents were sent to OASIS organization. Initially, two committees were formed to study and develop certain parts of the recently created standard. On the one hand, it was created the *WSRF Technical Committee*, which worked on four specifications: *WS-ResourceProperties, WS-ResourceLifetime, WS-Servicegroup*, and *WS-BaseFaults*. Moreover, the *WSN Technical Committee* was responsible for the rest of the specifications: *WS-BaseNotification, WS-Topics*, and *WS-BrokeredNotification*.

WS-Resource Framework is inspired by the work previously done by Global Grid Forum's Open Grid Services Infrastructure (OGSI) Working Group [12]. More specifically, WSRF can be seen as a simple refactoring of concepts and interfaces developed in the specification *OGSI V1.0*, but exploiting recent developments in the area of web services (e.g. WS-Addressing). WS- Resource Framework [4] is a specification, whose purpose is to define a generic framework for modelling and accessing WS-Resources and the relationships between them in a Grid/Cloud environment. In detail, WSRF defines the representation of the WS-Resource, specifying the messages exchanged and the XML documents required to manage the resource. A WS-Resource is defined as (i) the combination of an XML document with a type defined by one or more *portTypes* (a service may play different roles in the same interaction) and (ii) it must be addressed and accessed according to the implied resource pattern. This pattern is a derivation of the *Endpoint References* included in the standard WS-Addressing. WS-Addressing is used to standardise the endpoint reference of a WS-Resource. This endpoint reference is the address (identifier) of the WS-Resource at a given network and it must be used to identify the resource in any exchange of messages.

Typically, web service interfaces provide users with the ability to access and manipulate its state, e.g. data values that evolve by the interaction among various services. In other words, the message exchanges that are implemented in the behavior of the services are intended to allow persistent access to these resources. However, this notion is not as evident in the definition of the interface [8]. The messages sent and received by these services involve (or encourage the programmer to infer) the existence of an resource. Therefore, it is desirable the definition of standards that allow the discovery, creation, manipulation and destruction of these resources. These standards should make this complex environment as interoperable as possible. Furthermore, WSRF offers mechanisms to declare, access, monitor and destroy WS-Resources by using conventional techniques, which makes it easy to run in any platform. This is due to it is not necessary to take into account the decision logic of the resource owner. It also includes mechanisms to describe how to check the status of a resource and how to make it accessible through its interface (described in WSDL). In detail, WSRF includes the mechanisms defining the means by which []:

- a WS-Resource can be destroyed, either synchronously attending to a destroy request or a time-based (scheduled) destruction, and the resource properties may be used to inspect and monitor the lifetime of a WS-Resource (WS-ResourceLifetime);
- the state of a WS-Resource can be queried and modified via web services message exchanges (using the specification WS-ResourceProperties);
- an endpoint reference (WS-Addressing) can be renewed in the event the information contained becomes invalid or stale (WS-RenewableReferences);
- a collection of heterogeneous web services can be defined, whether or not the services are WS-Resources (WS-ServiceGroups); and
- fault reporting can be made more standardised through use of a predefined XML template (WS-BaseFaults).

2.5.1 WS-ResourceProperties

As mentioned above, WSRF uses a particular specification for defining the properties (attributes) of a WS-Resource, which is composed of the definition of the

interface in WSDL and an XML document (Resource Properties Document), specifying its properties. For example, these properties can be the disk size, processor capacity, etc. As usual, there is a bunch of messages in the specification to update one or more of theses properties or to retrieve this information.

From now on, we suppose that the Resource Properties Document is:

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

In WSRF, the operations that can be done are:

GetResourceProperty As the name suggests, this operation allows services to request the value of only one property of the document.

For instance, a possible request can be:

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

GetMultipleResourceProperties This method is equivalent to the last one, but it is intended to retrieve more than one property of the document. It can be used to prevent network congestion. The message would be:

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

SetResourceProperties This specification allows to change some properties in the document. There are three kinds of changes:

- Insert: It allows to add new properties to the document.
- Update: It is used to update the value of a property.
- Delete: To delete a property from the document.

A possible request can have the following form:

```
...
<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>

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

As it can be observed, it is possible to concate more than one operation in the same request. After proccesing this request, the document must look like this:

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

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

</GenericDiskDriveProperties>
...
```

QueryResourceProperties This method is used for querying resource properties. For example, if one wants to know if the number of blocks is greater than 20 and the block size is 1024, the following query gets this information:

```
...
<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>
...
```

The response must look like this:

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

2.5.2 WS-Base Faults

Normally, designers use interfaces defined by others, and, therefore, a method to standardise the format of error messages would facilitate the work of developers. This is the goal of WS-BaseFaults, where an error message has the following format:

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

where:

- Timestamp: It is the exact instant where the error happened.
- OriginatorReference: This is the endpoint reference of the service that originated the error.
- ErrorCode: Error code (e.g. POSIX errno) to be used by error handling systems .
- Description: Explanation of the cause (in natural language).
- FaultCause: Technical cause of the error.

Finally, note that it is possible to report an error whithout using this format.

2.5.3 WS-ServiceGroup

This specification allows users to create groups of services that share a number of properties in common, i.e., it is useful to group different web services with similar behaviours. This part of WSRf is not taken into account in BPELRF.

2.5.4 WS-ResourceLifetime

The lifetime of a WS-Resource is defined as the period between its instantiation and its destruction. The goal of this specification is to standardise the process of resource destruction and define mechanisms to monitor its lifecycle. Surprisingly, the proces to create the WS-Resource is not specified. The reason is that WSRF is intended to be used in the interaction and, therefore, the internal details of each participant are hidden. Thus, WSRF meets the requirements of SOC architecture presented previously and the transition from service-oriented architecture to resource-oriented architecture is short. For technical reasons, we have included in our language BPELRF a primitive to create the resource.

Generally, in distributed systems, clients just want to use a resource for a given time interval. For instance, in subscription systems, users decide normally the duration of the subscription. Nevertheless, in some scenarios is most appropriate to provide a manner to immediately destroy the resource. Following the last example, it could happen that the client wants to interrupt its subscription and hence the immediate destruction must be provided. As discussed above, WSRF gives two ways to destroy a WS-Resource: immediate or scheduled.

Immediate destruction To activate this kind of destruction, it is only required to add the attribute $< wsrl : Destroy/ >$ inside the field ($< Body >$) of the SOAP message that will be sent to the service To confirm the destruction, the receiver must send the same message including the attribute $< wsrl : DestroyResponse/ >$ in the field ($< Body >$) of the response SOAP message.

Scheduled destruction In this case, the WS-Resource has an associated deadline after which it is expected the resource has been destroyed. Moreover, it is reasonably expected that before this dealine the resource is available. An example of how to determine the completion time of a resource is:

```
...
  <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>
...
```

As we can see, the destruction requestor may indicate the exact destruction time as well as the local time (to avoid mismatches in how to represent the time zone). Once the *TerminationTime* is reached, the resource is destroyed without any further intervention and the requestor is reported that the resource is unavailable. WSRF has another message to inform the sender that the resource owner has received the destruction request. This option is not considered in our language BPELRF.

On the contrary, there may be a situation where more than one service is using the resource and, therefore, the resource owner can decide or not (this is not mandatory in WSRF) to implement a notification policy to inform other services that the resource is unavailable. The notification message must include the following fields:

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

where the attribute *TerminationTime* specifies the exact time of destruction and in the *TerminationReason* attribute it can be included the destruction reason.

The notification-based interaction pattern is a commonly used pattern for inter object communications. For example, the well-known publish/subscribe architecture uses this approach. In addition, it is increasingly being used in a web services context [].

In conjuction with WSRF, Web Services Notification (WSN) specifications are focused on the description of mechanisms to implement this notification-based pattern. As WSRF is based on web services, WSRF creators opted for the use of WS-Notification standard since the interoperability of both approaches is supposedly higher (they are based on web services).

WS-Notification is a family of specifications that uses a topic-based publish/subscribe pattern. It includes: standard message exchanges to be implemented by service providers that wish to participate in Notifications, standard message exchanges for a notification broker (allowing publication of messages from entities that are not themselves service providers), operational requirements expected of service providers and requestors that participate in notifications, and an XML model that describes the topics susceptible to generate notifications. The WS-Notification family includes three normative specifications: WS-BaseNotification, WS-BrokeredNotification, and WS-Topics.

In the notification process, there are three different steps:

1. First, the observation of the situation and its characteristics. This situation represents an event of interest for some services.
2. Second, the creation of notification messages that capture the characteristics of the situation; and
3. finally, the distribution of these messages to zero or more interested parties (notification consumers).

In WS-Notification, steps 1 and 2 are not taken into account since tey did not want to restrict the means by which these stages must occur. From now on, the entity in charge of performing the stages 1 and 2 is called Publisher. Other issue is how the publisher can disseminate the notification messages. In this case, two patterns can be followed: direct or brokered.

In the direct case, the publisher implements message exchanges associated with the notification producer interface (the details of this interface are out of the scope of this Thesis) and it is responsible for accepting suscription messages and sending notification messages to interested parties. Moreover, it can choose to include in its behavior the required logic or to delegate this task to specialized implementations. This last case is addressed by the WS-BaseNotification specification [].

An example of notification message (can include one or more notification messages) is:

```
...
<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>
...
```

In the brokered case, an intermediary (broker) is responsible for disseminating messages produced by one or more publishers to zero or more notification consumers. There exists three types of relationships between the publisher and the broker: simple publishing, composable publishing and demand-based publishing.

In the simple publishing scenario, the publisher entity is responsible only for the core publisher functions - observing the situation and formatting the notification message artifact that describes the situation. The dissemination step occurs when the publisher sends the notification message to the broker. In the composable publishing pattern, the publisher delegates its function to an external service that it is responsible to send the notification messages to the broker. Finally, demand-based publication is intended for use in cases where observing

the situation or formatting the messages is expensive, and therefore the notification should be avoided. To this end, the publisher will only send notifications to the broker if it is registered as a service interested in receiving notifications about a particular situation. Obviously, this will reduce the overload of the network [].

In Chapter 4, we will see that the language BPELRF avoids to include the broker role, and it is indeed the owner of the resource who sends the notifications. Moreover, the subscribers must show interest by sending a subscription message directly to the resource owner within the corresponding condition, thus reducing the overload of the net. This is due to the amount of notifications in the network is reduced. More technical details will be extended in that chapter.

2.5.5 Formal models of concurrency

Preliminaries

In this section, we will present the preliminary concepts used in this Thesis. The aim of this section is to provide the reader with a review of the main notions which are the basis of future sections as well as to fix the notion used throughout the Thesis. Thus, we will start presenting basic concepts such as the standard definition of Petri nets and we will continue with more technical details such as the addition of time features to this formal model.

Notation

The notation used in this work is the following:

1. **Numbers**
   We will denote by $\mathbb{N}_0 = \mathbb{N} \cup \{0\}$ the set of nonnegative integers including 0, and $\mathbb{R}^{\geq 0}$ be the nonnegative real numbers including 0. Obviously, $\mathbb{N}$ and $\mathbb{R}$ mean that zero is excluded from the set. Moreover, $\mathbb{N}_0^{\infty} = \mathbb{N}_0 \cup \{\infty\}$ is the set of nonnegative natural numbers including ∞.

2. **Sets and Multisets**
   We will use the standard delimiters for sets ($\{\}$) and multisets($\{\!\{\}\!\}$). As usual, let A be a set and $R : A \longrightarrow \mathbb{N}_0$, we say that $x \in A$ iff $R(x) > 0$. Moreover,

we abuse the notion using $x \in A$ to represent that x is an element of the multiset A. The cardinality of the set A is denoted by $|A|$. Given a set A, $\mathcal{B}(A)$ is the set of all finite multisets over A.

3. **Relations**
   Let X be a set, a relation over X is a set $R \subseteq X \times X$. The domain (or the set of departure) of R, denoted by $dom(R)$, is:

$$dom(R) = \{x \in X \mid \exists and \in X \, : \, (x, y) \in R\}$$

   and the codomain (or the set of destination) of R, denoted by $cod(R)$, is:

$$cod(R) = \{x \in X \mid \exists y \in X \, : \, (y, x) \in R\}$$

   Given a relation R, the *reflexive and transitive closure* of R, R^*, is defined as follows:

$$R^* = \{(x, y) \mid x = y \vee \exists x_1, \ldots, x_n, \ (x, x_1) \in R, \ldots, (x_n, y) \in R\}$$

   Moreover, the *transitive closure* of R, R^+, is given by:

$$R^+ = \{(x, y) \mid \exists x_1, \ldots, x_n, \ (x, x_1) \in R, \ldots, (x_n, y) \in R\}$$

4. **Vectores**
   La notación empleada para representar los vectores será la usual, mediante tuplas. En el caso de vectores con componentes en $\mathbb{N}_0$, diremos que $v \geq w$ sii todas las componentes de v son mayores o iguales que las correspondientes de w. Además, diremos que $v > w$ si $v \geq w$ y $v \neq w$.

2.6 Petri nets

Definition 1 (Basic Petri nets)

An *basic Petri net* (PN) is a triple $N = (P, T, F)$, where P and T are sets and F is a relation defined over $P \cup T$. Moreover, it has to satisfy the following constraints:

1. $P \cap T = \emptyset$

2. $F \subseteq (P \times T) \cup (T \times P)$

3. $dom(F) \cup cod(F) = P \cup T$

In a Petri net, P is known as the set of *places* of N, T is the set of *transitions* and F is a flow relation between the places in P and the transitions in T. This relation is graphically represented by arcs. In this Thesis, we suppose that the sets P and T are finite. Petri nets can be graphically represented by means of bipartite graphs (or bigraphs), which is a graph whose vertices can be divided into two disjoint sets (P and T in this case) such that every edge connects an element from P to T, and vice versa. In graphical representation, places are drawn as circles and transitions as rectangles or boxes. The places from which an arc runs to a transition are called the *input places* of the transition, whereas the places from which an arc runs from the transition are called the *output places*.

Let $X = P \cup T$ be a set and $x \in X$ an element of this set. The preset of x is ${}^\bullet x = \{y \in X \,|\, (y,x) \in F\}$, whereas the postset of x is defined as $x^\bullet = \{y \in X \,|\, (x,y) \in F\}$.

A net N is T-restricted iff ${}^\bullet t = t^\bullet = \emptyset \ \forall t \in T$. □

Example 2.6.1
Let $N = (P, T, F)$ be a Petri net such that:

$$P = \{p_1, p_2, p_3\}$$
$$T = \{t_1, t_2\}$$
$$F = \{(p_1, t_1), (p_2, t_1), (t_1, p_3), (p_3, t_2)\}$$

This net is depicted in Figure 2.10.

Graphically, places in a Petri net may contain a discrete number of marks called tokens.

Definition 2 (Marking on basic Petri nets)
Let $N = (P, T, F)$ be a basic Petri net. The function $M : P \longrightarrow \mathbb{N}_0$ is called *marking of* N. Then, (P, T, F, M) is called a *marked Petri net*.

Any distribution of tokens over the places will represent a configuration of the net called *marking*. The marking of a Petri net is graphically represented by

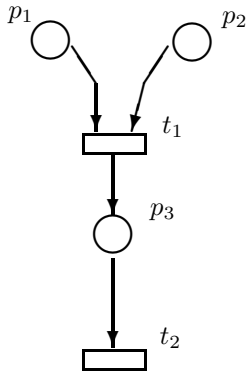


Figure 2.10: Example of a basic Petri net.

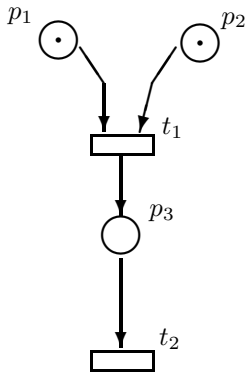


Figure 2.11: Example of a marked Petri net.

drawing in each place as many dots as tokens correspond, or putting into each place the number of tokens associated with it.

Example 2.6.2
In the net of Figure 2.10, we can consider the following marking:

$$M(p_1) = 1, \;\; M(p_2) = 1, \;\; M(p_3) = 0$$

Its graphical representation is shown in Figure 2.11.

The semantics of a Petri net is defined by the following *firing rule*, which represents the marking reached after firing a transition.

Definition 3 (Enabling rule)
Let $N = (P, T, F, M)$ a marked Petri net. A transition $t \in T$ se dice que *está*

permitida bajo el marcaje M, lo cual se denota por $M[t\rangle$, si para todo lugar $p \in P$ tal que $(p,t) \in F$ se verifica $M(p) > 0$.

Definition 4 (Firing rule)
The *firing* of a transition t enabled by the marking M produces a new marking on the net, M', defined as:

$$M'(p) = M(p) - W_f(p,t) + W_f(t,p) \quad \forall p \in P$$

where $W_f(x) = 1$ if $x \in F$ and $W_f(x) = 0$, $x \notin F$, for all $x \in (T \times P) \cup (P \times T)$. This is denoted by $M[t\rangle M'$.

Example 2.6.3
In Figure 2.11, the firing of the transition t_1 creates the marking M':

$$M'(p_1) = 0, \;\; M'(p_2) = 0, \;\; M'(p_3) = 1$$

Definition 5 (Concurrent enabling of transitions)
Let $N = (P,T,F,M)$ be a marked Petri net and $R \subseteq T$ a subset of transitions of N. The set of transitions R is *concurrently enabled*, denoted by $M[R\rangle$ iff $M(p) \geq \sum_{t \in R} W_f(p,t), \; \forall p \in P$, where $W_f(p,t)$ is defined as in the last definition.

We can also extend this definition to multisets, thus allowing multiple instances of the same transition to be fired in just one step. In this way, we say that the multiset of transitions R is enabled in M iff $M(p) \geq \sum_{t \in T} W_f(p,t) \cdot R(t)$, $\forall p \in P$.

The firing of the multiset of transitions R in M produces the new marking M' of N:

$$M'(p) = M(p) - \sum_{t \in T} (W_f(p,t) - W_f(t,p)) \cdot R(t)$$

This net evolution in just one step is denoted by $M[R\rangle M'$.

Chapter 3

Extended Petri nets

After introducing web services (and their composition) and the possible formal models that can be used to model and analyse them we will focus on this chapter in the introduction of the models used in this Thesis. Thus, we will present the extensions of the basic model of Petri nets stated previously and some properties that can be analysed. We will recall some notions (marking, firing, enableness and so on) defined in the preceding chapter and will adapt them to a particular case. This chapter is mainly divided in two parts. On the one hand, we will focus on the definition of Coloured Petri nets since they are used in the definition of the language BPELRF, whereas we will define timed-arc Petri nets, in its two variants (discrete and continuous), as they are the basis of the workflow model presented in the second part of this Thesis.

Definition 6 (General Petri nets)
A general Petri net is a 5-tuple $N = (P, T, F, K, W)$, where:

1. (P, T, F) is a basic Petri net.
2. $K : P \longrightarrow \mathbb{N}_0 \cup \{\infty\}$ is a function that indicates the maximum number of tokens in each place.
3. $W : F \longrightarrow \mathbb{N}_0^{\infty}$ is a function that indicates the multiplicity of the arcs (*weight of the arcs*).

When the context is clear, we will call them Petri nets. The function K can be ommited if it is infinite for all the places in the net.

Definition 7 (Firing rule for general Petri nets)
Let $N = (P, T, F, K, W)$ be a general Petri net.

1. A function $M : P \longrightarrow \mathbb{N}_0$ is a marking N iff $M(p) \leq K(p)$, for all $p \in P$.

2. A transition $t \in T$ is enabled in M, denoted by $M[t\rangle$, iff $W(p,t) \leq M(p) \leq K(p) - W(t,p)$, for all $p \in P$. The firing of t produces the marking M': $M'(p) = M(p) - W(p,t) + W(t,p)$, para todo $p \in P$. Again, this evolution is denoted by $M[t\rangle M'$.

3. A multiset of transitions R is enabled in M, written $M[R\rangle$, if and only if $M(p) \geq \sum_{t \in T} W(p,t) \cdot R(t)$. The firing of R produces M':

$$M'(p) = M(p) - \sum_{t \in T} (W(p,t) - W(t,p)) \cdot R(t), \ \forall p \in P$$

, denoted by $M[R\rangle M'$.

Definition 8
Let $N = (P, T, F, K, W, M_0)$ be a marked Petri net.

1. $\sigma = M_0 t_1 M_1 \dots t_n M_n$ is a finite occurrence sequence of N if and only if $\forall i \in \{1, \dots, n\}$, $M_{i-1}[t_i\rangle M_i$. Occasionally, we will write $t_1 \dots t_n$, ommiting the corresponding markings, since starting from M_0 it is easy to obtain the rest of the markings knowing the transitions fired. We extend the conventional notation to occurrence sequence, obtaining $M_0[\sigma\rangle M_n$. The set of occurrence sequences starting from M_0 are denoted by $L(N, M_0)$.

2. An ocurrence sequence $\sigma = M_0 R_1 M_1 \dots R_n M_n$ is finite iff $\forall i \in \{1, \dots, n\}$, $M_{i-1}[R_i\rangle M_i$. Again, we extend the notation to ocurrence sequence, $M_0[\sigma\rangle M_n$. The set of ocurrence sequence of N starting from M_0 is denoted by $P(N, M_0)$.

3.1 Petri nets analysis

When designing a new system, the construction of a graphical model of it is always helpful since it is interesting to broadly understand how this system works, helping designers to have a deeper knowledge about it. Nevertheless, the presence of a graphical model is not enough in many cases as the designers want the

system to meet some properties of interest. For instance, the system is useless if it reaches a dealock in each execution. To this end, it is important to have tools that allow to evaluate properties in the model. In finite sequential systems, it is not particularly challeging to check the fulfillment of certain statement, whereas the presence of concurrency complicates this task. The analysis of systems behaviour is intened to determe the compliance of certain properties such as the number of processes in a queue does not exceed certain threshold or that the mutual exclusion is guaranteed when accessing to a shared resource.

In Petri nets , there is a set of powerful tools to formally analyse the compliance of such properties. For instance, these properties check the absence of deadlocks, the reachability of a certain state, the possibility of reaching the situation after performing some computations and so on.

Normally, these properties are divided in two categories [**?**]:

3.1.1 Safety properties

A safety property asserts that *"nothing bad happens"*. Thus, they guarantee that a set of undesirable states are not reached or that the system does not execute an unwanted ocurrence sequence.

The *safety properties* are the following:

1. **Reachability**. A marking M of a marked Petri net $N = (P, T, F, W, M_0)$ is *reachable* in N iff there exists an ocurrence sequence $\sigma \in L(N, M_0)$ such that $M_0[\sigma\rangle M$. We will denote by $[M_0\rangle$ the set of reachable markings of N starting from M_0, and by $[M\rangle$ the set of reachable markings starting from the marking M

2. **Boundedness**. A marked Petri net $N = (P, T, F, W, M_0)$ is *k-bounded*, for some $k \in \mathbb{N}_0$, if all reachable marking M from M_0 hold that $M(p) \leq k$, for all $p \in P$. N is safe if it is 1-bounded. A place $p \in P$ is *n-safe* if $M(p) \leq k$, for all markings M reachable from M_0.

3. **Deadlock-free**. Let $N = (P, T, F, W, M_0)$ a marked Petri net and M be a reachable marking from M_0. M is

a *dead marking* if there is no $t \in T$ that is enabled in M. The net N is dealock-free iff there are no dead markings.

4. **Coverability**. Let $N = (P, T, F, W, M_0)$ be a marked Petri net and M be a marking of N. M is said to be *coverable* if there exists $M' \in [M_0\rangle$ such that $M' \geq M$.

3.1.2 Liveness properties

A liveness property asserts that *"something good eventually happens"*. Thus, thet guarantee that, independently of the current state of the system, a specific state can eventually be reached or that a certain ocurrence sequence can eventually be executed in the system.

The *liveness properties* are:

1. **Liveness**. Let $N = (P, T, F, W, M_0)$ be a marked Petri net. A transition $t \in T$ is said to be *live* if for all reachable marking $M \in [M_0\rangle$ there is an occurrence sequence σ starting from M such that $\sigma = t_1 \dots t_m$, with $t_m = t$. The N is *live* iff all the transitions are live.

2. **Home State**. Let $N = (P, T, F, W, M_0)$ be a marked Petri net. A marking M of N is a *home state* if for all $M' \in [M_0\rangle$, $M \in [M'\rangle$.

3. **Home Space**. Let $N = (P, T, F, W, M_0)$ be a marked Petri net. The set of markings $\mathcal{M}$ is a *home-space* of N if for all marking $M' \in [M_0\rangle$ there is marking $M'' \in \mathcal{M}$ such that $M'' \in [M'\rangle$.

4. **Cyclic**. Sea $N = (P, T, F, W, M_0)$ una Red de Petri Marcada. Se dice que N tiene un *comportamiento cíclico* si para todo marcaje $M \in [M_0\rangle$ existe una secuencia de ocurrencia σ que parte de M tal que $M[\sigma\rangle M_0$.

Let $\mathbb{N}_0 = \mathbb{N} \cup \{0\}$ and $\mathbb{N}_0^\infty = \mathbb{N}_0 \cup \{\infty\}$. A *discrete timed transition system* (DTTS) is a triple $(S, Act, \rightarrow)$ where S is the set of states, Act is the set of actions and $\rightarrow \subseteq S \times (Act \cup \mathbb{N}_0) \times S$ is the transition relation written as $s \xrightarrow{a} s'$ whenever $(s, a, s') \in \rightarrow$. If $a \in Act$ then we call it a *switch transition*, if $a \in \mathbb{N}_0$ we call it a *delay transition*. We also define the set of *well-formed time intervals* as $\mathcal{I} \stackrel{\text{def}}{=} \{[a, b] \mid a \in \mathbb{N}_0, b \in \mathbb{N}_0^\infty, a \leq b\}$ and its subset $\mathcal{I}^{\text{inv}} \stackrel{\text{def}}{=} \{[0, b] \mid b \in \mathbb{N}_0^\infty\}$ used in age invariants.

3.2 Timed extensions of Petri nets

Hablar de las redes de Merlin y luego introducir las nuestras

Definition 9 timed-Arc Petri Net

An *extended timed-arc Petri net* (ETAPN) is a 9-tuple $N = (P, T, T_{urg}, IA, OA, g, w, Type, I)$ where

- P is a finite set of *places*,
- T is a finite set of *transitions* such that $P \cap T = \emptyset$,
- $T_{urg} \subseteq T$ is the set of *urgent transitions*,
- $IA \subseteq P \times T$ is a finite set of *input arcs*,
- $OA \subseteq T \times P$ is a finite set of *output arcs*,
- $g : IA \to \mathcal{I}$ is a *time constraint function* assigning guards to input arcs,
- $w : IA \cup OA \to \mathbb{N}$ is a function assigning *weights* to input and output arcs,
- $Type : IA \cup OA \to \mathbf{Types}$ is a *type function* assigning a type to all arcs where $\mathbf{Types} = \{Normal, Inhib\} \cup \{Transport_j \mid j \in \mathbb{N}\}$ such that
  - if $Type(a) = Inhib$ then $a \in IA$ and $g(a) = [0, \infty]$,
  - if $(p, t) \in IA$ and $t \in T_{urg}$ then $g((p, t)) = [0, \infty]$,
  - if $Type((p, t)) = Transport_j$ for some $(p, t) \in IA$ then there is exactly one $(t, p') \in OA$ such that $Type((t, p')) = Transport_j$,
  - if $Type((t, p')) = Transport_j$ for some $(t, p') \in OA$ then there is exactly one $(p, t) \in IA$ such that $Type((p, t)) = Transport_j$,
  - if $Type((p, t)) = Transport_j = Type((t, p'))$ then $w((p, t)) = w((t, p'))$,
- $I : P \to \mathcal{I}^{inv}$ is a function assigning *age invariants* to places.

Note that for transport arcs we assume that they come in pairs (for each type $Transport_j$) so that their weights match. Also for inhibitor arcs and for input arcs to urgent transitions, we require that the guards are $[0, \infty]$. This restriction is important for some of the results presented in this paper and it also guarantees that we can use DBM-based algorithms in the tool TAPAAL [**?**].

The ETAPN model is not monotonic, meaning that adding more tokens to markings can disable time delays or transition firing. Therefore we define a subclass of ETAPN where the monotonicity breaking features are not allowed. In the literature such nets are often considered as the standard timed-arc Petri net model [**?**, **?**] but we add the prefix monotonic for clarity reasons.

Definition 10 Monotonic timed-arc Petri net
A *monotonic timed-arc Petri net* (MTAPN) is an extended timed arc Petri net with no urgent transitions ($T_{urg} = \emptyset$), no age invariants ($I(p) = [0, \infty]$ for all $p \in P$) and no inhibitor arcs ($Type(a) \neq Inhib$ for all $a \in IA$).

Before we give the formal semantics of the model, let us fix some notation. Let $N = (P, T, T_{urg}, IA, OA, g, w, Type, I)$ be an ETAPN. We denote by ${}^\bullet x \stackrel{\text{def}}{=} \{y \in P \cup T \mid (y, x) \in (IA \cup OA),\ Type((y, x)) \neq Inhib\}$ the preset of a transition or a place x. Similarly, the postset $x^\bullet$ is defined as $x^\bullet \stackrel{\text{def}}{=} \{y \in P \cup T \mid (x, y) \in (IA \cup OA)\}$. Let $\mathcal{B}(\mathbb{N}_0)$ be the set of all finite multisets over $\mathbb{N}_0$. A *marking* M on N is a function $M : P \longrightarrow \mathcal{B}(\mathbb{N}_0)$ where for every place $p \in P$ and every token $x \in M(p)$ we have $x \in I(p)$, in other words all tokens have to satisfy the age invariants. The set of all markings in a net N is denoted by $\mathcal{M}(N)$.

We write (p, x) to denote a token at a place p with the age $x \in \mathbb{N}_0$. Then $M = \{(p_1, x_1), (p_2, x_2), \ldots, (p_n, x_n)\}$ is a multiset representing a marking M with n tokens of ages x_i in places p_i. We define the size of a marking as $|M| = \sum_{p \in P} |M(p)|$ where $|M(p)|$ is the number of tokens located in the place p.

Definition 11 Enabledness
Let $N = (P, T, T_{urg}, IA, OA, g, w, Type, I)$ be an ETAPN. We say that a transition $t \in T$ is *enabled* in a marking M by the multisets of tokens $In = \{(p, x_p^1), (p, x_p^2), \ldots, (p, x_p^{w((p,t))}) \mid p \in {}^\bullet t\} \subseteq M$ and $Out = \{(p', x_{p'}^1), (p', x_{p'}^2), \ldots, (p', x_{p'}^{w((t,p'))}) \mid p' \in t^\bullet\}$ if

- for all input arcs except the inhibitor arcs, the tokens from In satisfy the age guards of the arcs, i.e.

$$\forall (p, t) \in IA. Type((p, t)) \neq Inhib \Rightarrow x_p^i \in g((p, t)) \text{ for } 1 \leq i \leq w((p, t))$$

- for any inhibitor arc pointing from a place p to the transition t, the number of tokens in p is smaller than the weight of the arc, i.e.

$$\forall (p,t) \in IA.\mathit{Type}((p,t)) = \mathit{Inhib} \Rightarrow |M(p)| < w((p,t))$$

- for all input arcs and output arcs which constitute a transport arc, the age of the input token must be equal to the age of the output token and satisfy the invariant of the output place, i.e.

$$\begin{aligned}\forall (p,t) \in IA.\forall (t,p') \in OA.\mathit{Type}((p,t)) = \mathit{Type}((t,p')) = \mathit{Transport}_j \\ \Rightarrow \left(x_p^i = x_{p'}^i \wedge x_{p'}^i \in I(p')\right) \text{ for } 1 \leq i \leq w((p,t))\end{aligned}$$

- for all normal output arcs, the age of the output token is 0, i.e.

$$\forall (t,p') \in OA.\mathit{Type}((t,p')) = \mathit{Normal} \Rightarrow x_{p'}^i = 0 \text{ for } 1 \leq i \leq w((p,t)).$$

A given ETAPN N defines a DTTS $T(N) \stackrel{\text{def}}{=} (\mathcal{M}(N), T, \rightarrow)$ where states are the markings and the transitions are as follows.

- If $t \in T$ is enabled in a marking M by the multisets of tokens In and Out then t can *fire* and produce the marking $M' = (M \setminus In) \uplus Out$ where $\uplus$ is the multiset sum operator and $\setminus$ is the multiset difference operator; we write $M \xrightarrow{t} M'$ for this switch transition.

- A time *delay* $d \in \mathbb{N}_0$ is allowed in M if

  - $(x + d) \in I(p)$ for all $p \in P$ and all $x \in M(p)$, and
  - if $M \xrightarrow{t} M'$ for some $t \in T_{urg}$ then $d = 0$.

  By delaying d time units in M we reach the marking M' defined as $M'(p) = \{x + d \mid x \in M(p)\}$ for all $p \in P$; we write $M \xrightarrow{d} M'$ for this delay transition.

A computation of a net N from the initial marking M_0 is $M_0 \rightarrow M_1 \rightarrow \cdots \rightarrow M_n$ is denoted by $\{M_i\}_{i=0}^n$ and we call it a *run*. If the sequence is infinite, we write $\{M_i\}_{i \geq 0}$. Moreover, we write $M \Rightarrow^* M'$ if M' is reachable from M and $[M\rangle$ represents the set of reachable markings of M.

Let $\rightarrow \stackrel{\text{def}}{=} \bigcup_{t \in T} \stackrel{t}{\rightarrow} \cup \bigcup_{d \in \mathbb{N}_0} \stackrel{d}{\rightarrow}$. The set of all markings reachable from a given marking M is denoted by $[M\rangle \stackrel{\text{def}}{=} \{M' \mid M \rightarrow^* M'\}$. By $M \stackrel{d,t}{\rightarrow} M'$ we denote that there is a marking M'' such that $M \stackrel{d}{\rightarrow} M'' \stackrel{t}{\rightarrow} M'$.

A marking M is a *deadlock* if there is no $d \in \mathbb{N}_0$ and no $t \in T$ such that $M \stackrel{d,t}{\rightarrow} M'$ for some marking M'. A marking M is *divergent* if for any $d \in \mathbb{N}_0$ we have $M \stackrel{d}{\rightarrow} M'$ for some M'.

In general, ETAPNs are infinite in two dimensions. The number of tokens in reachable markings can be unbounded and even for bounded nets the ages of tokens can be arbitrarily large. We shall now recall a few results that allow us to make finite abstractions for bounded ETAPNs, i.e. for nets where the maximum number of tokens in any reachable marking is bounded by a constant.

Let $N = (P, T, T_{urg}, IA, OA, g, w, \mathit{Type}, I)$ be a given ETAPN. In [**?**] the authors provide an algorithm for computing a function $C_{max} : P \rightarrow (\mathbb{N}_0 \cup \{-1\})$ returning for each place $p \in P$ the maximum constant associated to this place, meaning that the ages of tokens in place p that are strictly greater than $C_{max}(p)$ are irrelevant. In particular, places where $C_{max}(p) = -1$ are the so-called *untimed* places where the age of tokens is not relevant at all, implying that all the intervals on their ongoing arcs are $[0, \infty]$.

Let M be a marking of N. We split it into two markings $M_>$ and $M_\leq$ where $M_>(p) = \{x \in M(p) \mid x > C_{max}(p)\}$ and $M_\leq(p) = \{x \in M(p) \mid x \leq C_{max}(p)\}$ for all places $p \in P$. Clearly, $M = M_> \uplus M_\leq$.

We say that two markings M and M' in the net N are equivalent, written $M \equiv M'$, if $M_\leq = M'_\leq$ and for all $p \in P$ we have $|M_>(p)| = |M'_>(p)|$. In other words M and M' agree on the tokens with ages below the maximum constants and have the same number of tokens above the maximum constant.

The relation $\equiv$ is an equivalence relation and it is also a timed bisimulation where delays and transition firings on one side can be matched by exactly the same delays and transition firings on the other side and vice versa.

Theorem 3.2.1 [?]
The relation $\equiv$ is a timed bisimulation.

We can now define canonical representatives for each equivalence class of $\equiv$.

Definition 12 Cut

Let M be a marking. We define its canonical marking $cut(M)$ by $cut(M)(p) = M_{\leq}(p) \uplus \{ \underbrace{C_{max}(p)+1, \ldots, C_{max}(p)+1}_{|M_{>}(p)| \text{ times}} \}$.

Lemma 3.2.1 [?]

Let M, M_1 and M_2 be markings. Then (i) $M \equiv cut(M)$, and (ii) $M_1 \equiv M_2$ if and only if $cut(M_1) = cut(M_2)$.

Let M and M' be two markings. We say that M' *covers* M, denoted by $M \sqsubseteq M'$, if $M(p) \subseteq M'(p)$ for all $p \in P$. We write $M \sqsubseteq_{cut} M'$ if $cut(M) \sqsubseteq cut(M')$.

For monotonic timed-arc Petri nets we can now show that adding more tokens to the net does not restrict its possible behaviour.

Lemma 3.2.2

Let N be an MTAPN and $M, M' \in \mathcal{M}(N)$ be two of its markings such that $M \sqsubseteq_{cut} M'$. If $M \xrightarrow{d} M_1$ (resp. $M \xrightarrow{t} M_1$) then $M' \xrightarrow{d} M'_1$ (resp. $M' \xrightarrow{t} M'_1$) such that $M_1 \sqsubseteq_{cut} M'_1$ and $|M'| - |M| = |M'_1| - |M_1|$.

Proof. Let $M \xrightarrow{d} M_1$, resp. $M \xrightarrow{t} M_1$. As $M \equiv cut(M)$ by Lemma 3.2.1(i), we can by Theorem 3.2.1 conclude that also $cut(M) \xrightarrow{d} M_2$, resp. $cut(M) \xrightarrow{t} M_2$, such that $M_1 \equiv M_2$. Recall that $cut(M) \sqsubseteq cut(M')$ by the assumption of the lemma.

- Time delay case ($cut(M) \xrightarrow{d} M_2$). As the net does not contain any nontrivial age invariants and there are no urgent transitions, we know that also $cut(M') \xrightarrow{d} M_3$ such that $M_2 \sqsubseteq M_3$ as time delay preserves the $\sqsubseteq$-relation.
- Transition firing case ($cut(M) \xrightarrow{t} M_2$). As the net does not have any inhibitor arcs, we can see that also $cut(M') \xrightarrow{t} M_3$ by consuming exactly the same tokens in $cut(M')$ as we did in $cut(M)$. Clearly, $M_2 \sqsubseteq M_3$.

Because $cut(M') \equiv M'$ due to Lemma 3.2.1(i), we know by Theorem 3.2.1 that $M' \xrightarrow{d} M'_1$, resp. $M' \xrightarrow{t} M'_1$, such that $M_3 \equiv M'_1$. Hence $M_1 \equiv M_2 \sqsubseteq M_3 \equiv M'_1$. By Lemma 3.2.1(ii) we get $cut(M_1) = cut(M_2)$ and $cut(M_3) = cut(M'_1)$. Observe now a simple fact that $M_2 \sqsubseteq M_3$ implies that $cut(M_2) \sqsubseteq cut(M_3)$. This all together implies that $cut(M_1) = cut(M_2) \sqsubseteq cut(M_3) = cut(M'_1)$ which is another way of saying that $M_1 \sqsubseteq_{cut} M'_1$ as required by the lemma. As time delays do not change the number of tokens in M and M' and transition firing adds or removes an equal

number of tokens from both M and M', we can also conclude that $|M'| - |M| = |M'_1| - |M_1|$. □ □

3.3 Prioritized-Timed Colored Petri Nets

In this section we introduce the specific model of prioritized-timed colored Petri net that we consider for the translation. In the literature on timed extensions of Petri nets we can identify a first group of models, which assign time delays to transitions, by using either a fixed deterministic value [**?**, **?**, **?**] or choosing it from a probability distribution [**?**]. Other models use time intervals to establish the enabling times of transitions [**?**]. There are also models that introduce time on tokens [**?**, **?**, **?**]. In [**?**, **?**] a description is given of the different approaches to introduce time in Petri nets. Priorities were also introduced in Petri nets to extend the descriptive power of the model [**?**, **?**, **?**], usually by associating priority levels with transitions and modifying the firing rule to prevent the firing of a transition when another one with higher priority is enabled.

We use prioritized-timed colored Petri nets, which are a prioritized-timed extension of colored Petri nets [**?**], the well-known model supported by CPN Tools [**?**], developed by the CPN group at the University of Aarhus. In this model, places have an associated color set (data types). Each token then has an attached data value (*token color*), which belongs to the color to which the token is associated.

We will use timed colors, for which the first component will be a non-negative integer value, representing the data value, and the second component will be the token timestamp, a natural number representing the time at which the token will be available.

There is also a discrete global clock that represents the total time elapsed in the system model. Arcs also have an associated inscription (*arc expressions*), constructed using variables, constants, operators and functions. To evaluate an arc expression we need to bind the variables, which consists of assigning a value to the variables that appear in the arc inscription. These values are then used to select the token colors that must be removed or added when firing the corresponding transition.

Arc expressions can also have associated time information both for place-transition and transition-place arcs. However, only time inscriptions are needed in output arcs, and even, when all the output arcs of a transition have the same time inscription, there is a shorthand notation in CPN Tools by which this time information is associated with the transition instead of the output arcs. This is the specific model that we use in our WS-CDL semantics, i.e. we will only consider these time inscriptions in the transitions. We will therefore not use any time inscription in the arcs.

The time inscription associated with a transition is used to specify the delay that must be added to the current value of the global clock for every token generated by the firing of the transition.

Transitions can also have associated guards, which are Boolean expressions that can prevent their firing. Thus, when a transition has a guard, it must evaluate to true for the binding to be enabled, otherwise the binding is disabled and the transition cannot be fired.

Definition 13
(Notation)

The following notation will be used henceforth:

- For any $C_1, C_2 \in \mathcal{B}(X)$, we define:
  - $C_1 + C_2 \in \mathcal{B}(X)$, where $\forall x \in X : (C_1 + C_2)(x) = C_1(x) + C_2(x)$.
  - $C_1 \subseteq C_2$ if and only if $\forall x \in X : C_1(x) \leq C_2(x)$.
  - If $C_2 \subseteq C_1$ we can define the subtraction $C_1 - C_2 \in \mathcal{B}(X)$, where $\forall x \in X : (C_1 - C_2)(x) = C_1(x) - C_2(x)$.
- For any $C \in \mathcal{B}(\Sigma)$, we define the first projection $\Pi_1(C) \in \mathcal{B}(\mathbb{N}_0)$, as follows: $\forall n \in \mathbb{N}_0$, $\Pi_1(C)(n) = \sum_{m \in \mathbb{N}_0} C(n, m)$.
- For any $C \in \mathcal{B}(\Sigma)$ and $n \in \mathbb{N}_0$ we define the second projection $\Pi_2(C, n)$ as the ordered list that consists of the elements $(m_1, m_2, \ldots, m_{\Pi_1(C)(n)})$, such that $(n, m_i) \in C$, $\forall i = 1, \ldots, \Pi_1(C)(n)$ and $m_i \leq m_{i+1}$, $\forall i = 1, \ldots, \Pi_1(C)(n) - 1$.
- For any $C_1, C_2 \in \mathcal{B}(\Sigma)$, we say that $C_1 \preceq C_2$ if and only if the following conditions hold:
  - $\Pi_1(C_1) \subseteq \Pi_1(C_2)$.

- $\forall n \in \mathbb{N}_0$, taking $\Pi_2(C_1, n) = (m^1_1, \ldots, m^1_{\Pi_1(C_1)(n)})$ and $\Pi_2(C_2, n) = (m^2_1, \ldots, m^2_{\Pi_1(C_2)(n)})$, we must have $m^1_i \geq m^2_i$, $\forall i = 1, \ldots, \Pi_1(C_1)(n)$.

These conditions state that for every n the total number of elements (n, m) (moving m) must be lesser in C_1 than in C_2, and for every element (n, m) in C_1 there must be a corresponding (distinct element) (n, m') in C_2, with $m \geq m'$.

- For any $C_1, C_2 \in \mathcal{B}(\Sigma)$, with $C_1 \preceq C_2$, we define $C_2 \ominus C_1$ in the following (recursive) way:
  - For $C_1 = \emptyset$ we take $C_2 \ominus C_1 = C_2$.
  - For $C_1 \neq \emptyset$, let us consider that $C_2 = \{r^1_1.(n_1, m^1_1), \ldots, r^1_{i_{n_1}}.(n_1, m^1_{i_{n_1}}), \ldots, r^k_1.(n_k, m^k_1), \ldots, r^k_{i_{n_k}}.(n_k, m^k_{i_{n_k}})\}$, where $n_l \neq n_j$, $\forall l \neq j$, and $m^l_j < m^l_{j+1}$, $\forall l = 1, \ldots, k$ and $\forall j = 1, \ldots, i_{n_l}$.

    Since $C_1 \preceq C_2$, we can take one element $(n_l, m) \in C_1$, for some $l \in \{1, \ldots, k\}$, as well as the largest index j for which $m^l_j \leq m$. We then define recursively:

    $C_2 \ominus C_1 = (\{r^1_1.(n_1, m^1_1), \ldots, r^1_{i_{n_1}}.(n_1, m^1_{i_{n_1}}), \ldots, r^l_1.(n_l, m^l_1), \ldots, (r^l_j - 1).(n_l, m^l_j), \ldots, r^l_{i_{n_l}}.(n_l, m^l_{i_{n_l}}), \ldots, r^k_1.(n_k, m^k_1), \ldots, r^k_{i_{n_k}}.(n_k, m^k_{i_{n_k}})\}) \ominus (C_1 - \{1.(n_l, m)\})$.

    Thus, $C_2 \ominus C_1$ is obtained by removing from C_2 elements (n, m) that correspond to elements (n, m') of C_1, such that m is the largest value with $m \leq m'$.

    For instance, taking $C_1 = \{1.(2,3), 1.(2,5), 1.(1,4), 1.(7,6)\}$, and $C_2 = \{1.(2,0), 1.(2,1), 1.(2,2), 1.(1,3), 2.(7,6), 3.(3,3)\}$ it follows that $C_1 \preceq C_2$. Then, $C_2 \ominus C_1 = \{1.(2,0), 1.(7,6), 3.(3,3)\}$.

Definition 14

(Prioritized-Timed Colored Petri Nets)

We define a prioritized-timed colored Petri net (PTCPN) as a tuple $(P, T, A, V, G, E, \lambda, D, \pi)$, where[1]:

- P is a finite set of *places*, with colors in the set Σ. Thus, in our case, colors will be pairs $(n, x) \in \mathbb{N}_0 \times \mathbb{N}_0$, where n is the token value and x its timestamp.

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

$$\forall x \in P \cup T : {}^\bullet x = \{y \,|\, (y, x) \in A\} \qquad x^\bullet = \{y \,|\, (x, y) \in A\}$$

- 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 *typed variables* in Σ, i.e. $Type(v) \in \Sigma$, for all $v \in V$.
- $G : T \longrightarrow EXPR_V$ is the *guard function*, which assigns a Boolean expression to each transition, i.e. $Type(G(t)) = Bool$.
- $E : A \longrightarrow EXPR_V$ is the *arc expression function*, which assigns an expression to each arc, such that $Type(E(a)) = \mathcal{B}(\mathbb{N}_0 \times \{0\})$, which corresponds to untimed arcs, since, as mentioned above, we only attach time delays to transitions.
- λ is the *labeling function*, defined both on places and transitions.
  - Places are labeled as *entry places*, *exit places*, *error places*, *internal places* and *variable places*, which, respectively, correspond to the following labels: $\{in, ok, er, i, rv\}$. 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 of color $(0, 0)$. There is also an only *exit place* p_{ok}, such that $p_{ok}^\bullet = \emptyset$, which will be marked with one token when the system finishes correctly. Each PTCPN has also a single *error place* p_{er}, such that $p_{er}^\bullet = \emptyset$, which will become marked with one token in the event of a failure. Variable places are denoted by p_{rv}, to mean that they capture the value of variable v in role r. We will assume that a special value e is used to denote that the variable has not yet been assigned. Finally, all the remaining places are considered as *internal*.
  - Transitions are labeled as follows: $\lambda(t) \in L \cup \{\emptyset\} \cup \{fail\}$, where L is the set of basic activities, defined as follows:

$$L = \{time\_out, silent, noaction(r), assign(r, v, n), inter(r_1, r_2, v_1, v_2)\}$$

- $D : T \longrightarrow \mathbb{N}_0 \times \mathbb{N}_0$, which is the *delay function*, which associates a time interval to each transition. For $D(t) = [d_1, d_2]$, this means that a uniform probability function will be used when t is fired to select the specific discrete delay in that time interval.
- $\pi : T \longrightarrow \mathbb{N}_0$ is the *priority function*, which assigns a priority level to each transition.

In this definition, $EXPR_V$ denotes the expressions constructed using the variables in V, with the same syntax admitted by CPN Tools.

Definition 15

(Markings)

Given a PTCPN $N = (P, T, A, V, G, E, \lambda, D, \pi)$, a marking M is defined as a function $M : P \longrightarrow \mathcal{B}(\Sigma)$, which assigns a multiset of colors to each place (which can be empty).

A timed marking of a PTCPN N is a pair (M, x), where M is a marking of N and x is the current system time instant.

Initial markings are those markings for which the time annotation of every token is 0, and all variable places are marked with a single token $(0, 0)$. A marked prioritized-timed colored Petri net (MPTCPN) is then defined as a triple (N, M, x), where N is a PTCPN, and (M, x) a timed marking of it.

We define the semantics for MPTCPNs in a similar way as in [**?**], now taking into account that transitions have associated priorities. We first introduce the notion of *binding*, then the *enabling condition* and finally the *firing rule* for MPTCPNs.

Definition 16

(Bindings)

Let $N = (P, T, A, V, G, E, \lambda, D, \pi)$ be a PTCPN. A *binding* of a transition $t \in T$ is a function b that maps each variable $v \in Var(t)$ into a value $b(v) \in \Sigma$, where $Var(t)$ is defined as the set of variables that appear both in the guard of t and in the arc expressions of the arcs connected to t. We will denote by $B(t)$ the set of all possible bindings for $t \in T$.

Given an expression $e \in EXPR_V$, we will denote by $e\langle b \rangle$ the evaluation of e for the binding b.

A *binding element* is then defined as a pair (t, b), where $t \in T$ and $b \in B(t)$. The set of all binding elements is denoted by BE.

Definition 17

(Enabling condition)

Let $N = (P, T, A, V, G, E, \lambda, D, \pi)$ be a PTCPN, and (M, x) a timed marking of it. We

say that a binding element $(t, b) \in BE$ is *enabled* at the time instant x' in the timed marking (M, x) if and only if the following conditions are fulfilled:

1. $x' \geq x$.
2. $G(t)\langle b \rangle = true$.
3. For all $p \in {}^\bullet t$, $E(p,t)\langle b \rangle_{x'} \preceq M(p)$, where $E(p,t)\langle b \rangle_{x'}$ consists of the same colors as $E(p,t)\langle b \rangle$, but replacing their timestamp (which was 0) by x'.

   the notation C_{+x} is used to represent the color multiset obtained from C by delaying its timestamps by x time units:

$$C_{+x}(n,y) = \begin{cases} C(n, y-x) & \text{if } y \geq x \\ 0 & \text{otherwise} \end{cases}$$

4. There is no other binding element $(t', b') \in BE$ fulfilling the previous conditions such that $\pi(t') < \pi(t)$.
5. x' is the smallest time value for which there exists a binding element (t, b) fulfilling these conditions.

Definition 18

(Firing rule)

Let $N = (P, T, A, V, G, E, \lambda, D, \pi)$ be a PTCPN, (M, x) a timed marking of N, and an enabled binding element $(t, b) \in BE$ at instant x' in the timed marking (M, x).

The firing of (t, b) at instant x' is non-deterministic, depending on the chosen delay $d \in \mathbb{N}_0$ for the transition. This delay is randomly selected in the interval given by $D(t)$. Thus, the new timed marking (M', x') is:

$$\forall p \in P \,:\, M'(p) = M(p) \ominus E(p,t)\langle b \rangle_{x'} + E(t,p)\langle b \rangle_{d+x'}$$

Example 3.3.1

Let us consider the marked PTCPN depicted in Figure 3.1, obtained from CPN Tools. Observe that timed color tokens in CPN Tools are drawn using the notation $n`v@x$, meaning that we have n instances of a timed color token with color value v and *timestamp* x, which correspond to $n.(v, x)$ according to our formal notation. Besides, the symbol '+++' is there used to represent the union of timed multisets.

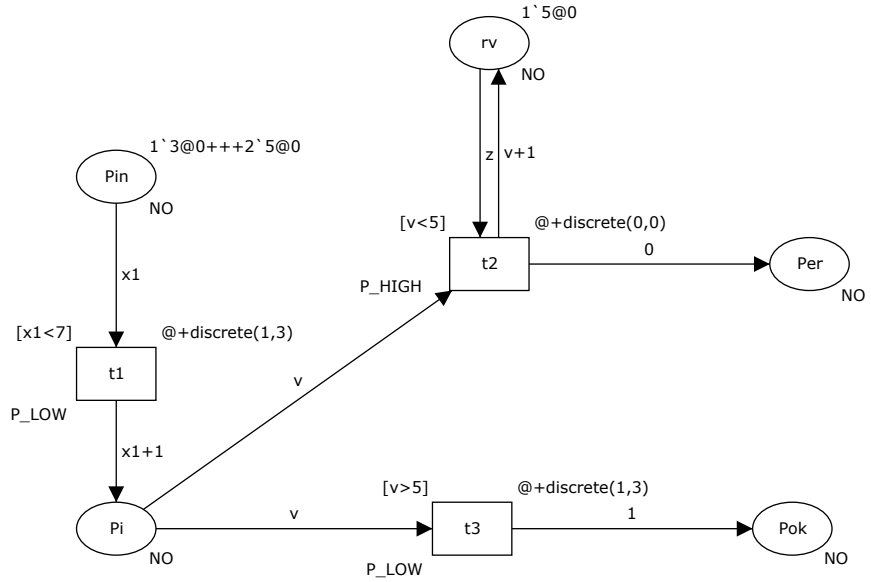


Figure 3.1: Graphical view of a PTCPN.

Thus, p_{in} is initially marked with one token of color $(3,0)$, and two tokens of color $(5,0)$, and the place rv has one token with color $(5,0)$. Transitions are labeled with their associated guard, time interval and priority information. Arcs are labeled with the corresponding expressions, in which no time delays appear, as we are considering that only transitions have associated time delays.

From the initial marking we can see that only transition t_1 can be fired (at instant 0), and any token of those in p_{in} can be used for that purpose. Taking $(5,0)$ we get the binding $x = 5$, which fulfills the transition guard. The firing of t_1 with this binding removes one instance of $(5,0)$ from p_{in}, and produces a new token on p_i. The timestamp of this new token is a discrete value in the interval $[1,3]$ (let us say 3). Thus, considering the output arc inscription we get a token $(6,3)$ on p_i.

Now, transition t_1 must fire again twice (until p_{in} becomes empty), due to the time constraints of this model. As a result we may obtain in p_i the following marking $\{1.(4,3), 1.(6,1),\ 1.(6,3)\}$ (the timestamp values depend on the values chosen from the interval $[1,3]$). The only transition that can be fired at this marking is t_3, because due to the time constraints we must first use the token $(6,1)$ and t_2 cannot be fired using this token. The firing of t_3 produces a new token on p_{ok}, whose color value must be 1, and the timestamp depends again

on the chosen delay in the time interval $[1,3]$. For instance, we could obtain the color token $(1,4)$.

Two tokens now remain in p_i, with colors $(4,3)$ and $(6,3)$, and t_2 becomes the only transition enabled (due to condition (4) of Def. 17). Its firing removes the token $(4,3)$ from p_i, the token on the place rv changes to $1.(5,3)$, and creates a new token on p_{er}, with color $(0,3)$. Finally, the remaining token $(6,3)$ on p_i only allows us to fire t_3, generating a new token on p_{ok}, with value 1 and a timestamp depending on the delay chosen for its firing.

In [**?**] there is a model that extends Merlin's nets by including dynamic priorities and resources.

The particular model that we use is also an extension of Merlin's nets, including priorities and three transition types (*black, grey* and *white*). All transitions are assigned both a time interval and a static priority. The time interval restricts the instants at which a transition is allowed to be fired. *White* transitions are not forced to fire when their clock reaches its latest firing time, grey transitions must fire once their clock reaches that value, and black transitions must fire immediately, once they fulfill all the required conditions. Of course, in the event of a conflict, any transition of those involved in the conflict can be fired (whichever type it has), although we take into account priorities to resolve the conflicts. Thus, priorities are only used in the case of conflict, when at a given marking two or more transitions are simultaneously enabled, then only those with the maximum priority are allowed to be fired at that moment.

Definition 19

(Prioritized-Time Colored Petri Nets)

We define a prioritized-time colored Petri net (PTCPN) as a tuple $N = (P, T, F, \alpha, \beta, \pi, \lambda, \Sigma, G)$, where[2]:

- P is a finite set of *places*, $P = P_i \cup P_c \cup \{p_{in}, p_{ok}, p_{er}\}$, where:
  - p_{in} is the only *entry* place of the PTCPN, initially marked with one token, fulfilling ${}^\bullet p_{in} = \emptyset$.

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

$$\forall x \in P \cup T : {}^\bullet x = \{y \,|\, (y,x) \in F\} \qquad x^\bullet = \{y \,|\, (x,y) \in F\}$$

- p_{ok} is the only *exit* place and it will be marked with one token when the system finishes correctly, fulfilling $p_{ok}^{\bullet} = \emptyset$.
- p_{er} is the only *error* place and it will become marked with one token in the event of a failure, fulfilling $p_{er}^{\bullet} = \emptyset$.
- P_i: These are called *internal* places, which have precondition and post-condition transitions, $\forall p \in P_i : \quad {}^{\bullet}p \neq \emptyset \ \vee \ p^{\bullet} \neq \emptyset$.
- P_c: These are the *colored* places, labeled with $r_i v_i$, to indicate that they are associated with variable v_i of role r_i. Their marking will be the corresponding value of v_i in role r_i, and initially they are unassigned (marked with a special value ϵ). For these places $p \in P_c$ we have ${}^{\bullet}p = p^{\bullet}$.

Markings are defined as an annotation of tokens over uncolored places (they will be represented with black dots), and with the value of the variable over the colored places. Hence, a marking M is formally defined as a function
$M : P \to \mathbf{Z} \cup \{\epsilon\}$, where $M(p_k)$ indicates the number of tokens on p_k, when it is an uncolored place, or it indicates the value of the variable v_i (or ϵ for an unassigned variable) for a colored place labeled with $r_i v_i$. We will denote by $\mathcal{M}$ the set of all possible markings of a Priorized-Time Colored Petri Net.

We consider as initial marking M_0, the marking in which the *entry* place is marked with one token, and all the colored places are marked with ϵ.

- T is a finite set of *transitions* ($P \cap T = \emptyset$), such that $T = T_w \cup T_g \cup T_b$, which are pairwise disjoint. Transitions in T_w are called *white*, transition in T_g are called grey, whereas transitions in T_b are called *black*.
- F is the *flow relation:* $F \subseteq (P \times T) \cup (T \times P)$.
- α and β define the time intervals that restrict the firing of transitions, $\alpha : T \to \mathbb{N}_0$, $\beta : T \to \mathbb{N}_0 \cup \{\infty\}$, fulfilling $\alpha(t) \leq \beta(t), \forall t \in T$.
- π is the *priority function*, $\pi : T \to \mathbb{N}_0$, which assigns a priority level to each transition.
- $\lambda : T \to L \cup \{\emptyset\} \cup \{fail\}$ is a labeling function, where L is the set of basic activities, defined as follows:

$$L = \{silent, noaction(r), assign(r, v, n), inter(r_1, r_2, v_1, v_2)\}$$

- Σ is the set of colors.

$$\Sigma : P_c \rightarrow \mathbf{Z} \cup \{\epsilon\}$$

- G is a guard function $G : T \rightarrow \mathcal{P}^{\mathcal{M}}$ where $\mathcal{P}^{\mathcal{M}}$ are predicates that are obtained by using the markings on the places and the classical logical and arithmetical operators. We will denote by $G(M,t)$ the evaluation of the predicate of t under marking M. We will omit this function in the graphical representation when it is *true*.

Example 3.3.2

In Figure **??** we can see the graphical representation of a PTCPN. Transitions are painted in black, grey and white according to their type. In Table **??** we indicate the guards, priorities and the static time intervals that the transitions have associated.

The semantics of PTCPNs is captured by the following definitions, which extend the firing rule of Merlin's nets by considering the priority and color information and the three transition types that we have introduced.

Definition 20

(Enabling transitions)

Given a PTCPN $N = (P, T, F, \alpha, \beta, \pi, \lambda, \Sigma, G)$, a marking M of it and a transition $t \in T$, we say that t is *enabled at* M if each of its input (uncolored) places contains at least one token, i.e. $\forall p \in {}^\bullet t,\ p \notin P_c\,,\ M(p) > 0$, i.e. it is enabled in the underlying Petri net (P, T, F)[3]. As usual, we denote this by $M[t\rangle$ and the set of transitions enabled at M by $E(N, M)$.

We restrict our attention to a particular class of PTCPNs, for which no transition will be enabled more that once at a time, i.e. it will never be the case that two or more instances of the same transition are enabled at a certain instant. With this restriction we avoid the semantic problems that appear in Merlin's nets when multiple enabling of transitions are allowed (see [**?**]).

Example 3.3.3

In the PTCPN of Fig. **??**, if we consider the marking M, where $M(p_2) = 1$, $M(p_7) = 1$, $M(r_1 v_1) = 1$, $M(r_2 v_2) = \varepsilon$, and the remaining places unmarked, then, we have an only enabled transition, t_3.

[3]Notice that colored places always have one colored token.

Definition 21

(States in PTCPNs)

Given a PTCPN $N = (P, T, F, \alpha, \beta, \pi, \lambda, \Sigma, G)$, we define a *state* of it as a pair (M, I), where M is a marking and I is a function $I : E(N, M) \to \mathbb{N}_0 \times (\mathbb{Z} \cup \{\infty\})$, which is defined for enabled transitions, and indicates the lower and upper time bounds that they have to be fired with respect to the current instant.

For $I(t) = (x_1, x_2)$, we will denote x_i by $\Pi_i(I(t))$, for $i = 1, 2$.

The *initial state* of a PTCPN is defined by considering an initial marking M_0 and the function I_0 defined by $I_0(t) = (\alpha(t), \beta(t))\,,\ \forall t \in E(N, M_0)$.

Example 3.3.4

For the PTCPN of Fig. **??**, taking the marking M of Example 3.3.3, its initial state is $I(t_3) = (2, 5)$.

The firing rule can now be precisely defined, but we first need a function capturing time elapsing.

Definition 22

(Time elapsing)

Given a PTCPN $N = (P, T, F, \alpha, \beta, \pi, \lambda, \Sigma, G)$ and a state of it (M, I), we say that x units of time can elapse if either:

- $E(N, M) \cap (T_b \cup T_g) = \emptyset$, or
- $\forall t \in E(N, M) \cap (T_b \cup T_g)$, $G(M, t) \equiv false$, or
- $\forall t \in E(N, M) \cap T_g$, $G(M, t) \equiv true$ and $\Pi_2(I(t)) \geq x$.
- $\forall t \in E(N, M) \cap T_b$, $G(M, t) \equiv true$ and $\Pi_1(I(t)) \geq x$.

In that case, the new state reached after that time will be (M, I'), where

$$\forall t \in E(N, M) : I'(t) = (x_1 \dot{-} x, x_2 - x)$$

taking $I(t) = (x_1, x_2)$ and $x \dot{-} y = Max\{0, x - y\}$.

From this definition we can see that *white* transitions may lose their opportunity to fire, if they are not fired when their clock has reached the latest firing time. This does not mean that they are definitely dead, because the tokens on

their preconditions can be used to fire other transitions, and they can become enabled again later. Besides, grey transitions correspond to Merlin's transitions, in the sense that time cannot elapse once the local clock of some enabled grey transition whose guard is true reaches its latest firing time. For black transitions we impose the condition that time cannot elapse once at least one of them is enabled, its guard is true and its local clock has reached the earliest firing time. Thus, our purpose is that black transitions must be fired immediately, once all the required conditions for that are fulfilled (unless they are involved in a conflict).

Example 3.3.5
For the PTCPN of Fig. **??**, taking again the marking M of Example 3.3.3, starting from the initial state $I(t_3) = (2, 5)$, 4 units of time could elapse, thus reaching a new state (M, I'), with $I'(t_3) = (0, 1)$.

The following definition introduces the so-called *potentially fireable transitions*, which are those that fulfill all the required conditions to be fired. However, taking into account that a Web Service is a reactive system, only those potentially fireable transitions that have been demanded for firing at a given instant can actually compete and therefore be fired at that instant. For this reason we split the firing into two steps. In the first one we consider a set of potentially fireable transitions (those that have been demanded), and in the second we select from this set the transition that is actually fired at that instant (one with the greatest priority in this set).

Definition 23
(Potentially Fireable Transitions)
Given a PTCPN $N = (P, T, F, \alpha, \beta, \pi, \lambda, \Sigma, G)$, a state (M, I) and an enabled transition $t \in E(N, M)$, we say that t is *potentially fireable* at that state if and only if the following conditions hold:

1. Its associated guard function is evaluated to true under M: $G(M, t) \equiv true$.
2. The earliest firing time of t is 0 and its latest firing time is greater than or equal to 0: $\Pi_1(I(t)) = 0 \ \wedge \ \Pi_2(I(t)) \geq 0$.

Definition 24
(Firing rule)

Given a PTCPN $N = (P, T, F, \alpha, \beta, \pi, \lambda, \Sigma, G)$, a state of it (M, I), and a set of potentially fireable transitions B, a transition $t \in B$ *can be fired* at that state if and only if there is no other potentially fireable transition in B having a greater priority: $\nexists t' \in B,\ \pi(t') > \pi(t)$.

The firing of t leads us to a new state, (M', I'), which is defined as follows:

1. The marking M' for uncolored places is obtained by applying the classical firing rule on Petri nets, i.e. $M'(p) = M(p) - W_F(p,t) + W_F(t,p)$, where $W_F(a) = 1$ for $a \in F$, and $W_F(a) = 0$ for $a \notin F$.

2. The marking of those colored places that are not precondition/postcondition of t keep their marking. For those colored places p_c (labeled with $r_i v_i$) that are precondition/postcondition of t ($p_c \in {}^\bullet t \cap t^\bullet$) we change their marking according to the label of t. For $\lambda(t) = asssign(r_i, v_i, n)$, we take $M'(p_c) = n$; for $\lambda(t) = inter(r_j, v_j, r_i, v_i)$, we take $M'(p_c) = M(\widetilde{p_c})$, where $\widetilde{p_c}$ is the colored place labeled with $r_j v_j$, and for any other transition labeling we also keep the same marking.

3. For every transition $t' \in E(N, M) \cap E(N, M')$, $t' \neq t$, we take $I'(t') = I(t')$.

4. For every transition $t' \in E(N, M') \setminus E(N, M)$ we take $I'(t') = (\alpha(t'), \beta(t'))$.

5. In the event that $t \in E(N, M')$ we take $I'(t) = (\alpha(t), \beta(t))$.

Notice that firing a transition takes no time to complete, so we keep in the new state the time restrictions of the transitions that were enabled before the firing and remain enabled after it. It can also be the case for the fired transition to become enabled again at the new marking, in which case it should be observed that its local clock is reset.

Example 3.3.6

For the PTCPN of Fig. **??**, taking the marking M of Example 3.3.3, from the state (M, I') with $I'(t_3) = (0, 1)$, t_3 is the only potentially fireable transition. We can then fire t_3, reaching the marking M' :

$$M'(p_3) = 1,\ M'(p_7) = 1,\ M'(r_1 v_1) = 1,\ M'(r_2 v_2) = \varepsilon,$$

and the remaining places unmarked.

3.4 Summary

Chapter 4

BPELRF

4.1 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 4.2, we present the basic concepts for a better understanding of this paper and some related works. In Section 4.3, we define the language itself and its operational semantics, whereas, in Section 4.7, 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.

4.2 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 are[1]:

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

[1] In 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 us 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.

4.3 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 ($EPRS \subseteq \mathbb{N}_0$), 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:

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

where $O \in ORCH, pl, pl_i \in PL, op, op_i \in OPS, timeout \in \mathbb{N}_0$, *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* ($\|$ 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 4.6. 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 25 State

We define a state as a pair s=(σ, ρ), where σ represents the variable values in the system and ρ captures the global resource state. We characterise σ as a global function in $\mathbb{Z}^{\text{VAR}}$, 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, $Subs_i = \{(O_{i_j}, cond'_{i_j}, A_{e_{i_j}})\}_{j=1}^{s_i}$, $i \in \{1, ..., r\}$, is the set of resource subscribers, their associated delivery conditions and the event handling activity $A_{e_{i_j}}$ that must be thrown in the case that $cond'_{i_j}$ holds; s_i is the number of orchestrators currently subscribed to this resource and $O_{i_j} \in ORCH$ 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 $\rho' = \{(O_i, EPR_i, v_i, Subs_i, t_i - 1, tag_i, A_{e_i}) |\ t_i > 1\}_{i=1}^{r}$. 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 $Add\_subs(\rho, vEPR_i, O_{i_j}, cond'_{i_j}, A_{e_{i_j}})$

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=(σ, ρ) 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 $j \in \{1, ..., s_i\}$.

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

1. (A,s)$\xrightarrow{a}$ (A', s'), a $\in$ 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 4.2.

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 $a \in Act$, which can be empty (τ). *Delay transitions* capture how the system state changes when one time unit has elapsed. In Tables 4.2, 4.3 we show the rules for these transitions.

Next, we only introduce a short explanation of some rules of Table 4.2. 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 4.6. Below, a toy example is depicted to explain how it works.

Example 4.3.1
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 $O_c = (A_c, empty)$ and $O_s = (A_s, empty)$, the BPEL-RF code for the primary activity of both participants is:

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

$A_s = receive(pl_2, info, id2); reply(pl_1, info, id4)$

According to rules **invoke**, **receive**, **reply** and $\overline{\text{reply}}$ in Table 4.2, the customer sends $\sigma(id1)$, which represents the product identifier, and starts the activity $\overline{reply}$ 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 4.3, notice that the activities for which the passage of time is allowed are *wait, empty, receive, invoke* and *pick*.

At the orchestrator level (Table 4.4), 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 4.4 captures.

Definition 27 Orchestration Operational semantics
The transitions in this level have the following form:

1. $(O : (A, s)^m) \xrightarrow{a} (O : (A', s')^{m'})$, a ∈ 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 4.5, we define the corresponding transition rules for the choreography semantics. Observe that Table 4.5 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 28 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.

4.4 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 *discov-*

ered 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:

$$Asys = assign(24, at); publishResource(1000, 24, \text{“bank”}, empty); while(actualTime() <= at, Abid)$$
$$Abid = getProp(epr, \text{“value”}, vEPR); assign(vEPR + bid(), vEPR); setProp(epr, vEPR); wait(1, 2)$$
$$A_1 = discover(\text{“bank”}, epr_1); getProp(epr_1, \text{“timeout”}, at_1); subscribe(O_1, epr_1, value < 980,$$
$$invoke(pl_1, buy, vEPR)); pick((pl_1, buy, vEPR_1, empty), empty, at_1)$$
$$A_2 = discover(\text{“bank”}, epr_2); getProp(epr_2, \text{“timeout”}, at_2); subscribe(O_2, epr_2, value < 990,$$
$$invoke(pl_2, buy, vEPR)); pick((pl_2, buy, vEPR_2, empty), empty, at_2)$$

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

4.4.1 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 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.

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.

4.5 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.

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

4.6.1 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 Tools[2].

Definition 29

(Prioritised-Timed Coloured Petri Nets)

We define a prioritised-timed coloured Petri net (PTCPN) as a tuple $(P, T, A, V, G, E, \lambda, D, \pi)$, where[3]:

- 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.
- $G : T \longrightarrow EXPR_V$ 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.
- $E : A \longrightarrow EXPR_V$ is the *arc expression function*, which assigns an expression to each arc.

[2] Official 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:

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

- 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*.

- $D : T \longrightarrow \mathbb{N}_0 \times \mathbb{N}_0$*(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.

- $\pi : T \longrightarrow \mathbb{N}_0$ 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 [**?**].

4.6.2 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 4.6 for the equivalence with the XML syntax of BPEL and WSRF):

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

where $O \in ORCH, EPR \in EPRS, pl, pl_i \in PL, op, op_i \in OPS, a, b \in \mathbb{N}_0, a \leq b$, $expr$ 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 $i \in \{1, m\}$:

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

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. 4.1. 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.

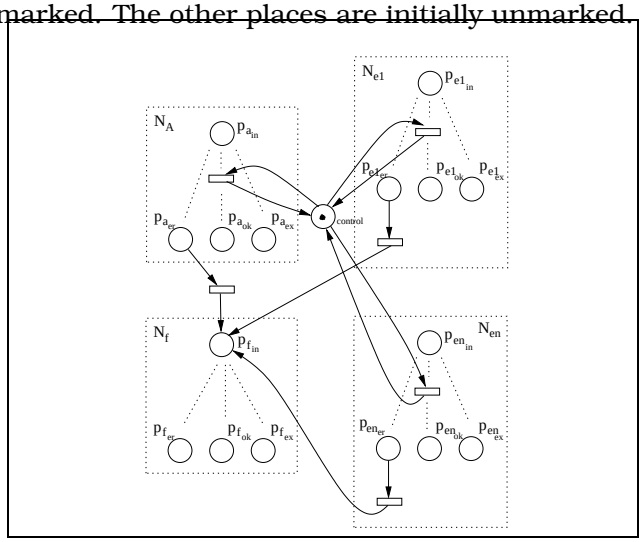


Figure 4.1: Orchestration Translation

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.

4.6.3 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 $p_{v1}, \ldots, p_{vn}$ 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 $\overline{reply}$ activity is optional in the syntax depicted in Table 4.6. Fig. **??** shows the translation for both the invoke/receive and the reply/$\overline{reply}$ 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.

4.6.4 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. 4.2), by just collapsing in a single place (this will be an internal place of the new PTCPN) the *output* place Pok 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.

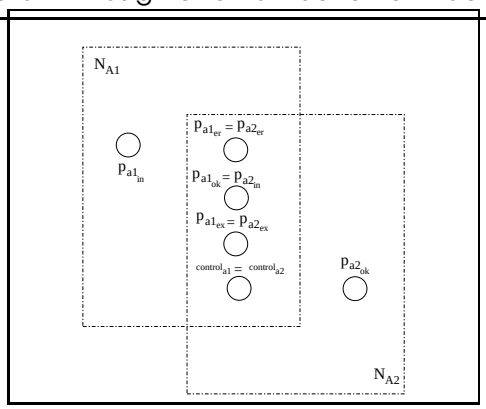


Figure 4.2: Sequence Translation

- *Parallel*: The translation for a parallel activity is depicted in Fig. 4.3, 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 <pick> 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. 4.4 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*.

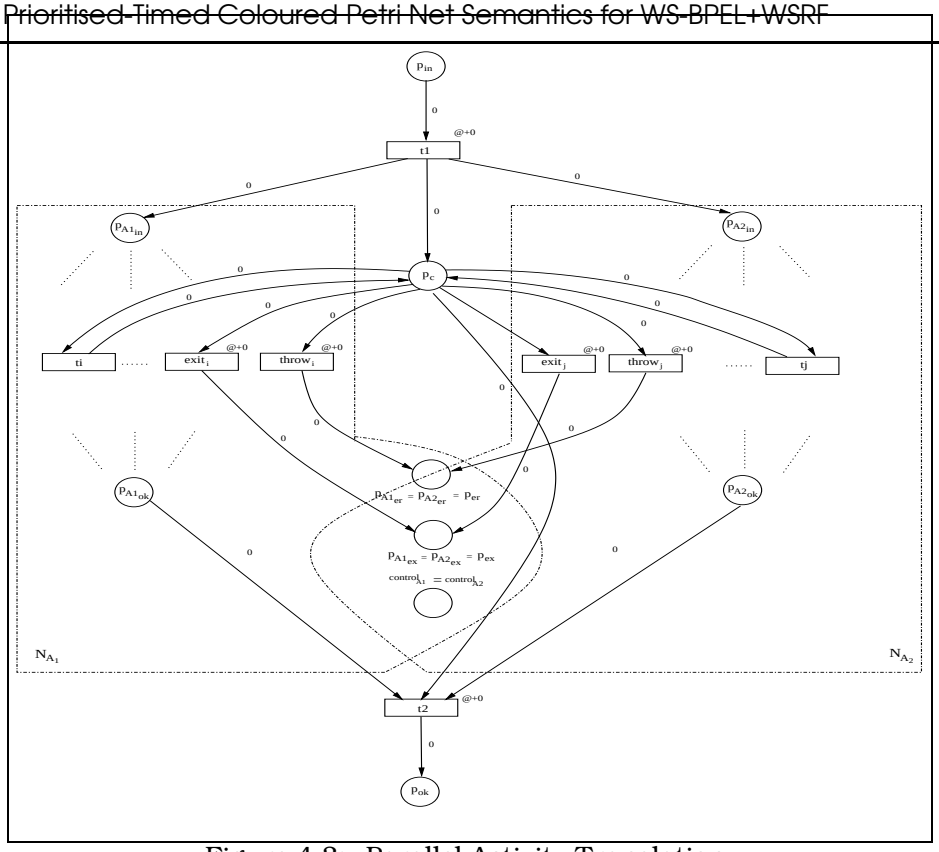


Figure 4.3: Parallel Activity Translation.

- *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. 4.5 shows this translation.

4.6.5 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. 4.6, 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. 4.6 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 life-time (max), and its value (val). Transition $t0$ is executed when the lifetime

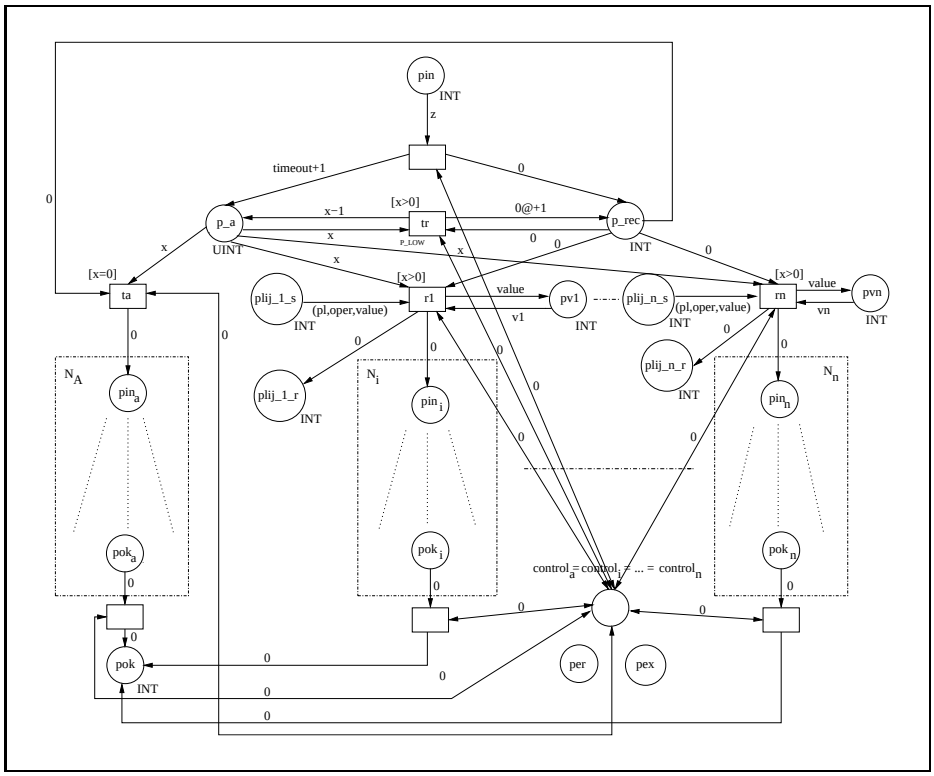


Figure 4.4: Pick Activity Translation.

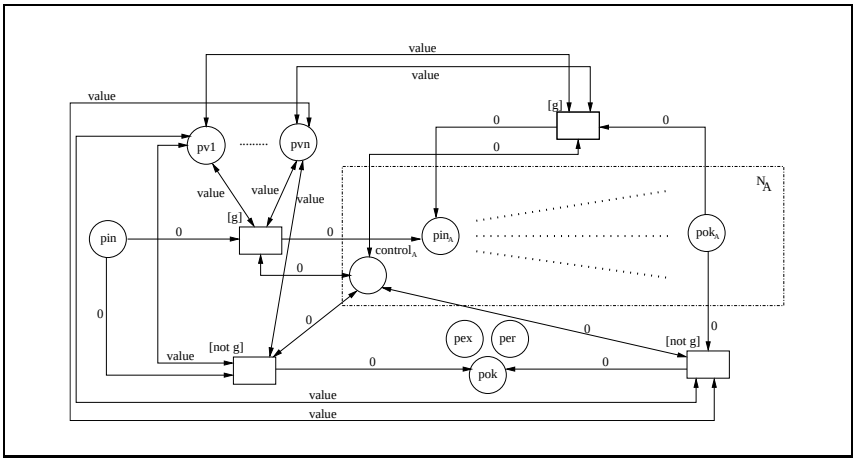


Figure 4.5: While Activity Translation.

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-

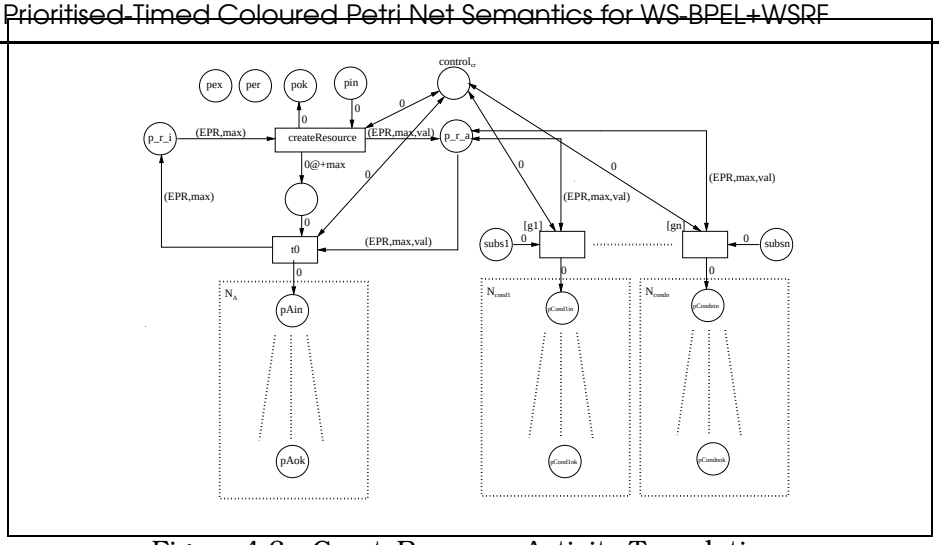


Figure 4.6: CreateResource Activity Translation.

ing activity *subscribe*. The PTCPNs $Ncond_i$ 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. 4.7 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.

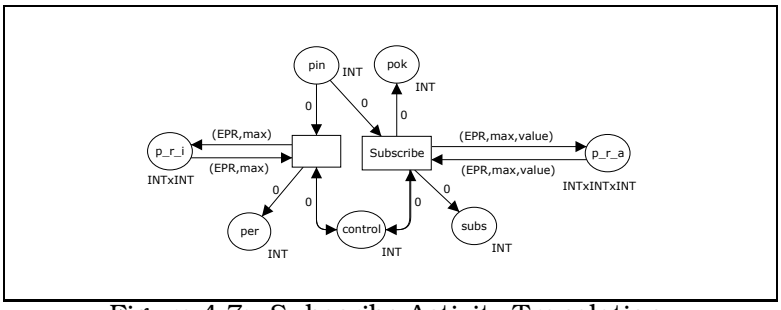


Figure 4.7: Subscribe Activity Translation.

- GetProp (EPR,v) and SetProp (EPR,expr): These are easily translated, as shown in Figs. 4.8 and 4.9, where the resource value is obtained and assigned to variable v (GetProp), or a new value is assigned to the resource (SetProp).

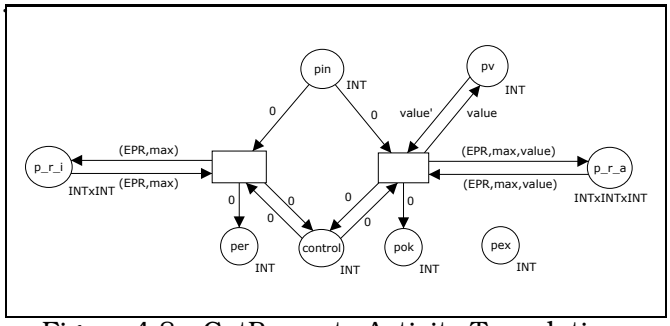


Figure 4.8: GetProperty Activity Translation.

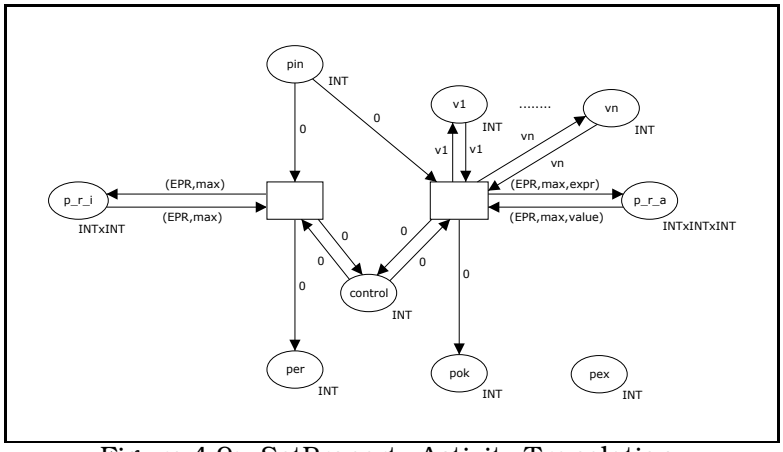


Figure 4.9: SetProperty Activity Translation.

- SetTimeout (EPR,timeout): This activity is analogous to *SetProp* activity. In this case, the resource lifetime is updated with a new value. Fig. 4.9 shows this translation.

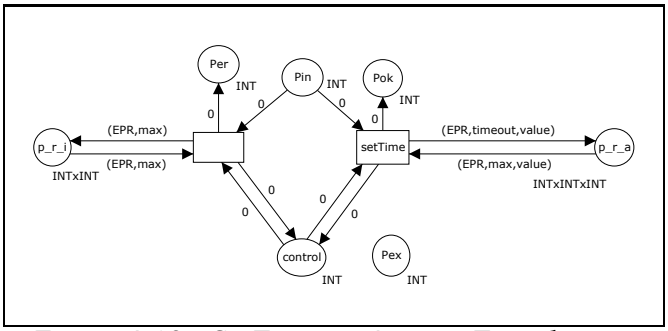


Figure 4.10: SetTimeout Activity Translation.

4.7 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 $O_i = (PL_i, Var_i, A_i, A_{f_i}, \mathcal{A}_{e_i})$, 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 $s_{0_{sys}}, s_{0_1}$ and s_{0_2} are the initial states of O_{sys}, O_1 and O_2, respectively, and all the variables are initially 0:

$$
\begin{aligned}
A_{sys} = {} & assign(10, at); createResource(EPR, 25, 11, A_{not}); \\
& while(actualTime() <= at, A_{bid}) \\
A_1 = {} & wait(1, 1); subscribe(O_1, EPR, EPR >= 0, A_{cond_1}); \\
& invoke(pl1, auction\_time_1, at1); \overline{reply}(pl1, auction\_time_1, at1); \\
& while(actualTime() <= at1, A_{bid_1}); receive(pl3, bid\_finish_1, vw_1, empty) \\
A_2 = {} & wait(1, 1); subscribe(O_2, EPR, EPR >= 0, A_{cond_2}); \\
& invoke(pl2, auction\_time_2, at2); \overline{reply}(pl2, auction\_time_2, at2); \\
& while(actualTime() <= at2, A_{bid_2}); receive(pl4, bid\_finish_2, vw_2, empty) \\
A_{not} = {} & ((invoke(pl_3, bid\_finish_1, v_w) || invoke(pl_4, bid\_finish_2, v_w)) \\
A_{bid} = {} & getprop(EPR, v_{EPR}); pick(\\
& (pl1, auction\_time1, t, reply(pl1, auction\_time1, at)),
\end{aligned}
$$

$$(pl2, auction\_time2, t, reply(pl2, auction\_time2, at)),$$
$$(pl_1, cmp, v_1, while(v_1 > v_{EPR}, assign(v1, v_{EPR});$$
$$setProp(EPR, v_{EPR}); assign(1, vw))),$$
$$(pl_2, cmp, v_2, while(v_2 > v_{EPR}, assign(v2, v_{EPR});$$
$$setProp(EPR, v_{EPR}); assign(2, vw))), empty, 1)$$
$$A_{cond_1} = getProp(EPR, v_{EPR}); invoke(pl_1, bid\_up_1, v_{EPR})$$
$$A_{cond_2} = getProp(EPR, v_{EPR}); invoke(pl_2, bid\_up_2, v_{EPR})$$
$$A_{bid_1} = receive(pl_1, bid\_up_1, v_1); assign(v_1 + random(), v_1);$$
$$invoke(pl_1, cmp, v_1); subscribe(O_1, EPR, EPR > v_1, A_{cond_1}); wait(1, 1)$$
$$A_{bid_2} = receive(pl_2, bid\_up_2, v_2); assign(v_2 + random(), v_2);$$
$$invoke(pl_2, cmp, v_2); subscribe(O_2, EPR, EPR > v_2, A_{cond_2}); wait(1, 1)$$

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_up_1 and bid_up_2 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_finish_1, bid_finish_2 update the value of v_w to inform the buyers who is the winner once the auction has expired.

In Fig. 4.11, 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 (p_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 A_1 and A_2, 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.

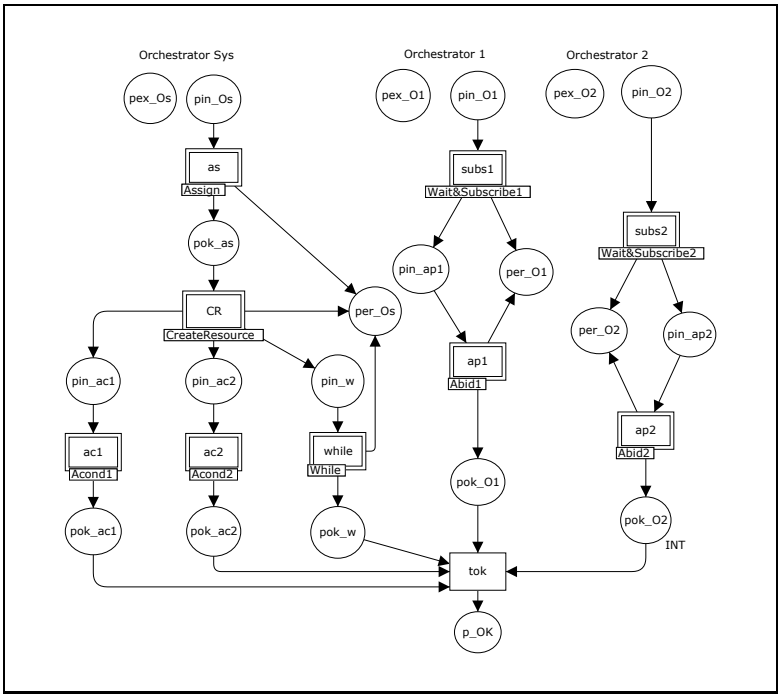


Figure 4.11: A simplified PTCPN for the online auction system.

4.8 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 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.

4.9 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 electronic 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
<process ...> <partnerLinks> ... </partnerLinks>? <Variables> ... </Variables>? <faultHandlers> ... </faultHandlers>? <eventHandlers> ... </eventHandlers>? (activities)* </process>	(A, A_f)
<throw/>/any fault	throw
<receive partnerLink="pl" operation="op" variable="v" createInstance="no"> </receive>	receive(pl,op,v)
<reply partnerLink="pl" operation="op" variable="v"> </reply>	reply(pl,op,v)
<invoke partnerLink="pl" operation="op"inputVariable="v_1" outputVariable="v_2"?> </invoke>	invoke(pl,op,v_1); $[\overline{reply}$(pl,op,v_2)]
<empty> ... </empty>	empty
<exit> ... </exit>	exit
<assign><copy><from>expr</from><to>v_1 </to></copy></assign>	assign(expr,v_1)
<wait><for>timeout</for> </wait>	wait(timeout)
<sequence> $activity_1$ $activity_2$ </sequence> <flow> $activity_1$ $activity_2$ </flow>	$\frac{A_1 \ ; \ A_2}{A_1 \parallel A_2}$
<while><condition>cond</condition>$activity_1$ </while>	while(cond,A)
<pick createInstance="no"> <onMessage partnerLink="pl" operation="op"variable="v"> $activity_1$ </onMessage> <onAlarm><for>timeout</for>$activity_1$ </onAlarm> </pick>	pick($\{(pl_i, op_i, v_i, A_i)\}_{i=1}^{n}$, A,timeout)
<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>	publishResource(O,val,timeout,tag,vEPR,A)
<wsrp:GetResourceProperty> <wsa:Address>vEPR< /wsa:Address> tns:value variable_identifier1 </wsrp:GetResourceProperty>	getProp(vEPR,v_1)
<wsrp:GetResourceProperty> <wsa:Address>vEPR< /wsa:Address> tns:timeout variable_identifier1 </wsrp:GetResourceProperty>	getTimeout(vEPR,v_1)
<wsrp:SetResourceProperties> <wsa:Address>vEPR< /wsa:Address> <wsrp:Update>expression</wsrp:Update> </wsrp:SetResourceProperties>	setProp(vEPR,expr)
<wsrl:SetTerminationTime> <wsa:Address>vEPR< /wsa:Address> <wsrl:RequestedTerminationTime> timeout </wsrl:RequestedTerminationTime> </wsrl:SetTerminationTime>	setTimeout(vEPR,timeout)
<wsnt:Subscribe> <wsnt:ConsumerReference>O</wsnt: ConsumerReference> <wsnt:ProducerReference>vEPR</wsnt: ProducerReference> <wsnt:Precondition>cond′ </Precondition> </wsnt:Subscribe>	subscribe(O,vEPR,cond′,A)
<wsnt:Notify> <wsnt:NotificationMessage> <wsnt:SubscriptionReference>O</wsnt:SubscriptionReference> <wsnt:ProducerReference>EPR</wsnt:ProducerReference> <wsnt:Message> ... </wsnt:Message> </wsnt:NotificationMessage> </wsnt:Notify>	Spawn the associated event handler activity
<invoke partnerLink="pl" operation="discover"inputVariable="tag1" outputVariable="vEPR"> </invoke>	discover(tag,vEPR)

Table 4.1: Conversion table

(Throw) $(throw, s) \xrightarrow{throw} (empty, s)$

(Exit) $(exit, s) \xrightarrow{exit} (empty, s)$

(Invoke) $(invoke(pl, op, v_1), s) \xrightarrow{invoke(pl,op,\sigma(v_1))} (empty, s)$

(Reply) $(\overline{\text{Reply}})\ (\overline{reply}(pl, op, v_2), s) \xrightarrow{\overline{reply}(pl,op,m)} (empty, s')$
where $v_2 \in VAR$, $m \in \mathbb{Z}$, $pl \in PL$, $op \in OPS$, and $s' = (\sigma[m/v_2], \rho)$.

(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 $s' = (\sigma[m/v], \rho)$

(Reply) $(reply(pl, op, v), s) \xrightarrow{reply(pl,op,\sigma(v))} (empty, s)$

(Assign) $(assign(expr, v_1), s) \xrightarrow{assign(expr,v_1)} (empty, s')$
where $v_1 \in VAR$, $expr$ is an arithmetic expression, and $s' = (\sigma[expr/v_1], \rho)$.

(Seq1) $$\frac{(A_1, s) \xrightarrow{a} (A_1', s'), a \neq exit, a \neq throw}{(A_1; A_2, s) \xrightarrow{a} (A_1'; A_2, s')}$$

(Seq2) $$\frac{(A_1, s) \xrightarrow{a} (empty, s'), a \neq exit, a \neq throw}{(A_1; A_2, s) \xrightarrow{a} (A_2, s')}$$

(Seq3) $$\frac{(A_1, s) \xrightarrow{a} (empty, s), (a = throw \vee a = exit)}{(A_1; A_2, s) \xrightarrow{a} (empty, s)}$$

(Par1) $$\frac{(A_1, s) \xrightarrow{a} (A_1', s'), a \neq exit, a \neq throw}{(A_1||A_2, s) \xrightarrow{a} (A_1'||A_2, s')}$$

(Par2) $$\frac{(A_2, s) \xrightarrow{a} (A_2', s'), a \neq exit, a \neq throw}{(A_1||A_2, s) \xrightarrow{a} (A_1||A_2', s')}$$

(Par3) $$\frac{(A_1, s) \xrightarrow{a} (empty, s), (a = throw \vee a = exit)}{(A_1||A_2, s) \xrightarrow{a} (empty, s)}$$

(Par4) $$\frac{(A_2, s) \xrightarrow{a} (empty, s), (a = throw \vee a = exit)}{(A_1||A_2, s) \xrightarrow{a} (empty, s)}$$

(Par5) $(empty||empty, s) \xrightarrow{\tau} (empty, s)$

(While1) $$\frac{cond(s)}{(while(cond, A), s) \xrightarrow{\tau} (A; while(cond, A), s)}$$

(While2) $$\frac{\neg cond(s)}{(while(cond, A), s) \xrightarrow{\tau} (empty, s)}$$

(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 $s' = (\sigma[m/v_i], \rho), \forall i \in \{1..n\}$.

(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 $s' = (\sigma[e/vEPR], \rho \cup \{O, e, val, \emptyset, t, tag, A\})$.

(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 $s' = (\sigma[val(\rho, vEPR)/v_1], \rho)$.

(GetT2) $$\frac{s = (\sigma, \rho), \sigma(vEPR) \notin \rho}{(getTimeout(vEPR, v_1), s) \xrightarrow{throw} (empty, s)}$$

(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 $s' = (\sigma[time(\rho, vEPR)/v_1], \rho)$.

(GetP2) $$\frac{s = (\sigma, \rho), \sigma(vEPR) \notin \rho}{(getProp(vEPR, v_1), s) \xrightarrow{throw} (empty, s)}$$

(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 $s' = (\sigma, \rho[expr/vEPR]_1)$.

(SetP2) $$\frac{s = (\sigma, \rho), \sigma(vEPR) \notin \rho}{(setProp(vEPR, expr), s) \xrightarrow{throw} (empty, s)}$$

(SetT) $$\frac{s = (\sigma, \rho), \sigma(vEPR) \in \rho}{(setTimeout(vEPR, t), s) \xrightarrow{setTimeout(\sigma(vEPR),t)} (empty, s')}$$
where $t \geq 1$, $s' = (\sigma, \rho[t/vEPR]_2)$.

(Subs2) $$\frac{s = (\sigma, \rho), \sigma(vEPR) \notin \rho}{(subscribe(O, vEPR, cond', A), s) \xrightarrow{throw} (empty, s')}$$

(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 $s' = (\sigma, Add\_subs(\rho, vEPR, O, cond', A))$

(SetT2) $$\frac{s = (\sigma, \rho), \sigma(vEPR) \notin \rho}{(setTimeout(vEPR, t), s) \xrightarrow{throw} (empty, s)}$$

(D1) $$\frac{s = (\sigma, \rho), tag \in \rho}{(discover(tag, vEPR), s) \xrightarrow{discover(tag,sel(\rho,tag))} (empty, s')}$$
where $vEPR \in VAR$, $s' = (\sigma[sel(\rho, tag)/vEPR], \rho))$

(D2) $$\frac{s = (\sigma, \rho), tag \notin \rho}{(discover(tag, vEPR), s) \xrightarrow{discover(tag,-1)} (empty, s')}$$
where $vEPR \in VAR$, $s' = (\sigma[-1/vEPR], \rho))$

Table 4.2: Action transition rules.

(Wait1D) $$\frac{t > 1}{(wait(t), s) \rightarrow_1 (wait(t-1), s^+)}$$

(Wait2D) $(wait(1), s) \rightarrow_1 (empty, s^+)$

(SequenceD) $$\frac{(A_1, s) \rightarrow_1 (A_1', s^+)}{(A_1; A_2, s) \rightarrow_1 (A_1'; A_2, s^+)}$$

(EmptyD) $(empty, s) \rightarrow_1 (empty, s^+)$

(ParallelD) $$\frac{(A_1, s) \rightarrow_1 (A_1', s^+) \wedge (A_2, s) \rightarrow_1 (A_2', s^+)}{(A_1||A_2, s) \rightarrow_1 (A_1'||A_2', s^+)}$$

(InvokeD) $(invoke(pl, op, v_1), s) \rightarrow_1 (invoke(pl, op, v_1), s^+)$

(ReceiveD) $(receive(pl, op, v), s) \rightarrow_1 (receive(pl, op, v), s^+)$

(PickD) $(pick(\{(pl_i, op_i, v_i, A_i)\}_{i=1}^n, A, 1), s) \rightarrow_1 (A, s^+)$

(Pick2D) $$\frac{t > 1}{(pick(\{(pl_i, op_i, v_i, A_i)\}_{i=1}^n, A, t), s) \rightarrow_1 (pick(\{pl_i, op_i, v_i, A_i\}_{i=1}^n, A, t-1), s^+)}$$

Table 4.3: Delay transition rules without notifications.

$$\textbf{(ORCH1)}\ \frac{(A,s) \xrightarrow{a} (A',s'), a \neq exit, a \neq throw}{(O:(A,s)^m) \xrightarrow{a} (O:(A',s')^m)} \qquad \textbf{(ORCH2)}\ \frac{(A,s) \xrightarrow{throw} (empty,s)}{(O:(A,s)) \xrightarrow{throw} (O:(A_f,s)^f)}$$

$$\textbf{(ORCH3)}\ \frac{(A,s) \xrightarrow{throw} (empty,s)}{(O:(A,s)^f) \xrightarrow{throw} (O:(empty,s)^f)} \qquad \textbf{(ORCH4)}\ \frac{(A,s) \xrightarrow{exit} (empty,s)}{(O:(A,s)^m) \xrightarrow{exit} (O:(empty,s)^m)}$$

$$\textbf{(ORCH5)}\ \frac{(A,s) \rightarrow_1 (A',s^+)}{(O:(A,s)^m) \rightarrow_1 (O:(A',s^+)^m)}$$

Table 4.4: Action and delay transition rules for orchestrators.

$$\textbf{(Chor1)}\ \frac{(O_i:(A_i,s)^{m_i}) \xrightarrow{exit} (O_i:(empty,s)^{m_i})}{\{(O_k:(A_k,s)^{m_k})\}_{k=1}^c \xrightarrow{exit} \{(O_k:(A'_k,s)^{m_k})\}_{k=1}^c}$$
where *if* $k \neq i, A'_k = A_k$.

$$\textbf{(Chor2)}\ \frac{(O_i:(A_i,s)^{m_i}) \xrightarrow{throw} (O_i:(A'_i,s)^{m'_i})}{\{(O_k:(A_k,s)^{m_k})\}_{k=1}^c \xrightarrow{throw} \{(O_k:(A'_k,s)^{m'_k})\}_{k=1}^c}$$
where *if* $k \neq i, A'_k = A_k,$ *and* $m'_k = m_k$.

$$\textbf{(Chor3)}\ \frac{(O_i:(A_i,s)^{m_i}) \xrightarrow{a} (O_i:(A'_i,s')^{m'_i}),\ a \neq exit,\ a \neq \underline{throw},\ a \neq receive,\ a \neq invoke,\ a \neq reply,\ a \neq \overline{reply},\ a \neq pick}{\{(O_k:(A_k,s)^{m_k})\}_{k=1}^c \xrightarrow{a} \{(O_k:(A'_k||N(O_k,s'),s'')^{m'_k})\}_{k=1}^c}$$
where *if* $k \neq i, A'_k = A_k, m'_k = m_k$ *and* $s'' = Subs(s')$.

$$\textbf{(Chor4)}\ \frac{(O_i:(A_i,s)^{m_i}) \rightarrow_1 (O_i:(A'_i,s^+)^{m_i}),\ \forall i \in \{1,\ldots,c\},\ \text{and rules chor5, chor6, chor7 are not applicable}}{\{(O_k:(A_k,s)^{m_k})\}_{k=1}^c \rightarrow_1 \{(O_k:(A'_k,s^+)^{m_k})\}_{k=1}^c}$$

$$\textbf{(Chor5)}\ \frac{(O_i:(A_i,s)^{m_i}) \xrightarrow{invoke(pl,op,n)} (O_i:(A'_i,s)^{m_i}),\ pl=(O_i,O_j),\ (O_j:(A_j,s)^{m_j}) \xrightarrow{receive(pl,op,n)} (O_j:(A'_j,s')^{m_j})}{\{(O_k:(A_k,s)^{m_k})\}_{k=1}^c \xrightarrow{invoke(pl,op,n)} \{(O_k:(A'_k,s')^{m'_k})\}_{k=1}^c}$$
where *if* $k \neq i, k \neq j, A'_k = A_k, m'_k = m_k$.

$$\textbf{(Chor6)}\ \frac{(O_i:(A_i,s)^{m_i}) \xrightarrow{reply(pl,op,n)} (O_i:(A'_i,s)^{m_i}),\ pl=(O_i,O_j),\ (O_j:(A_j,s)^{m_j}) \xrightarrow{\overline{reply}(pl,op,n)} (O_j:(A'_j,s')^{m_j})}{\{(O_k:(A_k,s)^{m_k})\}_{k=1}^c \xrightarrow{reply(pl,op,n)} \{(O_k:(A'_k,s')^{m'_k})\}_{k=1}^c}$$
where *if* $k \neq i, k \neq j, A'_k = A_k, m'_k = m_k$.

$$\textbf{(Chor7)}\ \frac{(O_i:(A_i,s)^{m_i}) \xrightarrow{invoke(pl,op,n)} (O_i:(A'_i,s)^{m_i}),\ pl=(O_i,O_j),\ (O_j:(A_j,s)^{m_j}) \xrightarrow{pick(pl,op,n,A)} (O_j:(A'_j,s')^{m_j})}{\{(O_k:(A_k,s)^{m_k})\}_{k=1}^c \xrightarrow{invoke(pl,op,n)} \{(O_k:(A'_k,s')^{m'_k})\}_{k=1}^c}$$
where *if* $k \neq i, k \neq j, A'_k = A_k, m'_k = m_k$.

Table 4.5: Choreography transition rules.

WS-BPEL/WSRF Syntax	Model
<process ...> <partnerLinks> ... </partnerLinks>? <Variables> ... </Variables>? <faultHandlers> ... </faultHandlers>? <eventHandlers> ... </eventHandlers>? (activities)* </process>	(PL,Var,A,A_f,$\mathcal{A}_e$)
<throw/>/any fault	throw
<receive partnerLink="pl" operation="op" variable="v" createInstance="no"> </receive>	receive(pl,op,v)
<reply partnerLink="pl" variable="v"> </reply>	reply(pl,v)
<invoke partnerLink="pl" operation="op" inputVariable="v_1" outputVariable="v_2"?> </invoke>	invoke(pl,op,v_1); $[\overline{reply}$(pl,op,v_2)]
<empty> ... </empty>	empty
<exit> ... </exit>	exit
<assign><copy><from>expr</from><to>v_1</to></copy></assign>	assign(expr,v_1)
<wait><from>a</from><to>b</to> </wait>	wait(a,b)
<sequence> $activity_1$ $activity_2$ </sequence> <flow> $activity_1$ $activity_2$ </flow>	$\frac{A_1 \ ; \ A_2}{A_1 \ \| \ A_2}$
<while><condition>cond</condition>$activity_1$ </while>	while(cond,A)
<pick createInstance="no"> <onMessage partnerLink="pl" operation="op"variable="v"> $activity_1$ </onMessage> <onAlarm><for>timeout</for>$activity_1$ </onAlarm> </pick>	pick($\{(pl_i, op_i, v_i, A_i)\}_{i=1}^{n}$, A,timeout)
<invoke partnerLink="Factory"operation="CreateResource" inputVariable="val,timeout" outputVariable="EPR"> </invoke><assign><copy><from variable="EPR">part="ref" query="/test:CreateOut/wsa:endpointreference"</from> <to> partnerlink="Factory"</to></copy></assign>	createResource(EPR,val,timeout,A)
<wsrp:GetResourceProperty> <wsa:Address>EPR< /wsa:Address> variable_identifier </wsrp:GetResourceProperty>	getProp(EPR,v)
<wsrp:SetResourceProperties> <wsa:Address>EPR< /wsa:Address> <wsrp:Update>expression</wsrp:Update> < /wsrp:SetResourceProperties>	setProp(EPR,expr)
<wsrl:SetTerminationTime> <wsa:Address>EPR< /wsa:Address> <wsrl:RequestedTerminationTime> timeout </wsrl:RequestedTerminationTime> </wsrl:SetTerminationTime>	setTimeout(EPR,timeout)
<wsnt:Subscribe> <wsnt:ProducerReference>EPR</wsnt: ProducerReference> <wsnt:Precondition>cond′ </Precondition> </wsnt:Subscribe>	subscribe(EPR,cond′,A)
<wsnt:Notify> <wsnt:NotificationMessage> <wsnt:ProducerReference>EPR</wsnt:ProducerReference> <wsnt:Message> ... </wsnt:Message> </wsnt:NotificationMessage> </wsnt:Notify>	Executes the associated event handler activity

Table 4.6: Conversion table

Chapter 5

Timed-arc workflow nets

Chapter 6

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] What is bpel and why is it so important to my business? Technical report.

[2] 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.

[3] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the clouds: A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009.

[4] T. Banks. Web Services Resource Framework (WSRF) - Primer. *OASIS*, 2006.

[5] Rajkumar Buyya, James Broberg, and Andrzej M. Goscinski. *Cloud Computing Principles and Paradigms*. Wiley Publishing, 2011.

[6] E. Clarke and J. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 1996.

[7] O. Grumberg E. Clarke and D. Peled. *Model Checking*. MITPress, Cambridge, MA, 1999.

[8] 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.*

[9] Organization for the Advancement of Structured Information Standards (OASIS). *Web Services Business Process Execution Language (WS-BPEL) Version 2.0*, April 2007.

[10] 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.

[11] A. Pnueli. The temporal logic of programs. *Proc. 18th IEEE Symp. on Foundations of Computer Science*, 1977.

[12] 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.

[13] J. Sifakis T. Henzinger, X. Nicollin and S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111(2):193–244, 1994.

[14] Luis M. Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner. A break in the clouds: Towards a cloud definition. *SIGCOMM Comput. Commun. Rev.*, 39(1):50–55, December 2008.

Bibliography

[1] What is bpel and why is it so important to my business? Technical report.

[2] 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.

[3] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the clouds: A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009.

[4] T. Banks. Web Services Resource Framework (WSRF) - Primer. *OASIS*, 2006.

[5] Rajkumar Buyya, James Broberg, and Andrzej M. Goscinski. *Cloud Computing Principles and Paradigms*. Wiley Publishing, 2011.

[6] E. Clarke and J. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 1996.

[7] O. Grumberg E. Clarke and D. Peled. *Model Checking*. MITPress, Cambridge, MA, 1999.

[8] 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.*

[9] Organization for the Advancement of Structured Information Standards (OASIS). *Web Services Business Process Execution Language (WS-BPEL) Version 2.0*, April 2007.

[10] 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.

[11] A. Pnueli. The temporal logic of programs. *Proc. 18th IEEE Symp. on Foundations of Computer Science*, 1977.

[12] 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.

[13] J. Sifakis T. Henzinger, X. Nicollin and S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111(2):193–244, 1994.

[14] Luis M. Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner. A break in the clouds: Towards a cloud definition. *SIGCOMM Comput. Commun. Rev.*, 39(1):50–55, December 2008.