Abstract. Web services are becoming a standard means to enable distributed business in a variety of scenarios. Nevertheless, the widespread adoption of Web services relies on the development of tools that support the whole development cycle of Web services. This document describes the implementation of a software architecture that enables the automated composition, monitoring and verification of Web services.
Acknowledgements.

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

The partners in this project are: Istituto Trentino di Cultura (Coordinator), Università degli Studi di Trento, Università degli Studi di Genova, Università degli Studi di Roma “La Sapienza”, DeltaDator S.p.A.
Chapter 1

Executive Summary

Web services are becoming a standard means to enable distributed business in a variety of scenarios. The enactment of services on the web is extremely promising to realize a number of novel applications, and improve existing ones. However, as pointed out by several sources, acting on the basis of a variety of independently designed and deployed services collaborating poses a substantial challenge that requires the provision for powerful support tools. The goal of the ASTRO project is that of providing a platform that supports the development process of Web services, and in particular provides powerful automated functionalities for composing Web services (i.e. building novel services that rely on existing ones), verifying them against some formal requirements, and monitoring their execution. The deliverable D7.1 [IioTD] spells out in detail these requirements and provides a high level view of the ASTRO architecture meant to support them. In this deliverable, we aim at providing a more detailed architectural view in terms of software components, providing the reader a full-blown description of the tools involved in the ASTRO chain, and of their functionalities, clarifying their roles and describing the language choice taken for the current architectural instantiation.
Contents

1 Executive Summary i

2 Introduction 1

3 The Software Architecture 3

4 The languages 7
  4.1 The input languages ........................................ 7
    4.1.1 The Web service description languages ............... 7
    4.1.2 The requirements languages .......................... 11
    4.1.3 The choreography language ............................ 12
  4.2 The intermediate language ................................ 14
  4.3 The output languages ..................................... 16
    4.3.1 The composition output language ..................... 16
    4.3.2 The monitor output language ........................ 16
    4.3.3 The verification output language .................... 16

5 The wsCHAINMANAGER tool 17
  5.1 Overview .................................................. 17
  5.2 Design and implementation ............................... 18

6 The wsREQUIREMENT tool 22
  6.1 Overview .................................................. 22
  6.2 Design and implementation ............................... 23

7 The wsTRANSLATOR tool 27
  7.1 Overview .................................................. 27
  7.2 Design and implementation ............................... 28

8 The wMON, wSYNTH, and NuSMV tools 30
  8.1 The wMON tool ............................................ 30
  8.2 The wSYNTH tool .......................................... 31
  8.3 The NuSMV tool ............................................ 31
9 History of the Deliverable

9.1 1st year ................................................................. 33
9.2 2nd Year ............................................................... 33
9.3 3rd year ............................................................... 33
9.4 4th Year ............................................................... 34

A Knowledge-level representation: impact on languages 35

B XML-schema for choreography files 38
Chapter 2

Introduction

The Web services paradigm supplies a universal technological foundation for the integration of business processes, be they localized within a company or distributed across different organizations. The adoption of Web services paves the way to large improvements in a variety of scenarios, e.g. in e-Government, where linking different services of the state and region administrations can enable the user to save considerable effort and time. In distributed business, it makes it possible to combine different sources into an integrated customer service.

However, the composition of existing services is a complex task that can be extremely time-demanding and error prone even for specialists, let alone for non-experienced users. Therefore, the success of the Web service approach strongly relies on the development of powerful support tools that enable the composition of services. On top of that, verification and monitoring capabilities are also crucial to make sure that the design of novel services is satisfactory w.r.t. some user requirements, and that these guarantees are respected during the actual execution. This justifies the growing interest, in the Web service area, on this genre of issues, and more generally on those techniques and tools that might address the issues relevant to the evolution and adaptation life cycle of distributed Business Processes as a whole.

The design of support tools for automated Web service composition, verification and monitoring poses relevant research and technological challenges. Both in principle and in practice, such functionalities are extremely complex and need to manipulate complex inputs, such as e.g. sets of published protocols and formal logic requirements. The goal of the ASTRO project is the design and implementation of a software platform for the automated composition, monitoring and verification of Web services, capable of showing the results of the research conducted in the fields above. The deliverable D7.1 [IioTD] presents the requirements of such a platform, along with a generic architecture able to cope with those requirements. In this document we delve into the details of a concrete implementation of that reference architecture; namely, we present the detailed software architecture that enables the automated composition of Web services as well as other useful
crucial business process management activities, as for instance, design-time verification and run-time monitoring.

In particular we will detail on the languages and the tools that have been devised for addressing the issues of the Web service development and adaptation life-cycle. We will present the implementations of the algorithms and tools that cooperate for achieving in a flexible and effective manner the objectives of composition, monitoring and verification of Web services. After describing the overall picture in which the tools operate we will describe their internal behavior in order to point to the reader’s attention their capability of being extended and improved for several purposes and their ability of being adapted to different contexts and different languages.

More specifically, the deliverable is structured as follows: chapter 3 describes the general architectural design; chapter 4 describes the languages that are used for representing information along the elaboration and the results of the elaboration itself; chapters 5, 6, 7 and 8 describe the various tools that compose the architecture, namely the coordinator of the activities, the editor of the requirements, the translator, and the tools that implement the composition, monitoring and verification algorithms.

In chapter 9 we wrap up with the description of the historical evolution of the development during the years of the project.
Chapter 3

The Software Architecture

The architecture described in the deliverable D7.1 [IioTD] aims at setting up a framework enabling the application of state-of-the-art techniques for the composition, monitoring and verification of Web services. The deliverable D7.1 [IioTD] also presents a generic reference implementation of such an architecture that shows which are the macro-steps that we envisage as necessary in order to satisfy in a flexible way the requirements of the ASTRO project and briefly hints at the tools that are necessary to perform the tasks.

This chapter presents the instantiation of that reference architecture into a software platform. The aim of this section is that of describing the mechanisms that govern the execution of the composition, monitoring and verification tasks, along with other important steps of the Web services life cycle. Here we provide an overview of the overall framework; a deeper analysis of each component within such a background will take place in the following chapters.

The design of the software architecture stems in a natural way from the acknowledgement of the existence of an ordered sequence of phases that is necessary to fulfill the requirements. The reference architecture implicitly asserts the existence of a sequence of elaborations that must be carried on, to go from the definition of the Web services domain to the attainment of the expected results. This series of steps are conceptually common to all the different kinds of elaborations prescribed in the requirements and can be summed up as follows:

**Definition of the domain and requirements.** In this phase, the user points out the elements of the domain being investigated, along with the requirements or goal of the elaboration. This phase produces as a result a document called *choreography file* which lists all the elements and pieces of information that are to be elaborated. This phase is common to all the different kinds of elaborations.

**Translation into an intermediate representation.** This phase consist of the translation of the input sources into an intermediate formal language for the representation of
state transition systems for business processes, the IL language. This step represents an important junction that decouples the definition phase, where several possible different input languages can be used, from the real elaboration that is carried on using a neutral representation.

**Elaboration of the intermediate representation.** The elaboration branches depending on the objective. Due to the different algorithms that are necessary for composing, monitoring and verifying the Web services, the operations that are carried out on the intermediate representation obtained in the previous phases, are different. What is important to note is that, despite this branching of the activities, the application of the search techniques takes still into consideration the same abstract intermediate representation of the problem, and the results of the elaborations conducted in this phase can be still expressed using the neutral IL formalism. This guarantees, once again, a complete independence of the elaborations from the source languages.

**Generation of the results.** The results determined by the composition, monitoring and verification algorithm are translated into those languages that are more suitable for their exploitation. In particular, in the case of composition the emission target language is the BPEL4WS [ACD^+03] language that can be deployed and run onto a BPEL execution engine; SMV [McM] is the target language for verification while JAVA [GJJB05] is the language that is used for emitting the monitors that will be deployed onto a Web Server for performing their task.

**Deployment/Run.** The results achieved at this stage are not useful unless they are deployed and run on their relevant execution tools. The newly composed process is to be deployed onto a process execution engine, while the generated monitors of the services must be deployed onto a Web server. In the case of verification the results of the previous elaboration, namely the possible counterexample, can be used as a mean of analysis for the improvement of the Web service life-cycle. It is only after these further steps, that the objectives they such activities were thought for can be considered achieved.

The identification of such phases represents an important achievement of the ASTRO project in that they both represent the common execution pattern for composition, monitoring and verification activities and at the same time form the skeleton of the architecture in which the different involved tools and languages find their settlement.

The conceptual view of the software architecture implementing the principles explained in the deliverable D7.1 [IioTD] is shown in Fig. 3.1.

The picture includes both the tools that represent the execution engine of the platform and the languages that are used by the tools for exchanging information during the execution. The layout of the architecture follows a linear pattern that reflects the idea of a tool-chain elaboration that is shared amongst the different goals of the platform following the phases mentioned above. Therefore the ASTRO platform fulfills the variety of its
requirements, basing on a common schema that is adapted where necessary to the different Web services elaboration tasks. This allows to maintain as small as possible the set of tools and languages that are needed to satisfy the requirements. Besides, most of the management of the execution can be easily governed by a unique coordination tool, the **wsChainManager**.

The starting point of the execution is the choreography file **CHOR**; the user specifies within this file both the domain and the requirements of the problem to be solved. This file can either be compiled by the user or can be the result of the elaboration of the **wsRequirement** tool. In fact, such a tool is designed to help the user in the filling in of the choreography file. Following the chain, the problem represented in the choreography is then transformed, by the **wsTranslator**, into a neutral formalism, the **IL** intermediate language. This formalisms is central to the architecture and to its execution. As a matter of fact it represents the means for capturing the **State Transition Systems** that are the basis of the elaboration performed onto the Web services. All the elaborations that are carried on for the composition, monitoring and verification of the Web services can be thought of as transformations of State Transition System that the **IL** can express. The fact that we adopt this neutral formalism for applying the algorithmic transformation relevant to the composition, monitoring and verification tasks, let us free to open to other possible different tools and technologies for the accomplishment of the same tasks.

The elaboration then is carried on by different tools, **wsYn**th for composition, **wmOn** for monitoring and **NuSMV** for verification, depending on the target objective. All the

---

**Figure 3.1**: The ASTRO software platform.
tools share the SMV concrete language as their native input language for historical or practical reasons. The SMV language represents itself a specific way of capturing a state machine that is equivalent to IL. At the end of execution the different elaborations produce results that are quite different both as for the language and for their usage. The result of composition is a BPEL4WS file that can be deployed and run onto a BPEL execution engine. The result of monitoring is a JAVA file that can be deployed and run onto a Web Server endowed with a relevant monitoring framework. The results of verification may include a counterexample execution trace that can be visualized or even run by a simulator.

Summing up, the architecture comprises a list of reference languages that are used as milestones for the input and output of the tools executions and a wide set of tools that performs the real work. These tools have an heterogeneous nature and serve for different purposes. In order to coordinate this flows of elaboration a dedicated tool as been conceived as another important part of the architecture: the wsCHAINMANAGER. The wsCHAINMANAGER implements the intuition that there is a common flow of execution, separated in several phases, that is shared amongst the different Web services activities. The role of the wsCHAINMANAGER is that of coordinator and launcher of the actions that occur during single executions of the tool chain. It consist of a unique graphical interface that can cope with the different kind of elaborations that are considered within the ASTRO platform. The wsCHAINMANAGER is a versatile tool that, given a definition of the problem to be solved, that is to say a choreography file, automatically perform the tasks that are necessary to achieve the desired results. The wsCHAINMANAGER triggers the actions that go from the representation of the problem in the IL format to the deployment of the results. From this console the user can decide which Web services elaboration to perform, possibly specifying execution options and monitoring the execution of the single tools of the suite.

It can be easily seen that such an architecture, addresses both the functional requirements expressed in the deliverable D7.1 [IioTD] and the fundamental non functional requirement of independence with respect to the input languages used for expressing the elements of the domain. Furthermore the architecture is open with respect to other possible forms of Web services elaboration and with respect to the integration of other complementary activities of the Web services life cycle.

In the following chapters we will detail the different languages used and the tools that have been introduced in this chapter.
Chapter 4

The languages

The architecture of the platform contemplates the usage of different languages for the exchange of information from and to the user and, furthermore, between the tools of the platform itself. In this chapter we briefly review all of the formalisms that are used for representing the info that is processed during the Web service elaborations and in particular to describe the languages and the schemas that have been explicitly devised for the ASTRO software.

4.1 The input languages

Several different languages are used for supplying the sources of info for the elaborations. Some of them (e.g., BPEL4WS, used for representing business processes and LTL, used for specifying the verification requirements) have been adopted since they are de facto standards in their relevant area, others (e.g., the language for expressing the composition requirement and for describing the choreography of processes) have been appositely devised for the ASTRO project. The general criteria used for the choice and the design of such languages are of course their expressive power, their friendliness towards the users and their adherence to current standards. The specific criteria for each language will be explained in the following sections.

4.1.1 The Web service description languages

The language used within the ASTRO platform for the description of business processes is BPEL4WS[ACD+03], an XML based language for the orchestration of Web services. One of its most important features is its ability of describing the processes at different level of abstractions thus supplying the opportune level of visibility and details to the activities. In particular, if on one side it is necessary to be able to define business pro-
cesses with all those details that make them really executable on the relevant engines, on the other side it is important to be able to specify only an abstract view on the behavior of the process itself, thus hiding the implementation details. This is useful when processes are to be published and for some reasons only the abstract interaction protocol of the process with the external agents is necessary. This may be the case when the detailed info about the internal elaboration of the processes are not relevant or they are to not to be disclosed. These two aspects of business process description languages are also known respectively as executable representation of the process and abstract representation of the process. BPEL4WS is endowed with such a twofold way of defining and publishing business processes: the choreography concepts it expresses are meant to be used both by the external (abstract) and internal (executable) views of a business process. The business processes defined using BPEL4WS interact with each other and with other external entities through Web service operations. In order to represent these operations BPEL4WS takes advantage of the representational power of the XML based language WSDL [WSD], that provide for the definition of the stational elements of the domain where business processes interacts via Web services operations. While BPEL4WS is powerful in expressing the dynamic work flow of activities that processes are made of, WSDL is necessary to define all those functional components that are put in place within a service oriented infrastructure.

BPEL4WS has been chosen within the ASTRO platform as a reference language for the description of input processes since it is de-facto a standard for the definition of business processes and its expressiveness contemplates all those features (e.g., data manipulation functions, correlations mechanisms, recovery and exception handling, transaction management, ...) that are relevant for the ASTRO requirements.

In the following we introduce a sample of BPEL4WS expressive power. The process that is presented, and whose graphical representation is reported in Fig. 4.1, defines an Hotel reservation service endowed with the following behavior: the process receives a request for a reservation; after verifying the availability of rooms the process prepares either a not-available or a valid offer response message to be sent to the inquiring service. In the case a valid offer is sent, then the Hotel process waits for a positive or negative acknowledgement of the offer.

The BPEL4WS code that implements such a process is listed below. The header of the process definition includes the declaration of the namespaces, both used and defined by the process itself. Then the description of the elements of the process starts. Only one partnerLink definition is necessary to describe the relationship of this process with the other processes. The variables section introduces the names and the types of the message variables that are used by the process. The reservation request received by the Hotel, the offer that the Hotel makes to the customer as well as the final positive or negative acknowledgements that terminates the process are the defined variables. The correlationsSets section provides the means for recognizing the possible different instances of this process that might be running simultaneously, and, thus, to deliver the messages to the right instance of the process. The interaction of the process with the other processes
Figure 4.1: A graphical representation of the Hotel BPEL process.

during the execution of such a flow is defined by message-based primitives for sending and receiving information through Web service calls (e.g., receive, invoke). The behavior of the process is then determined by a sequence of imperative style constructs (e.g., sequence, switch) that establishes the flow of the activities and by other activities for the handling of messages and variables (e.g., pick, assign).

```xml
<?xml version="1.0" encoding="UTF-8"?>
<process name="Hotel" suppressJoinFailure="yes"
targetNamespace="http://astroproject.org/BusinessProcesses/Hotel"
xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/
xmlns:ns2="http://astroproject.org/BusinessProcesses/Hotel"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <partnerLinks>
    <partnerLink myRole="Hotel_Service" name="Hotel_PLT" partnerLinkType="ns2:Hotel_PLT" partnerRole="Hotel_Customer"/>
  </partnerLinks>
  <variables>
    <variable messageType="ns2:hRequestMsg" name="hRequestMsg"/>
    <variable messageType="ns2:hOfferMsg" name="hOfferMsg"/>
    <variable messageType="ns2:hNot_availMsg" name="hNot_availMsg"/>
    <variable messageType="ns2:hAckMsg" name="hAckMsg"/>
    <variable messageType="ns2:hNackMsg" name="hNackMsg"/>
  </variables>
  <correlationSets>
    <correlationSet name="H_CS" properties="ns2:key"/>
  </correlationSets>
  <sequence>
    <receive createInstance="yes" name="Receive_request" operation="hRequest"

<correlation initiate="yes" set="H_CS" />
</correlations>
</receive>

<switch name="Is_available">
  <case condition="bpws:getVariableData('hRequestMsg', 'location', '/ns2:location')="Paris"">
    <sequence>
      <assign name="Prepare_offer">
        <copy>
          <from expression=""Hilton"" />
          <to part="hotel" query="/ns2:hotel" variable="hOfferMsg" />
        </copy>
        <copy>
          <from part="key" variable="hRequestMsg" />
          <to part="key" variable="hOfferMsg" />
        </copy>
      </assign>
      <invoke inputVariable="hOfferMsg" name="Hotel_offer" operation="hOffer"
        partnerLink="Hotel_PLT" portType="ns2:Hotel_CallbackPT">
        <correlations>
          <correlation pattern="out" set="H_CS" />
        </correlations>
      </invoke>
      <pick>
        <onMessage operation="hAck" partnerLink="Hotel_PLT" portType="ns2:Hotel_PTT"
          variable="hAckMsg"/>
        <correlations>
          <correlation set="H_CS" />
        </correlations>
        <empty name="Ack" />
      </onMessage>
      <onMessage operation="hNack" partnerLink="Hotel_PLT" portType="ns2:Hotel_PTT"
        variable="hNackMsg"/>
      <correlations>
        <correlation set="H_CS" />
      </correlations>
      <empty name="Nack" />
    </sequence>
    <otherwise>
      <sequence>
        <assign name="Prepare_answer">
          <copy>
            <from part="key" variable="hRequestMsg" />
            <to part="key" variable="hNot_availMsg" />
          </copy>
        </assign>
        <invoke inputVariable="hNot_availMsg" name="Not_available" operation="hNot_avail"
          partnerLink="Hotel_PLT" portType="ns2:Hotel_CallbackPT">
          <correlations>
            <correlation pattern="out" set="H_CS" />
          </correlations>
        </invoke>
      </sequence>
    </otherwise>
  </case>
  <otherwise>
    <sequence>
      <assign name="Prepare_answer">
        <copy>
          <from part="key" variable="hRequestMsg" />
          <to part="key" variable="hNot_availMsg" />
        </copy>
      </assign>
      <invoke inputVariable="hNot_availMsg" name="Not_available" operation="hNot_avail"
        partnerLink="Hotel_PLT" portType="ns2:Hotel_CallbackPT">
        <correlations>
          <correlation pattern="out" set="H_CS" />
        </correlations>
      </invoke>
    </sequence>
  </otherwise>
</switch>
</sequence>
</process>
4.1.2 The requirements languages

Composition

The identification of the right language for expressing the requirements for the composition of Web services must take into consideration the particular nature of the interactions that is established among the business processes that are involved in the composition. This nature is twofold: on one side, when composing business processes that interact with each other one has to cope with the relationship that must be established amongst the behavior of the new composed service with respect to the behaviors of the composing services. Thus one part of the requirement must define the newly composed service in terms of the goal it must achieve. Of course, this is not the explicit specification of the new process itself but rather an implicit description of the protocol its behavior must adhere to without any of the implementation details that will be the result of the composition. On the other side the interaction between processes has also to do with another aspect represented by the relationship between the data that are passed back and forth between the processes. When a new process is being composed starting from the initial ones, a new set of relationships between these data and, possibly other newly defined data that comes into play, is to be derived. Thus another part of the composition requirement must specify constraints on the possible flow of messages that the behavioral part of the composition makes possible.

For these reasons the language for expressing the Web service composition goal is actually made by two distinct components: the control flow language that is in charge of the behavioral aspects of the composition and the data flow language that is in charge of the aspects related to data constrainment. A single goal for the composition problem is thus composed by two distinct parts that concur in driving the composition of the new service. The control flow portion of the requirement specifies the requirement of the new interaction protocol established among services. With this language it is possible to express prioritized complex goals that takes into account high level constructs expressing the conjunction or the prioritized choice between simpler goals. The basic goals, that concur to form complex goals, are represented by propositional formulae, which in turn, consist of predicates making explicit references to the processes elements (variables and messages). The exact formal definition of the language considered can be found in [PTB05, PTBA05]. The data flow portion of the composition goal that asserts the correlation between the messages that are exchanged amongst the processes in the choreography is conceptually a hyper graph where the messages (the nodes of the graph) are linked together by arches that witness the kind of relationship between those pieces of info. A complete description of the expressiveness capabilities of the data flow language can be found in [APT05, MPT06].
Monitoring

The input language for specifying the Web service properties to be monitored within the ASTRO platform must fulfill the following requirements. First of all, owing to the Web service domain we deal with, it must be able to express properties about the exchange of messages that occur between the processes and about the beginning and termination of the processes themselves. Besides, such a language must be able to deal both with \textit{instance} monitors and with \textit{class} monitors. The difference between these two kinds of monitors is that the former are monitors that control a single instance of a given process while the latter are able to collect info about all the instances of a given class of processes. Finally, with respect to the type of information that is monitored, we can have two kind of properties: boolean ones that witness the occurrence or not of a given event, and numerical one which assess measurable quantities and time intervals.

A language that comprises all such features is a derivation of the \textit{past fragment} of the LTL logic whose detailed description can be found in [BTPT06a].

Verification

The input language for the specification of the requirements to be verified is designed in order to satisfy the following expressivity constraints. First of all it has to provide for two distinguished kind of properties: \textit{assertion} properties and \textit{possibility} properties. The former are required to hold on every execution of the composition, while the latter are required to hold on at least one execution of the system. Besides, there is another kind of assertion that is desirable when verifying systems and that correspond to \textit{deadlock} property. Checking them means to verify that the system does not comprise the possibility that the evolution gets stuck.

In order to represent all of such requirements we exploit temporal logic formalism with linear model of time (LTL logic[Eme90]). Although other variants of time interpretations are used in literature, the linear model allows for more natural and adequate representation of behavioral properties of concurrent systems. This language allows one to express requirements on ordering of events and interactions, reachability of particular states, etc. The requirements for the verification of Web services are then expressed using the LTL language.

4.1.3 The choreography language

All the languages introduced in the previous sections concur to generate the definition for the composition, monitoring and verification problems. By defining the elements of the domain (the Web services) and the goals (the requirements) of the computations, such languages are able to completely define the inputs that are to be elaborated by the tools of the ASTRO platform. All those pieces of information are collected into a unique input
source that is the choreography file.

The choreography file is an XML file that acts as a container for all the definitions that are relevant for a given Web service composition, monitoring or verification task. Since it assembles together all the definition of the domain and all the intended goals, the choreography file represents the global input source that is necessary for the ASTRO tools.

In particular the pieces of information we are dealing with are the instantiation of the ones that are described in [IioTD] and in the relevant papers where the requirements for the composition, monitoring and verification problems are expressed. Thus, for composition, the choreography file presents the list of processes that are to be composed and the goal that is to drive the synthesis of the new service. For monitoring, again the list of processes that are relevant to the problem is presented along with a formal definition of the properties to be monitored. For verification, similarly to monitoring, we require the list of processes involved in the computation plus the properties to be verified opportunely expressed into a formal language. The complete definition of the schema of the choreography language is contained in an XSD document that is reported in appendix B.
4.2 The intermediate language

The intermediate language, IL, is a language for the specification of State Transition Systems that plays a pivotal role within the ASTRO platform. The need of a new language for representing State Transition Systems within the ASTRO platform stems from two main reasons:

1. The ASTRO platform deals with a particular kind of State Transition Systems that are those deriving from the representation of business processes. Hence, there is the need of a language whose expressiveness matches the operations that are involved in the web services interactions that occur within business processes.

2. The ASTRO platform aims at being independent from the formalisms used for describing the input sources that are the Web services. Hence, there is the need for an abstract internal representation and a (human readable) language that define unambiguously such elements of the domain. Such a representation should not be bound to any particular language, but it should cope with all of the possible way of representing business processes. In this way the algorithms for the composition of Web services preserve their universal flavor and their results can be explicitly represented in a human readable form at any stage of the elaborations.

As such, the role of such a language within the ASTRO platform is pivotal. IL is the core of all the elaborations that are carried on along the tool chain, that permit the composition, monitoring and verification tasks. All the transformations that are performed on the Web services occur on internal data structures that mirror the IL and that can serialized into a string of this language at any stage of the elaboration.

The formal description of IL reflects the adoption of a generic language for the representation of state transition to the field of business processes. A generic state transition system is defined in the IL languages by a name (i.e., the name the process it corresponds to), a set of type definitions (the types of the data that are elaborated), a set of states (represented as a set of variables that with their values concur to define the global state of the system), an initial state (i.e., a particular assignment to the variables of the state section), a set of input actions (i.e., the input messages/events that the process can accept and manage), a set of output actions (i.e., the output messages/events that the process can generate), a set of transitions of the state machine underlying the process, expressed as assignment of opportune values to the variables of the state section.

The existence of such a language makes it possible not only to perform elaborations on business processes in an abstract fashion, but also to define directly, if necessary, business process in this formalism and let them be elaborated without getting involved in the choice of a fixed language.

In the following, we give a sample of the usage of the IL language. The text below represents the translation into IL of the Hotel business process that was presented in section 4.3.1. Hence it is possible to establish a parallelism between these two representation.
The BPEL4WS variables become IL input and output messages. The *partnerLinks* and *correlationSets* elements of the BPEL4WS language need not be represented within the IL state machine. The *state* of the IL system is the aggregation of the set of variables derived by the input and output messages of the BPEL4WS process with their input parameters, and of a *program counter* that must keeps track of the current state of the system. The initial state is determined by a trivial assignment to the variables. The set of transition of the state machine is constructed with a forward approach starting from the initial state and providing all the evolution steps that the activities defined in the BPEL4WS allow.

```plaintext
PROCESS Hotel;
TYPE
  flight: { AZ500, AZ600 };
  location: { Paris, NewYork };
  hotel: { Palace, Hilton };
  time: { Aug13, Aug15 };
INPUT
  hAck ();
  hNack ();
  hRequest (time, location);
OUTPUT
  hNot_avail ();
  hOffer (hotel);
STATE
  _pc : { end_Is_available, seqHotel3, FAIL, seqHotel2,
    seqHotel1, end_pickHotel1, Prepare_offer,
    Hotel_offer, Receive_request, FAIL_NACK,
    Not_available, SUCC, Prepare_answer, START,
    END, Is_available, pickHotel1 };
  hRequestMsg_location: location;
  hRequestMsg_time: time;
  hOfferMsg_hotel: hotel;
INIT
  _pc := START;
  hRequestMsg_location := UNDEF;
  hRequestMsg_time := UNDEF;
  hOfferMsg_hotel := UNDEF;
TRANS
  _pc = FAIL_NACK -[TAU]-> _pc := end_pickHotel1;
  _pc = end_Is_available -[TAU]-> _pc := END;
  _pc = seqHotel1 -[TAU]-> _pc := Receive_request;
  _pc = seqHotel3 -[TAU]-> _pc := Prepare_answer;
  _pc = seqHotel2 -[TAU]-> _pc := Prepare_offer;
  _pc = Hotel_offer -[OUTPUT hOffer(hOfferMsg_hotel)]
    _pc := pickHotel1;
  _pc = Is_available \
    (!_isAvailable) -[TAU]-> _pc := seqHotel3;
  _pc = START -[TAU]-> _pc := seqHotel1;
  _pc = Not_available -[OUTPUT hNot_available()]-> _pc := FAIL;
  _pc = end_pickHotel1 -[TAU]-> _pc := end_Is_available;
  _pc = Prepare_answer -[TAU]-> _pc := Not_available;
  _pc = Prepare_offer -[TAU]
    _pc := Hotel_offer, hOfferMsg_hotel IN hotel;
  _pc = Is_available \& (_isAvailable) -[TAU]-> _pc := seqHotel2;
  _pc = SUCC -[TAU]-> _pc := end_pickHotel1;
  _pc = pickHotel1 -[INPUT hAck()]-> _pc := SUCC;
  _pc = pickHotel1 -[INPUT hNack()]-> _pc := FAIL_NACK;
  _pc = Receive_request -[INPUT hRequest(hRequestMsg_location, hRequestMsg_time)]
    _pc := Is_available;
  _pc = FAIL -[TAU]-> _pc := end_Is_available;
```

15
4.3 The output languages

4.3.1 The composition output language

Of course, the language used for the description of the services obtained from the composition activity of the ASTRO platform is the same used for the definition of the input processes, that is BPEL4WS.

4.3.2 The monitor output language

Monitoring business process at run-time means both to detect the possible violation of given properties during the execution of such processes and to trace the value of specific numeric variables related to the processes themselves. In the case of the ASTRO platform this amounts to provide a mechanism, embedded within the execution engine, that alerts the user when such violations occur. At a deeper level this means to endow the engine both with an infrastructure for handling the generic management of the monitoring activity and to supply the pieces of code that implements the real monitors. It is well known that most of the engines for the execution of business processes can be extended with JAVA code and this in particular holds for the ActiveBPEL engine [Act] that is used within the ASTRO platform. These are the main reasons for the choice of JAVA as the language for plugging the generated monitors into the monitoring framework of the ActiveBPEL engine. This technological background concur to create an ideal execution environment where the monitors can run. Of course, it is to say that JAVA in itself comprises those qualities that are typical of standard general purpose, imperative and high level language that makes them attractive: high expressive power and easiness of programming. As a matter of fact JAVA has grammar constructs that are sufficient to represent the monitoring state machines that result from the monitor elaboration and it is adapt for being easily generated at run-time.

4.3.3 The verification output language

The output of the verification activity within the ASTRO platform consist of a yes/no answer to the property being verified, possibly enriched by a counterexample trace. In particular when an assertion property is not verified, such a trace demonstrates an execution path of the system, where the analyzed property is violated. When a possibility property is verified, the trace represents a witness of the property satisfaction.

The counterexample trace is a merely a (potentially infinite) sequence of activities performed by the composition participants while executing their own processes. Such a trace corresponds to a State Transition System and, as such, it is equivalent to its IL representation. Furthermore, the tool allows for representation of the counterexample as a Message Sequence Chart diagram that shows an ordered execution of service interactions.
Chapter 5

The wsCHAIN manager tool

The wsCHAIN manager is the tool that controls the execution of the steps that are necessary to complete the Web service tasks within the ASTRO project. We start by giving an overall description of the tool, then we hint at its architecture, and supply some implementation details.

5.1 Overview

The wsCHAIN manager acts as a command executor for launching the tools that perform the basic work inherent to the functionalities of the ASTRO architecture, that are composition, monitoring and verification. Namely, the wsCHAIN manager is a console that offers the user the possibility of controlling every phase of the ASTRO chain, starting from the definition of the requirements until the actual execution of the business processes. In this respect the wsCHAIN manager is the trait d’union between the wsREQUIREMENT tool that helps the analysts in defining the problem to be solved and those tools (business process execution engines, web servers, ...) on which the resulting solutions actually run.

The wsCHAIN manager represent a solution to the problem of easily enabling the selective execution of those chain steps that have been described in the chapter dealing with the ASTRO architecture. In particular, the wsCHAIN manager is designed and implemented for supplying a unique environment for the execution of the synthesis, monitoring and verification of Web services. The wsCHAIN manager is a configurable Eclipse plug-in whose behavior at run-time is determined by a choreography file that the user can select and by options that the user can choose within the program graphical user interface.

The application is designed for being easily extended. The control that the wsCHAIN manager performs on the Web service activities is implemented within the architectures
through a sort of plugins components. It is then possible to add new chain activities or to extend or change the behavior of the existing one by providing new plugin components or extending the existing ones. Provided an opportune implementation of the steps that compose the activities is supplied, the enabling of such new features can take place off line by editing a configuration file.

5.2 Design and implementation

The architecture of the WSCHAINMANAGER is shown in Fig. 5.1. It is composed by three layers. The top layer is the graphical user interface that consist of a window that is common for all the possible chain, activities or complex tasks that are controlled by the WSCHAINMANAGER. The GUI is laid upon the execution engine of the tool that is in charge of triggering the launch of the proper atomic steps that concur to the elaboration chain. The engine, in turn, is laid on a layer consisting of the code that defines the atomic actions (steps) that are executable, and of the code that accounts for their assembly into more complex activities (chains).

![Figure 5.1: The architecture of the WSCHAINMANAGER.](image)

The WSCHAINMANAGER is an open and extensible framework that guarantees the maximum flexibility in enhancing its capabilities. In particular, in the bottom layer of the architecture of the WSCHAINMANAGER it is easy to add new control chains for new Web services activities. At the same time the framework is easily extensible in order to define new elaboration steps to be easily implemented and integrated into on or even more than one more activity.

The single steps or actions correspond to the launch of one of the basic tool of the ASTRO platform (WSTRANSLATOR, WSYNTH, WMON, NuSMV) or to other operations such as the deployment and undeployment of business processes and code on the exe-
cution engines. The inputs and outputs of such operations are determined by a set of assumptions and conventions on the names of the files. These atomic actions concur to define the global activities performed by the WSCHAINMANAGER, following what is prescribed in the configuration file. Such a file states the execution flow of the activities by defining sequences of atomic steps. The choreography file, on the other side, concur to the execution of the activities, and of course of the singles steps, by providing both the the declaration of the objectives of the elaboration and the source elements that are to be elaborated. In other words, the choreography file asserts what is the goal of the given Web service activity (whether it be composition, monitoring or verification), while the configurable part of the tool determines how, that is to say, through which steps, such a goal is obtained.

The WSCHAINMANAGER is implemented as an Eclipse plug-in and it is written in Java. The GUI of the tool consists of a window where the user can interact with a set of controls in order to supply the input, can see the log of the execution and can possibly browse the results of the elaborations. Figures 5.2 and 5.3 shows the window appearing in the case of the Web service composition task and in the case of the Web service verification task.

![Figure 5.2: The WSCHAINMANAGER window for the composition of Web services.](image)

The tool is actionable from the toolbar of the Eclipse GUI. For each of the different
ASTRO activities, that is composition, monitoring and verification, a window is opened whose appearance is the same as in the other cases but the contents is different. Both the area of the window where the steps of the activity are listed and the sections where the possible execution options can be checked, present different elements depending on the kind of elaboration chosen by the user.

Figure 5.3: The wsCHAINMANAGER window for the verification of Web services.

The GUI allows the user to select which steps of the activity to execute and which execution option to activate or deactivate; the sequence of actions is carried on automatically without delays after the execution button is pressed; the tool logs its activity with a verbosity that the user can control.

Figure 5.4: A results window associated to the verification of a series of Web services properties.

In the case of Web service verification when the results of the verification have been computed, the wsCHAINMANAGER lets a browser appear that shows the results. In Fig. 5.4 the window corresponding to the display of such results are displayed. Further info
on the results, that is the traces of the counterexamples, shows up by clicking on the the relevant button as Fig. 5.5 demonstrates.

Figure 5.5: The counterexample associated to the verification of a Web service property.

The tool allows for customization that can be obtained through a property file. This file whose settings are loaded when the Eclipse platform gets started allows to change some properties of the GUI, to set some properties and defaults options about the execution of the ASTRO tools, and to control the execution chains for synthesis, monitoring and verification by defining the possible steps for each chain.
Chapter 6

The wsREQUIREMENT tool

In this chapter we present wsREQUIREMENT, the tool that is in charge of managing the choreography file. As for the wsCHAINMANAGER we start by giving a general description of the tool, then we go on by describing the architecture of the tool, and, finally, we provide some implementation details.

6.1 Overview

The choreography file plays a crucial role in supplying the inputs and in expressing the requirements for the elaboration that is carried on within the ASTRO platform. The goal of the wsREQUIREMENT tool is that of easing the definition of such a choreography file.

The main motivation behind the project of the wsREQUIREMENT tool is the usability of the ASTRO platform and the possibility of enlarging the basis of its potential users. As a matter of fact, the platform, as showed in the architecture and languages chapters, uses languages that are difficult to control. As showed in chapter 4 the languages used for specifying the requirements are derivation of logical languages, very near to the low level representation of the tools that use them and quite far from human comprehension. The choreography file itself, written in XML, is quite far from being an example of human friendly compilation of the requirements. The wsREQUIREMENT takes in charge this problem, and in particular the fact that the drafting of the specifications for the composition, monitoring and verification problem is not easy at all.

The wsREQUIREMENT is a tool that, by means of a user friendly graphical interface, obtains a twofold purpose: on one side it allows the user to avoid the manual filling in of the XML tags of the choreography file that otherwise would be unpractical and error prone and, on the other side, relieve the user from the burden of writing obscure logical formulae for expressing concepts that are conceptually simple.

The main goal of the tool is to let the user the of writing brand new choreography
files and modifying existing ones so that that can be used by the \texttt{wsChainManager}, and possibly by the other tools of the platform. Besides, by using opportune visual masks it allows to easily enter the elements of the domain of interest that represent the requirements of the composition, monitoring and verification tasks as they have been stated in the deliverable D7.1 [IioTD]. For example, in the case of the composition task, it is possible to select which are the BPEL4WS processes (i.e., the list of BPEL and WSDL files) that must be composed and it is possible construct the goal that has to drive the composition, both in its control flow part and in its data flow part. In the case of the data flow portion of the composition requirement a graphical editor allows the user to select particular elements of the domain, namely the messages that are exchanged between processes, as nodes of a diagram that are linked by arches in order to form the data net for the composition requirement.

### 6.2 Design and implementation

The architecture of the \texttt{WSRequirement} is shown in Fig. 6.1. The tool is designed as an Eclipse plugin and thus the base setting of the architecture modules is represented by the Eclipse Plugin Development software layer. Since the tool is essentially a graphical interface towards an XML file, the other main components of the architecture are a version of the general Model-View-Controller design pattern and a module for the parsing of the XML file.

<table>
<thead>
<tr>
<th>GUI (GMF + SWT)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHOR file model</td>
</tr>
<tr>
<td>XML parser</td>
</tr>
<tr>
<td>Eclipse Plug−in Development</td>
</tr>
</tbody>
</table>

Figure 6.1: The architecture of the \texttt{WSRequirement} tool.

The XML parsing module is in charge of performing the exchange (serialization and de-serialization) of the information between the internal model of the file and the file system. The Model View Controller architecture that is on top of the parsing module consists of a model of the choreography file and a graphical component based on the Eclipse Graphical Modeling Framework (GMF) enriched with a set of standards SWT components. The GMF is a framework for the aided and semi-automatic generation of graphical editors starting from a given domain model. Within the \texttt{WSRequirement}, the GMF component takes care of the graphical management of the data flow portion of the composition requirement, supplying a graphical editor for it. Its enrichment by standards SWT elements (the common Java graphic component library) is for the handling of the
other textual element of the choreography file that are managed through the use of input masks.

The **wsREQUIREMENT** is a Java tool implemented as an Eclipse plug-in. It is implemented as a multi editor where each window represents a partial view on the choreography file and is in charge of managing a specific portion of it. Hence the tool, within a typical Eclipse plug-in windows presents a number of tabs for:

- choosing the processes of interest (see Fig. 6.2);
- defining the control flow portion of the composition requirement (see Fig. 6.3);
- drawing the graph that correspond to the data flow portion of the composition requirement (see Fig. 6.4);
- defining the properties to be monitored and for which processes (see Fig. 6.5);
- defining the properties to be verified and for which processes (the tab is identical to the monitoring one);
- directly editing the XML choreography file (see Fig. 6.6);

Figure 6.2: The process definition editor window.

The majority of the partial editors are graphical masks built by using SWT components. Only the editor of the data flow component of the composition requirement, which is a hyper graph editor, consists of a more sophisticated software module that is generated taking advantage of the Eclipse Graphical Modeling Framework (GMF), a generative component and runtime infrastructure for developing graphical editors based on EMF and GEF. Finally, the textual editor of the XML file takes advantage of Eclipse standard development facilities. All these editing components are set within a standard Eclipse Plugin Development project. Such an environment also provides the means for the implementation of a wizard that helps the user in a step by step generation of a choreography file from scratch.
Figure 6.3: The composition control flow editor window.

Figure 6.4: The composition data flow editor window.
Figure 6.5: The monitoring properties editor window.

Figure 6.6: The XML editor window.
Chapter 7

The **WSTRANSLATOR** tool

The **WSTRANSLATOR** is the tool that is in charge of translating the business process that are given in input to the ASTRO tool chain from their original language into a neutral intermediate representation, that is the IL language. In the following we give an overview of the tool and then describe its design and implementation.

7.1 **Overview**

The goal of the **WSTRANSLATOR** is the translation of automata representing business processes across different formalisms. Owing to its open and extendible architecture such a translation can, in principle, take into account different formalisms both as input and as output. This possibility of performing translation back and forth amongst such different, though equivalent, languages is granted by an internal representation that is designed to guarantee the possibility of representing business processes in an abstract fashion by a State Transition System.

Notwithstanding this versatile attitude, the current implementation **WSTRANSLATOR** is mainly aimed at translating business processes from a the BPEL4WS language to the IL language. This two languages are considered the milestones of input and output amongst the other that are possible and are effectively managed by the tool (SMV, SPIN, XML) for the reasons expressed in chapter 4. The transformation from BPEL4WS to IL as we have seen in chapter 3, correspond to the second phase that is common to all the different Web service elaboration chains (for composition, monitoring and verification) that are implemented within the ASTRO architecture. As already hinted the translation is performed by means of a data structure, called **State Transition System or STS**, that represents an automaton derived from the description of how business processes are composed within a choreography. This automaton can be emitted into different output languages and in particular it has a direct correspondence with the IL language that has been devised as an abstract way of dealing with business processes.
The WSTRANSLATOR is a versatile tool not only with respect to the input and output formalisms but also with respect to the different kind of elaboration targets it can manage. As a matter of fact the WSTRANSLATOR is capable of performing those particular elaborations that slightly differs for the different types of Web services elaborations that are comprised in the ASTRO requirements. The translation performed for the composition of Web services driven by a goal is different from the translation that has, as its objective, the monitoring of Web services and the same holds for the verification of properties relevant to the input processes. And even for the same activity the WSTRANSLATOR is capable of performing the elaboration in different flavors as for instance it happens for composition where it can take into account different representation axioms (ground level, knowledge level and datanet) or for verification where different underlying models can be used.

By an opportune use of the several options that the WSTRANSLATOR make available several different behaviors and results can be achieved serving for possibly different purposes within the business process elaboration life-cycle.

### 7.2 Design and implementation

The architecture of the WSTRANSLATOR tool is presented in Fig. 7.1. In the architecture different kind of components are present: libraries, generated code and developed code. The bottom LIBS layer of the architecture consists of a set of basic software components that are used by the WSTRANSLATOR both for critical activities such as XML parsing and for complementary functionalities such as for instance command line management, graphic visualization and logging. The intermediate XML parser layer correspond to a heap of generated code that is used for managing the XML files that are used by the tool. This code is generated from the definition of the schema of the languages defined in chapter 4, in particular of the choreography, the BPEL4WS and the WSDL languages. The actual code implemented for the WSTRANSLATOR tool lies in the upper layers of the
architecture, namely the STS and the Translation components. The STS component is in charge of managing the internal, abstract representation of the State Transition Systems, or automata, that are central to the computation performed by the ASTRO platform. The data structures and the API contained in this layer offer the primitives for constructing, transforming and browsing a generic automaton that is capable of representing business processes. This portion of the architecture of the WSTRANSLATOR is tightly linked to the IL language that was presented in chapter 4. Finally, the translation layer is the one that is in charge of performing the actual translation, basing on the lower level of the architecture and by following a sequence of steps that starts with the acquisition of definition in the input formalisms, goes on with its internal representation by an automaton and ends by emitting the result of the translation into the target language.

The WSTRANSLATOR is a command line tool. It is implemented in JAVA and it is designed for open source distribution.
Chapter 8

The WMON, WSYNTH, and NUSMV tools

The architecture of the ASTRO platform presented in chapter 3 provides a framework where most of the activities relevant to the different requirements of the ASTRO project can be carried on following a common pattern and sharing tools and representation formalisms. Obviously, the specific activities carried for the different targets of composition, monitoring and verification of Web services need to differentiate each other, when approximating the end of the elaboration chain. In this chapter we discuss the specific tools that, through further elaboration steps, produce those final results that are prescribed by the ASTRO requirements. First we deal with the WMON and the WSYNTH tools; then, we discuss the role played by the NUSMV tool within the tools chain.

8.1 The WMON tool

WMON is a tool for the automatic synthesis of Web services run-time monitors. The synthesis is achieved through static analysis of Web services. Given in input a formal representation for one or more asynchronous processes, WMON outputs an external execution monitor, i.e. an executable piece of code whose function is to detect and signal whether the processes behave consistently with their specified nominal protocol and/or with some given invariant properties, by observing only processes’s input and output. WMON input processes are modeled as stateful, non-deterministic and partially observable labeled-transition systems. Partial observability is modeled by using tau-transitions, i.e. elements of the transition relation whose associated action is a special one called tau, that has neither input nor output associated to it. Intuitively, tau-transitions represent internal, unobservable service evolutions.

One of the most important activities performed by WMON is making deterministic those Web services that originally are not. In other words, WMON builds a version of the state machines that represents Web services that is equivalent to the original ones, as for their behavior, though fully observable. This is achieved through a so called power set
construction algorithm that performs a transitive closure with respect to \( \tau \) actions on the input processes (see for example [BTPT06a]). Since such a construction is useful for WSYNTH as well, WMON acts also as a pre-requisite of WSYNTH execution.

Output monitors are modeled as fully observable, state-transition systems, and can be emitted by WMON in various formats. In particular, it is possible to emit a state machine in the WSYNTH input format, or as JAVA piece of source code suited for embedding into the a business process execution engine endowed with monitoring infrastructure (e.g. the ActiveBPEL[Act] engine).

8.2 The WSYNTH tool

WSYNTH is the tool that takes care of performing Web service composition, by emitting an orchestrator service that, interacting with some given component services, satisfies a user requirement. As such, WSYNTH receives as input an appropriate representation, i.e., a fully observable version, of the (IL conversion of) the available component services, and of the composition requirement, in terms of its data-flow and control-flow portions. Based on this, WSYNTH examines all possible interactions that the orchestrator protocol may perform with the other components, and selects one which is satisfactory for the user requirements. As a result, an IL structure is produced that represents the desired executable orchestrator, and that is viable for being emitted in terms of an executable BPEL service. Therefore, WSYNTH realizes a rather general form of program synthesis, a task whose high complexity is well known, and essentially stems from the combinatorial blowup that takes place when considering all possible message-based service interactions. To effectively deal with this even in the case of real scenarios, where participating components and user requirements are rather complex per se, WSYNTH relies on advanced search and representation techniques. In particular, WSYNTH is based on symbolic representation techniques that allow the analysis of large sets of protocol combinations at once; as witnessed in several documents [PTB05, PMBT05, PTBA05], this makes it possible to confront with considerably complex scenarios in practical time. A detailed account of WSYNTH’s working, and of the various search techniques it implements, is presented in the aforementioned papers.

8.3 The NuSMV tool

NuSMV is the component that within the ASTRO tool chain performs the design-time verification of Web services. NuSMV is a tool, that has been developed jointly by ITC-irst and by Carnegie Mellon University, independently from the ASTRO project, for the formal verification of finite state systems. NuSMV allows to check finite state systems against specifications in the CTL and LTL temporal logics. The input language of NuSMV is designed to allow for the description of synchronous finite state systems.
Such system are described by a set of finitely typed variables and a transition relation defined over such variables.

Within the ASTRO tool chain, NuSMV is used for verifying assertion and possibility properties (see also Section 4.1.2) of a group of Web services that interacts within a choreography. To bridge the gap between the definition of the component Web services and the NuSMV language, the intermediate representation of such component services in the IL language is translated into the NuSMV input language by the WSTRANSALATOR tool. The requirement to be verified is supplied by the user already in the LTL or CTL logic language. The tool checks the property against the system represented by the set of service components and delivers a positive or negative answer. In the negative case the tool exhibits a proof of the reason why the property does not match the specification. Such a proof comes in the flavor of an execution trace of the system, that in the case of Web service verification can be represented by a sequence of the messages that are exchanges between the component services.
Chapter 9

History of the Deliverable

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

9.1 1st year

The first year of the project was mostly devoted to establishing the feasibility of adapting state-of-the-art search techniques to pursue automated composition and verification of web services. As such, the architectural issues were just analyzed in a preliminary way, awaiting for the results of such feasibility study.

9.2 2nd Year

At this stage, the ongoing research activity on verification and monitoring, as well as on composition, made it evident the possibility and advantages of designing a structured development process that would cover all phases in the life-cycle of Web services. Parallel to a technology acquisition phase, aiming at determining a service development platform, we started the design of our architecture, and started inquiring on the possibility of integrating it into an open development platform. The most interesting prospect in this sense was identified being Eclipse[Ecl], and we acquired its technology.

9.3 3rd year

In the 3rd year, we settled to the ActiveBPEL[Act] web service platform, which is integrated within the Eclipse development platform, and we realized our architecture. This
also implied completing the design of our task description language, based on an XML format. This way, we finally achieved an integrated platform where all of the activities related to development and test of services can be exploited. This platform and its successive developments has been, since then, the core of our demos at international workshops and conferences such as ICAPS’05, ICAPS’06, and ECAI’06 [TPC+05, TPB+06a, TPB+06b].

9.4 4th Year

In the 4th year, we thoroughly tested our platform over different scenarios, as witnessed in several publications [PTB05, PMBT05, BTPT06b, MPT06]. This led us to refine the capabilities of our platform and to ease its user interaction. For instance, our current instantiation allows for a variety of verbosity levels during the composition and verification phases, and we introduced a way to graphically analyze intermediate code. This activity is ongoing, as well as the refining of the underlying techniques, by considering more and more elaborate testbeds.
Appendix A

Knowledge-level representation: impact on languages

The BPEL4WS language chosen to represent component services in our reference instantiation of the architecture is extremely rich. For this reason, in the reference instantiation of the architecture we discussed so far, we adopted some restrictions that, while still making it possible to represent very relevant scenarios, simplify the task of devising search techniques to deal with the problem. One such notable restriction involved ruling out the use of expressions involving (possibly non-interpreted) functions. This choice is motivated by the fact that introducing function symbols greatly complicates all phases in the composition process, and in particular the combinatorial analysis of data based on ground values, which defines the search space for the WSYNTH engine. On the other side, allowing for expressions with function symbols opens the way to a more immediate and compact representation of certain scenarios, which if handled correctly, may lead to a better performance of composition.

Here, we discuss the impact of the introduction of expressions and functions into the BPEL4WS on the various architecture functions and modules, and describe the way the internal representation is used to support a kind of reasoning that acts at the knowledge level rather than over ground values.

In particular, we will show that the choice taken for the intermediate language provides us with the necessary degree of flexibility to also deal with such a different setting - all that is required is a different mechanism for translation and emission.

In particular, the following elements are affected, to some extent, by the introduction of function-rich expressions into BPELFORWS.

- First of all, of course, the BPEL4WS parsing phase is suitably extended. In BPELFORWS, expressions are presented using the Xpath [XPA] grammar, for which the parser internal to wSTRANSlator is extended, based on the associated XSD schema.
The extension in the BPEL4WS language reflects into an extension to the symbol table data structure built by WSTRANSLATOR, which now must cover the possibility of functional terms. And since functions may be uninterpreted, for the sake of uniquely identifying the type of functional terms, we have to introduce an additional function declaration file, an XML-format file which is also given as input to WSTRANSLATOR. We remark that, for the sake of uniformity, also standard infix operators, such as e.g. arithmetic operators, are treated as functions; however their declaration is not required since a 'standard function declaration file' is added by default.

More specifically, the function declaration file is a list of prototype declarations, each stating the return and argument types for a function. For instance, the following listing that represents the content of a possible function declaration file of a flight reservation service, declares two functions. The first function is named fChoose_flight and, on the basis of the two input parameters that represents the destination location and the leaving time for a flight, returns an identifier of an existing flight that matches those inputs; the second function, given a flight identifier, returns the time of such a flight.

```xml
<tns:functions xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:tns="http://www.astroproject.org/schema/functions/"/>
  <function name="fChoose_flight">
    <retType>{http://astroproject.org/BusinessProcesses/Flight}flight</retType>
    <params>
      <parType>{http://astroproject.org/BusinessProcesses/Flight}location</parType>
      <parType>{http://astroproject.org/BusinessProcesses/Flight}time</parType>
    </params>
  </function>
  <function name="fChoose_time">
    <retType>{http://astroproject.org/BusinessProcesses/Flight}time</retType>
    <params>
      <parType>{http://astroproject.org/BusinessProcesses/Flight}flight</parType>
    </params>
  </function>
</tns:functions>
```

The translation into intermediate language is the portion that requires most of the intervention. This is due to the fact that, once reasoning on variables in terms of explicit ground values is not possible, we need to reason over the knowledge we have on variables and their relationships. Therefore, the finite state machine that defines the search space for the composition algorithm has a very different nature from the one used in the ground case. In particular, as discussed in [PMBT05], the machine represents the evolution of a so-called knowledge base, a collection of variables that represent (a) whether the values of terms are known, (b) whether the equalities amongst terms are known. Each operation between web services corresponds to an evolution of the knowledge base, e.g. if service $W_1$ sends variable $v_1$ to service $W_2$, which applies function $f$ to it and stores it into variable $v_2$, then we get to know that $W_2.v_2 = f(W_1.v_1)$, and we derive every logical consequence coming from
that. The translation, starting from the expression-rich BPELFORWS, must build such a finite state machine, therefore extracting the required knowledge variables and computing every possible evolution for them; as pointed out in [PMBT05], this is far from trivial since a transitive closure operation is required to make sure that the knowledge is consistently maintained. In a nutshell, every communication involves the establishing of new known facts, the deletion of formerly known facts, and the computation of all logical consequences based on a set of axioms (namely those over equality). The translation follows these steps, and is applied as well to the composition requirement, which needs to be converted to speak of knowledge variables; the whole procedure is detailed out in [PMBT05].

- The emission of the orchestrator is also partially affected. While the search takes place in a space that represents the knowledge over domain variables, the actual orchestration must refer to the actual messages and functions used by the components. That is, a link must be maintained between the variables representing knowledge and their actual concrete sources, so that emission of a concretely executable orchestrator is possible in spite of reasoning at an abstract knowledge level. This is easily enforced by adopting certain naming conventions in the translation of variables, which allows a unique backward reconstruction at emission time. The modification of the emit module is therefore rather trivial.

We stress that neither the intermediate language, nor the search mechanisms on it, need any adaptation to deal with the knowledge level representation, confirming the intuition that a general design should be able to cope with different levels of abstractions. As shown in [PMBT05], the whole knowledge level chain has been tested with success, and shown to perform reasonably well in terms of composition effort.
Appendix B

XML-schema for choreography files

```xml
<?xml version = "1.0" encoding = "UTF-8"?> <!--Generated by XML Authority. Conforms to w3c http://www.w3.org/2001/XMLSchema-->
<xsd:schema xmlns:xsd = "http://www.w3.org/2001/XMLSchema">
  <xsd:element name = "domain">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref = "process_def" maxOccurs = "unbounded"/>
        <xsd:element ref = "choreography"/>
        <xsd:element ref = "function_refs" minOccurs = "0"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

  <xsd:element name = "process_def">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref = "bpe1" minOccurs = "0"/>
        <xsd:element ref = "sts" minOccurs = "0" maxOccurs = "unbounded"/>
        <xsd:element ref = "wsdl" minOccurs = "0" maxOccurs = "unbounded"/>
        <xsd:element ref = "function_refs" minOccurs = "0"/>
        <xsd:element ref = "define" minOccurs = "0" maxOccurs = "unbounded"/>
      </xsd:sequence>
      <xsd:attribute name = "id" use = "required" type = "xsd:ID"/>
    </xsd:complexType>
  </xsd:element>

  <xsd:element name = "choreography">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref = "compose" minOccurs = "0"/>
        <xsd:element ref = "verify" minOccurs = "0"/>
        <xsd:element ref = "monitor" minOccurs = "0"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

  <xsd:element name = "bpe1" type = "xsd:string"/>
  <xsd:element name = "sts" type = "xsd:string"/>
  <xsd:element name = "wsdl">
    <xsd:complexType>
      <xsd:simpleContent>
        <xsd:extension base = "xsd:string">
          <xsd:attribute name = "primary" use = "required" type = "xsd:boolean"/>
        </xsd:extension>
      </xsd:simpleContent>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```
<xsd:element name = "define">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element ref = "symbol"/>
            <xsd:element ref = "value"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>

<xsd:element name = "compose">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element ref = "process_ref" maxOccurs = "unbounded"/>
            <xsd:element ref = "goalVarList" minOccurs = "0"/>
            <xsd:element ref = "goal"/>
            <xsd:element ref = "servicename"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>

<xsd:element name = "goalVarList">
    <!--xsd:attribute name = "tns" type = "xsd:string"/>-->
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element ref = "goalVar" maxOccurs = "unbounded"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>

<xsd:element name = "goalVar">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element name = "name" type = "xsd:string"/>
            <xsd:element name = "type" type = "xsd:string"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>

<xsd:element name = "verify">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element ref = "process_ref" maxOccurs = "unbounded"/>
            <xsd:element ref = "property" maxOccurs = "unbounded"/>
            <xsd:element ref = "channel" maxOccurs = "unbounded"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>

<xsd:element name = "monitor">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element ref = "process_ref" maxOccurs = "unbounded"/>
            <xsd:element ref = "property" maxOccurs = "unbounded"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>

<xsd:element name = "symbol" type = "xsd:string"/>
<xsd:element name = "value" type = "xsd:string"/>
<xsd:element name = "process_ref">
    <xsd:complexType>
        <xsd:attribute name = "direction" use = "required">
            <xsd:simpleType>
                <xsd:restriction base = "xsd:NMTOKEN">
                    <!--If 'property' element is missing then monitoring is only about protocol-->"\begin{math}\end{math}
    </xsd:complexType>
</xsd:element>
<xsd:enumeration value = "uses"/>
<xsd:enumeration value = "implements"/>
</xsd:restriction>
</xsd:simpleType>
</xsd:attribute>
<xsd:attribute name = "process_id" use = "required" type = "xsd:IDREF"/>
</xsd:complexType>
</xsd:element>
<xsd:element name = "servicename" type = "xsd:string"/>
<xsd:element name = "property">
<xsd:complexType>
<xsd:sequence>
<xsd:element ref = "property_name"/>
<xsd:element ref = "property_type" minOccurs = "0"/>
<xsd:element ref = "property_spec" minOccurs = "0"/>
<xsd:element ref = "property_description"/>
<xsd:element ref = "process_name"/>
</xsd:sequence>
</xsd:complexType>
</xsd:element>
<xsd:element name = "process_name" type = "xsd:string"/>
<xsd:element name = "property_name" type = "xsd:string"/>
<xsd:element name = "property_description" type = "xsd:string"/>
<xsd:element name = "property_spec" type = "xsd:string"/>
<xsd:element name = "channel">
<xsd:complexType>
<xsd:sequence>
<xsd:element ref = "src_process"/>
<xsd:element ref = "dest_process"/>
<xsd:element ref = "operation"/>
<xsd:element ref = "ch_pair" minOccurs = "0" maxOccurs = "unbounded"/>
</xsd:sequence>
</xsd:complexType>
</xsd:element>
<xsd:element name = "src_process" type = "xsd:string"/>
<xsd:element name = "dest_process" type = "xsd:string"/>
<xsd:element name = "operation" type = "xsd:string"/>
<xsd:element name = "ch_pair">
<xsd:complexType>
<xsd:sequence>
<xsd:element ref = "src_ch_pair"/>
<xsd:element ref = "dest_ch_pair"/>
</xsd:sequence>
</xsd:complexType>
</xsd:element>
<xsd:element name = "property_type">
<xsd:simpleType>
<xsd:restriction base = "xsd:string">
<xsd:enumeration value = "possibility"/>
<xsd:enumeration value = "deadlock"/>
<xsd:enumeration value = "assertion"/>
<xsd:enumeration value = "instance"/>
<xsd:enumeration value = "class"/>
</xsd:restriction>
</xsd:simpleType>
</xsd:element>
<xsd:element name = "src_ch_pair" type = "xsd:string"/>
<xsd:element name = "dest_ch_pair" type = "xsd:string"/>
<xsd:element name = "function_refs">
<xsd:complexType>
<xsd:sequence>
<xsd:element ref = "function_ref" maxOccurs = "unbounded"/>
</xsd:sequence>
</xsd:complexType>
</xsd:element>
</xsd:complexType>
</xsd:element>
<xsd:element name="function_ref" type="xsd:string"/>
\end{math}

\begin{math}
<!-- goal 29/07/2005 -->
\end{math}

\begin{math}
<xsd:element name="goal">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="desc"/>
      <xsd:element ref="complexgoal"/>
      <xsd:element ref="datanet" minOccurs="0"/>
    </xsd:sequence>
    <xsd:attribute name="name" use="optional" type="xsd:string"/>
  </xsd:complexType>
</xsd:element>

<xsd:element name="desc" type="xsd:string"/>

<xsd:element name="complexgoal">
  <xsd:complexType>
    <xsd:choice>
      <xsd:element ref="simplegoal"/>
      <xsd:element ref="oneof"/>
      <xsd:element ref="all"/>
    </xsd:choice>
  </xsd:complexType>
</xsd:element>

<xsd:element name="simplegoal">
  <xsd:complexType>
    <xsd:choice>
      <xsd:element ref="true"/>
      <xsd:element ref="false"/>
      <xsd:element ref="predicate"/>
      <xsd:element ref="not"/>
      <xsd:element ref="imply"/>
      <xsd:element ref="iff"/>
      <xsd:element ref="and"/>
      <xsd:element ref="or"/>
      <xsd:element name="predicatekl" type="xsd:string"/>
    </xsd:choice>
  </xsd:complexType>
</xsd:element>

<xsd:element name="oneof">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="prioritygoal" maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:element name="all">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="complexgoal" maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:element name="predicate">
  <xsd:complexType>
    <xsd:attribute name="process" use="required" type="xsd:string"/>
    <xsd:attribute name="expr" use="required" type="xsd:string"/>
  </xsd:complexType>
</xsd:element>

<xsd:element name="true">
  <xsd:complexType/>
</xsd:element>
<xsd:complexType name="nodesType">
  <xsd:sequence>
    <xsd:element name="node" type="nodeType" minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="nodeType">
  <xsd:attribute name="id" use="required" type="xsd:ID"/>
  <xsd:attribute name="type" use="required">
    <xsd:simpleType>
      <xsd:restriction base="xsd:NMTOKEN">
        <xsd:enumeration value="in"/>
        <xsd:enumeration value="local"/>
        <xsd:enumeration value="out"/>
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:attribute>
</xsd:complexType>

<xsd:complexType name="elementsType">
  <xsd:choice minOccurs="0" maxOccurs="unbounded">
    <xsd:element name="identity" type="identityType"/>
    <xsd:element name="fork" type="forkType"/>
    <xsd:element name="last" type="lastType"/>
    <xsd:element name="oper" type="operType"/>
    <xsd:element name="merge" type="mergeType"/>
    <xsd:element name="clone" type="cloneType"/>
    <xsd:element name="filter" type="filterType"/>
  </xsd:choice>
</xsd:complexType>

<xsd:complexType name="singleInputsType">
  <xsd:sequence>
    <xsd:element name="in" type="inType" minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="multiInputsType">
  <xsd:sequence>
    <xsd:element name="in" type="inType" minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="singleOutputsType">
  <xsd:sequence>
    <xsd:element name="out" type="outType" minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="multiOutputsType">
  <xsd:sequence>
    <xsd:element name="out" type="outType" minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="inType">
  <xsd:attribute name="idref" use="required" type="xsd:IDREF"/>
</xsd:complexType>

<xsd:complexType name="forkType">
  <xsd:sequence>
    <xsd:element name="inputs" type="singleInputsType"/>
    <xsd:element name="outputs" type="multiOutputsType"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="mergeType">
  <xsd:sequence>
    <xsd:element name="inputs" type="multiInputsType"/>
    <xsd:element name="outputs" type="singleOutputsType"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="cloneType">
  <xsd:sequence>
    <xsd:element name="inputs" type="multiInputsType"/>
    <xsd:element name="outputs" type="singleOutputsType"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="filterType">
  <xsd:sequence>
    <xsd:element name="inputs" type="singleInputsType"/>
    <xsd:element name="outputs" type="singleOutputsType"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="identityType">
  <xsd:sequence>
    <xsd:element name="inputs" type="singleInputsType"/>
    <xsd:element name="outputs" type="singleOutputsType"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="lastType">
  <xsd:sequence>
    <xsd:element name="inputs" type="singleInputsType"/>
    <xsd:element name="outputs" type="singleOutputsType"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="operType">
  <xsd:sequence>
    <xsd:element name="inputs" type="multiInputsType"/>
    <xsd:element name="outputs" type="singleOutputsType"/>
  </xsd:sequence>
  <xsd:attribute name="name" use="required" type="xsd:string"/>
</xsd:complexType>

<xsd:complexType name="outType">
  <xsd:attribute name="idref" use="required" type="xsd:IDREF"/>
</xsd:complexType>
Figure B.1: The schema of the choreography file.

Figure B.2: The composition goal portion of the schema of the choreography file.
Bibliography


[IioTD] ITC-irst, University of Trento, and Delta Dator. Platform for the composition of web services: Requirements and architecture.


[XPA] XPATH. Xml path language (xpath) version 1.0 - http://www.w3.org/tr/xpath.