A. D. MCCXXIV

UNIVERSITA' DEGLI STUDI DI NAPOLI FEDERICO II

Dottorato di Ricerca in Ingegneria Informatica ed Automatica

Comunità Europea
Fondo Sociale Europeo

Methodologies, architectures and tools for automated service composition in SOA

GIUSY DI LORENZO

Tesi di Dottorato di Ricerca

XXI Ciclo

Novembre 2008

Il Tutore

Prof. Valeria Vittorini

Il Coordinatore del Dottorato

Prof. Luigi P. Cordella

Dipartimento di Informatica e Sistemistica

Methodologies, architectures and tools for automated service composition in SOA

Giusy Di Lorenzo

A Thesis submitted for the degree of Doctor of Philosophy

Dipartimento di Informatica e Sistemistica

University of Naples Federico II

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Service Oriented Computing

Service oriented computing (SoC) is an emerging cross-disciplinary paradigm for distributed computing that is changing the way software applications are designed, architected, delivered and consumed [69]. Services are autonomous, platform-independent computational elements that can be described, published, discovered, composed, integrated and programmed using standard protocols to build networks of collaborating applications distributed within and across organizational boundaries. In other words, they represent a new way in the utilization o f the web, by supporting rapid, low-cost and easy interoperation of loosely coupled heterogeneous distributed applications. The SOC paradigm envisions to wrap and adapt existing application, such as legacy systems, and expose them as services. Thus, services help to integrate applications that were not written with the intent to be easily integrated with other applications and define architectures and techniques to build new functionalities while integrating existing application functionalities. SOA(Services Oriented Architecture) is the architectural infrastructure built on the services concept and allows to implements the SOC paradigm. In the SOA context, many challenging research efforts such as services modeling, composition and managements, quality of services evaluation and application integration have received relevant interests from both the academic and industry community.

In the last years, also, the emerging Web 2.0 movement is emphasizing the same things of the services computing - which is rapidly emerging as the best practice for building services, reusing IT assets, and providing open access to data and functionality. One of the goals of the Web 2.0 is to make it easy to create, use, describe, share, and reuse resources on the Web. To achieve that, a lot of technologies have flourished around this concept such as blogs and social networks. The capabilities of Web 2.0 are further enhanced by many service providers who expose their applications in two ways; one is to expose application functionalities via Web APIs such as Google Map[1], Amazon.com, and Youtube[2], the other is to expose data feeds such as RSS and ATOM. This opened up new and exciting possibilities for service consumers and

[1] http://maps.google.com/
[2] http://youtube.com

providers as it enabled the notion of using these *services*[3] as "ingredients" that can be mixed-and-matched to create new applications. To achieve this goal, and maybe to anticipate future needs in Web 2.0, a new framework, called *Mashup*, is surfacing. *Mashup* is an application development approach that allows users to aggregate multiple services, each serving its own purpose, to create a service that serves a new purpose.

A combination of principles from both Web 2.0 (user self-service and collective end-user intelligence) and SOA (a composition of reusable building blocks) can facilitate the wide dissemination of many resources. Examples include professional business applications, value-added services (including location-based services), and interoperability services (for example, applications that can be leveraged by trading partners to initiate business-to-business transactions). Enterprise mashups represent one specific use case of this type of architecture that could easily be situated at the interstice of Web 2.0 and SOA. The interconnection of presentation layer focused Web applications to internal SOA implementations could be of significant value for enterprises, as this could extend their services' reach to the Web for further use and composition by their business partners and customers [82]. The interaction and the integration of service computing and Web 2.0 technologies led to the definition of a new architecture *Global SOA*, that refers to as *Internet of Services* [82]. In an *Internet of Services* all people, machines, and goods will have access to it by leveraging the network infrastructure of tomorrow. The Internet will thus offer services for all areas of life and business, such as virtual insurance, online banking and music, and so on.

From the above discussion, it derives that the SOA and its combination with the Web 2.0 paradigm pose many challenging research issues, both from a conceptual and from a technological perspective. In this thesis we concentrate on the former ones.

1.2 Services and Services Composition

In this section we define the basic concepts which are investigated in this thesis, namely, services and the problem of service composition.

1.2.1 Service Definition

In the literature there is no common understanding of what services are, since the term *service* not always is used with the same meaning. In [18] an interesting and detailed discussion is provided on existing definitions, on top of which we base what follows. Current definitions of the term *service* range from very generic and somewhat ambiguous, to very restrictive and too technology-dependent. On one end of the spectrum, there is the generic idea that every application that is characterized by an URL is a service, thus focusing on the fact that a service is an application that can be invoked over the Web. On the other end of the spectrum, there are very specific definitions such as for example,a service is *a standardized way of integrating Web-based applications using the XML, SOAP, WSDL, and UDDI open standards over an Interned protocol backbone*. This definition tightly relates services to today state-of-the-art web standards, such as Web Service Description Language (WSDL [7]), Simple Object Access Protocol (SOAP [6]), Universal Description, Discovery and Integration repository (UDDI [13]), which are, respectively, the description language for service, the protocol supporting interactions among services and the distributed repository where services are published.

However, such standards evolve continuously and may be subject to revisions and extensions, due to new requirements and possible changes in the vision of SOC. Other definitions, which lie in the middle of the spectrum are the following ones, provided by two standardization consortia, respectively, UDDI and

[3] These services can be a data service, such as news, or a process/operation service such as placing an order to Amazon.com.

World Wide Web (W3C [148]). According to the former, services are *self-contained, modular business applications that have open, Internet-oriented, standard-based interfaces*.

It is not our intention to discuss here and provide the right definition of service, rather, we want to observe that all the above definitions agree on the fact that *a service is a distributed application that exports a view of its functionalities*. In general, a service can be described either from an Input/Output perspective only, or also in terms of its preconditions and effects. So the world state pre-required for the service execution is the precondition, and new states generated after the execution is the effect. A typical example is a service for logging into a web site. The input information is the username and password, and the output is a confirmation message. After the execution, the world state changes from $notLoggedIn$ to $loggedIn$.

1.2.2 Research Issues on Service and Service Composition

The commonly accepted and minimal framework for Service Oriented Computing is the Service Oriented Architecture (SOA) [72]. It consists of the following basic roles: (i) the service provider is the owner of a service, i.e., the subject (e.g., an organization) providing services; (ii) the service requestor, also referred to as client is the subject looking for and invoking the service in order to fulfill some goals, e.g., to obtain desired information or to perform some actions; and (iii) the service directory is the subject providing a repository/registry of service descriptions, where providers publish their services and requestors find services. When a provider wants to make available a service, he publishes its interface (input/output parameters, message types, set of operations, etc.) in the directory, by specifying information on himself (name, contacts, URL, etc.), about the service invocation (natural language description, a possible list of authorized clients, on how the service should be invoked, etc.) and so on. A requestor that wants to reach a goal by exploiting service functionalities, finds (i.e., discovers) a suitable service in the directory, and then binds (i.e., connects) to the specific service provider in order to invoke the service, by using the information that comes along with the service. Note that this framework is quite limited, for instance, a service is characterized only in terms of its interface: no support for behavioral description of services is considered. Also, SOA conceives only the case when the client request matches with just one service, while in general it may happen that several services collaborate to its achievement. Despite this, SOA clearly highlights the main research challenges that SOC gives rise to. Research on services SOA and SOC spans over many interesting issues. *Service Description* is concerned with deciding which services are only needed for implementation purposes and which ones are publicly invocable; additionally, it deals with how to describe the latter class of services, in order to associate to each service precise syntax and semantics. *Service Discovery*, selection and invocation considers how customers can find the services that best fulfill their needs, how such services, and, consequently, the providers that offer them, can be selected, how the clients can invoke and execute the services. Other interesting areas are: *Service Advertisement*, which focuses on how providers can advertise their services, so that clients can easily discover them; *Service Integration*, that tackles the problem of how services can be integrated with resources such as files, databases, legacy applications, and so on; *Service Negotiation*, dealing with how the entities involved negotiate their role and activities in providing services. The notions of reusability and extensibility are key to the SOC paradigm: they consider how more complex or more customized services can be built, by starting from other existing services, thus saving time and resources. *Security* and *Privacy* issues are of course important, since a service should be securely delivered only to authorized clients and private information should not be divulged. Last but not least *Quality of Services* (QoS) should be guaranteed. It can be studied according to different directions, i.e., by satisfying reasonable time or resource constraints; by reaching a high-level of

data quality, since data of low quality can seriously compromise results of services. Guaranteeing a high quality of services, in terms of service metering and cost, performance metrics (e.g., response time), security attributes, (transactional) integrity, reliability, scalability, availability, etc. allows clients to trust and completely depend upon services, thus achieving a high degree of dependability, property that Information Systems should have. Note that all the research areas highlighted above are tightly related one with the other.

In this thesis, we focus on another very interesting research topic, *Service Composition*. Service composition addresses the situation when a client request cannot be satisfied by any available service, but a composite service, obtained by combining parts of available component services, might be used. Services used in a composite application may be new service implementation, they me be old applications such as legacy system or existing web application that are adapted and wrapped, or they may be composition of services [72]. The services used in the context of a composite service are called its component services. Actually two main approaches are used to build *composite services*. They are called *orchestration* and *choreography*.

Orchestration and *choreography* describe two aspects of creating business processes by composing stand alone web service. Both the approaches used to create composite web services lead to some component web services interactions, but orchestration refers to a business process which components can interact both with internal and external web services. It creates statefull composite processes combining stand alone web services. Choreography, instead, is related to message sequences among different parties and not to the specific business process the parties execute. Orchestration represents control from one party perspective while choreography describes collaborations (exploited through a message passing paradigm) among parties involved in a business process. Notice that orchestration and choreography can both be used to compose VAS: orchestration is usually used to compose web services belonging to a specific process participant, while choreography can be used to allow interaction of component web services of different parties. Our main focus in this thesis is on services orchestration. The research in orchestration is based on research in workflows, which models business process as sequence of activities and focuses on both data and control flow among them.

As an example, let us consider an organization that wants to offer a *travel planning* service that allows users to build their own travel itinerary. The travel planning service can be obtained by combining several elementary services: *Ticket Booking*, to search for a plane or train ticket and eventually to proceed with the booking and payment; *Hotel Booking*, to search and book for an hotel room; and *Tourist Attraction Searching*, to provide information about the main tourist attractions such as monuments and museums.

To offer this service, a *human* has first to select the individual services and then manually integrate the responses. The service composition proposed by the example requires: 1) discovering the services that expose the wished functionalities; 2) knowing the services interface description, in order to invoke and integrate them into the composed process (in particular, the integration requires the knowledge of the semantics and the structure of the data and operations provide by the service); 3) describing the composed process model, defining the control and data flow (the control flow establishes the order in which component services have to be invoked, the data flow captures the flow of data between component services); 4) implementing the composite service using a process-modeling language like BPEL [8].

Each of these steps has its intrinsic complexity, which, in general, comes from the following sources. First, the number of services available over the Web increased dramatically during the recent years, imposing searches in huge Web service repositories. Second, some applications providing services are implemented by legacy systems or web applications which cannot be modified in order to accommodate new interaction requirements. Third, services can be created and updated on the fly, thus the composition system

needs to detect the updating at runtime and the decision should be made based on the most up to date information. Fourth, services can be developed by different organizations, which use different concept models to describe the services. Finally, the majority of web services composition languages lacks of formal definitions of their construct, leaving the task of (correctly) defining them to the developers of appropriate interpreters [67]. Therefore, building composite web services with an automated or semi-automated tool is needed but at the same time is a critical task.

Although, in recent years a wide research effort has been done to automate the composition of web services more of the problems mentioned above still remain to be clarified [36, 69, 78].

The aim of this thesis is to propose a formal unified composition development process for the automated composition of web services. The process is based on the usage of Domain Ontologies for the description of data and services, and on workflow patterns for the generation of executable processes. The approach produces workflows and executable processes that can be formally verified and validated. Moreover, our approach considers integrating services whose functionalities are provided by legacy or existing web applications.

The research reported in this thesis tackles the following iusses:

a) Turning Web Application into Web Services by Wrapping Technique

b) Automated Web Service Composition

c) Automatic Analysis of the Semantics Correctness of the Data and Control Flow of Orchestration

Turning Web Application into Web Services by Wrapping Technique

The integration of old legacy systems and web application into new thecnologies like SOA requires the maintenance interventions involving reverse engineering, reengineering and migration approaches to be planned and executed. An interesting classification of approaches for integrating legacy systems in SOAs has been presented by Zhang and Yang [95] who distinguish between the class of black -box re-engineering techniques, which integrate systems via adaptors that wrap legacy code and data and allow the application to be invoked as a service, the class of white-box re-engineering techniques, which require to analyse and modify the existing code in order to expose the system as Web services, and the class of grey-box techniques that combine wrapping and white-box approaches for integrating parts of the system with a high business value. In this context, a specific and relevant migration problem consists of turning Web applications into Web services: here, the basic challenge is that of transforming the original (non programmatic) user-oriented interface of the Web application into a programmatic interface that exposes the full functionality and data of the application.

Some relevant research questions that still need to be addressed in this field include:

1. Which criteria can be used to establish which parts of a Web application (e.g., data layer, functional layer, or presentation layer) can be migrated to a SOA?

2. Which are the migration strategies and techniques applicable to turn a Web application into a Web service?

3. For each type of migration strategy, what is the migration process to be adopted?

The thesis contributes to give an answer to these question, by addressing the specific problem of migrating the functionality of traditional Web application to Web service architecture using black box strategies [59]. In particular, the technique is based on wrapping to migrate the functionalities implemented by

an interactive, form-based legacy system towards Web services. A black box technique aims at exposing interactive functionalities of form-based systems as services. The problem of transforming the original user interface of the system into the request/response interface of SOA is solved by a wrapper that is able to autonomously interact with the legacy application by knowing the rules of the dialogue between user and application. These rules are specified by a User Interface (UI) model based on Finite State Automata that is interpretable by an automaton engine, and that can be obtained by UI reverse engineering techniques. This migration approach has been validated by experiments that showed its effectiveness.

Automated Web Service Composition

The problem of automated web service composition can be formulated as following:

Given a description of a requested service, in which way is it possible to compose at run time a set of available basic services which composition satisfies the request?

To achieve this goals an *intelligent* composition engine should be able:

a) to perform the automatic and dynamic selection of a proper set of basic services whose combination provides the required capabilities;

b) to generate the process model that describes how to implement the requested service;

c) to translate the process model into an executable definition of the services composition, in case the selection is successful;

d) to verify the correctness of the definition of the composition;

e) to validate the composite web service against the initial description.

Each of these steps has its intrinsic complexity. Services descriptions, relations among the involved data and operations, composition definitions should be unambiguously computer-interpretable to enable the automation of web services discovery, selection, matching, integration, and then the verification and validation of web services compositions [63].

To solve the automated composition problem, we propose an unifying composition development process, which copes with several aspects related to service discovery, integration, verification and validation of the composite service. Our approach uses domain ontologies to describe operations, data and services, and aims at producing an executable process expressed by a standard workflow language that can be formally verified and validated [61].

The composition development process is realized by means of the following phases:

1) *Logical Composition*. This phase provides a functional composition of service operations to create a new functionality that is currently not available.

2) *Transformation Feasibility*. This phase verifies the feasibility and the correctness of the transformation of the new functionality into a executable process expressed by a standard workflow language.

3) *Physical Composition*. This phase aims at producing an executable process, which is formally verified and validate.

An operational semantics is the formal basis of all the composition process phases: it is used to define the relations between operations and data, to express the flow of the operations which realizes the

composition goal, to identify the composition pattern described by the composition flow and automate its translation into an executable process expressed using a composition language, to formalize the flow constructs of a composition language (so that the resulting executable process can be automatically verified), and to support the validation of the composition.

Automatic Analysis of the Semantics Correctness of the Data and Control Flow of Orchestration

As mentioned above, *orchestration* requires that the composite service is completely specified, in terms of both the specification of how various component services are linked, and the internal process flow of the composite one. Several standard for the orchestration are emerged, lending to the definition of different languages and architecture able to define and enact composite service [87]

Nevertheless,the standard languages used to create business processes from composite web services lack of formal definition of their semantics and tools to support the analysis of a business process. As a consequence, ambiguities and errors are possible in the definition of the control flow of a composition process that are not detected before the enactment of the process itself [87].

In the thesis, a pattern analysis methodology is proposed , which is founded on the concept of constructs operational semantics of a given language [67]. In brief this approach allows a syntax driven definition of a given language semantics. Formally a language behavior is described through transitional rules that specify how language expressions have to be evaluated and how to execute commands. The proposed methodology has two advantages:

1. It forces a formal description of semantics of a given language, defining once for all which kind of steps are needed to execute language constructs and not leaving its definition to workflow engine vendors.

2. It allows a fully automatic way to investigate if a given pattern (or even a given workflow process) can be executed by a given language.

The methodology is applied to the BPEL language, since it is now begin standardize by OASIS. Moreover for the BPEL tool for the formal verification of the composition process has been developed [62]. That tool aiming at the *formal verification of BPEL executable processes*, by defining a syntax-driven operational semantics for BPEL. The goal of this work is to provide the developer of BPEL processes with a light, practical means to debug BPEL compositions against the most common semantic errors.

1.3 Outline of the Dissertation

The thesis is structured as follows. In Chapter 2 we discuss the state of the art and several solutions for the automated web service composition. In particular, the analysis is done comparing the approaches through seven dimensions which cover several aspects related to service discovery, integration, verification and validation of the composite service. A first study regarding the combination of Web 2.0 and SOA approaches is presented. In particular, the objective of this study is to analyze the richnesses and weaknesses of the Mashup tools.Chapter 3 describes a methodology for addressing the problem of migrating the functionalities of traditional Web applications to Web services using black box strategies. A toolkit for the migration is also presented and its application described. In Chapter 4 the problem of automatic composition is formalized and the several steps of the proposed approach for solving it are presented. A unified composition

development process to the automated composition of web services is presented. Such process is based on Domain Ontologies and workflow patterns for the generation of executable processes. Finally, the problem of the analysis of the semantics correctness of the data and control flow of orchestration is tackled. In Chapter 5, we present the prototype framework that implements the proposed approach and a case study. Finally, in Chapter 6 we summarize our work and discuss research directions.

Chapter 2

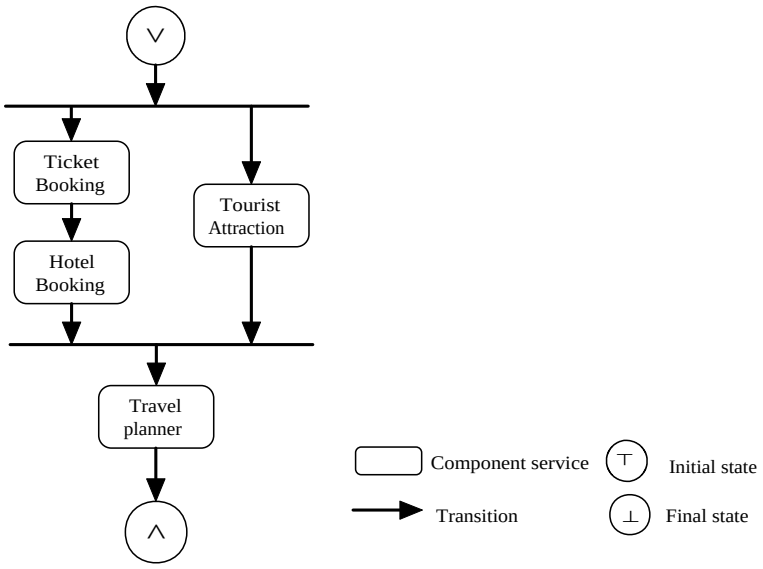
Data and Service Integration: an Overview

This chapter is dedicated to an overview of the rising field of the composition of Web services and integration of Data services available on the web.

2.1 Service Composition

The Web has become the means for organizations to deliver goods and services and for customers to search and retrieve services that match their needs. Web services are self-contained, internet-enabled applications capable not only of performing business activities on their own, but also possessing the ability to engage other web services in order to complete higher-order business transactions. The platform neutral nature of web services creates the opportunity for building *composite services* by combining existing elementary services, possibly offered by different enterprises. As mentioned in the Section 1.2.2, two main approaches are used to build *composite services*. They are called *orchestration* and *choreography*. Mainly, *Orchestrated* composite services allow a centralized management, while *Choreography* defines a more collaborative approach to the process management. In the following we will mainly analyze *Orchestrated* composite services. The research in orchestration is based on research in workflows, which models business process as sequence of activities and focuses on both data and control flow among them. As an example, we can consider an organization that wants to offer a *travel planning* service that allows users to build their own itinerary in a given city. The travel plan service can be obtained by combining several elementary services: *Ticket Booking*, to search for a plane or train ticket and eventually to proceed with the booking and payment; *Hotel Booking*, to search and book for an hotel room; and *Tourist Attraction Searching*, to provide information about the main tourist attractions such as monuments and museums. In Figure 2.1, the control flow of the *travel planning* service is depicted. When the process start, both the *Ticket Booking* and *Tourist Attraction Searching* services are executed in parallel. If no plane or train ticket satisfying the user request is found the process ends, otherwise the *Hotel Booking* service is invoked. When both the *Tourist Attraction Searching* and *Hotel Booking* services end, the *Travel Planning* service will send the user a notification

message containing the travel plan information.



To offer this service, a human has first to select the individual services and then manually integrate the responses. The service composition proposed by the example requires: 1) discovering the services that expose the wished functionalities; 2) knowing the services interface description, in order to invoke and integrate them into the composed process (in particular, the integration requires the knowledge of the semantics and the structure of the data and operations provide by the service); 3) describing the composed process model, defining the control and data flow (the control flow establishes the order in which component services have to be invoked, the data flow captures the flow of data between component services); 4) implementing the composite service using a process-modeling language like BPEL [8].

In a nutshell, the composition problem can be formalized as following: *Given a description of a requested service and the descriptions of several available basic services, the ultimate goal is to be able*:

a) to perform the automatic and dynamic selection of a proper set of basic services whose combination provides the required capabilities;

b) to generate the process model that describes how to implement the requested service;

c) to translate the process model into an executable definition of the services composition, in case the selection is successful;

d) to verify the correctness of the definition of the composition;

e) to validate the composite web service against the initial description.

Each of these steps has its intrinsic complexity, which, in general, comes from the following sources. First, the number of services available over the Web increased dramatically during the recent years, imposing searches in huge Web service repositories. Second, some applications providing services are implemented by legacy systems or web applications which cannot be modified in order to accommodate new interaction requirements. Third, services can be created and updated on the fly, thus the composition system needs to detect the updating at runtime and the decision should be made based on the most up to date information. Fourth, services can be developed by different organizations, which use different concept models

to describe the services. Finally, the majority of web services composition languages lacks of formal definitions of their construct, leaving the task of (correctly) defining them to the developers of appropriate interpreters [67]. Therefore, building composite web services with an automated or semi-automated tool is needed but at the same time is a critical task.

In recent years a wide research effort has been done to automate the composition of web services. Several solutions have been proposed for the automation of the web services composition. Automation means that the method can automatically generate a process that satisfies given composition requirements and that can be deployed and executed on a workflow engine and published as a Web service. Alternatively, automation can mean that, given an abstract process model, the method can locate the correct services.

In this chapter we want to compare and analyze currently available approaches for the web service composition:

- **Synthy**, proposed by **IBM, India Research Laboratory** [16];
- **Meteor-S**, proposed by **John Miller and his group** [17, 91, 92];
- **Composer**, proposed by **Jim Hendler and his group** [85, 86];
- **ASTRO**, proposed by **Traverso and his grouup** [64, 74];
- **Self-Serv**, proposed by **Benatallah and his group** [24];
- **eFlow**, proposed by **HP, Palo Alto Software Technology Laboratory** [30, 31].

In the following, we describe the dimensions used to analyze and compare the approaches. The scope is to analyze how the several solutions for the web service composition cope with several aspects related to service discovery, integration, verification and validation of the composite service.

Discovery of Candidate Services

This dimension deals with the process of discovering the candidate services for a composition process. The discovery of services is based on functional requirements (represented via the service input-output parameters) and non-functional (QoS, etc.) requirements. To enable the discovery of a desired service functionality, we need a language to describe the available services.

In the SOA context services are defined by standards languages for service description and discovery, e.g. Web Service Description Language(WSDL) [7] and Integration Registry(UDDI) [13] [73]. However, these language are not expressive enough to allow automating the discovery process. In fact, the data and the operations inside the WSDL documents are not unambiguously computer-interpretable since they are not semantically annotated.

For example, let us suppose that we want to discover an on-line translator service. The automation of the discovery is difficult using just the WSDL description of service since the service description would designate strings as input and output, rather than the necessary concepts for identifying them. For example, some strings could be the name of original/destination languages, others could be the input words. Standards such as OWL-S(formerly DAML-S) [5] ontology (Service Profile and Service Grounding) or ASWDL [4] can be used to provide those services with a semantic annotation.

Another element to consider in the discovery of services, is that in a business to business context most applications which provide services are implemented by legacy systems or web applications. In such a scenario, it is of great interest the use of functionalities of legacy or existing web applications, which however do not provide web service interfaces [58, 73]. Legacy systems and web applications do not

provide a web service standard interface, making it difficult to both discovery and integrate them with other services. To allow this integration, maintenance interventions involving reverse engineering, re-engineering and migration approaches are needed.

The discovery process includes the following three indicators: 1) **What** are the information needed to select the candidate service; 2)**When** are the candidate services discovered and integrated; 3) **Which** kind of service can be integrated (web service e/o web applications). For the **What** issue, service selection can be based on the description of *functional* and/or *no-functional* requirements. Besides for the **When** issue, the discovery may be done either at *design time* when the human selects the candidate services, or at *planning time* e.g. during the automatic generation of the plan(workflow), or at *execution time* e.g during the execution of the of the executable process.

Composition Technique

This dimension focuses on the evaluation of the possible approaches to generate a model of the composite service and then an executable composed process, which can be deployed and executed.

As mentioned above, a composite service is similar to a workflow. Therefore, the definition model of a composite service specifies both the order in which services should be invoked (control flow) and the flow of data between those services (data flow). In the following we will use workflow model and composition model as synonymous.

The *control flow* is designed evaluating the requested service specifications and the functional requirements of available services. For example, in an automated composition approach, the functional requirements are expressed in terms of preconditions and effects. The precondition is the world state pre-required for the service execution, and the effect is the new states generated after the execution. The control flow is generally generated by applying planning techniques [78].

The design of the *data flow* is needed for the generation of an executable process. The data mediation is a critical process since we need to convert a message *M1* into a message *M2*, which may have structural and semantics heterogeneities[1].

Semantics heterogeneities in message schema means that terms with the same name may refer to different concepts or vice-versa. For example, in Figure 2.1 (a), the output of service S1(Out_{S1}) and the input of service S2(In_{S2}) have both the Location entity, but they have different attributes. In fact, Out_{S1} includes Address, City and Zip-Code and In_{S2} includes latitude and longitude information.

In Figure 2.1 (b), another example of data mediation problem is depicted. In this case, the output of service S1(Out_{S1}) and the input of service S2(In_{S2}) have both semantically similar entities and attributes but the number of attributes is different. Therefore, the right mapping between the parameters requires additional information about the context.

Once the the workflow model has been generated, the execution of the composition model can be done either by translating it in an executable process expressed using standard composition languages, or by developing an ad-hoc framework that allows for its management and execution.

Two main composition techniques can be identified: the **Operational Based Technique** and the **Rule Based Technique**.

The **Operational Based Technique** aims at generating of a state-based model. The model describes the order of execution of services and each state handles the invocation of the services. Generally a repository of state-based models is provided. The generated model is not translated in any executable process expressed by standard composition languages such as BPEL. In fact, an ad-hoc orchestration engine is

[1] Web services standard are XML based, therefore there is no syntactic heterogeneity problem.

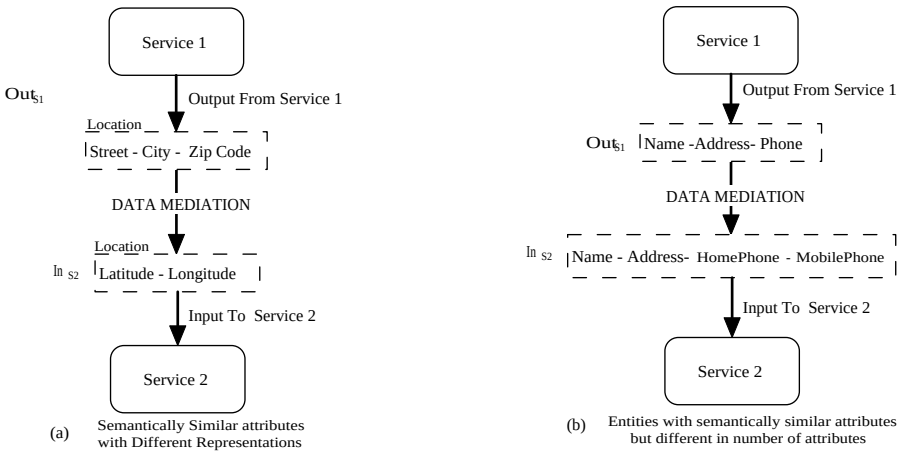


Figure 2.1: Data Mediation

needed to execute the composite process. An example of operational based technique approach is *Self-Serv* [24]. In *Self-Serv* the business logic of a composite service operation is expressed as a state chart [83] that encodes a flow of component service operations. A basic state in a composite service operator 's state chart can invoke a service container operation. A container is a service that aggregates several other substitutable services which provide a common capability (the same set of operations) either directly or through a mapping. At run time, the container selects the candidate service based on attributes defined in a policy (e.g. reliability or execution time). The interaction between the services are done using set conversations defined for a set of predefined service templates. The composite service is executed on a Peer-to-Peer Orchestration engine.

The **Rule-Based technique** aims at automatically generating the workflow model and then translating it in an executable process. In the Rule-Based Approach we can identify the following phases: (1) *Logical Composition*, (2) *Physical Composition*.

(1) Logical Composition aims at defying the workflow model from the description of candidate services and requested service. The definition is done by reasoning on the service description and by applying a set of composition rules. In the majority of the approaches, the control flow is obtained by planning technique and different approaches are adopted to solve the data mediation problem. In this phase the data flow between a pair of services is generally defined evaluating semantics correspondences between schema of the input/output parameters.

(2) Physical Composition aims at defying an executable process expressed by a standard composition language that can be deployed and executed on a workflow engine. The control flow is implemented using the constructs provided by the composition language. The implementation of the data flow is usually done using ad-hoc service wrappers which implement the physical representations of the semantic data match, establishing the schema matching and rules for transforming elements of one schema to another one. Moreover, the physical realization of the data flow depends on the constructs made available by the composition language.

As an example, in *Synthy* [16] a **Rule-Based technique** is used to generate the executable composition process. In particular, for the *Logical Composition* the control flow is generated by matching on functional service requirements and by planning technique. Then, the data flow is generated by reasoning the context of the input/output parameters of services. The logical phase does not operate on the services instances but

only on their interfaces. In fact the instances of the services are selected during the *Physical Composition* phase. Moreover in this last phase, an executable BPEL process is generated by constructing the control and data flow.

Service Composition Approach

To generate the composite service from a set of available services components, different strategies might be provided.

Manual, when the composition process model and the executable process are manually specified, one by one, by the application designer. Actually, several graphical tools like *ActiveBpel Designer* [15] are available to facilitate the definition of the executable process.

Semi-Automatic, when the system can provide facilities either to define the workflow model or to generate the executable process. In particular during the *Wokflow Model Generation*, the system could provide facilities to automatically select the candidate services or a list of them. Generally, the system exploits some metadata (e.g. input/output parameters and type) to propose some possible services that functionally match with a given request. However, the user needs to confirm these propositions and design the control flow. An example of semi-automatic approach is proposed in [86]. Each time a user has to select a web service, all possible services that match with the selected service are presented to the user. The choice of the service is made taking into account both functional and no-functional attributes. Besides, during the *Executable Process Generations* phase, facilities can be made available to the user to generate or adjust the control and data flow. For example, *Synthy* gives facilities to the developer to manually handle the messages conversations by resolving: input-output type matching and transformation, mismatch in invocation protocols that are used (synchronous versus asynchronous), ordering parameters etc. and editing this data in the generated BPEL process.

Automatic, when the system, once the service request requirements are known, automatically 1) selects the candidate services based generally on functional and non-functional requirements, 2) generates the control flow, 3) resolves the data mediation problem and eventually, 4) translates the obtained workflow model into an executable process using a composition language. Generally the last step is not often possible since it depends on the expressiveness of the languages used both to model the workflow model and the executable process (see the subsection related to Expressiveness below). An example of automatic tool is *Meteor-s* [91, 92]. With this tool, the service selection is based on functional (represented via input-output parameter) and non-functional (e.g. QoS) requirements. Planning technique is used to generate the control flow, while the data flow is generated by reasoning with the context of input/output parameters of the services. Finally the workflow model is automatically translated in an executable BPEL process.

Formalization

A formalization (formal semantics) is important for unambiguously interpreting and analyzing composite specifications. This dimension evaluates the formalization used in the composition process.

Moreover, if more than one phase are considered in the composition approach, we also have to consider whether a formal approach is used to guarantee that feasibility and correctness of the transformation between phases. In fact, the transformation from the workflow model to the executable process should not alter the semantics of the original workflow model and as such it should be equivalence preserving.

Note that the feasibility of the transformation may not always be guaranteed, since many workflow languages have been proposed without a formal description [87, 89]. This is not a desirable situation, since the use of an informal language can easily lead to ambiguous specifications.

For this dimension, we will evaluate if a formalization is used during the logical and physical phases separately, and on the whole composition process.

Expressiveness

The *Expressiveness* is an object criterion. Given a certain set of modeling constructs, it is often possible to show that certain processes can, or cannot be modeled [54]. The expressiveness depends on the language used to describe both the workflow model and the executable process.

In literature, the expressiveness of different workflow modeling languages is evaluated for the control flow prospective, since that provides an essential insight into workflow languages's effectiveness [88].

From the control flow prospective, a workfow pattern approach was introduced by V.M.P. van der Aalst in [88, 89] to analyze languages' expressiveness. As described in [80], a pattern is "an abstraction from a concrete form which keeps recurring in specific nonarbitrary contexts". In the case of workflow patterns, they represent the solution to the problem of composing web services in order to build a VAS through the use of workflow models and languages. They typically describe certain business scenarios in a very specific context.

The evaluation of the expressiveness in the description of the business process depends on the technique used for the composition. In fact, into an **Operational Based Technique** the language used to describe the state-based model defines the expressive power in the description of the business process. For example, if the workflow process is modeled through an *Automaton*, it is not possible to model the choice and synchronization pattern.

On the other hand, in a **Rule Based Technique**, the overall expressiveness depends on the ones of the languages used in the two phases to describe the workflow model and the executable process. In this approach, the transformation from the workflow model to the executable process should not alter the semantics of the original workflow model and as such it should be equivalence preserving. Similarly, when accessing the the expressive power of a given workflow language the issue of equivalence is crucial. If you would like to prove that for a certain workflow a corresponding workflow in another language exists, that will also depend on the notion of equivalence chosen.

Observe that the feasibility of the transformation may not always be guaranteed, since many workflow languages have been proposed without a formal description [87, 89]. This is not a desirable situation, in fact the use of an informal languages can easily lead to ambiguous specifications. It means that, such specifications cannot be formally evaluated or analyzed with respect to their expressive power.

The example in Figure 2.1 shows the scenario in which at logic level a petri net model is adopted to define the workflow, and at physical level the graph is translated to a BPEL workflow. It is easy to recognize that the overall expressiveness is limited by the expressiveness of BPEL. In fact, if we consider the multi-merge pattern in the petri net model and want to translate it to BPEL we are limited by the expressiveness of the last language that does not allow implementing it (BPEL does not allow activating two threads following the same path without creating new instances of the process [89]).

For this dimension, the expressiveness is evaluated on the capability to express the following workflow patterns. 1) Sequence, 2) Parallel Split, 3) Synchronization, 3) Exclusive Choice, 4) Simple Merge, 5) Multi Choice, 6) Multi Merge, 7)Discriminator, and 8) Loop [2]. We selected these patterns since they are the only ones implemented into the considered approaches. Moreover, we will evaluate if the transformation feasibility is analyzed in the rule-based approach.

[2] A description of these patterns is presented in the Appendix A.

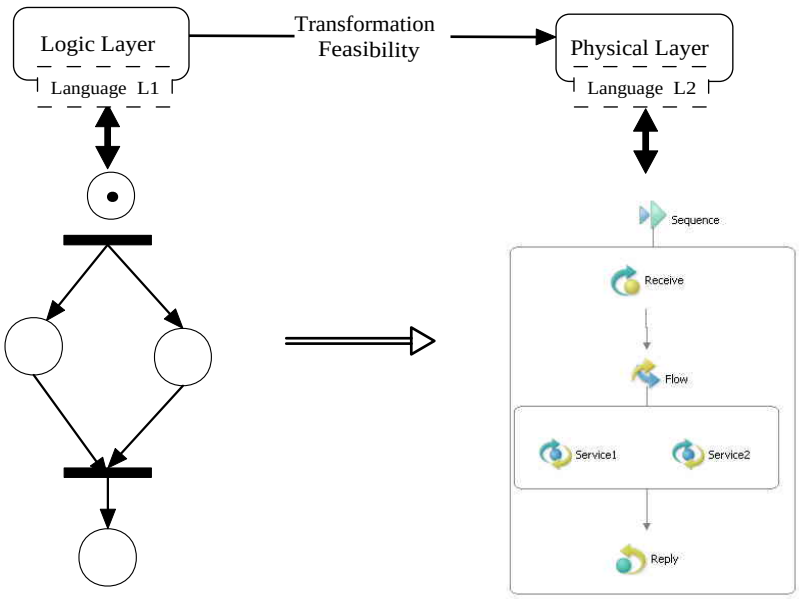


Figure 2.2: Expressiveness

Reusing/ Adaptation

This dimension considers how more complex or more customized services can be built starting from other existing composite services, thus saving time and resources. In this comparison, we consider the following reuse policies:

- **Generated Executable Process**, when the process is reused as a stand alone web service;
- **Workflow Model Adaptation**, when the user can customize the workflow model generated by the first phase, before generating the executable process.

Coding effort

Composite e-services have to cope with a highly dynamic business environment in terms of services and of service providers. In addition, the increased competition forces companies to provide customized services to better satisfy the needs of every individual customer. Ideally, a service process should be able to transparently adapt to changes in the environment and to the need of different customers with minimal or no user intervention. In addition, it should be possible to automatically create the composite service definition and dynamically modify it.

This dimension focuses on the coding effort that a user (a programmer) needs to perform in order to implement all, or some, of the aspects of service composition. We can distinguish the following:

- **Considerable effort**: in this case, the whole composite process needs to be manually implemented.
- **Medium effort**: in this case, only some functionalities need to be coded. For example,the control flow is automatically generated by the tool but a graphical interface is offered to the user to express the data flow.

- **Small effort**: this is the most suitable case in the context of automated web service composition. In this case, small or no programming effort is needed to perform operations or to build the composed service. All the aspects regarding service composition such as service discovery, integration, verification and validation of the composite service are automatically done.

2.1.1 Analyzed Approaches

In the next section we will compare and analyze the approaches listed above, based on the presented dimensions.

Synthy

Synthy is a system for end to end composition of web services. It differentiate between services *types* and *instances*. The *types* define the capability of the services in terms of functional and non-functional requirements, which are described by OWL-S service profile model. The *instances* are described using service grounding. Different instances may be associated to the same type.

The technique used for the composition is *rules-ased*. During the logical composition, given an OWL-S description of the user request, the composition and a set of services descriptions, a BPEL abstract model is built through planning. At physical level, the plan is translated in an abstract BPEL process. Once generated the executable process, the developer may manually handle the messages conversation by resolving: input-output type matching and transformation, mismatch in invocation protocols that are used (synchronous versus asynchronous), ordering parameters etc. and editing this data in the generated BPEL process. The service selection is done at planning time. In particular, during the logical composition a matching is done on functional requirements. Differently, during the physical phase the selection of the best web service instance is done in terms of non-functional parameters.

Since a formal approach (planning technique) is used to generate the model during the logical phase, the generated workflow model is formally correct. On the other hand, the authors do not consider both the problems of evaluating the feasibility of the transformation and the informal definition of BPEL. The expressiveness of the approach is limited by the planner that generates the plan, which can have only sequences, choices and concurrences among actions.

Synthy could be considered a semi-automatic approch, since the user has to resolve manually the messages conversation.

For the reusing, the created service is itself available as a component for further use.

The evaluation of the coding effort is medium since the developer have to know the BPEL language syntax to restructure the executable process.

Meteor-s

In **Meteor-s** [17] an automatic approach for web services composition is presented, which addresses the problem of process and data heterogeneities by using a planner and a data mediator.

The authors suppose that all web services are already semantically annotated and described using SAWSDL[3] languages [4]. Moreover, SAWSDL is extended by adding precondition and effects for an operation in order to improve the discovery and composition process.

The framework uses METEOR-S web service discovery infrastructure for the semantic discovery [17]. This allows for selecting the candidate services based on data and functional semantics.

[3] SAWSDL is a standard for specifying semantics annotation for web services into the WSDL.

The composition used for the composition is *rules-based*. The requirements of the requested service are described in a Semantic Template model, which describes the Input/output data and the functional and non-functional specifications. Once the request requirements and the candidate services are known, *GraphPlan* [81] algorithm is used to generate the control flow. The authors extend that algorithm either to take into account the input/output messages of actions during the planning and to generate plan with loops. The data mediation and heterogeneity problem is handled by assignment activities in BPEL or by introducing data mediator modules, which may be embedded in the middelware or an externalized Web service to handle different structures and/or semantics. The workflow model once generated is automatically translated in BPEL.

From the formalism point of view, *METEOR-S* like Synthy, uses a formal approach (planning technique) to generate the workflow model in during the logical phase. As a consequence, the workflow model is formally correct. On the other hand, in *METEOR-S* the authors do not consider both the problem of evaluating the feasibility of the transformation and the informal definition of BPEL. The expressiveness of the approach is limited by the planner that generates plan, which can have only sequence, And-Split and loops.

METEOR-S is an automatic approach, since the obtained executable process can be deployed and executed without any *human* intervention. Therefore, the coding effort is small.

For the reusing, like *Synthy*, the created service becomes available as a component.

Composer

In [85], the authors present a semi-automatic method for web service composition. Web services parameters and functionalities are specified in DALM-S. Each time a user selects a web service, all possible web services that match with the selected one are presented to him/her. The choice of the candidate services is based on their functionalities specified into the service *profile* and non-functional requirements. A match is defined between a pair of services, whose output type is a OWL class or a subclass of an input parameter of the other service. If more than one match is found, the system filters the services based on non-functional attributes that are specified by the user as constraints. the selection of the candidate services is done at *design time*. The composition technique, can be considered *rules-based*. A graphical interface is offered to the user to define the workflow model during the logical phase. Each composition generated by the user can itself be realized as a DAML-S composite process (physical phase), thus allowing it to be advertised, discovered and composed with other services. In conclusion, a semi-automatic approach is proposed. No formalization is used to model the composition process, therefore the composite process cannot be interpreted and analyzed. From the expressiveness point of view, the generated composite flow can have sequence and And-Split workflow patterns.

The current composer implementation allows executing the composite service by invoking each individual service and passing the data between the services according to the flow constructed by the user.

That approach has a small coding effort, since a graphical tool is provided to the user to define the control flow and also the discovery process and the physical phase are automatically managed by the tool.

ASTRO

In [64], the authors present the ASTRO approach for addressing the problem of automatic composition of an executable process that satisfies given composition requirements by communicating with a set of existing Web services, and that can be published itself as a Web service providing new higher level functionalities.

In ASTRO, the candidate services are selected at design time. ASTRO implements a *rules-based* technique. The logical phase aims at producing an initial version of the executable process starting from initial composition requirements. In particular, the composition algorithm takes as input a set of partially specified services, modeled ad non-deterministic finite state machines, and the client goal expressed in EaGLE. The phase then returns a plan that specifies how to coordinate the concurrent services in order to realize the client's goal [74]. The planning problem is modeled as a model checking problem on the message specification of partners. During the physical phase, the obtained plan is then translated in the BPEL language and executed on a BPEL engine [75]. ASTRO provides an automatic composition approach, and a graphical interface for the adaptability of the process is offered to the uses. For the adaptability, after the generation of the BPEL process, the developer, on the basis of the automated composition outcomes, can refine both the composition requirements and the customer interface and automatically re-compose. Since, the component services are modeled with a non-deterministic finite state machine and the workflow model is generated through a model checking formalism, the correctness of the workflow model can be formally evaluated. Moreover, in [53] an unified framework for the analysis and verification of Web service compositions provided as BPEL specifications is presented. The framework is based on the special form of composition, namely extended parallel product, that allows one to analyze whether the given system is valid under particular communication models. Whenever the model is valid, the actual verification of the system can be performed and different properties of interest can be checked. Therefore, the formal correctness of the BPEL executable process can be evaluated. Besides, it is not clear if the authors have considered the problem of transformation feasibility.

Finally, as the other approaches, the expressiveness is defined by the expressive power of BPEL.

SELF-SERV

Self- Serve is a middleware infrastructure for the composition of web services [24]. That framework is designed so that the composition process is created at design time, while the involved web services are selected and invoked on the fly at run time (discovery at execution time).

Therefore, a **operational based** composition technique is adopted. In **Self-Serv**, web services are declarative composite and executed in a dynamic peer-to-peer environment. Three types of services are defined: elementary, composite service, and service community. The elementary and composite services should provide a programmatic interface based on SOAP and Web Service Description Language. Besides, service community can be seen as container of alternative services, which provide a common capability (the same set of operation). Service composition is based on state-charts, gluing together an operation's input and output parameters and produced events. Each operation's state chart can invoke a service container operation instead of an elementary or composed service. At run time, the container is responsible for selecting the service that will execute the operation based on non-functional requirements. The utilization of a state chart allows both to describe the control flow of composite process and to analyze the formal semantics of the composition process. The data flow is manged though mapping between data items. The expressiveness of the process is defined by the expressive power of the state chart model. The approach is semi-automatic since the workflow models are defined at designed time and stored into a model repository. Therefore a considerable coding effort is required to the user. In fact, if a user wants a new service not available in the repository, he/she has to manually design the state chart model. Service execution is monitored by software components called coordinators, which initiate, control, and monitor the state of the composite service they are associated with.

eFlow

eFlow is a platform for specifying, enacting and monitoring adaptive and dynamic composite services [30, 31]. It provides features that support service specification and management, including a language for the composition(CSDL), exception handling, ACID transactions and security management. The composition technique is *operational based*. In fact, the composite service process model is designed in a static way but is dynamically updated. In particular, the process is modeled by a graph similar to HTML activity diagrams [83], which defines the order of execution of the component services. The graph may include service, decision and event nodes. Service nodes define the context in which the invocations are performed (for instance the specification of the service to be invoked). The decision nodes specify the alternative and rules controlling the execution flow, and finally the event nodes enable the service process to send and receive messages. In order to manage and even take advances of the frequent changes in the environment, the composite process needs to be adaptive. E-flow implements several strategies to achieve this goal with minimal or no manual intervention. These include 1) *dynamic service discovery* that allows to select at *execution time* the appropriate service, evaluating some service selection rules; 2) *Dynamic conversation selection*, which allows the user to select the conversation with a service node at run time (adaptation on the fly of the workflow process); 3) *multiservice nodes* for invoking parallel instances of the same process and 4) *generic nodes* for the dynamic selection of several service nodes.

2.1.2 Discussion

A summary of the different approaches is provided in Table 5.4, which specifies, for each dimension, whether the approaches addresses the issues presented previously.

Table 2.1: Summary of the considered dimensions for the web service composition methods analysis. (+) means the dimension (i.e. functionality) is provided, (-) means the dimension is not provided. These marks do not vehicle any ”positive” or ”negative” information about the approaches except the presence or the absence of the considered dimension.

		Self-Serv	eFlow	Synthy	Meteor-s	Composer	Astro
Service Discovery							
Which	Web Service	+	+	+	+	+	+
	Legacy/Web Application	+	-	-	-	-	
What	Functional Requirements	-	-	+	+	+	+
	Non-Functional Requirements	+	+	+	-	+	-
When	Design Time	-	-	-	-	+	-
	Planning Time	-	-	+	+	-	+
	Execution Time	+	+	-	-	-	-
Composition Technique							
Composition Technique	Operational Based	+	+	-	-	-	-
	Rule Based	-	-	+	+	+	+
Composition Approach	Manual	-	-	-	-	-	-
	Semi-Automatic	+	+	+	-	+	-
	Automatic	-	-	-	+	-	+
Formalization	Logic Phase	-	-	+	+	-	+
	Physical Phase	-	-	-	-	-	+
	Whole Process	+	+	-	-	-	-
Expressiveness	Sequence	+	+	+	+	+	+
	Parallel Split	+	+	+	+	+	+
	Synchronization	-	+	-	-	-	-
	Exclusive Choice	+	+	+	-	-	+/-
	Simple Merge	+	+	+	+	-	-
	Multi Choice						
	Multi Merge						
	Discriminator						
	Loop	-	-	-	+	-	-
	Transformation Feasibility	-	-	-	-	-	+/-
Composition Framework Characteristics							
Coding Effort	Considerable Effort	+	+	-	-	-	-
	Medium effort	-	-	+	-	-	-
	Small Effort	-	-	-	+	+	+
Reusing	Generated Executable Process	-	-	+	+	+	+
	Workflow Model Adaptaion	-	+	-	-	-	+

2.2 Data Integration Analysis

One of the goals of the Web 2.0 is to make it easy to create, use, describe, share, and reuse resources on the Web. To achieve that, a lot of technologies have flourished around this concept such as blogs and social networks. The capabilities of Web 2.0 are further enhanced by many service providers who expose their applications in two ways; one is to expose application functionalities via Web APIs such as Google Map[4], Amazon.com, and Youtube[5], the other is to expose data feeds such as RSS and ATOM. This opened up new and exciting possibilities for service consumers and providers as it enabled the notion of using these *services*[6] as "ingredients" that can be mixed-and-matched to create new applications.

To achieve this goal, and maybe to anticipate future needs in Web 2.0, a new framework, called *Mashup*, is surfacing. *Mashup* is an application development approach that allows users to aggregate multiple services, each serving its own purpose, to create a service that serves a new purpose. Unlike Web services composition where the focus is on the composition of business services, the Mashup framework goes further in that it allows more functionalities and can compose heterogeneous services such as business services, data services, etc. Applications built using the Mashup technique are referred to as *Mashups* or *Mashup applications*, which are built on the idea of reusing and combining existing services, i.e. existing search engines and query services, data, etc.

The recent proliferation of Mashup applications demonstrates that there is high level of interests in the Mashup framework [19,39,48,49,90]. It also shows that the needs for integrating this rich data and service sources are rapidly increasing. Although the Mashup approach opens new and broader opportunities for data/service consumers, the development process still requires the users to know, not only understand how to write code using programming languages (e.g., Java Script, XML/HTML, Web services), but also how to use the different Web APIs[7] from all services. In order to solve this problem, there is increasing effort put into developing tools which are designed to support users with little programming knowledge in Mashup applications development.

The objective of this study is to analyze the richnesses and weaknesses of the Mashup tools. Thus, we identify the behaviors and characteristics of general Mashup applications and analyze the tools with respect to the key identified aspects. We believe that this kind of study is important to drive future contributions in this emerging area where a lot of research and application fields, such as databases, user machine interaction, can meet.

Focusing on the data integration part of the Mashups, we organize the remainder of this paper as follows: Section 2 introduces the different levels of Mashups. After presenting, and illustrating using examples, Section 3 introduces the dimensions of analysis, and describes different mashup tools according to the considered dimensions[8]. We finish by a discussion and a conclusion in Section 4 summarizing our study and highlighting some future directions of the work.

2.2.1 Mashups Description and Modeling

The Web is certainly the location where we can find the most of heterogeneous components. This heterogeneity can be seen on data, processes, and even visual interfaces. Conceptually, a Mashup application is a

[4] http://maps.google.com/

[5] http://youtube.com

[6] These services can be a data service, such as news, or a process/operation service such as placing an order to Amazon.com.

[7] A lot of APIs are available on the Web: http://www.programmableweb.com/apis/directory/

[8] We have selected these tools not because they are the best or the worst tools, but we have tried to consider as much representative tools as possible.

Web application that combines informations and services from multiple sources on the Web. Generally, web applications are developed using the *Model-View-Controller* pattern (MVC) [76], which allows to separate core business model functionalities from the presentation and control logic that uses these functionalities. In the MVC pattern, the *model* represents the data on which the application operates and the business rules used to manipulate the data. The Model is independent of the view and controller, it passively supplies its services and data to the other layers of the application.

The *view* represents the output of the application. It specifies how the data, accessed through the model, are presented to the user. Also, it has to maintain its presentation when the model changes. The *controller* as for it represents the interface between the model and the view. In fact, it translates interaction with the view into actions to be performed on the model.

A Mashup application includes all the three components of the MVC pattern. In fact, according to Maximilien et al. [11], the three major components of a Mashup application are (1) data level, (2) process level, and (3) presentation level. They are depicted in Figure 2.3. Moreover, each data source needs to be first analyzed and modeled in order to perform the required actions of retrieval and preprocessing. For the completeness of the study, we first identify all different levels of concerns in a Mashup application. We will then focus our analysis specifically on the data level.

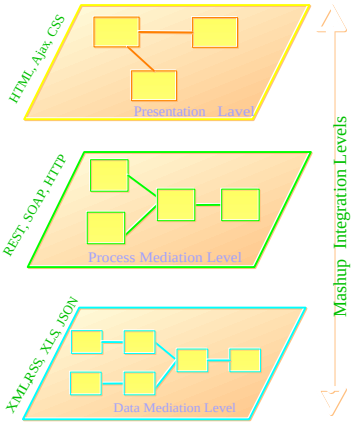


Figure 2.3: Mashup Application Level

1. *Data Level*: this level concerns the data mediation and integration mainly. Challenges at this level involve accessing and integrating data residing in multiple and heterogeneous sources such as web data and enterprise data [45]. Generally, these resources are accessible through REST [41] or SOAP[9] web services, HTTP protocol and XML RPC. Regarding the data mediation, the basic problem to be dealt with, when integrating data from heterogeneous sources, come from structural and semantics diversities of the schema to be merged [45] [22]. Finally, data sources can be either structured for which a well defined data model is available (e.g., XML-based document, RSS/ATOM feed), or unstructured (e.g., audio, email text, office documents). In the latter case, the unstructured data needs to be pre-processed in order to extract meaning and create structured data. So, this level consists of all possible data manipulations (conversion, filtering, format transformation, combination, etc.) needed to integrate different data sources. Each manipulation could be done by analyzing both syntax and semantics requirements.

[9] Simple Object Access Protocol, http://www.w3.org/TR/soap/

2. *Process Level*: the integration at the application level has been fully studied specially in the workflow and service oriented composition areas [37] [3]. The *Process Level* defines the choreography between the involved applications. The integration is done at the application layer and the composed process is developed by combining functions, generally exposed by the services through APIs. In the Service Oriented Architecture (SOA) area, the composition focuses on the behavioral aggregation of services and the interaction is considered between the resources only [32]. In contrast, since the Mashup applications do not only focus on the data integration but also the connection to different remote data services, for instance REST resources or functions available through Java or Java Script methods, the interaction with the clients browsers needs to be handled. So, the models and languages from the SOA approach must be adapted in order to model and describe interactive and asynchronous processes. Currently, languages like *Bite* [32] or *Swashup* [11] have been proposed to describe the interaction and the composition model for the Mashup applications.

3. *Presentation Level*: every application needs an interface to interact with users, and Mashup application is not an exception. Presentation Level (or User Interface) in Mashup application is used to elicit user information as well as to display intermittent and final process information to the user. The technologies used to display the result to the user can be as simple as an HTML page, or a more complex web page developed with Ajax, Java Script, etc. The languages used to implement the integration of UI components and the front-ends visualization support server-side or client-side Mashup [1] [2]. In a server-side Mashup, the integration of data and services is made on the server. The server acts as a proxy between the mashup application and the other services involved in the application. On the other hand, a client-side Mashup integrates data and services on the client. For example, in a client-side Mashup, Ajax application will do the required composition and parse it into client's web browser.

   Today, in the Mashup application, the integration at Presentation Level is developed manually. That is, a developer needs to combine the User Interface of the wished components using either server-side or client-side technologies. This area is an emerging area and a lot of efforts are done in this direction [33] [93]. From the Mashup point of view, there is still a lot of work to be done.

After this brief description of the different levels in the Mashup, we introduce in the next section the dimensions used in our analysis. These dimensions concern mainly the data level, the focus of this analysis.

2.2.2 Analysis Dimension

In this section we describe the dimensions used in our analysis. To understand why some automatic support is needed to create a Mashup application, we give the following example. Let us suppose that a user wants to implement a *News Mashup* that lets her select news on a news service like CNN International and display both the list of the news and a map that highlights the location of the results; she typically needs to do a lot of programming which invokes fetching and integrating heterogeneous data. In fact the user needs to know not only how to write the code but also (1) to understand the available API services in order to invoke them and fetch the data output; (2) to implement screen scraping techniques for the services that do not provide APIs, and (3) to know the data structure of the input and output of each service in order to implement data mediation solution.

The Mashup tools provide facilities to help the user to solve some of the above mentioned problems. The analysis provided in this section aims at studying how the tools[10] address the data mediation problems

[10]The list of the considered tools can be found in Table 5.4

discussed in the previous section: we will be asking questions like which operators do they provide for data transformation and for creating the data flow? or, which are the types of data supported by the available operators?

Therefore, the objective of this section is not to make a comparative, qualitative or quantitative, study of the considered tools but only to analyze how they manage and deal with the different described issues at the data integration level.

Data Formats and Access

In a Mashup application, a user can integrate several format of data as: web feed formats, used to publish frequently updated content such as blog entries, news and so on; table based formats, used to describe any of various data models (most commonly a table) as a plain text file, e.g. csv, xls; markup based formats as HTML and XML; multimedia content as video, images and audio. These types of data can be available to the user from different data sources. The most common data sources can be traditional database systems, local files that are available in the owner's file system, web pages, web services and web applications. For the web data to facilitate the data retrieval, providers often expose their content through web APIs. APIs can be seen as useful, in the same time, as a means for data and application mediation. Here we consider the role of APIs from the data integration point of view in the sense that they offer specific types and formats of data. It should be noted that an API can offer several formats of data, e.g. csv, xml, etc.

Murugesan [68] defines an API as an interface provided by an application that lets users interact with or respond to data or service requests from other programs, applications, or web sites. Thus, APIs facilitate the integration between several applications by allowing data retrieval and data exchange between applications. APIs help to access and consume resources without focusing on their internal organization; simple and known examples of APIs include Open DataBase Connectivity (ODBC) and Java DataBase Connectivity (JDBC). On the Web, providers like Microsoft, Google, eBay, and Yahoo allow to retrieve content from their web sites by providing web APIs that are generally accessible through standard protocols such as REST/SOAP web services, AJAX (Asynchronous Javascript + XML) or XML Remote Procedure Call.

APIs can also be used to access resources which are not URL addressable such as private or enterprise data [51]. However, some common data sources do not expose their contents through APIs. So, other techniques as screen scraping are needed to extract information.

Internal Data Model

As stated before, the objective of a Mashup application is to combine different resources, data in our case, to produce a new application. These resources come generally from different sources, are in different formats, and vehicle different semantics. To support this, each Mashup tool uses an internal data model. An internal data model is a single global schema that represents a unified view of the data [22]. A Mashup tool's internal data model can be either *graph-based* or *Object-based*.

- *Graph-based model*: by graph we refer to the model based on XML and those consumed as they are (i.e. XML). This can include pure XML, RDF, RSS, etc. Most of the Mashup applications use a graph-based model as an internal data model. This is certainly motivated by the fact that most of today's data, mainly on the Web such as RSS feeds, are available in this format and also, most of the Mashup tools are available via the Web. That is, all the data that are used by the Mashup tools, in this category, transform the input data into an XML representation before processing it. For example, *Damia* translates the data into tuples of sequences of XML data [19].

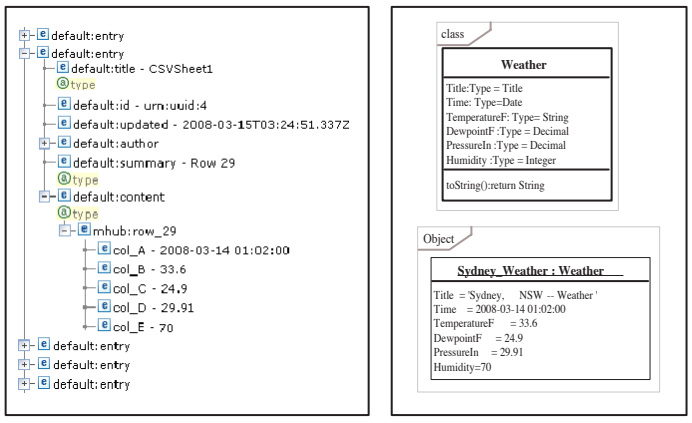
- *Object-based model*: in this case, the internal data is in the form of objects (in the classical sense of the object-oriented programming). An object is an instance of a class which defines the features of an element, including the element's characteristics (its attributes, fields or properties) and the element's behaviors (methods). It should be noted that in this case, there is no explicit transformation, performed by the tool, like in the case of the graph-based model, but the programmer needs to define the structure of the object according to her data.

To illustrate the differences in the internal data models, let us consider the example of Figure 2.4 which shows an extract of a spreadsheet (or csv) file containing the information of the national parks in the world[11].

title	link	description	pubDate
Grand Canyon National Park	www.grand.canyon.national -park.com /	Grand Canyon National Park...	19/03/2008
Big Bend National Park	http:// www.big.bend.national -park.com /	Big Bend National Park...	19/03/2008
Gulf Islands National Seashore	http:// www.hikercentral.com /parks/ guis /	Gulf Islands National Seashore.....	19/03/2008

Figure 2.4: National Parks Data Source

The illustrated data in Figure 2.4 is given as an input to two different tools, namely *Damia* and *Popfly* and the obtained result is shown Figure 2.5. Figure 2.5(a)shows the translation operated by *Damia* on the input data. That is, each row in the *csv* file is transformed to an XML representation contained in the element $< damia : entry >$. Each entry is composed of some elements containing general information regarding the data source such as file name (i.e. $< default : id >$), last updating (i.e. $< default : update >$) and by the $< default : content >$ element in which the national parks information is stored. Figure 2.5(b) as for it shows how the data could be represented using an object based notation is the case of *Popfly*.



(a) DAMIA - Weather Internal Data Model (b) Popfly - Weather Internal Data Model

Figure 2.5: Representation of the National Park *csv* file in DAMIA and Popfly

Data Mapping

To instantiate an internal data model from an external data source, the Mashup tools must provide strategies to specify the correspondences between their internal data model and the desired data sources. This is

[11] Some elements are omitted for readability matters

achieved by way of data mapping. Data mapping is then the process needed to identify the correspondences between the elements of the source data model and the internal data model [77]. Generally speaking, a data mapping can be: (i) *manual*, where all the correspondences between the internal data model and the source data model are manually specified, one by one, by the application designer. In this case, the tool should then provide some facilities for the user to design the transformation. (ii) *semi-automatic*, where the system exploits some metadata (e.g., fields names and types) to propose some possible mapping configurations. However, the user needs to confirm these propositions and, usually, correct some of them. At this stage, only *Yahoo Pipe* supports the semi-automatic mapping, offering some hints for the user about possible mapping. (iii) *automatic*, where all the correspondences between the two data models are automatically generated, without user intervention [77]. This is a challenging issue in the data integration area. Since the Mashup area is in its "early stage", this type of mapping is not supported by any Mashup tool. It should be noted that the mapping process can necessitate an intermediary step, i.e. a wrapping step, in order to transform the source format to the internal format, e.g. from csv to XML. It is also interesting to point out that the mapping in the currently available Mashup tools is only done at schema level, while no semantic information is being considered so far.

Data Flow Operators

Data flow operators allow to perform operations either on the structure of the data (similar to the data definition language/operators in the relational model), or on the data (content) itself (similar to the data manipulation language/ operator in the relational model). Here we consider the operators and the expression languages provided by the tools for processing and integrating data.

More concretely, data flow operators allow: (i)to restructure the schema of the incoming data, e.g. adding new elements, adding new attributes to elements; (ii) to perform elaboration on a data set such as extracting a particular piece of information, combining specific elements that meet a given condition, change the value of some elements; (iii) to build a new data set from other data sets such as merging, joining or aggregating data (similar to the concept of *views* in databases).

The implementation of the data flow operators depends strongly on the main objective of the tool, i.e. integration or visualization. Some operators, e.g. *Union*, are implemented in different tools, e.g. *Damia* and *MashMaker*, but the attached interpretation is also different, e.g. a materialized union of two data sets in *Damia* and a virtual union of two Web pages in *MashMaker* (virtual in the sense that the schemas associated to the main page is not altered with the new page). The main data integration oriented operators, implemented in several tools are the following: *Union, Join, Filter,* and *Sort*. We give hereafter a general description of these operators. A more detailed discussion of all the operators of all the considered tools is given in [60].

Data Refresh

In some cases, e.g. stock market, data are generally generated and updated continuously. A lot of strategic decisions, especially in enterprises, are generally taken according to the last status/values of the data. It is then important that a system propagates the updates of the data sources to the concerned user(s). There are generally two strategies dealing with the status of the data in the source, depending on the objective of the user: *pull strategy* and *push strategy* [25]. In the *pull strategy*: this strategy is based on frequent and repetitive requests from the client, based on a pulling frequency. This pulling frequency is generally chosen to be lower than the average update frequency of the data in the source. The freshness of the data depends mainly on the pulling frequency, i.e. higher is the pulling frequency, fresher will be the data and

Table 2.2: Main and common operators

Operator	Description
Union	Combines two data sets in one. The resulting set contains all the data from the participating sets.
Join	Combines different data sets according to a condition.
Filter	Selects a specific subset (entities and attributes) from an original subset.
Sort	Presents the selected data in a specific order.

vice-versa. One of the main disadvantages of a high refresh frequency is that unnecessary requests may be generated to the server. *push strategy*: in this case, the client does not send requests but needs to register to the server. The registration is necessary to specify/identify the data of interest. Consequently, the server broadcasts data to the client when a change occurs on the server side. The main disadvantage of this model is that the client can be busy performing other tasks when the information is sent which implies a delay in it's processing.

Another important parameter to point on here is the way the tool manages the pull interval. We can define two possible strategies to handle this issue: a *global strategy* and a *local strategy*. For the *Global strategy*, the pull interval is set for the whole application. This supposes that the data sources have the same updating interval. That is, the data sources are requested at the same time interval, corresponding to the one of the Mashup tool. As a result, the user keeps better the trace of some sources (the ones having low refresh interval compared to the defined one) than the others (the ones having high refresh interval compared to the defined one). In the *Local strategy*: in the local strategy, for each data source is affected it's own refresh interval. This pull interval is supposed to correspond to the one of the data source refresh itself. As a result, a better trace is kept of each data source. From the tools point of view, only *Damia* allows to define the pull interval and to hand it with a *Local strategy*. In fact, to set the pull interval, each source component has the *Refresh Interval* parameter. After the time has exceeded, the data from the specified URL is reloaded.

Mashup's Output

We consider the output as a dimension in this study since a user can be interested in exporting her Mashup (the data flow) result in another format in order to reuse it or to process it with another particular application (e.g. spreadsheet) for further processing instead of visualizing it. That is, we can distinguish two main output categories: *Human oriented output* and *processing oriented output*. In the *Human oriented output*, the output is targeted for human interpretation, e.g. a visualization on a map, on an HTML page, etc. That is, for this category, the output can be considered as a "final product" of the whole process. For the *processing oriented output*, the output is mainly targeted for machines processing. This is interesting in the case where the considered data needs to be further processed for, e.g. knowledge extraction. It should be noted that this category can, at some stage, include the first category, e.g. an RSS output can be at the same time visualized on an HTML page and also can be used by other applications for other processing tasks.

The provided output depends on the main objective of the tools. In fact, tools like *Popfly*, *GME*, and *MashMaker*, that are much more about data visualization, provide rich and dynamic visualizations of

the mashup applications. Besides, *Damia* and *Apatar* aim to aggregate and manipulate data that can be reused from other applications. The output is exported into RSS, Atom, or XML feed by adding header information and content specific to the feed, and converting the tuples of sequences to the specified output feed type. Only, *Pipes* and *Exhibit* make available both the output's categories.

Extensibility

Extensibility defines the ability of the tool to support additional, generally user defined, functionalities. There can be two possible ways to define and use these functionalities. A functionality can be either (i) embedded inside the tool, i.e. the corresponding code of that functionality is added to the tool using a specific programming language, or (ii) external, i.e. invoking the corresponding service containing such function. Extensibility depends mainly on the architecture and the spirit of the tool. In some cases, the extension can be done by embedding the code of the desired functionality in the tool (e.g. *Popfly*); in other cases, services are invoked like REST services, SOAP, etc. (e.g. *Pipes*). In addition, this feature is managed differently by the different tools. In fact, in one case, the added function/service is shared with the whole community that uses the tool (e.g. *Popfly*). In the other case, the extension is visible only for the specific user (e.g. *Pipes*).

Sharing

Mashups are based on the emerging technologies of the Web 2.0 in which people can create, annotate, and share information in an easy way. Enabling security and privacy for information sharing in this huge network is certainly a big challenge. This task is made more difficult especially since the targeted public with the Web 2.0 is, or supposed to be, a general public and not experts in computing or security. This dimension defines the modality that the tool offers to enable resources sharing by guaranteeing privacy and security in the created Mashup applications. This is a challenging area in the current Mashup and a lot of work remains to be done. Also, this dimension includes the following three indicators: 1) **What** is shared in the Mashup?, 2) **How** is this shared? and 3)**Who** are the users with whom this (the shared resource(s)) is shared with? For the **What**, the shared resource can be *total*, *partial*, or *nothing*. The shared resource can be given different wrights such as read only (user can read all entries but cannot write any entry), read/write (user can read and write all entries in the data), no access (user cannot read or write any entries). The **Who** as for it can be **All people**, **Group**, or **particular User**. It should be noted that for each member, different sharing policies (what and how) can be specified and applied.

For example, *GME* and *Yahoo Pipes* allow implementing sharing policies. In *GME*, the sharing policy can be: (1) total, i.e. read access to source code, data and output. (2) Partial, i.e. read access to source code. (3) Nothing, where the Mashup is not shared. When a Mashup is shared in GME, for the data used to build the application, the designer can decide to share it with a group or with all users by specifying Read/Write policies. In *Yahoo Pipes*, if a private element is used (Private string or Private text input) the code of the shared Mashup is available, but it is not possible to visualize the intermediate outputs, but the Mashup output is available.

2.2.3 Analyzed Tools

To understand why some automatic support is needed to create a Mashup application, we give the following example. Let us suppose that a user wants to implement a *News Mashup* that lets her select news on a news service like CNN International and display both the list of the news and a map that highlights the locations

associated with the news; she typically needs to do a lot of programming which involves fetching and integrating heterogeneous data. In fact the user needs to know not only how to write the code but also (1) to understand the available API services in order to invoke them and fetch the data output; (2) to implement screen scraping techniques for the services that do not provide APIs, and (3) to know the data structure of the input and output of each service in order to implement data mediation solution.

The Mashup tools provide facilities to help the user to solve some of the above mentioned problems. The analysis provided in this section aims at studying how the tools[12] address the data mediation problems discussed in the previous section, according to the different discussed dimensions. We will be asking questions such as, how the tools handle data? what kind of processes are performed on the data? what is the result of Mashups? which operators do they provide for data transformation and for creating the data flow? or, which are the types of data supported by the available operators? etc.

Currently a lot of Mashup tools are available, we have selected only the following tools since our objective is not to analyze all the tools but to give a view on the current state of these tools.

2.2.4 Damia

IBM provides a tool, *Damia [19],* to assemble data feeds from Internet and enterprise data sources. This tool is dedicated for data feeds aggregation and transformation in enterprise Mashups. Additional tools or technologies like *QEDWiki*[13] and feed readers, that consume Atom and RSS, can be used as presentation layer for the data feed provided by *Damia*.

Damia supports *REST* and *SOAP* web services and allows to fetch local Excel, CSV and XML files. It should be noted that these files must be first uploaded to the server, making them addressable, and can be then invoked through REST protocol. In addition if used in combination with Mashup Hub, *Damia* allows to assemble feeds obtained as results of query of data stored in relational databases like *Microsoft Access*[14] and *DB2*[15].

Damia provides two main widgets, for data access: URL and Catalog widgets. *URL* widget is used to extract the repeating elements from a feed, and the *Catalog* widget is used to fetch feeds from the Mashup Hub Catalog[16]. For processing data feeds, *Damia* engine translates all data sources into tuples of sequences of XML, which constitutes it's internal data model. That is, if the data source is not in the same formalism like the internal data model, e.g. MS Excel, and since the source data are all stored in the 'content' field of the internal data model without taking in account the schema of the data source, a special container is created to receive that data. In this case, the mapping is manually performed between the internal data model if needed to perform more operations.

To consume and produce data, several operators are made available by *Damia*. We can distinguish between two categories of operators:

1. *Data elaboration and presentation operators*: these operators, shown in the Table 2.3, are used to perform modifications on the data or their structure,.

2. *Building operators*: these operators, shown in the Table 2.4, are used to produce new data starting from a data source.

[12] The list of the considered tools can be found in Table 5.4
[13] http://services.alphaworks.ibm.com/qedwiki/
[14] http://office.microsoft.com/access
[15] http://www.ibm.com/db2
[16] http://services.alphaworks.ibm.com/Mashuphub/

Table 2.3: Data Elaboration and Presentation Operators offered by DAMIA

Operator	Description
Transform	used for restructuring the schema of an incoming feed by adding/removing elements, adding/removing attributes to elements, or manipulating values of the elements. The transformation is accomplished by creating an output structure that is used to create a new feed.
Sort	used to sort feeds based on their values, in an ascending or descending order. Also, multiple sort keys can be used to perform the sort.
Group	used to gather entries with similar elements into a single entry based on a grouping expression. The grouping expression evaluates to a text value.

Table 2.4: Building Operators offered by DAMIA

Operator	Description
Merge	the operator is used to combine two source feeds based on an expression that is applied to the feeds. The expression compares an item value from the first feed with an item value from the second feed. All of the entries that satisfy the condition of the expression are merged, or joined, resulting in a new feed.
Union	the Union operator is used to combine two or more feeds into one feed. The entries from the first feed are added to the new feed, then the entries from the second feed.
Filter	used to extract specific items from a feed that meet the filter conditions.
Augment	allowed to combine the data from two incoming feeds into a single output feed of data. One must link an expression from the upper feed to a variable that he/she defines in the lower feed.

Damia caches all the data declared as data sources in a Mashup on its own server. A *pull strategy* is implemented to update the data on the *Damia* server and the pull interval is handled with a *Local strategy*. To set the pull interval, each source component has the *Refresh Interval* parameter for defining the time that the data from the specified URL is cached. After the time has exceeded, the data from the specified URL is reloaded. By doing this, *Damia* offers the possibility to consume it's output using other task specific tools, techniques, etc. (e.g. analysis tools).

As mentioned above, *Damia* aims to aggregate and manipulate data that can be reused from other applications. The output is exported using the *Publish* operator which transforms the output of the data flow into RSS, Atom, or XML feed by adding header information and content specific to the feed, and converting the tuples of sequences to the specified output feed type. Also, *Damia* is an extensible tool in that the user can either embed new functionalities inside the tool or invoke external services. The new operators can be written in PHP language and can be plugged into the engine or can be made available as web services(SOAP or REST).

From a sharing policy management point of view, the tool offers the possibility of sharing the whole Mashup, i.e. total sharing, the output of the Mashup, i.e. partial sharing, or no thing. In the first case, another user might access all the information used by the Mashup creator. The Mashup is completely shared meaning that other users have access to source code, data and output. In the second case, the only

shared thing is the final output. The resources are shared following one policy, which is the read only policy. The user can't specify another policy. The Mashup can be shared either with all people or no one. There is no possibility to share the application with a specific user or with a specific group of users.

2.2.5 Yahoo pipes

Yahoo Pipes allows to build mashup applications by aggregating and manipulating data feeds from web feeds, web pages and other services. A pipe is composed of one or more modules, each one performing a single task like retrieving feeds from a web source, filter, sort or merge feeds. The modules are broken into different categories such as **Data Source** for data accessing, **Operators** for data manipulation [17] and so on.

Yahoo pipes supports mainly *REST* web services, but provides also specific modules to access services as *Flicker* for searching for photographs by keyword and geographic location, *Google Base* for allowing anyone to create and publish web-accessible information, *Yahoo Local* for searching for services in a particular area, *Yahoo Search* to build custom searches as a starting point for Pipes and to fetch the source of a given web site(*Fetch Page Module*) and a CSV file(*Fetch CSV Module*).

To combine data feeds, *Yahoo Pipes* translates the source formats (which can be RSS, Atom or RDF) into its internal RSS feed data model. The data mapping between the source data model and the internal data model is **Semi-Automatic**. In fact, if the name of the input fields of a feed match with the name of the RSS fields, the conversion into the is done automatically; otherwise many facilities are provided to help the user for the data mapping. An example is the "Item Building" module which is used to restructure and rename multiple elements in the feed, in order to convert a source data model to the Internal RSS feed data model.

To restructure the schema of incoming data, *Pipes* provides three operators described in the Table 2.5.

Table 2.5: Data flow operators offered by Yahoo Pipes

Operator	Description
Regex	allows to modify fields in an RSS feed using regular expressions.
Rename	used to rename elements of the input feed and add new items in the inputs feeds.
Sub-Element	allows extracting sub-elements from the feed, which are buried in its hierarchy.
Union	allows to combine a list of item into a single list.

For data flow specification, *Yahoo pipes* provides only the *Union* operator to combine a list of item into a single list. Besides, to perform an elaboration on the data set, the following operators described in Table 2.6 are made available.

Pipes caches all the feeds it visits on its own server. A *pull strategy* is also implemented here to update the data on the server and the pull interval(that in our tests resulted to be 1 hour) is set for the whole application(*Global strategy*). The created pipes are hosted at Yahoo server and can be either accessed by a RSS or JSON client via it unique URL or visualized on the provide yahoo map. Besides, the pipes can be used like Mashup components to build a more complex pipe or their outputs can be combined with other tools that can process RSS feeds.

Yahoo Pipes is an extensible tool. If some functionalities the end-user needs are not offered, he/she can create a web service and invoke it from the system through the **Web Service** interface. This external

[17] http://pipes.yahoo.com/pipes/

Table 2.6: Data flow operators offered by Yahoo Pipes

Operator	Description
Reverse	If the feeds are ordered, the **Reverse** module provides a way to change the order of feeds, by flipping the order of all items in a data feed.
Sort	Used to sort a feed by any item element, such as title. The items can be sorted in ascending or descending order.
Truncate	This module returns a specified number of items from the top of a feed.
Tail	This module truncates a feed to the last N items, where N is a number you specify.
Count	This module counts the number of items in the input feed, and outputs that number.
Filter	The Filter operator is used to extract specific items from a feed that meet the filter condition.
Unique	This module removes items that contain duplicate strings.
String Operators	These modules help manipulate and combine strings.
Simple Math	This module performs basic arithmetic, such as addition and subtraction.
Date Operators	These modules can perform elaboration on date as create a date object from a string value.
URL	This module builds URLs in either traditional or Web 2.0 style query-string format from a series of input fields.

service is accessible through JSON Interface and its output has to be a data type supported by the tool. The added functionality is visible only to the owner and cannot be shared with the whole community.

Finally, the Mashup can be shared with either all people or no one, in particular the sharing can be: (1) Total, meaning that there is a read access to source code, to the data, and to the output. The source code of the pipe and the output are shared. In this way another user might access all the information used by the Mashup creator. (2) Partial sharing meaning that the people with whom the Mashup is shared have read access to source code and the output only. The data are not shared in this case. If a private element is used (Private string or Private text input) the code of the shared Mashup is available, but it is not possible to visualize the intermediate outputs. The Mashup output is available. The most restrictive policy is the Nothing policy which allows to have a read access to output. In this case, only the Mashup output is shared.

2.2.6 Popfly

Popfly is a web-based Mashup application by Microsoft[18] that allows users to create a Mashup combining data and media sources. The Mashup is built by connecting *blocks*. Each block is associated to a service like “Flicker”[19], ”Facebook”[20] and ”Virtual Earth”[21], and exposes one or more functionalities. A block is characterized by one or more operations with mandatory, optional input variables and an output. An operation defines the functionality exposed by the block such as display resources like photos or videos. The input variables are the parameters of the query to invoke the services, for instance URL of service. The

[18] $http://www.popfly.net/$
[19] www.flickr.com/
[20] www.facebook.com/
[21] www.microsoft.com/virtualearth/

output represents the way in which the output of the operation is provided to the user. The output can be either a data object or an HTML object that can be added to Mashup web page. The *Popfly* block supports mainly REST and SOAP services. In addiction, a WSDL block generator, that automatically produces a stub for a WSDL file, is made available.

In *Popfly*, the internal data model is in the form of object. The designer defines himself the characteristics and the behaviors of a block based on the source data model. Since, there is not an explicit internal data model, any transformation is performed by the tool for the mapping between the source data model and the internal data model. Therefore, the mapping is **Manual** give that it is manually specified by the designer.

Popfly is much more about data visualization than data manipulation, consequentially few operators are made available for data processing and integration. For data processing, operators for restructuring the schema of incoming data are not provided, since the Popfly's internal data model is object based, but some operators, described in Table 2.7 for object elaborations are made available.

Table 2.7: Operators offered by Popfly

Operator	Description
Sort	Sort operator is used to sort a list of input objects based on the values of the object.
Filter	This module filters the input list based on an arbitrary condition.
Truncate	This module returns a specified number of objects from the top of a input list.
Calculate	This module allow to do different match operation on the numbers.
Text Helper	This module allows to do some operators on the text as: (1)Split (returns an array of the substrings separated by a given separator), (2)getSubString: (Returns a portion of the input text, given a position and length)an so on.

Besides, for data Integration, *Popfly* makes available only the **Combine** module to join two sets of data of different types into one. For the data refresh, a *pull strategy* is implemented by *Popfly* to update the data. The pull interval depends on the frequency with the Mashup web page is reloaded by the user and concerns the whole application(*i.e. a global strategy*).

Popfly does not offer any true output function. Once a Mashup application has been developed and shared, it can be embedded into a web page, downloaded as a gadget, etc. But the mashed data cannot be exported in standard format for further processing instead of visualizing it.

For the extensibility, in *Popfly* the user can create his/her own blocks either by writing the Java Script code or by developing a SOAP web service. The created block can be plugged into the engine and shared with the whole community. Finally, the created Mashup can be shared with either all or no one. That is, two sharing policies are managed: (1) Total sharing where the user are given read access to the Mashup implementation, the data, and the output. The implementation of the Mashup and the output are shared. In this way another user might access all the information used by the Mashup creator. (2) Nothing where the Mashup is accessible only by the owner.

2.2.7 Google Mashup Editor

Google Mashup Editor(GME)[22] is an interactive environment to build, deploy, and distribute Mashup applications. The Mashup can be created using technologies like HTML, JavaScript, CSS along with GME XML tags and JavaScript API that further allows a user to customize the presentation of the Mashup output. *GME* allows also to consume RSS and Atom feeds accessible via *REST* web services. Local files contained data feed can be uploaded on the *GME* server and can be used through REST protocol. The user can also create his/her own feeds using Gdata API[23] and embed them in the Mashup's web page.

To operate on different types of data from different sources, the data in *GME* applications is managed with an Atom based data model named *Google Data* feed. *Google Data* feed is a data protocol based on Atom. The data from RSS feeds, are automatically converted to *Google Data* by *GME* through an XSL transformation. In reality, there is not an explicit mapping between the source data model and the internal data model, given that they have the same schema.

To handle the data, *GME* makes available operators for modifying the incoming data feed by sorting and filtering, shown in Table 2.8.

Table 2.8: Operators offered by Google Mashup Editor

Operator	Description
Sort	The sorting operator allows sorting the data on various types of elements as title, e-mail address, etc.
Filter	The filtering operator as for it allows retrieving specific data that meet the filter condition. The filter condition can be applied to various types of elements that appear in the feeds[24].

Operators for data merging and data schema manipulation are not explicitly provided but can be implemented using Javascript APIs, and XPath queries for data field access, and be plugged into the application. All the feeds visited by GME are cached on its own server. A *pull strategy* is implemented to update the data on the server but it does not support variable cache refresh frequencies, i.e.*Global Strategy*.

From the output point of view, *GME*, like *Popfly*, does not offer any true output function. The output of the Mashup can be only visualized using the provided visualization tool. Then, in what concerns the extensibility, if some functionalities the end-user needs are not offered, he/she can write Java Script functions that implement them. Like *Pipe*, the added functionality is visible only to the specific user and cannot be shared with the whole community.

Unlike the other Mashup tools, *GME* allows to specify sharing policies for the Mashup application. The sharing policy can be: (1) total, i.e. read access to source code, data and output. This means that the source code of the *GME* graph and the output are shared. In this way another user might access all the information used by the Mashup creator. (2) Partial, i.e. read access to source code. This means that If data is used to build the Mashup, and "no access" to other users is set, the code of the shared Mashup is available, but it is not possible to visualize or manipulate the data of the Mashup. (3) Nothing, where the Mashup is not shared. When a Mashup is shared in GME, for the data used to build the application, the designer can be define to share it with a group or with all users by specifying Read/Write policies. Finally, the data feeds retrieved from external sources can be read and accessed by any user. The data of the application can be accessed in Read/Write mode by the designer, and can be shared by specifying the classical Read/Write

[22] http://code.google.com/gme/index.html
[23] http://code.google.com/gme/docs/data.html

policies for data accessing.

2.2.8 Apatar

Apatar[25] is a Mashup data integration tool that helps users join desktop data with the web. Users install a visual job designer application to create integration called DataMaps. A Data Map is composed of data storage type and operator (which modify data in different ways) and defines the flow of data from the source(s) to the target(s). *Apatar* supplies the connectivity to applications such as MySQL, PostreSQL, Oracle, MS SQL, Compiere, SugarCRM, XML and flat files. In addition, it has connectors for the most popular Web 2.0 APIs such as Flickr, Salesforce.com and Amazon. The data sources are accessible mainly through REST web services. Besides, local files like Excel, RSS, Text file can be uploaded on *Apatar* server and then invoked through REST protocol. In *Apatar*, the internal data model is an object based. A specific object is automatically created for each data sources. Like *Popfly*, since there is not a specific internal data model, any transformation is performed by the tool for the mapping between the source data model and the internal data model.

Apatar is actually the only Mashup tool that provide a wide range of operators to consume and manipulate different types of data. In particular, to restructure the schema of incoming data, the user must define first the structure of the output, configuring the wished output connector(text file, or database, or whatever connector blocks), and then using the the *Transform* operator which can specify the correspondences between the input and the output fields. To perform elaboration on the data sets, *Apatar* makes available different sets of functions each one associate to a kind of data types such as string, date, number and so on. These functions are available in each operator blocks. Finally, for data integration the provided operator are shown in the Table 2.9.

Table 2.9: Operators offered by Apatar

Operator	Description
Aggregate	used to combine two different data sources. The user first must define the structure of the output and then in the *Aggregate* operator, can specify the correspondences between the fields of the input data and the output.
Distinct	similar to the 'DISTINCT' operator in SQL, this operator eliminates the data duplications for the columns specified by the user.
Filter	used to extract specific data fields that satisfy the filter conditions.
Join	combined two different data sources based on the join condition that is applied to the input fields.
Split	this operator is named *Validate* in the terminology of *Apatar*, split a data sources in two separate tables according to a specific criteria. The first table contains all data for which the criteria is **True**, the second contains the rest of the records.

To refresh the data uploaded, a *pull strategy* is implemented in *Apatar* to update the data and the pull interval is set for the whole application(*Global strategy*). The created Data Map, one created and shared, is hosted on *Apatar* web site and its output can be either exported in standard format like RSS or can be redirected to whatever storage data such as MS SQL, Compiere and SugarCRM. *Apatar* like *DAMIA*

[25] www.apatar.com/

mainly aims to aggregate and manipulate data that can be reused from other applications, so additional tools that consume the *Apatar* output formats can be used as presentation layer.

Apatar is an extensible tool. In fact new connector and operator blocks and new functions can be developed in Java and plugged inside the engine(like new plug-in). These new functionalities can be also shared with the whole community. Finally, from a sharing policy management point of view, the tool offers the possibility of sharing the whole Mashup, i.e. total sharing, or no thing. Total sharing means that other users have access to source code, data and output. There is the possibility neither to specify policy for the accessing of the data nor to share the application with a specific user or a group.

2.2.9 MashMaker

MashMaker [39] is a web-based tool for editing, querying and manipulating web data. *MashMaker* is different from the described tools in that it works directly on web pages. In fact, *MashMaker* allows users to create a mashup by browsing and combining different web pages. A Web page is seen as two parts: the presentation and the data through *HTTP* protocol. To handle the data part, an RDF schema[26] is associated to the Web page by the user. For this part, users can use the *Extractor Editor* offered by the tool to edit the data model and formulate the XPath query for data extracting. The extracted schema is used by the tool to extract and structure the corresponding data included in the Web page.

It should be noted that the schema of the Web page is stored on the *MashMaker* server. The corresponding data are extracted when the Web page is retrieved by the navigator[27]. To build a mashup, different Web pages are combined in one. This combination is done by mean of widgets, a mini-application that can be added to an existing web page to enhance it in various ways such as provide data to other widgets. The final goal of this tool is to suggest to the user some enhancements, if available, for the visited web pages. The enhancements can be mashups or widgets which have been defined before by other users on top of the visited web page.

Back to the RDF description of a Web page. Such a description is composed of a set of nodes and properties. A node corresponds to a location on the web page. It is characterized by an XPath[28] expression to identify the position of the location and the corresponding value of that location. A node can have only a concrete value (*leaf node*) or properties (*compound node*). To create a mashup application, *MashMaker* must first extract the RDF description of the data, representing the internal data model of the tool, from the HTML pages [38]. For each web page, a different schema can be created: an URI-comprehension mechanism is used for normalizing URIs that are different but refer to the same web page.

The particularity and the interest of *MashMaker* is in it's operators. In fact, the idea is to offer the user the possibility of using operation that he is used to use in his desktop (e.g. copy and paste). That is, *MachMaker* offer the following basic operators, *Other services are added every day to the tool. Here we focus on the main operators which are directly related to data integration.*, described in Table 2.10.

As explained before, once the description schema is defined for a Web page and saved on the server, the data of the corresponding Web page are extracted when it's retrieved via a navigator. *MashMaker* implements then a *pull strategy* to update the data. The refresh interval is fixed by default but the user can refresh manually the data. From the output point of view, *Mashmaker* does not offer any true output function. In fact, once a Mashup application has been developed and shared, the mashed data cannot be exported in standard format, but can be only visualized.

[26] Resources Description Framework: http://www.w3.org/RDF/
[27] The navigator is supposed to have MashMaker installed.
[28] XML Path Language: http://www.w3.org/TR/xpath

Table 2.10: Operators offered by Intel MashMaker

Operator	Description
Match	This operator is similar to the *left join* operator in SQL. The output contains all the items of the main Web page (left relation or table in SQL) plus the matched items from the targeted Web page (the right relation or table in SQL).
Copy	takes a copy of the selected Web page. The copied object contains the presentation, i.e. the Web page itself, as well as it's RDF description if it is available.
Paste	Reproduces the copied object, i.e. the Web page and it's description. From the visualization point of view, this operator allows to past the data of a new web page into the main web page of the Mashup. From the data point of view, the two data sets, corresponding to the two Web pages, are merged.

As the other tools, *MashMaker* is extensible since users can create new widgets and add and share them with other users. In addition to the possibility of extending the tool itself with new widgets, a particular schema can be enriched and extended by other users. The sharing of the created RDF schema of a particular Web page is automatically done by the tool. This one is shared with the whole community. For the widgets sharing, the user can create a widget containing private data, this widget and the Mashup containing it cannot be accesed by other users. Finally, the sharing can be done with all the users or no one. This means that it is not possible to manage group of users or particular users.

2.2.10 Discussion

In this section, we make a general discussion on the tools by considering their advantages and disadvantages. We aim in the same time to try to give a look out on the possible points to consider for further improvements.

Mushup tools are mainly designed to handle Web data. This can be seen, in the same time, as an advantage and an inconvenient. In fact, it is an advantage since it offers access and management of some data available only on the Web, e.g. RSS feeds. To access these Web data, the tools support the two most used protocols for exposing APIs, i.e. REST and SOAP protocol. This is a consequence of the success, the utility, and the popularity of these protocols. A disadvantage since by doing this, user's data, generally available on desktops, can not be accessed and used. This is a considerable disadvantage since users bring a lot of data on their desktops for cleaning, manipulation, etc. There is a lot of work done to help the user to put and manage data, even personal data, on the web [42], but since this is not completely adopted, local data should be considered.

As discussed in the previous point, i.e. consuming Web data, the majority of tools have an internal data model based on XML; this design choice is motivated by the fact that the data available on the web is mainly exposed in an XML format. Also, the communication protocols for the data exchanging over the network use generally XML messages. The other dominant internal data model in Mashup tools is object based. This data model is much more flexible to use, even if more programming is required to implement operations on it, especially for programmers. This diversity can explained targeted/origin community. In fact, XML is much more for databases community where as object is for applications and development

community.

To manage data, the tools make available only a small set of operators for data integration and manipulation. The set of provided operators is usually designed based on the main goal of the tool. For example, if the tool is visualization oriented, only few operators for data elaboration such as filtering and sorting are available. In addition, the offered operators are not easy to use, at least from a naive user point of view. Also, the tools do not offer powerful expressiveness since they allow expressing only simple operations, e.g. simple joins, and cannot be used to express more complicated queries such as joins with conditions, division, etc. This means that, from the expressiveness point of view, these tools are far from reaching the database languages, i.e. integration languages, such as SQL.

None of the analyzed tools implement a *Push strategy* for the data refreshing, the reason is that the majority of the currently available APIs are REST based. The style of the REST protocol requires all communication between the browser and the server to be initiated by the client and no support is offered to maintain the state of the connection [26]. All the analyzed tools use a *Pull strategy* for data freshness handling. This can be motivated also by the fact that the tools providers wish to control (or prevent) the overloading of their servers. In addition, they implement a *Global strategy* for the pull interval setting. This strategy however does not allow developing applications in which processed data are characterized by a high refresh frequency, since it is not possible to explicitly specify the refresh rate for each source.

One of the main goals of the Web 2.0 technologies is the creation, the reuse, the annotation and the sharing of web resources in an easy way. Based on these ideas, the Mashup tools are all extensible in the sense that new operators, and in some cases data schema, can be developed and invoked or/and plugged inside the tools. However, at this stage, the majority of tools do not support the reuse of the created Mashups. This feature could allow developing complex applications by integrating the results of different Mashups (also built with different Mashup tools). Some tools start to consider this issue such as Potluck [49][29] which can use the Exhibit output. However, this is a limited cooperation between tools. This is a very important point especially that the tools have a lot of limitations and a user cannot express his wishes using only one tool.

The current development of Mashup tools is mainly focused on offering features to access, manage and present data. Less consideration has instead been given to the issue of data sharing and security so far. The security criterion needs to be taken into account inside the tools since communication problems could make a Mashup perform too many requests to source data servers, causing overload for those servers. At this time, only *Intel Maskmaker* takes into account this problem applying some performance restrictions on the Mashup application [39].

Also, all the analyzed tools are server side applications, meaning that both the created Mashup and the data involved in it are hosted on a server, which is owned by the tool's provider. Therefore, the tool's provider has the total control on the Mashup and, if a user wants to build an application containing that Mashup, the dependability attributes [21] of that application cannot be properly evaluated. In addition, from the performance point of view, no tools provides information regarding the analysis of the performances and in particular information regarding the evaluation of the scalability. That information is needed to know the capability of a system to handle a growing amount of the data and the user request.

Finally, all the tools are supposed to target 'non-expert' users, but a programming knowledge is usually required. In particular, some tools require considerable programming effort, since the whole process needs to be implemented manually using instructions expressed in programming language such as Java Script. Others necessitate medium programming effort given that only some functionalities need to be coded in an

[29] This tool is not discussed further in this paper.

explicit way using a programming language; a graphical interface is offered to the user to express most of operations. At this time, there is no tool that requires low or no programming effort to the user to build a Mashup, which is necessary to claim that the tools are targeted for end-users.

Table 2.11: Summary of the considered dimensions for the tools analysis. (+) means the dimension (i.e. functionality) is provided, (-) means the dimension is not provided. These marks do not vehicle any ”positive” or ”negative” information about the tools except the presence or the absence of the considered dimesion.

		Damia	**Yahoo Pipes**	**MS Popfly**	**GME**	**Exhibit**	**Apatar**	**MashMaker**
Data Format & Access	Protocol[a]	P_2	P_2	P_2 ,P_3	P_2	P_2	P_1, P_2	P_1
	Data Format[b]	D_1, D_2, D_7, D_8	D_1, D_2, D_3 D_4, D_5, D_6 D_8, D_9	D_1, D_2, D_9 D_{10}	D_2, D_3, D_4	D_1, D_4, D_6 D_7, D_8, D_9	D_1, D_2, D_6 D_7, D_9	D_5
Internal Data Model	XML-based	+	+	-	+	-	-	+
	Object-based	-	-	+	-	+	+	-
Data Mapping	Manual	+	-	+	-	-	+	+
	Semi-Automatic	-	+	-	-	-	-	-
	Automatic[c]	-	-	-	+	+	-	-
Data Refresh	Pull strategy	+	+	+	+	+	+	+
	Push strategy	-	-	-	-	-	-	-
	Global pull interval	-	+	+	+	+	+	+
	Local pull interval	+	-	-	-	-	-	-
	Interval Setting	+	-	-	-	-	-	-
Mashup’s Output	Machine oriented	+	+	-	-	+	+	-
	Human oriented	-	+	+	+	+	-	+
Extensibility	Components	+	+	+	+	+	+	+
	Data	-	-	-	-	-	-	+
Sharing	Total	+	+	+	+	-	+	+
	Partial	-	+	-	-	-	+	-
	No thing	+	+	+	+	-	+	+
	Read only	+	+	+	+	-	+	+
	Read/Write	-	-	-	+	-	-	-
	All users	+	+	+	+	-	+	+
	Groups	-	-	-	+	-	-	-
	Particular user	+	+	+	+	-	+	+

[a] P_1 = HTTP; P_2 = REST; P_3 = SOAP
[b] D_1 = XML; D_2 = RSS; D_3 = ATOM; D_4 = JSON; D_5 = HTML; D_6 = CSV; D_7 = XLS; D_8 = RDF; D_9 = Image; D_{10} = Video
[c] This is considered true if the source data has the same data model as the internal data model.

Chapter 3

Turning Web Application into Web Services

In the era of Service Oriented Architectures a relevant research problem consists of turning Web applications into Web Services using systematic migration approaches. This chapter presents the research results that were obtained by adopting a black-box migration approach based on wrapping to migrate functionalities of existing Web applications to Web services. This approach is based on a migration process that relies on black-box reverse engineering techniques for modeling the Web application User Interface. The reverse engineering techniques are supported by a toolkit that allows a semi-automatic and effective generation of the wrapper. The software migration platform and the migration case studies that were performed to validate the proposed approach will also be presented in this chapter.

3.1 Introduction

The diffusion of the Service Oriented computing paradigm [10] is radically changing the ways of developing and delivering software: the keyword is now integration of services i.e., software functional units offered by different providers, that can be interconnected using the common infrastructure provided by Service Oriented Architectures (SOAs) for obtaining new services and applications. In such a scenario, the coexistence of new and already existing software assets becomes feasible, and legacy systems can take the opportunity of extending their lifetime exploiting the new architectures. Of course, this integration requires that maintenance interventions involving reverse engineering, reengineering and migration approaches are planned and executed. An interesting classification of approaches for integrating legacy systems in SOAs has been presented by Zhang and Yang [94] who distinguish between the class of black-box re-engineering techniques, which integrate systems via adaptors that wrap legacy code and data and allow the application to be invoked as a service, the class of white-box re-engineering techniques, which require analysis and modification to existing code in order to expose the system as Web services, and the class of grey-box techniques that combine wrapping and white-box approaches for integrating parts of the system with a high business value. White-box approaches are usually more invasive to legacy systems than black-box ones,

but they are able to prolong the lifetime of a system, saving on maintenance, and improving the efficiency of processes relying on legacy code. Vice-versa, black-box approaches that leave the original system configuration untouched, as well as its execution environment, may represent a cost-effective and practicable solution for rapid migration processes. In this context, a specific and relevant migration problem consists of turning Web applications into Web services: here, the basic challenge is that of transforming the original (non programmatic) user-oriented interface of the Web application into a programmatic interface that exposes the full functionality and data of the application. Wrapping techniques exploiting black-box knowledge of the Web application represent candidate solutions to implement effective and efficient migration processes. Using wrapping techniques in the field of Web applications is not new: in the last years several Web wrapping techniques and tools have been developed with the aim of providing alternative solutions to manual Web browsing. While the first versions of these tools were just data wrappers supporting the task of extracting data from HTML interfaces using specific extraction rules, more sophisticated versions of Web wrappers have been successively developed to automate the interaction with the Web application to aid or to completely substitute the user in some simple interaction tasks. More recently, these tools are being proposed as integration solutions, and visual environments (called Web wrapping toolkits) supporting semi-automatic or automatic generation of wrappers are emerging as commercial solutions. Although these tools provide technological solutions to some specific Web integration problems, there is still methodological immaturity in the field of Web service migration. Some relevant research questions that still need to be addressed in this field include:

1. Which criteria can be used to establish which parts of a Web application (e.g., data layer, functional layer, or presentation layer) can be migrated to a SOA?

2. Which are the migration strategies and techniques applicable to turn a Web application into a Web service?

3. For each type of migration strategy, what is the migration process to be adopted?

This chapter answers these questions, by addressing the specific problem of migrating the functionality of a 'traditional' Web application to Web service architectures using black-box migration strategies. In previous works [28, 29] we proposed a black-box modernisation technique based on wrapping to migrate the functionalities implemented by an interactive, form-based legacy system towards Web services. In this chapter, we present the research results we obtained by adopting a similar migration approach in the different field of Web applications. In particular, this chapter will preliminarily analyse similarities and differences between migrating a form-based legacy system and migrating a Web application, and will present the wrapper architecture and the migration process that were defined to solve this specific migration task. The results of two case studies where different functionalities of a Web application were successfully turned into Web services will also be discussed in this chapter. The chapter is organised as follows. Section 3.2 describes related works on the migration of existing applications to Web services, while Section 3.3 presents the migration problem addressed in this chapter and the proposed wrapping solution. Section 3.4 shows the characteristics of the wrapping toolkit designed to support the wrapper generation, while Section 3.5 introduces and discusses case studies of migrating a Web application. Finally, Section 3.6 provides concluding remarks and outlines directions for future work.

3.2 Related work

In the last years, several Web wrapping techniques and tools have been developed to solve the problem of automatically extracting structured information from Web sites and applications that, due to the specific nature of the HTML language, normally provide unstructured information. These techniques are usually based on extraction rules which may be less or more sophisticated, and defined either semi-automatically through demonstration [35, 40, 47, 55–57, 66], or automatically by using machine-learning algorithms or other artificial intelligence methods requiring training examples [55, 56]. A survey of several Web data extraction tools is presented in [57].

While these wrapping approaches essentially address the problem of data extraction from Web applications for the aims of an automatic and integrated navigation of Web pages, or for the migration towards the Semantic Web, other authors have recently presented their experiences with the migration of Web application functionalities to Web services.

Guo et al. [44] propose a white box reverse engineering technique and a tool for generating wrapper components that make the functionalities of a Client - Server .NET application available as Web services. Jiang and Stroulia [52] describe their work for constructing Web services from functionalities offered by Web sites: their approach is based on the analysis of the pairs of browser-issued HTTP requests and corresponding HTML responses produced by the server for selecting the functionalities to be specified in terms of WSDL specifications. Baumgartner et al. [23] have proposed a suite for obtaining Web services from applications with Web-based interfaces: the suite includes a visual tool that allows the extraction of relevant information from HTML documents and translation of them into XML which can be used in the context of Web services.

A black-box technique for migrating a legacy system functionality towards SOAs has recently been proposed by Canfora et al. [28]. This technique aims at exposing interactive functionalities of form-based systems as services. The problem of transforming the original user interface of the system into the request/response interface of SOA is solved by a wrapper that is able to interact with the legacy application autonomously by knowing the rules of the dialogue between user and application. These rules are specified by a User Interface (UI) model based on Finite State Automata that is interpretable by an automaton engine, and that can be obtained by UI reverse engineering techniques. This migration approach has been validated by case studies that showed its effectiveness.

3.3 The Migration Approach

In a form-based legacy system, the human-computer interaction is session-based and composed of an alternating exchange of messages, based on forms, between the user and the computer [35].

The model of the interaction with a traditional Web application can be, thus, considered as a special type of form-based interaction model where the concept of form is substituted by the concept of Web page: the user provides his input on a Web page and the remote component of the Web application sends back output data or new input requests by means of other Web pages.

As a consequence, since the black-box wrapping approach proposed in [28, 29] just relies on this type of interaction model, we can hypothesise that the same methodological approach is still adoptable to turn a Web application into a Web service, while the technologies supporting it will have to be adapted to the new context.

As an example, a relevant difference between Web applications and form-based legacy systems regards the techniques and protocols adopted for the communication between parties, where legacy applications

often adopt terminal servers to support various types of terminals using specific communication protocols, while Web applications are always based on the HTTP communication protocol and mostly exploit the HTML language (with eventual components coded in script languages) to implement the user interface. For the aim of implementing the wrapper, a possible advantage of using HTML will consist of the possibility of automatically interacting with the Web application using open technologies (such as Http Unit [12]) or data wrapping technologies, rather than using specific Terminal emulators.

In the following, details of the wrapper, of the migration platform providing the environment for the wrapper execution, and the migration process supporting this approach will be presented.

3.3.1 The Wrapper

The wrapper is the core component of the migration approach, whose mission consists of interacting with the Web application to obtain the execution of the functionalities that must be exposed as Web services. Of course, since the rules of the interaction with a Web application depend both on the specific Web application and on the specific functionality to be migrated, the wrapper will have to behave differently depending on the service to obtain.

Therefore, we separated the wrapper into two main components, an Automaton Interpreter and the Automaton to be interpreted. The Automaton is actually a non deterministic Finite State Automaton [84] that provides the model of the interaction associated with a given functionality: the Automaton is defined by specifying a set of interaction states, the actions to be performed at each state, and the set of transitions between states. On the other side, the Automaton Interpreter is an engine that executes the actions associated with each state.

According to the Automaton conceptual model shown in Figure 3.1, any specific Wrapped Service will be associated with its Automaton, and with a set of Service Input and Service Output, respectively. Moreover, each automaton will own a set of automaton variables used to buffer intermediate results of an application execution. Automaton variables may be associated with Web pages input or output fields, or Web service input or output data.

The logical architecture of the wrapper is reported in Figure 3.2 as a layered architecture where the higher level layer is the Web Service interface manager that manages external Web service requests and responses, the intermediate layer is the Automaton Interpreter that coordinates the Web application execution by interacting with two lower level layers, the former storing the Automaton specification obtained from an Automata Repository, and the latter executing the Automatic interaction with the wrapped Web application, respectively.

3.3.2 The Migration Platform

This platform provides an environment for the wrapper execution comprising the Automaton Interpreter component besides additional components that are delegated by the automaton engine to implement the following key actions:

1. data input in a Web page associated with an Input interaction state;
2. output data extraction from a Web page associated with an Output interaction state;
3. identification of the current interaction state based on screen templates and evaluation of the discriminating expressions;
4. access to the automaton specification and to its automaton variables.

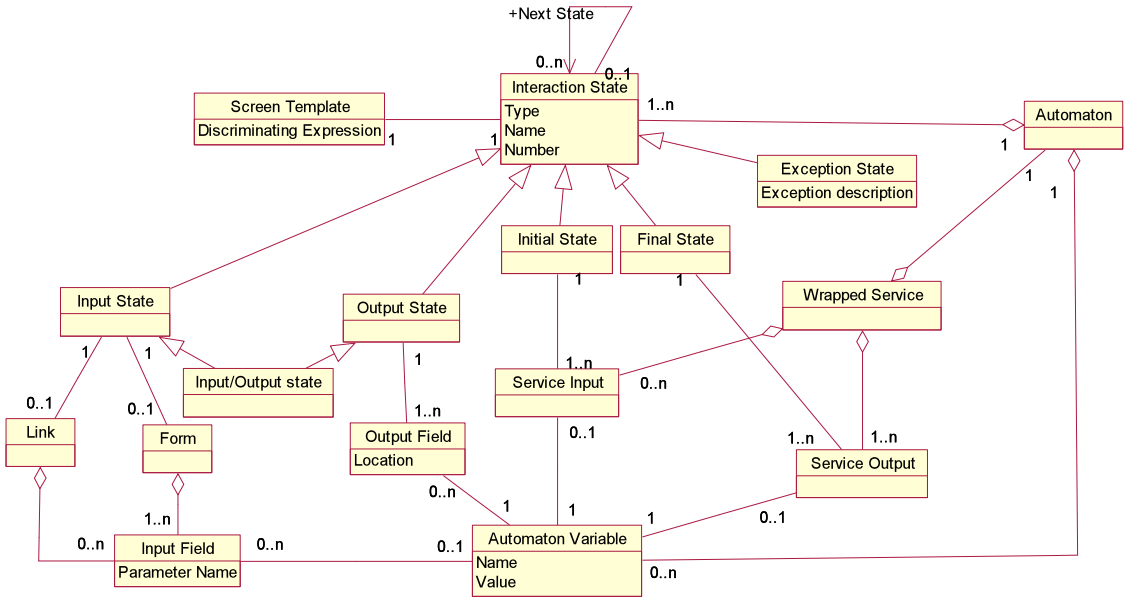


Figure 3.1: The Automaton conceptual model

All the platform components have been implemented using Java technologies (cfr. the UML component diagram reported in Figure 3.3 showing the physical organisation of the platform). WA Interaction Manager is the component implementing the automatic interaction with a Web page by using the HttpUnit framework [12], to accomplish both the first and the second type of actions listed before. The second action type is implemented by using the XPath technology [14] for automated data extraction from Web applications. The identification of the current interaction state is realised by the State Identifier component that exploits the technique for Web page classification presented in [34]. Finally, the automaton specification is persistently stored in an XML based database, and the access to it and to its variables is allowed by generic data driver technologies (such as DOM or SAX libraries).

3.3.3 The Migration Process

The wrapping technique presented is based on a migration process that defines all the steps needed for transforming a selected functionality (i.e., a use case) of a Web application into a Web service. The process includes four consecutive steps:

1. Selection of the Web application functionality to be turned into a Web service
2. Reverse Engineering of the Web application User Interface
   (a) identification of execution scenarios
   (b) characterisation of execution scenario steps
3. Interaction Model design
   (a) Evaluation of alternative modelling solutions

(b) XML-based design of the model

4. Wrapper Deploy and Validation

The first step of the process aims at selecting the functionality to be exposed as a service from the set of functionalities implemented by the Web application. This task will be driven by the business goals of the Web application proprietary organisation that may be interested both in turning an already existing functionality into a Web service, and in developing a completely new service by reusing the existing functionalities of the application.

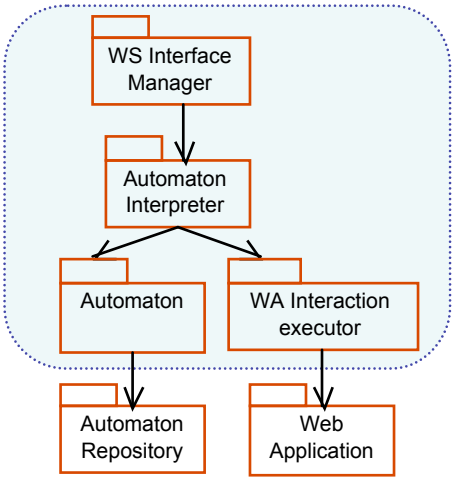


Figure 3.2: Wrapper Logical Architecture

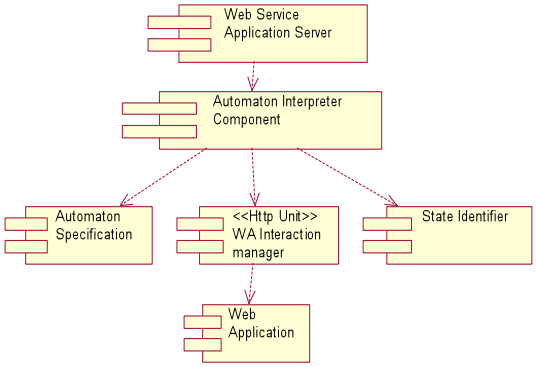


Figure 3.3: Migration Platform organisation

The choice of the functionality will be based on several factors like its reusability, its business value, its state independence, and so on. A special activity of this step will be the evaluation of the degree of cohesion of a candidate use case, in order to establish whether it may be transformed into a single, high-cohesion service or, vice-versa, it needs to be broken down into more elementary use cases, each of which can be wrapped into a more cohesive service. In the latter case, the selected use case will not be presented as a simple service but as a composite one obtainable by a workflow based composition of the single wrapped services [47].

The Reverse engineering step will be executed to obtain a sufficiently comprehensive model of the interaction between the user and the Web application for each selected use case. To reach this goal, an extensive dynamic analysis of the application will be performed to exercise and identify each possible flow of actions (i.e., normal and alternative/exceptional execution scenarios) associated with the use case. During this analysis, for each distinct scenario the sequence of pages returned by the system and user actions performed on these pages will have to be traced and collected. On the basis of this data analysis, each scenario will have to be associated with the correct sequence of interaction states, and each state will have to be characterised by a screen template of the corresponding Web page. This template will be own by all the equivalent Web pages (i.e., pages associated with the same logical state, but obtained during different application executions) and it can be univocally identified by a discriminating expression, as explained in Section 3.3.1. The technique presented in [34] can be used to solve the discriminating expression generation problem.

The discriminating expression-based identification technique allows the wrapper to automatically solve the problem of identifying the current state of the interaction with a Web application on the basis of the analysis of the last returned Web page.

The third step of the migration process aims at designing the Web application interaction model needed to implement a given functionality, and at producing an engine-interpretable version of it. The design activity will have to evaluate the opportunities of producing a single (but providing a complex interaction logic) automaton, comprehensive of all the scenarios' interaction states, or of obtaining the same interaction model using one or more simplified automata (each one implementing a simpler interaction logic). In the latter case a workflow of activities has to be defined in order to implement the selected use case by means of a composite Web service. Of course, these alternative approaches will differ in several aspects, such as the design complexity of involved automata, modularity and reusability of the models, performances of obtained wrappers. These aspects will have to be carefully considered to take the more effective design decisions. An example of such a decision-making will be presented in Section 3.5 of this chapter by means of a case study.

The second activity to be performed in the third step of the migration process consists of developing the specification of the selected interaction model that could be interpreted by the wrapper. This specification will include the XML-based specifications of the single automata, and the (eventual) workflow specification expressed by a workflow description language (such as BPEL).

The fourth and final step of the migration process will be devoted to the wrapper deploy and validation activities. The deploy activity will include all the operations needed to publish the service and export it to an Application Server. As the service must be exposed as a Web service, the WSDL document [7] describing input data contained in request messages and output data contained in response messages will have to be written and stored on the Application Server, while a UDDI description document [13] of the service can be registered in a public UDDI repository.

The validation activity will be a testing activity aiming at discovering failures of the service execution due to eventual Automaton design defects. Possible failures may consist of State Identifier exceptions (i.e., corresponding to unidentified Web pages) or unexpected output responses produced by the service.

In order to maximise the effectiveness of this testing activity, different and complementary testing strategies, such as the ones proposed in [29], can be used to design test cases.

3.4 The Migration Toolkit

The activities of the migration process proposed in the previous Section are supported and partially automated by a set of tools integrated in an execution framework. This toolkit includes the Page Collector and Discriminating Expression Generator tools that support the Reverse Engineering step of the migration process, besides the Classifier and Automaton Designer tools which support the Interaction Model Design step. All the tools have been developed using Java technologies. The integrated platform of tools with their main input and output flows of data is presented in Figure 3.4, while a description of services they offer follows.

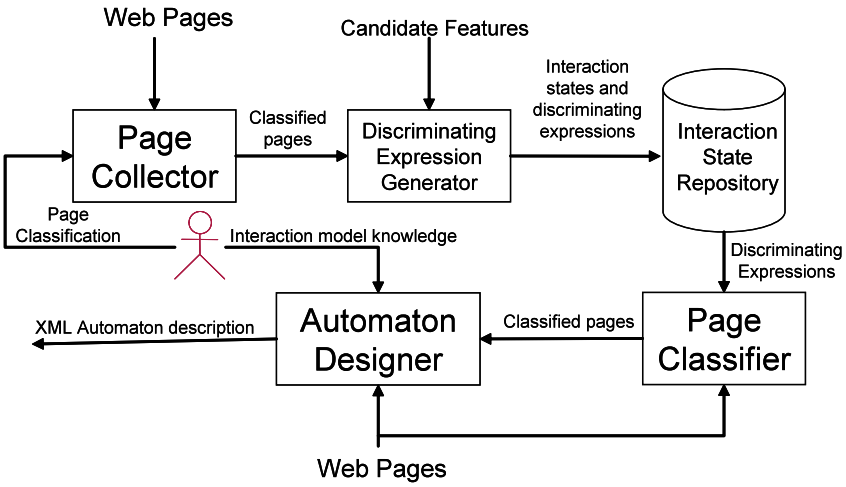


Figure 3.4: Wrapper Logical Architecture

Page Collector is an interactive tool aiding the software engineer during the task of dynamic analysis of the Web application. The tool is responsible for storing the source code of Web pages that are generated during the navigation of the Web application, whose HTML code is transformed into XHTML using JTidy libraries. For each Web page, the tool also stores its equivalence class provided by the software engineer (where an equivalence class is composed by all those pages associated with the same logical interaction state, but obtained during different application executions).

The Discriminating Expression Generator tool implements the technique presented in [34] for generating discriminating expressions for classes of equivalent pages. This technique requires that relevant page features are preliminarily selected from equivalent pages, and therefore exploits Formal Concept Analysis for identifying the combination of page features that will be able to discriminate the class of the pages.

The Page Classifier tool implements the task of classifying a given Web page using the set of discriminating features: the tool, by evaluating the presence and absence of discriminating features in a given page, proposes a unique classification of it, or eventually reports the impossibility of classifying the page.

The Automaton Designer tool has been developed to aid the generation of the XML-based specification of the automaton associated with the use case to be wrapped. This specification will include the set of automaton interaction states, transitions between states, and actions to be performed in each state by the automaton interpreter.

The generation of this specification is based on an assisted navigation of the Web application, implemented by means of HttpUnit [12] where, for each scenario of the use case, each page will have to be associated with a type of Interaction State (input, output, exception, initial, or final) and the information

needed to characterise that state is collected. In particular, the Automaton Designer supports the characterisation of the states in the following ways:

- In the Input states, the tool automatically retrieves and displays all the page input fields and stores the correspondence (provided by the software engineer) between each field and the associated automaton variable.
- In the Output states, the tool stores the relevant output fields selected by the software engineer from the page, associates them with the corresponding automaton variables, and automatically generates the XPath queries that are able to extract them from the page.
- In the Exception states, an exception message will have to be associated with each state: this message will be either retrieved from the Web page content (via an XPath query) or will be a constant message defined by the software engineer. In both cases, the message will be associated with an automaton variable.
- In the Initial state, the tool supports the software engineer task of associating the input variables (provided by the service request message) with Automaton variables.
- In the Final state, the tool supports the software engineer task of associating the Automaton variables to output variables (composing the service response message).

After completing the characterisation of each state, the tool automatically generates the XML specification of the automaton.

3.5 Case Studies

This Section presents some case studies that were carried out for exploring the feasibility of the proposed wrapping approach. The case studies involved a real world Web application providing user functionalities for planning and booking journeys via trains. Selected functionalities of this application were transformed into Web services using the migration process presented in Section 3.3. Alternative wrapping design approaches have been adopted and compared, too.

3.5.1 First case study

The aim of this case study was to illustrate the steps of the process we performed to turn a Web application use case into a Web service using a single Automaton.

In the first step of the migration process, the Web application use cases were analysed in order to determine possible candidate services. The main application functionalities included: a) Timetable browsing, b) Seats Booking, c) Ticket Purchase. A black-box analysis of the Web application showed that these three functions were not completely independent since the second one requires the first one execution, and the third one requires the execution of both the first and the second one. As a consequence, three candidate Web services were identified that expose the following functionalities respectively: 1) Timetable browsing (a); 2) Booking (a+b); 3) Purchase (a+b+c). The Booking Web service only suits users that want information on tickets availability to book a seat, but do not want to buy tickets on-line by means of this rail enquiries application. In this case study, we focused our attention on the Booking service (that is just accessible to registered users) and, since the subject Web application was able to provide several Booking functions

with different business logic rules, we had to decide both the semantic and the interface of the service we wanted to obtain.

We decided to wrap a booking service (WS1) that, provided with Departure station, Arrive Station, Date, and starting Time of the journey, besides user Login and Password, books one seat on the first available train solution listed in the timetable iff it can be purchased on-line and the Standard fare can be applied. Other journey search criteria are intended to be set to the default values stated by the application (e.g. 2nd Class, ticketless mode, etc). The Fare is not included in the service input list since it is a pre-defined value for this service. The output data included all booking details (Train number, Departure time, Coach number, and seat number), or an exception message in case of unsuccessful booking.

The second step of the process was devoted to the Reverse Engineering activity that aimed at identifying the interaction model of the booking use case, made of normal (i.e., successful) and exceptional (i.e., unsuccessful) execution scenarios. These scenarios had to be preliminarily identified before proceeding with the automaton design.

To reach this aim, in this step we exercised the Web application with several input data that were selected in order to trigger all possible behaviours of the selected booking functionality. This task was supported by the Page Collector tool that, for each execution, stored the corresponding sequence of Web pages and the other information needed to characterise these pages.

This analysis discovered that the use case included 1 successful scenario (allowing the correct booking to be accomplished) and 14 exception scenarios (corresponding to various types of exceptions that may occur during the execution). These scenarios were grouped into 6 classes, where scenarios associated with similar logical exceptions were clustered into a single class. Table 1 reports for each class its cardinality (#SC), a textual description, and the sequence of states associated with the scenario class.

The interaction states reported in the last column of Figure 3.5 refer to the graphical representation of scenarios provided by the UML State Diagram in Figure 3.6. This diagram includes three macro-states (namely, state 8, 10 and 11) representing further states (not shown in the figure for brevity) belonging to a same class of equivalent scenarios.

ID	#SC	Description	Interaction States
SC1	1	Seat booked with success	S-1-2-3-5-6-7-E
SC2	1	Not Available Trains	S-1-9-E
SC3	5	Available Trains but other journey search criteria cannot be satisfied	S-1-2-8-E
SC4	3	Failed User Authentication	S-1-2-3-5-6-11-E
SC5	3	Wrong or Incomplete journey specification	S-1-10-E
SC6	1	Not Available trains with Standard Fare	S-1-2-4-E

Figure 3.5: Booking Scenarios

In Figure 3.6, each scenario class is represented by a distinct path between the starting and final states.

As an example, the scenario SC1 allowing a user to book a seat with success was associated with the path where the user initially submits the requested journey data (i.e., Departure station, Arrive Station, Date, and starting Time) (State 1-Home) and, thus, if input data are correct and trains satisfying his/her request are available, State 2 -Available Trains is reached where a list of available train options is shown. The user selects the first available solution listed in the timetable and the State 3-Available Standard Fare is reached if the Standard fare can be applied and the solution satisfies the other search criteria. In this

state, the user clicks on the reservation button and the application shows train details and ticket cost in the State 5-Train details shown. In this state, the user is required to make the authentication procedure and the application enters the State 6-Authentication, where the user provides correct login and password, and the State7- Booking is reached where booking details are shown and the interaction ends with success.

As to the remaining scenarios, SC2 is the one associated with the exception that no train is available between Departure and Arrival station in the requested date and hour. SC3 is the set of exceptional scenarios executed when one or more train exist, but none of them can be booked for several reasons (such as the 2nd class does not exist, or it is full). SC4 groups exceptional scenarios executed when user authentication fails, while SC5 represents exceptions due to incomplete or incorrect input data. SC6 is the scenario triggered when there are more available trains but they do not offer the standard fare.

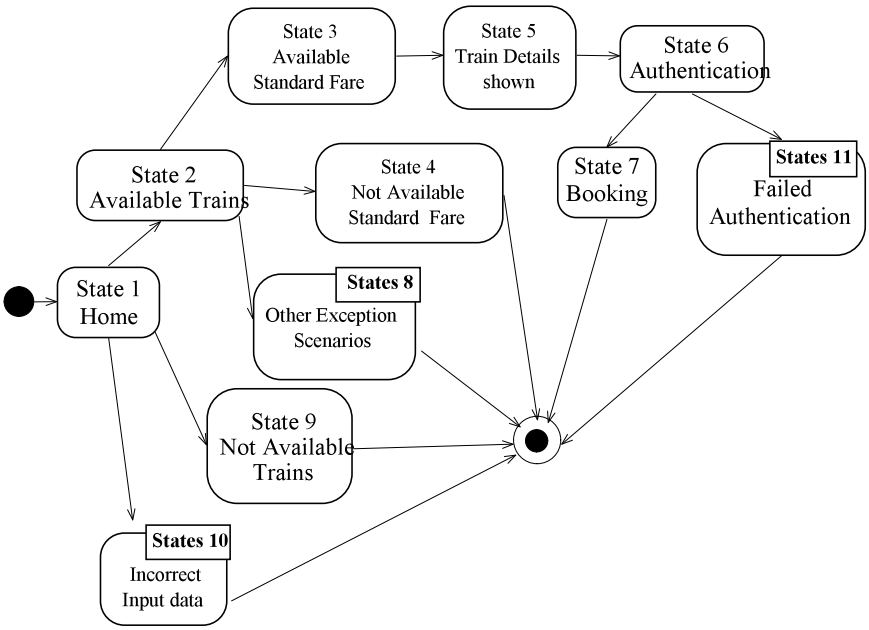


Figure 3.6: Wrapper Logical Architecture

During the third process step, the interaction model needed by the wrapper to implement the required use case had to be designed. In this case study, a single automaton was designed that implemented all and the same interaction rules represented by the model reported in Figure 3.6. In particular, this automaton had to allow the wrapper to look just for the first solution listed in the timetable and satisfying all the search criteria, returning an exception code if no seat can be booked at the first train attempt (see scenarios SC2, SC3 and SC6). This Finite State Automaton comprised the same set of scenarios, states and interactions included by the model reported in Figure 3.6. Of course, all the additional information needed by the Automaton Interpreter to interpreting this model had to be designed too: this task was performed with the support of the Automaton Designer tool that finally produced the automaton XML specification.

In the fourth step of the process, the operations needed to deploy and validate the wrapped use case were executed. The Automaton Specification was deployed on the migration platform (described in Figure 3.3) and the Application Server was configured to manage the wrapped Web Service. The service was therefore submitted to a validation activity in its execution environment, where a client application invoking the service and providing it with input data was used. Test cases that covered all the scenarios reported in Table 1 were designed and executed. Thanks to this analysis, some errors in the definition of the discriminating

expressions and some Automaton description faults were detected.

3.5.2 Second Case Study

The aim of this second case study was to explore the possibility of implementing another Booking service (WS2) having the same interface of the service WS1 considered in the first case study, but implementing the booking functionality with different (and more effective) business rules. In particular, we removed the main limitation of WS1 that consisted of making just a single booking attempt involving the first train listed by the timetable. We added the rule that the service makes at least a number Nmax of booking attempts involving distinct trains from the timetable.

Implementing this service required a new Interaction Model to be executed by the wrapper. We decided to compare two different approaches of implementing this model, a first one (A) based on a single (but implementing a more complex business logic) automaton, and a second one (B) where the business logic implemented by the automaton is left 'simple' and new business logic is added outside the automaton. These solutions are presented in the following.

Business logic internal to the automaton

This solution required a modification to the automaton designed in the previous case study for implementing the new business logic. In particular, the new logic was implemented with the support of an automaton variable N that stored the current number of attempts. The value of this variable had to be checked in some automaton states and modified in other ones. An excerpt of the corresponding new automaton is reported in Figure 3.7. As the figure shows, once the State 2 (Available Trains) is reached, four possible transitions can be now executed:

1. The first transition leads to State 3 (Available Standard Fare) and is executed if the first train in the list complies with all user's requirements;

2. The second transition leads to State 4 (Not Available Standard Fare), meaning that the first train in the list does not comply with the user's requirements and the number N of performed attempts is less than the maximum allowed number (N≤Nmax). When this state is reached, the automaton variable N is incremented.

3. The third transition leads to the exception State 12 (Too much Attempts) and is performed when the first train in the list does not comply with the user's requirements and N¿Nmax. When this state is reached, an exception is returned.

4. The fourth transition leads to the same State 8 of the automaton designed in the first case study.

The main difference between these automata is that in the second automaton two of the next states of the State2 (State4 and State12) cannot be established just using the features of the corresponding Web page class (i.e.,$Discriminating_Features_State4$ and $Discriminating_Features_State12$, respectively) but also the current value of the Automaton variable N must be considered. Hence, more complex conditions will be evaluated. The following conditions trigger the transitions to State 4 and the State 12, respectively:

1. $State4 : Discriminating_Features_State4$ AND $N <= Nmax$

2. $State12 : Discriminating_Features_State12$ AND $N > Nmax$

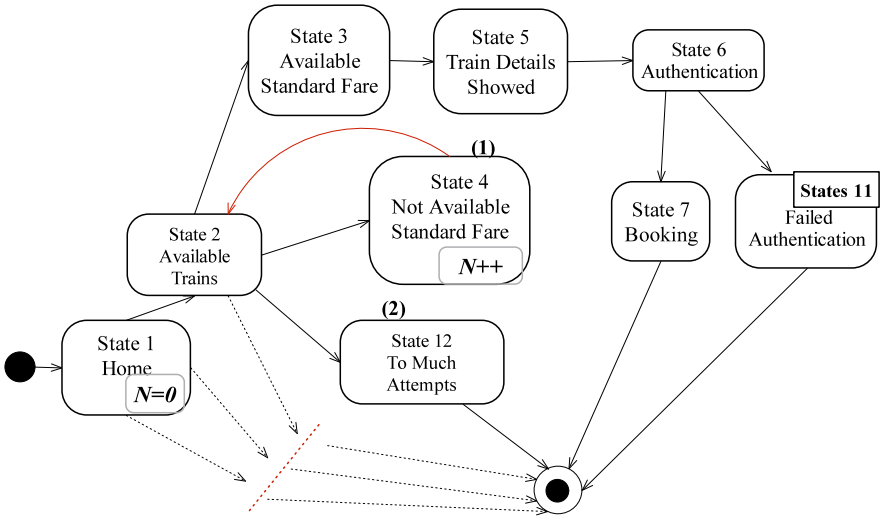


Figure 3.7: Wrapper Logical Architecture

TC#	Description	Interaction States
1	Seat booked with success	S-1-2-3-5-6-7-E
2	Not Available Trains	S-1-9-E
3	Available Trains but other journey search criteria cannot be satisfied	S-1-2-8-E
4	Failed User Authentication	S-1-2-3-5-6-11-E
5	Wrong or Incomplete journey specification	S-1-10-E
6	Not Available trains with Standard Fare	S-1-2-12-E
7	Booking of the second available train in timetable	S-1-2-4-2-3-5-6-7-E
8	Booking of the fourth available train in timetable	S-1-2-4-2-4-2-4-2-4-3-5-6-7-E

Figure 3.8: Test cases for the second case study (sol.A)

As a consequence, the XML specification of the second automaton will be more complex than the first one. This is an additional cost to be paid for obtaining a more effective booking service. As to the validation activity of the wrapped service, it was carried out with the same approach used for validating the automaton developed in the previous case study. In this case, due to the greater number of scenarios included by the automaton, additional test cases had to be designed (cfr. Figure 3.8). In particular, the scenarios including the loop were exercised by two test cases (namely, 7 and 8) where the loop was executed just once, and Nmax times, respectively.

Business logic outside the Automaton

To implement the solution where business rules are not added to the automaton but are left outside it, we had to design an orchestration process defining the private workflow to be executed by a deployment engine to realise a composition of services. This orchestration process was specified as a BPEL executable process [20].In BPEL, the sequence of the operations is specified by means of a set of primitives called activities, according to the terminology of the workflow languages [9]; BPEL provides basic and structured activities. A typical scenario is the following. The BPEL executable process waits for a message from a

requestor (which represents a client) by using the receive activity; when the message arrives, an instance of the process is created. Then the process instance might execute one or more partners by means of the invoke activity and finally sends back a response to the requestor by using the reply activity. The workflow process we designed is reported in Figure 3.9. The process logic is the following: the process starts with the Receive Activity (1) where a request for a Booking Service is waited. When a request from a client is received, an integer process variable N (counting the number of search attempts) is set to 0, and the Booking Service activity (2) (implemented by the wrapped service WS1) is invoked. When WS1 returns its response message, this message will be evaluated by the decision activity (3) in order to distinguish between two cases:

- if WS1 returned a 'Not Available Fare' exception message AND the number of search attempts is less than the maximum allowed number NMax, WS1 will be invoked again but with a new Depart Hour. The Depart Hour modification will be performed by the workflow activity (4) that will also increment the number of attempts stored in N.
- if WS1 returned a message different from the ÔNot Available Fare' exception message OR no further attempts can be made, the Reply activity (5) will be invoked that will send back a response message to the process requestor.

After the design of the workflow, the wrapped service had to be deployed and validated. In this case, the service platform where the service was deployed had to include a BPEL engine that supports the execution of BPEL processes. The BPEL engine used to deploy the service during this case study is ActiveBpel [15]. The testing activity was carried out in two phases, the first one devoted to the validation of the single activities included by the workflow, and the second one aiming at testing the equivalent classes of execution paths of the workflow.

A comparison between the wrapping solutions designed in these case studies should consider several aspects. A preliminary consideration regards the applicability of the approaches in case of state-dependent Web services that are services where distinct service invocations may return different results, depending on the state of data managed by the corresponding Web application. For this category of services, the workflow based solution may not be applicable, because the state of the application may change between two consecutive invocations, while the single-automaton solution is always applicable. As to the wrapper design activity, the approach adopted for the solution (A) will generally produce automata with greater internal complexity (both of states and transitions) which requires greater effort of designing and validating the automata. As to the resulting Web service, in general, creating small Web services promotes the development of scalable computing components and the creation of goal-oriented services that are specific for classes of users.

Vice-versa, the approach adopted in the case (B) generally produces more modular solutions, requiring smaller automaton design effort, but an additional effort for the workflow design and validation. Moreover, wrapping use cases according to a modular solution may efficiently face partial updates of the Web application and also may result more fault tolerant then building a complex Web service. A price has to be paid in terms of performance of the wrapped service, since a further level is added introducing the needs for the business process execution and multiple invocations to the Web service(s). Hence, the choice among these two approaches should be driven by the required characteristics of the service to be obtained.

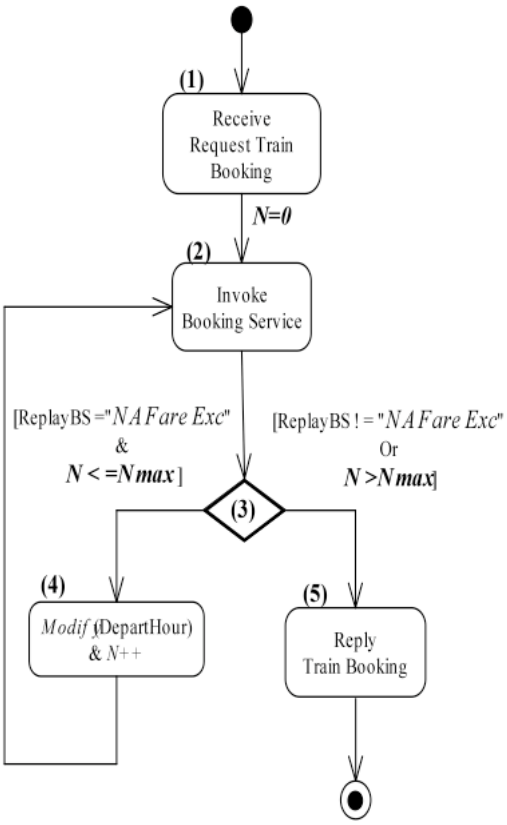


Figure 3.9: Wrapper Logical Architecture

3.6 Discussion

Nowadays, defining and validating systematic approaches for exporting existing software applications towards the new Service Oriented architectures is a relevant research issue.

This chapter addressed the problem of migrating the functionality of existing Web applications towards Service Oriented architectures by a black-box wrapping approach. This approach considers a Web application as a special type of 'form-based' system, and proposes to adopt Automaton-based wrappers for transforming the original (non programmatic) user-oriented interface of the Web application into a programmatic one that exposes the full functionality and data of the application. In this chapter, the logical architecture of the wrapper, the migration platform providing the environment for the wrapper execution, the migration process allowing the wrapper design, and a toolkit developed to assist the migration process execution have been presented. Some migration case studies that were carried out for exploring the feasibility of this approach, and evaluating and comparing alternative wrapping design solutions, have also been discussed in this chapter.

Chapter 4

Automated Service Composition Methodology

Web service composition is a very active area of research due to the growing interest of public and private organizations in services integration and/or low cost development of value added services. The problem of building an executable web service from a service description has many faces since it involves web services discovery, matching, and integration according to a composition process. The automated composition of web services is a challenge in the field of service oriented architecture and requires an unambiguous description of all the information needed to select and combine existing web services.

In this chapter, we propose an unified composition development process to the automated composition of web services which is based on the usage of Domain Ontologies for the description of data and services, and on workflow patterns for the generation of executable processes. In particular the chapter focuses on the integration of the matching and composition phases. The approach aims at producing executable processes that can be formally verified and validated. In order to meet this goal an operational semantics and the Prolog language are used throughout the composition process.

4.1 Introduction

The SOA (Service Oriented Architecture) foundation relies upon basic services, services descriptions and operations (publication, discovery, binding) [70]. One of the most promising benefits of SOA based web services is enabling the development of low cost solutions/applications by composing existing services. Web service composition is an emerging approach to support the integration of cross-organizational software components [50] whose effectiveness may be severely compromised by the lack of methods and tools to automate the composition steps. Given a description of a requested service and the descriptions of several available basic services, the problem is to create an *executable composition process* that satisfies the requested requirements and that can be programmatically deployed, discovered and invoked.

To achieve this goal, a composition process has to be able:

a) to perform the automatic and dynamic selection of a proper set of basic services whose combination provides the required capabilities;

b) to generate the process model that describes how to implement the requested service;

c) to translate the process model into an executable definition of the services composition, in case the selection is successful;

d) to verify the correctness of the definition of the composition;

e) to validate the composite web service against the initial description.

Each of these steps has its intrinsic complexity. Services descriptions, relations among the involved data and operations, composition definitions should be unambiguously computer-interpretable to enable the automation of web services discovery, selection, matching, integration, and then the verification and validation of web services compositions [63, 65].

To solve the automated composition problem, we propose an unifying composition development process. Our approach uses domain ontologies to describe operations, data and services, and aims at producing an executable process expressed by a standard workflow language that can be formally verified and validated.

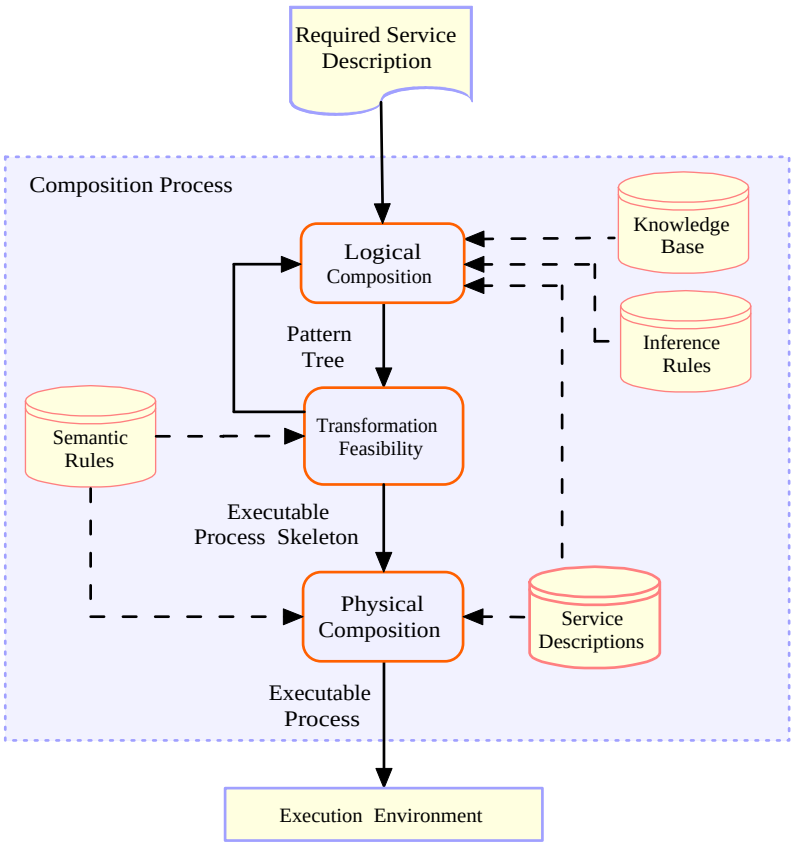


Figure 4.1: Life Cycle

The composition development process is realized in terms of the following phases:

1) *Logical Composition*. This phase provides a functional composition of service operations to create a new functionality that is currently not available.

2) *Transformation Feasibility*. This phase verifies the feasibility and the correctness of the transformation of the new functionality into a executable process expressed by a standard workflow language.

3) *Physical Composition*. This phase aims at producing an executable process, which is formally verified and validate.

This basic approach is illustrated in Figure 4.1. A *Service Description* contains the information about the services available *Domain*. The service functionalities are described formally, using a domain-specific terminology that is defined in the *Knowledge Base*. In particular, the *Knowledge Base* contains the domain specifications expressed by a *domain ontology* and two main Upper Ontologies for defining concepts related to the entities "Service" (e.g. concepts related to security, authentication, fault tolerance, etc.) and "Data" (type, casting, relationships among types, etc.). Moreover to evaluate the feasibility and the correctness of the transformation, we have formally defined a workflow language through operational semantics. That semantics is contained into *Semantics Rules* knowledge base.

When a new service have to be created, the user has to specify the *functional requirements* of the requested service using the concepts defined into the *Knowledge Base* and the workflow language to use to describe the executable process.

Driven by the user request specification, the *Logical Composition* module first synthesizes an *operational flow graph (OF)* by reasoning on the facts contained in the *knowledge base* and applying the *inference rules* (*IR*). Then, the *operation flow graph* is modified by inserting opportune service wrappers in order to resolve Input/Output mismatches. Finally, a graph transformation technique is applied to the *operation flow graph* in order to identify the workflow patterns which realize the composite service. The output of the first phase is a *pattern tree (PT)*, which will be consumed by the *Transformation Feasibility* module in the second phase.

The *Transformation feasibility* module checks by inferencing on the *semantic rules*, if the patterns (belonging to the *pattern tree*) can be realized by composing the language constructs. Generally, one or more construct combinations that realize the pattern tree are provided. The output of this phase is a representation of the pattern tree through the language constructs. If the pattern tree cannot be implemented, the user can choose either to repeat the process with more relaxed requests or to select another available composition language to implement the executable process.

Finally, in order to turn the abstract process into an executable process, the *Physical Composition* module generates the code by analyzing the PT and recursively associating proper construct skeletons to PT nodes. The output of this phase is an executable process, which can be verified before being enacted to detect syntax or semantic errors in the code.

An operational semantics is the formal basis of all composition development process phases: it is used:

- to define the relationships between operations and data;
- to express the flow of the operations which realize the composition goal;
- to identify the composition pattern described by the composition flow;
- to formalize the workflow language constructs in order to either evaluate the feasibility and the correctness of the transformation of the composition flow into an executable process and verify the syntax and semantics correctness of the obtained executable process;

- to support the validation of the composition.

4.1.1 Service Description

To enable automatic discovery and composition of desired functionalities, we need a language to describe the available web services. At the logical composition stage, the composition process typically involves reasoning procedures. To enable those, services need to be described in a high-level and abstract manner.

Therefore, at this stage it suffices to describe the capabilities of web services, using semantic annotations. Once the language is known, the basic terms used in the language have to be drawn from a formal domain model. This is required to allow machine-based interpretation while at the same time preventing ambiguities and interoperability problems. The DARPA Agent Markup Language (DAML, now called OWL)[1] is the result of an ongoing effort to define a language that allows creation of domain models or concept ontologies. We use it to create the domain model using which services are described. The OWL-S [5] markup language (previously known as DAML-S) is also being defined as a part of the same effort, for facilitating the creation of web service ontologies. OWL-S partitions the semantic description of a web service into three components: the *service profile*, *service model*, and *service grounding*(see Figure 4.2).

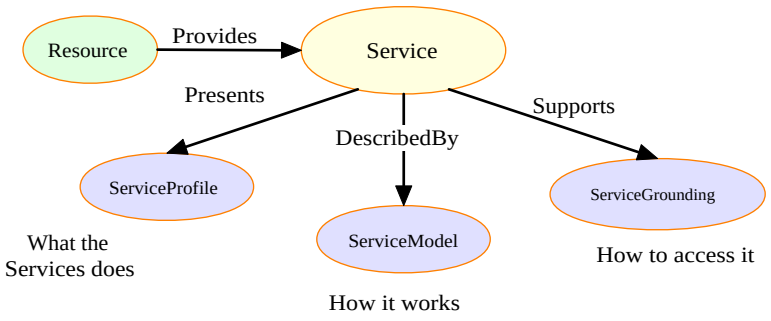


Figure 4.2: OWL-S Service Description

The *ServiceProfile* describes what the service does by specifying the input and output types, preconditions and effects(IOPE). The *ProcessModel* describes how the service works; each service is either an atomic process that is executed directly or a composite porcess that is a combination of subprocesses (i.e., a composition). The *Grounding* contains the details of how an agent can access a service by specifying a communications protocol, parameters to be used in the protocol, and the serialization techniques to be employed for the communication.

The proposed approach exploits OWL for the definition of the *Domain*, *Data* and *Service* ontologies and OWL-S for the description of service operations. We consider an operation as an atomic function described by the OWL-S Profile and Grounding[2]. Hence the proposed composition process takes into account the fact that a service may provide more operations.

The *Service Profile* descriptions are used during the logical phase either for the discovery and selection of the candidate service operations, and for the generation of the *OF* graph. Besides, the *Grounding* is used during physical composition for the generation of the executable process.

[1] http://www.daml.org/

[2] The Service Profile says what an operation does by specifying its IOPE, and Grounding is a mapping from OWL-S to WSDL [86]

4.2 Logical Composition

The main phases of the *Logical Composition* are the following:

1. Requested Service Description: service request expressed by an IOPE[3] description
2. Synthesis of the Operation-Flow graph (OF): the flow of the service operations that satisfies the Request is generated by reasoning on the facts contained in the *knowledge base* and applying the composition inference rules (stored in *Inference Rules*). Operations compatibility is guaranteed by matching Pre-conditions and Effects.
3. I/O Mapping: the input/output mappings among the parameters needed to invoke the operations provided by the selected services are considered. In this phase the Data Upper Ontology is used and the operation-flow graph is modified (if it is necessary), by inserting sequences of wrapping services;
4. Generation of the Pattern Tree of the composition (PT): graph transformation techniques are applied to the operation-flow graph in order to identify the workflow patterns which realize the composite service.

4.2.1 Requested Service Description

We suppose that the user's request contains a description of the service $\mathcal{W}$ to be provided, expressed by using the Domain Ontology concepts and relations. This description must specify the semantics of $\mathcal{W}$, its IOPE parameters and the conditions that it must verify. The conditions may be defined by means of the logical connectives of the propositional calculus. From this information, a Prolog Query $PQ_{\mathcal{W}}$ is generated.

The Preconditions and effects are used during the generation of the OF graph in the logical composition. The Inputs and Outputs are expressions involving general data types in the Data Ontology, which are used during the flow concretization in the physical composition. Moreover, they are used during the IO mapping in the the logical composition.

4.2.2 Synthesis of the Operations Flow Model

The synthesis of the operations flow model is achieved by analyzing the Request and the OWL-S definitions of the operations, and then issuing a Prolog query on $\mathcal{KB}$. The inference rules in $\mathcal{IR}$ defines services compatibility in terms of pre-conditions and effects.

The Operation Flow(OF) Graph is defined through *nodes* and *transitions*. Nodes are related to services to activate (*activities*) during the execution of the composed process, while edges (*transitions*) define precedences in services activations. Edges may be labeled with predicates (*conditions*) representing flows conditional activations. Each activity specifies also *SPLIT* and *JOIN* conditions. Split conditions are used to identify which (outgoing) flows have to be activated during process enactment when activities terminate. Join Conditions are used to define when an activity can be enacted when all or part of the activities on its incoming flows terminate. Basically three types of split and join conditions exist: AND, XOR and OR. For what concerning split conditions, (1) AND states that all outgoing flow paths have to be activated "simultaneously", (2) OR that all outgoing paths having transition conditions which evaluate *true* can be activated, (3) XOR that only one among outgoing paths has to be activated (depending on transition conditions). For

[3] Input, Output, Pre-condition and Effect

what regarding join conditions, (1) AND requires that the activity can be enacted only when all activities on all incoming paths have been terminated, (2) XOR allows for a single activity enactment when the first of the incoming paths is activated, (3) OR allows for multiple enactment of the activity (once every time an incoming path terminates).

IOPE Matching

Filtering and selection of services is achieved by using matchmaking algorithms similar to those implemented in the DAML-S Matchmaker [71]. In that system matchmaking uses ServiceProfiles to describe service requests as well as the services advertised. A service provider publishes a DAML-S (or, presumably in a successor, updated Matchmaker, OWL-S) description to a common service repository. When someone needs to locate a service to perform a specific task, a ServiceProfile for the desired service is created. Request profiles are matched by the service registry to advertised profiles using OWL-DL subsumption as the core inference service. In particular, the DAML-S Matchmaker computes subsumption relations between each individual IOPE's of the request and advertisement ServiceProfile. If the classes of the corresponding parameters are equivalent, there is an exact and thus best match. If there is no subsumption relation, then there is no match. Given a classification of the types describing the IOPEs, the Matchmaker assigns a rating depending on the number of intervening named classes between the request and advertisement parameters.

- **Perfect Matching**: predicate concepts are the same;
- **Exact**: predicate concepts are equivalent;
- **PlugIn**: a predicate concept is a subconcept of another one;
- **Subsume**: a predicate concept is a superconcept of anothe one;
- **Fail**: no matchs among predicate concepts.

Finally, the ratings for all of the IOPEs are combined to produce an overall rating of the match.

Our system uses the same basic typology of subsumption based matches, but we match based on the subsumption of the entire profiles.

Clearly, only **Exact** and **Plugin** matches between parameters of *ServiceProfile* would yield useful results. Sometimes, for input/output parameters we can consider a more relaxed matching that can be obtained considering the cases when the output of a service is subsumed by the input of another service. In fact the output type can be viewed as a specialized version of the input type and these services can still be chained.

Workflow Operators

The workflow constructs we use to build OF are the following: *sequence*, *split* and *join*. Sequence allows for sequential activation of operations; split and join allows for concurrent execution of operations and synchronization. Choices and loops can be introduced by means of proper conditions on OF edges.

In the following the PE matching and flow rules are formally defined.

We remember that *P* and *E* are sets of predicates, as explained in Section 4.1.1. Let $Predicate$ be the set of all predicates that appear in *P* and *E* sets of all operations in the domain and let σ be the set of evaluations of all predicates in $Predicate$. The function $eval(Predicate)$, for short's $eval$, associates to each element in the set $Predicate$ the couple $(predicate, value)$, where $value$ is the truth value of $predicate$.

In the following P_A (E_A) will denote the preconditions (effects) set associated to an operation A. In order to allow for A operation activation (i.e. to make A $activable$), all predicates in P_A must evaluate true. After a correct termination of the A operation, predicates in E_A evaluates true. In addition we call Act_A the activation of the operation A. Finally we indicate with Eff_A the set of all $Predicates$ that evaluates true after the Act_A:

$$Eff_A = \{p \in Predicates | eval(p) = (p, true) after Act_A\}$$

In the following the operational semantics of the *rules* we introduced before is reported. These definitions are translated into Prolog rules which are used during the synthesis phase. *Rules* are defined in terms of precondition that can enable the activation of a given operation composition and in terms of the changes in the σ set depending on *E* sets of component operations and composition operators.

The semantics of the activation of an operation A is the following:

$$\frac{eval(P_A)}{\sigma_A \xrightarrow{Act_A} \sigma_A^{'}} \tag{4.1}$$

where $\sigma_A = eval(Predicate)$ before the activation of the operation A and $\sigma_A^{'}$ is the same $eval(Predicate)$ but after the activation of the operation. Notice that only predicates in E_A may change their evaluation, and then $\sigma_A^{'} = eval(\neg(Predicate \cap E_A) \cup E_A)$

In order to synthesize the requested composed service all possible combinations of services are analyzed by the means of a Prolog engine, that tries to find a services composition to which corresponds the requested *P* and *E* sets.

Proper pruning techniques are used to allows for termination of the inferences even when loops are created in the problem state space exploration.

Sequence

In a sequence, an operation is activable after the completion of another operation in the same process. Let A and B two web services operations where the B operation can be activated only after the completion of the A operation. We denote with $Seq(A, B)$ the sequential activation of A and B where the activity order inside the brackets is related to the activation order.

In order to activate the Sequence, the following conditions must happen:

$$Eff_A \supseteq P_B, eval(p) = (p, true)$$
$$\forall p \in P_A \longrightarrow activable(Seq(A, B))$$

Obviously $P_{Seq} = P_A$ and $E_{Seq} \supseteq E_B$ because E_{Seq} contains all predicates in the last sequence operation, but also predicates that belong to the E sets of the other operations which maintain their truth values during the execution of the whole sequence (i.e., in sequence with two operations, the $p \in E_A$ such that $eval(p) = (p, true)$ even after the B execution). It is possible to prove that $E_A \cup E_B \supseteq E_{Seq}$. The relation between the sets is not an equality since it the B operation should request the invalidation of a previous effect. For example a service that first request an authentication for a session, can also request the end of the authentication session after the execution of a given task. The predicate *hasAuthentication* is an effect of the first operation, but not of the last one (and thus it is not one of the sequence).

It is also true that $Eff_{Seq} = Eff_B$.

Notice that associative property can be applied to Sequence operator, and it is possible to state that $Sequence(A, Seq(B, C)) = Seq(Seq(A, B), C) = Seq(A, B, C)$. In the case of multiple sequence component operations, the previous definition can be extended by recursion.

If L_n denotes a list of n operations $L_n = (A_1, \cdots, A_n)$ to execute in a sequence then:

$$Seq(L_n) = Seq(Seq(L_{n-1}), A_n)$$
$$Seq(A) = A$$

The semantics of a sequence composition is the following one:

$$\frac{\sigma_{Seq(L_{n-1})} \overset{Act(Seq(L_{n-1}))}{\longrightarrow} \sigma'_{Seq(L_{n-1})},\ Eff_{Seq(L_{n-1})} \supseteq P_{A_n}, \sigma'_{Seq(L_{n-1})} \overset{Act(L_n)}{\longrightarrow} \sigma'_{Seq(L_n)}}{\sigma_{Seq(L_n)} \overset{Act(Seq(L_n))}{\longrightarrow} \sigma'_{Seq(L_n)}} \tag{4.2}$$

(1) and (2) are the rules which define the execution of sequential web services operations. They state that in order to allow for sequential execution of a list of operations, the last one (A_n) has to be activable and the other ones have to be previously activated. In order to allows for last operation activation, it must be $Eff_{Seq(L_{n-1})} \supseteq P_{A_n}$.

For example, the rules are recursively applied for $Seq(A, B)$ in the following manner:

$$\frac{\frac{eval(P_A)}{\sigma_A \overset{Act(A)}{\longrightarrow} \sigma'_A}, Eff_A \supseteq P_B, \frac{eval(P_B)}{\sigma'_a \overset{Act(B)}{\longrightarrow} \sigma'_{Seq(A,B)}}}{\sigma_{Seq(A,B)} \overset{Act(Seq(A,B))}{\longrightarrow} \sigma'_{Seq(A,B)}}$$

Notice that $Eff_A \supseteq P_B \Longrightarrow eval(P_B)$, $\sigma_{Seq(A,B)} = \sigma_A$ and $\sigma'_{Seq(A,B)} = \sigma'_B$.

Split

A Split is a point in the OF with a single incoming control flow path and multiple outgoing paths (Figure 4.3). Three types of splits are defined in order to describe different kind of outgoing paths executions: AND, XOR and OR splits.

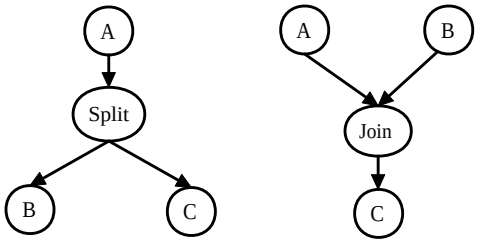


Figure 4.3: Split and Join

AND splits allow for parallel execution of outgoing paths. All preconditions needed to enact all paths executions have to be verified in order to consider the OF activable. XOR splits allow for the enactment of

only one of the outgoing paths. The preconditions of only one outgoing path have to be be verified in order to consider the OF activable. If more paths can be enabled, the system must choose one to activate. OR splits allow for the enactment of one or more of the outgoing paths at any time, as soon as paths enabling preconditions are evaluated true. Obviously any type of split with one outgoing edge has to be considered a sequence. For brevity's sake we will show in the following only the AND split activation conditions and semantics. Complex splits can be achieved by associating predicates called *conditions* on each outgoing path and discriminating their activation depending on these predicates values.

Let us consider the AND split in Figure 4.3 and let us indicate a split with $Split^{AND}_{(A,B,C)}$ where the first operation in the brackets is related to the incoming split path and the other ones to the outgoing paths . The condition that makes activable the AND split is the following one:

$$Eff_A \supseteq P_B \cup P_C$$
$$eval(p) = (p, true) \forall p \in P_A$$

Let us consider for simplicity's sake (here and in the other following constructs) that the E sets of split outgoing operations have no predicates that appear in one of the other set in the negate form. We can say that $P_{Split^{AND}_{(A,B,C)}} = P_A$ and, $E_{Split^{AND}_{(A,B,C)}} = E^*_A \cup E_B \cup E_C$, where E^*_A is the set of E_A predicates which do not appear in a negative form in E_B and E_C . The equality is because we assume that parallel outgoing operations cannot act concurrently executing conflicting operations.

More generally we denote with Inc the operation on the incoming path and with $Out_n = (O_1, \cdots, O_n)$ the operations on the outgoing paths, indicating the split with $Split^{AND}_{(Inc,Out_n)}$. Let be $Out_{n-1} = (O_1, \cdots O_{n-1})$, We can thus recursively define $Out_n = (Out_{n-1}, O_n)$, with $O_0 = \emptyset$ and $Split^{AND}_{(Inc,\emptyset)} = Inc$

Since we assume no conflicts on outgoing operations, the AND Split operator can be considered commutative on Out_n list. Thus if we denote with $Perm(Out_n)$ the set of all possible permutations on Out_n list, and with $Perm_i$ an element of this set,

$$Split^{AND}_{(Inc,Out_n)} = Split^{AND}_{(Inc,Perm_i)} \forall i \in \{1, n!\}$$

It is possible to associate the position in the list O_n with the order of operation completion. With the previous relation we state that the AND Split execution is independent on the completion order of outgoing operations.

It is now possible to describe the semantics of the AND Split:

$$Eff_{Inc} \supseteq P_{O_{n-1}},$$
$$\frac{\sigma_{Split^{AND}_{(Inc,Out_{n-1})}} \xrightarrow{Act(Split^{AND}_{(Inc,Out_{n-1})})} \sigma'_{Split^{AND}_{(Inc,Out_{n-1})}}}{\sigma_{Split^{AND}_{(Inc,Out_n)}} \xrightarrow{Act(Split^{AND}_{(Inc,Out_n)})} \sigma'_{Split^{AND}_{(Inc,Out_n)}}}$$

Notice that $\sigma'_{Split^{AND}_{(Inc,Out_{n-1})}}$ is the set of all predicates evaluations of all the Split operations evaluations except the operation O_n. This may resume the case of having all operations terminated but the O_n. Thanks to the commutative property previously described, this is not a loss of generality and the rule can be applied independently on outgoing operation terminations: the final state will be $\sigma'_{Split^{AND}_{(Inc,Out_n)}}$ in every case.

Furthermore,

$$Eff_{Inc} \supseteq \bigcup_{i=1,\cdots,n} P_{O_i} \Rightarrow$$
$$Eff_{Inc} \supseteq P_{O_i} \forall i \in \{1, \cdots, n\}$$

and the precondition $Eff_{Inc} \supseteq P_{O_{n-1}}$ is true at any level of inference tree.
We do not report the rules of other Split types due to the lack of space.

Join

A Join in the OF is a point with multiple ingoing paths and a single outgoing path (Figure 4.3). It is usually a synchronization point of concurrent or parallel activities. Three types of joins are defined in order to describe different kind of synchronization: AND; XOR and OR joins.

An AND join is a point where all operations on incoming paths have to terminate their execution in order to activate the outgoing control flow path. An XOR join allows for activation of the outgoing path whenever the operations of one of the incoming paths terminate their executions. Finally an OR join allows for activation of the outgoing path every time an incoming path operations terminate; the outgoing path can be activated more than once. In addition complex synchronization patterns can be defined by associating predicates (*conditions*) on incoming paths. The joins rules in such case apply only to paths with conditions that evaluate true. In the following for brevity's sake, we will describe only AND Join semantics.

With reference to the Figure 4.3 we denote with $Join^{AND}_{(A,B,C)}$ the activity with operation A and B on incoming join path and with operation C (the last in the join list) on the outgoing path. The conditions that make activable the AND join are the following:

$$Eff_A \cup Eff_B \supseteq P_C$$
$$eval(p) = (p, true) \forall p \in P_A \cup P_B$$

In order to activate the C operation, the paths with A and B operations must terminate. This implies that join can be activated only when all precondition of operations on incoming paths evaluate true and it is possible to state that: $P_{Join^{AND}_{(A,B,C)}} = P_A \cup P_B$. Further, if no conflicting operations on incoming paths, it can be trivially proved that $E_{Join^{AND}_{(A,B,C)}} = E^*_A \cup E^*_B \cup E_C$ where E^*_X is the set of all E_X predicates except for predicates that appear in the negative form in E_C. The definitions can be adapted to a generic number of operations on incoming paths (in a list Inc_n) and one on outgoing path (Out) like as we did previously with Split.

In order to define the semantics of Join, let us extend the semantics of the Act function to a list of operations. Let be Inc_n a list of n operations $(I_i, \cdots, I_n)$. We can extend the Act semantics as follows:

$$\frac{\sigma \xrightarrow{Act(Inc_{n-1})} \sigma'_{n-1}, \sigma'_{n-1} \xrightarrow{Act(I_n)} \sigma'}{\sigma \xrightarrow{Act(Inc_n)} \sigma'}$$

Where σ is the state before the activation of all operations, σ' is the state after the activation of all operations and σ'_i is the state after the activations of the first i activities in the Inc_n list. In brief the activation of n operations in a list evolves by activating component operations in turn.

With this definition and thanks to the commutative properties on incoming paths operations, it is possible to define the semantics of a Join:

$$\frac{\sigma_{Join^{AND}_{Inc_n,Out}} \xrightarrow{Act(Inc_n)} \sigma'_{Inc_n}, \quad \bigcup_{i\in\{1,\cdots,n\}} Eff_{I_i} \supseteq P_{Out}, \sigma'_{Inc_n} \xrightarrow{Act(Out)} \sigma'_{Join^{AND}_{(Inc_n,Out)}}}{\sigma_{Join^{AND}_{(Inc_n,Out)}} \xrightarrow{Act(Join^{AND}_{(Inc_n,Out)})} \sigma'_{Join^{AND}_{(Inc_n,Out)}}}$$

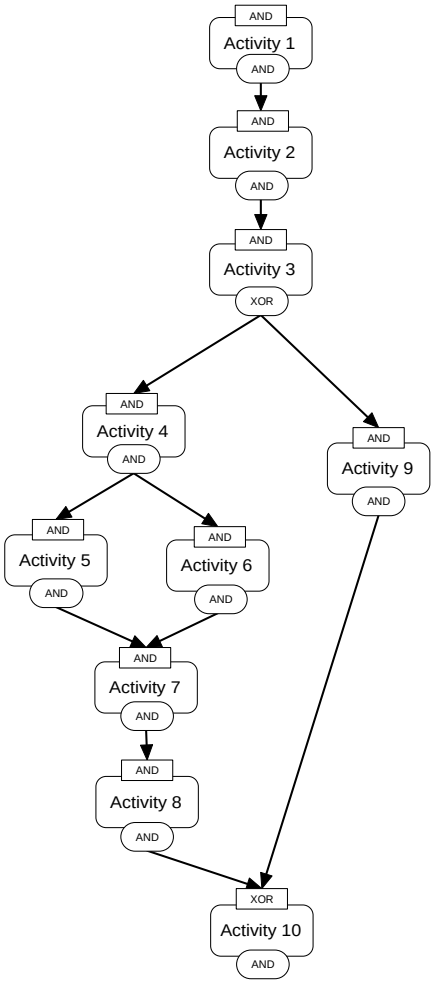


Figure 4.4: OF Graph

4.2.3 I/O Mapping

This section considers the input/output mapping among the parameters needed to invoke the operations provided by the selected services.

Since the previous synthesis process does not take in account the I/O operation descriptions, the OF graph may generate compositions that, even if semantically correct, may be incorrect in terms of Inputs

and Outputs. In fact, the generated OF graph describes the control flow(the dependence among activities) of the composite service, but generating the complete data flow needs reasoning with context of inputs and outputs. The data flow defines the dependences among data manipulations, and has to be produces between component services to make the composite model executable. For the definition of the data flow the semantics of each input and output parameter may be expressed. That semantics needs to be expressed along two dimension. The first one specifies the meaning of parameters as intended by the service provider. The second dimension is dictated by the composition of which this service become a components, so that the relation between the I/O parameters of component services can be determinated. To solve the problem of IO parameter context, we introduce a specific Data Upper ontology, which specifies the meaning, the type and relations among the types.

In this phase the complete data flow is obtained by reasoning on the Data Upper ontology and by inserting sequences of wrapping services(defined by the designer at design time) that execute I/O format translations.

4.2.4 Generation of the Pattern Tree of the Composition PT

Once the rules explained before have been applied in order to build the OF graph, it is necessary to translate this representation into a control flow graph which elements are organized in workflow patterns [88]. We will call this graph *Service Workflow* graph ($\mathcal{SW}$). This step of the approach is implemented applying the graph transformation algorithms defined in [43].

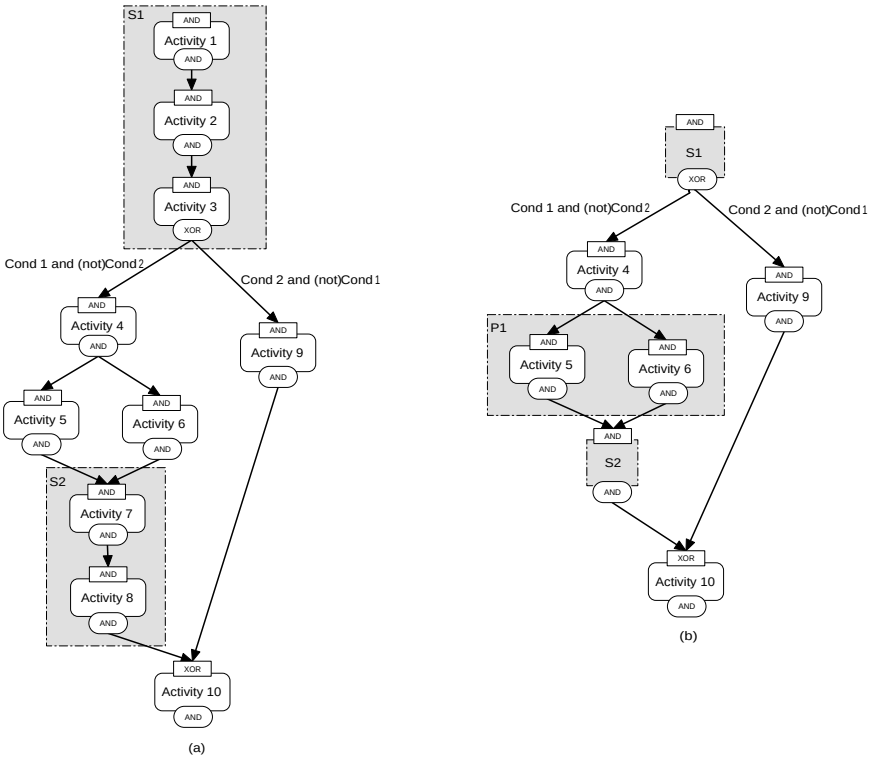


Figure 4.5: Graph Analysis 1

Let us suppose that we have to discover patterns in the graph reported in Figure 4.4. The steps required in order to discover pattern in the graph are the following. First of all (a), the Sequences **S1** and **S2** are

identified(see Figure 4.5 (a)). On the graph with the two sequences macro-nodes, the Parallel execution pattern **P1** is identified (see Figure 4.5 (b)). Then, the Sequence **S3** and the (Exclusive) Choice **C1** are identified (see Figure 4.6 (c-d)). Finally the whole graph is reduced to the sequence **S4** containing all the other detected patterns (see Figure 4.6 (e-f)).

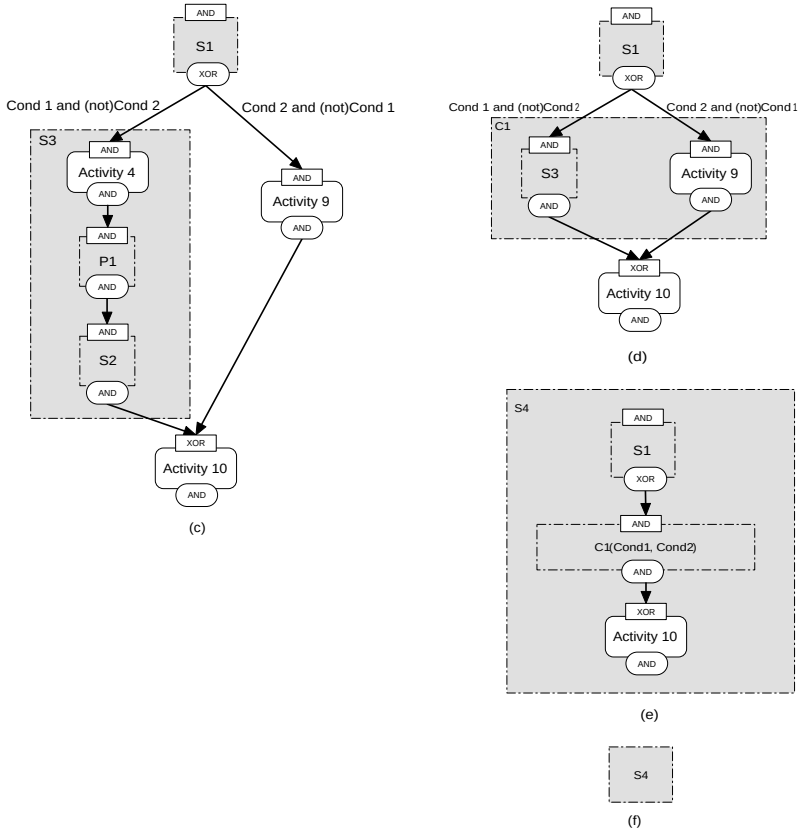


Figure 4.6: Graph Analysis 2

Some parts of the OF Graph can be expressed in terms of workflow patterns described in Section 4.3.3. The algorithm used to detect the workflow patterns belong to the operation flow graph is reported in Figure 4.7. It actually detect sequences, parallel executions and choices patterns.

The algorithm iteratively identifies sequences in the OF Graph. A Sequence is detected when a two or mode activities are linked each other by a single transition, independently from split and join conditions of component activities. Parallels and choice patterns are identified depending on transitions, join and split conditions of *three-level activities* OF subgraphs. A three-level activities subgraph is identified in the graph if two activities exist having respectively only one incoming(first-layer activity, FLA) and one outgoing(last layer activity, LLA) transition and if some other activities (middle-layer activities, MLA) exist having only LLA and FLA linked to their incoming and outgoing transitions.

The algorithm for sequences detection is reported in Figure 4.8. Notice that if a transition condition is specified on an activity, only at run time it is possible to state if the transition will be able to activate the following activities. This is notified as warning to the user during the OF Graph analysis.

The algorithm for detecting choices and parallels is reported in Figure 4.9.

The pattern macro node to substitute depends on the FLA split, LLA join and on transition conditions. The Table 4.1 resumes relations among conditions and detected patterns. Notice that defining a transition

repeat
identify sequences
create new nodes for sequences
substitute sequences nodes in the OF Graph
identify parallels and choices
create new nodes for parallels and choices
substitute parallels and choices in the OF Graph
until until no more substitution done
Envelop non-pattern nodes in $link\ nodes$
Produces $Pattern\ Tree$

Figure 4.7: OF Graph Analysis Algorithm

while Performing a Depth First Search on OF Graph **do**
if a Sequence macro-node was not active and current-activity has only one outgoing edge **then**
create a new active Sequence macro-node.
add the current-activity to Sequence macro-node list
$starting - activity = current - activty$
if a condition is defined on outgoing arc transition **then**
Warning notify
end if
else if a Sequence macro-node is active and current-activity has only one outgoing arc **then**
add the current-activity to Sequence macro-node list
if a condition is defined on outgoing arc transition **then**
Warning notify
end if
else
$ending - activity = current - activity$
de-activate Sequence macro-node
substitute all activities beetween $starting - activity$ and $current - activity$ with the Sequence macro-node
end if
end while

Figure 4.8: Sequences Detection

condition predicate as*true*, is the same of defining no condition.

The output of this phase is a tree (*Pattern Tree* (PT))containing the nested structure of workflow patterns that can be detected in the OF graph. Sometimes, it is simple to represent the output of the OF Transformation phase with a tree structure (since it is similar to a parse tree). In general some nodes and edges may not be enveloped into pattern nodes during the enactment of OF Graph analysis. In this case, these nodes and linked pattern macro-nodes are included into special *link nodes*.

```
while Performing a Depth First Search on OF Graph do
    detects three-level activities
    if a three-level activities group is detected then
        analyze FLA split, LLA join and transitions conditions
        Create a new Parallel/Choice Macro-node containing MLA.
        substitute all activities in MLA with the Parallel/Choice macro-node
        if any problem on transitions conditions is detected then
            Warning notify
        end if
    end if
end while
```

Figure 4.9: Parallel/Choice Detection

4.3 Transformation Feasibility

This phase verifies the feasibility and the correctness of the transformation of the Pattern tree into an executable process expressed by a standard workflow language.

To evaluate the feasibility of the transformation, we first need to analyze whether and how a given Pattern tree can be implemented using a workflow language. A common way to perform this analysis, is to verify if the language is able to express common workflow patterns. This methodology is presented in Section 4.3.1. However, a main problem in analyzing patterns is that orchestration languages lack of formal definitions of their constructs semantics. We have selected an orchestration language WS-BPEL[4], standard the facto in the composition of web services, and have formalized the semantics of the BPEL constructs (see section 4.3.2). Finally, we have been able to apply the pattern analysis methodology to BPEL in section 4.3.3.

4.3.1 Pattern Analysis Methodology

The proposed pattern analysis methodology is founded on the concept of constructs *operational semantics* of a given language.

The operational semantics of a workflow process describes the sequences of the computational steps generated by its execution by providing a state transition system for a language. Of course the a workflow language is not a programming language It is a flow language that allows to define executable processes. Nevertheless, its structured activities are very closed to the classical control flow constructs of an imperative programming language. Thus it makes sense to formalize them by defining a set of derivation rules for the execution of commands.

We use a structured operational semantics since the rules are syntax driven. The formal description of a language requires to define: a) a set of states, b) the evaluation of the expressions, c) the execution of the commands.

An expression that must be evaluated has the form:

$$\langle e, \sigma \rangle$$

where e is an expression and σ is the state in which e is evaluated. The evaluation produces a value n, according to the evaluation relation:

$$\langle e, \sigma \rangle \rightarrow n$$

[4]In this thesis for brevity we use the notation BPEL to refer to the standard WS-BPEL

Table 4.1: Choices and Parallels Detection rules

FLA Split	**LLA Join**	**FLA outgoing trans.conf**	**FLA outgoing trans.conf**	Pattern
AND	AND	all true (or Warning)	all true (or Warning)	Parallel: (Parallel Split + Synchronization)
AND	XOR	all true (or Warning)	all true (or Warning)	Parallel Split + Discriminator)
AND	OR	all true (or Warning)	any	(Multiple)Choice+ Multiple Merge
XOR	XOR	only one true (or Warning)	true on the active path (or Warning)	(Exclusive)Choice + Simple Merge
XOR	OR	only one true (or Warning)	true on the active path (or Warning)	Choice: (Simple Choice + Simple Merge)
OR	AND	all true (or Warning)	all true (or Warning)	Parallel Split + Synchronization)
OR	OR	at least one true (or Warning)	at least true on the active paths	(Multiple)Choice + Multiple Merge
OR	XOR	at least one true (or Warning)	at least true on the active paths	(Multiple)Choice + Discriminator

The pair:

$$\langle c, \sigma \rangle$$

where c is a command of the language, is said to be a *command configuration*. It specifies that the command c is executed in the state σ.

The execution of a command causes a state transition. The execution of a command may terminate in a final state or it may never reach a final state. The following evaluation relation means that the complete execution of c in the state σ terminates in the state σ':

$$\langle c, \sigma \rangle \rightarrow \sigma'$$

The evaluation relation is driven by the syntax of the language. The derivation rules say how to evaluate the expressions and the commands execution. A rule may define preconditions (that must be satisfied to apply the rule) and a conclusion:

$$\frac{preconditions}{conclusion}$$

The horizontal line reads ”implies”. Thus, rules represent logical truths. It follows that rules with nothing above the line are *axioms* since they always hold.

A *derivation tree* is a tree structure built by composing derivation rules so that the preconditions of each rule R are the conclusion of a rule R' that is a parent of R.

The derivation rules provides an algorithm to evaluate expressions and the behaviour of commands that is based on the search of a derivation tree.

Thus rules can be derived like *Prolog* rules by an inferential engine. For example, to evaluate the arithmetic expression $3 + (5 \times 2)$ in the state σ_0 it is necessary to: 1) evaluate the value of the literal 3;

2) evaluate the literal 5,2 and the value of the expression 5×2, 3) add the first value to the second one producing the result. The operational semantics of this operation is:

$$\frac{\frac{}{\langle 3,\sigma_0\rangle \rightarrow 3}, \frac{\langle 5,\sigma_0\rangle \rightarrow 5, \langle 2,\sigma_0\rangle \rightarrow 2}{\langle 5*2,\sigma_0\rangle \rightarrow 10}}{\langle 3+(5*2),\sigma_0\rangle \rightarrow 13}$$

The proposed methodology needs the definition of operational semantics of the language to analyze in order to state if a given workflow pattern can be described by the given language or not. As introduced in the previous sections, the main problem of workflow language for orchestration is that they lack a formal definition of their semantics.

So the proposed methodology has two advantages:

- It forces a formal description of semantics of a given language, defining once for all which kind of steps are needed to execute language constructs and not leaving its definition to workflow engine vendors.
- It allows a fully automatic way to investigate if a given pattern (or even a given workflow process) can be executed by a given language.

Our approach is different from the approach described in [88, 89]. In that approach the analyst has to know in which way a pattern can be implemented by using language constructs. It proceeds by example and without a fixed methodology.

The steps needed to perform the pattern analysis are:

1. A definition of the operational semantics of a given language must be provided.
2. The semantics rules must be translated into a *prolog* rule based system.
3. A description of the patterns to analyze as rules to prove in the *prolog* system must be provided.
4. The prolog system evaluates if a given pattern could be realized by composing the language constructs. In the case of affirmative results, the inferential engine also provides all the possible construct combinations to realize the pattern.

In the following we apply our methodology to the BPEL language.

4.3.2 BPEL Semantics

The first step of our approach requires the formal definition of the BPEL constructs and the subsequent implementation of derivation rules.

In the following the state of an activity a will be denoted by the symbol σ_a and the state of a link k will be denoted by the symbol σ_{L_k}.

Furthermore we will denote structured activities by:

$$A = \top s_0 a_1 s_1 \cdot \cdots \cdot s_{N-1} a_N s_M \bot$$

where:

- $\top$ is the activity that precedes the construct in the process,

- $\perp$ activity specifies that the construct is terminated and no more activities have to be processed within the construct,
- the symbols $s_i \in S$ denote a construct;
- $a_i \in L_A$ is the list of the activities specified within the construct.

The state σ_a of the basic activity a, can be:

- $ready$: a is ready to start;
- $exec$: a is started;
- $terminated$: a is ended; depending on the cause of the termination, this state can assume the values:
  - $noexec$: if the activity completes correctly without faults;
  - $undefined$: if the activity is terminated in an abnormal way or if it generates a not handled fault.

Notice that the state of a structured activity depends on the states of its component activities.
The state of a link k (σ_{L_k}) can assume the following values:

- $undefined$: the state of the link before the evaluation of its *Transition Condition*;
- $positive$: if the *Transition Condition* associated to the link evaluates *true*;
- $negative$: if the *Transition Condition* associated to the link evaluates *false*;

Derivation Rules

In the following the operational semantics for some of the BPEL language constructs are reported.

For brevity's sake in this paper we will only report the semantics of the sequence and flow (with links) constructs. The sequence is simple enough to let us explain how the semantic rules are defined and derived; flow with links is complex enough to describe the semantics of complex BPEL processes and to make possible the definition of a non trivial example.

In order to define the semantics of the BPEL constructs, it is needed to introduce some rules that are related to (implicit) constructs not explicitly provided by the BPEL language.

Implicit constructs

Let us introduce two basic transitions:

$$\frac{}{\langle a, \sigma_a = ready\rangle \xrightarrow{\mu} \langle a, \sigma'_a = exec\rangle}$$

$$\frac{}{\langle a, \sigma_a = exec\rangle \xrightarrow{\tau} \langle a, \sigma'_a \in terminated\rangle}$$

the μ and τ that respectively enable the execution and the termination of an activity:

Now let us introduce two operators frequently used in workflow languages: *split* and *join*. *Split* operator is used, when an activity terminates, to choose the next activity that may be activated depending on some boolean *conditions* defined over outgoing links. *Join* operator is similar to *split* operator but it applies to incoming links. It is important to notice that BPEL does not support explicitly these operators, but they are used to define links behaviors. For these reasons, here we define the semantics rules for *join* and *split*

operator.

As for the join operator, let L_A be a list of activities, in the $ready$ state, to analyze and L_L the list of the links defined in the process.

In the following, the recursive definition of a list will be used: *a list is either the empty list, or it is a head followed by a tail, where the head is an element and the tail is a list*. With this definition, it is possible to define the following rule:

$$\frac{}{L_A \overset{first}{\rightarrow} l_H \cdot L_T}$$

The $first$ transition extracts the first activity l_H from the list of the activities L_A. The L_T list is composed by the remaining activities (the tail).

With this definition it is possible to introduce the basic rules for join semantics:

Rule 1 (Join1):

$$\frac{L_A \overset{first}{\rightarrow} a_H \cdot L_{A_T}, \sigma_{a_H} = ready, \frac{\langle a_H, L_L \rangle \overset{vC}{\rightarrow} true}{\sigma_{a_H} \overset{\mu}{\rightarrow} \sigma_{a_H}^{exec}}}{\langle L_A, L_L, \epsilon_A, \sigma_A \rangle \overset{join}{\rightarrow} \langle L_{A_T}, L_L, a_H^{exec}, \sigma'_A \rangle}$$

A join of some activities in the state σ_A is enacted only if at least one activity exists such the status of all its incoming link in the L_A list is known, and the Join condition can be evaluated. This check is performed by the rule vC that is omitted for brevity's sake. If **Join1** evaluates $true$, the join operator, through the μ transition, allows the activation of the activity; changing the state σ_A into σ'_A that is the same of the state σ_A except for the state of the component activity $\sigma_{a_H}^{exec}$ that becomes *exec*.

If the join condition is $false$ the **Join2** and **Join3** rules must be applied and the sJF value is evaluated. sJF is used to establish if the *death-path-elimination* must be applied or if a standard fault must be propagated. *Death-path-elimination* does not allows the activation of other activities which are on the same path of the activity whose join condition evaluates false, assigning a negative status to its outgoing links.

Rule 2 (Join2):

$$\frac{L_A \overset{first}{\rightarrow} a_H \cdot L_{A_T}, \sigma_{a_H} = ready, \frac{\langle \sigma_{a_H}, L_L \rangle \overset{vC}{\rightarrow} false, sJF=no}{\sigma_{a_H} \overset{\mu}{\rightarrow} \sigma_{a_H}^{undefined}}}{\langle L_A, L_L, \sigma_A \rangle \overset{join}{\rightarrow} \langle L_{A_T}, L_L, \sigma'_A \rangle}$$

Rule 3 (Join3):

$$\frac{L_A \overset{first}{\rightarrow} a_H \cdot L_{A_T}, \sigma_{a_H} = ready, \frac{\langle \sigma_{a_H}, L_L \rangle \overset{vC}{\rightarrow} false, sJF=yes}{\sigma_{a_H} \overset{dPE}{\rightarrow} \langle \sigma_{a_H}^{undefined}, L_L^{New} \rangle}}{\langle L_A, L_L, \sigma_A \rangle \overset{join}{\rightarrow} \langle L_{A_T}, L_L^{New}, \sigma'_A \rangle}$$

The **Join3** rule handles join failure suppressions, performing death path elimination while the **Join2** rule put the activity with false transition condition into an undefined state.

The $Split$ operator is similar but concerning the activation of outgoing links of terminated activities and is omitted for brevity's sake. Another important construct, that we have defined to handle the termination

of the activities in BPEL is the $TauCostruct$ transition. The semantics of $TauCostruct$ transition is:

Rule 1 (Tau 1):

$$\frac{L_A \overset{first}{\rightarrow} a_H \cdot L_{A_T}, \sigma_{a_H} = exec, \frac{\sigma_{a_H} \overset{tau}{\rightarrow} \sigma_{a_H}^{noexec}}{\langle a_H, L_L \rangle \overset{Split}{\rightarrow} L_{L_{New}}}}{\langle L_A, L_L, \sigma_A \rangle \overset{TauCostruct}{\rightarrow} \langle L_{A_T}, L_{L_{New}}, \sigma'_A \rangle}$$

Rule 2 (Tau 2):

$$\frac{L_A \overset{first}{\rightarrow} a_H \cdot L_{A_T}, \sigma_{a_H} = exec, \frac{\sigma_{a_H} \overset{tau}{\rightarrow} \sigma_{a_H}^{undefined}}{\langle a_H, L_L \rangle \overset{Split}{\rightarrow} L_{L_{New}}}}{\langle L_A, L_L, \sigma_A \rangle \overset{TauCostruct}{\rightarrow} \langle L_{A_T}, L_{L_{New}}, \sigma'_A \rangle}$$

L_A is the list of activities to analyze, L_L is the list of the **links** and $L_{L_{New}}$ is the list of links activated after a $split$ operation. A **TauCostruct** transition is activated when at least one activity is no more in execution and applies the split transition to a_H outgoing links to verify if some other activity can be execute. This operation changes the state of the process links and may activate some other activities. Since this transition terminates an activity, its state becomes $noexec$, if the activity termination is normal, $undefined$ otherwise. The state of process activities changes from σ_A to σ'_A where σ'_A is the same of σ_A except for the state of the activity σ_{a_T} (because the τ transition changes the state of the a_T activity).

It is now possible to introduce the sequence and flow constructs:

Sequence

Let be :

$$S = \top \cdot a_1 \cdot a_2 \cdot a_3 \cdots a_n \cdot \bot$$

the definition of a sequence of activities a_i. $\top$ activity is the activity that precedes the sequence in the process and the $\bot$ activity specifies that S is terminated and no more activities have to be processed.

Let L_A be the activity list of S and σ_s the state of S. A sequence activity may be in execution (state $exec$) or not (state $noexec$). Let be σ_a the state of the a activity in the sequence ($exec$ or $noexec$). The state of the sequence depends on the state its component activities:

$$\begin{aligned} \sigma_s &= exec \Leftrightarrow \forall i, j \in \{1, \cdots, n\} \sigma_{a_i} = exec, \sigma_{a_j | j \neq i} = ready; \\ \sigma_s &= noexec \Leftrightarrow \forall i \in \{1, \cdots, n\} \sigma_{a_i} = noexec \end{aligned}$$

The operational semantics specification of the *sequence* construct consists of the following four rules:

Rule 1 (S1):

$$\frac{\sigma_S = ready, L_A \overset{first}{\rightarrow} \top \cdot L_{A_T}, \sigma_S \overset{\mu}{\rightarrow} \sigma_S^{exec}}{\langle L_A, \sigma_S \rangle \overset{sequence}{\rightarrow} \langle L_{A_T}, \sigma_S^{exec} \rangle}$$

This rule applies when the sequence is not started yet. In such case the first activity of the sequence is the $\top$ activity. The *first* transition put in the L_{A_T} list the remaining sequence activities and the state of the sequence through the application of the μ becomes $exec$. The new state of the sequence is then σ_S^{exec} and

the remaining activities to process (L_{A_T}) were pruned of the $\top$ activity.

Rule 2 (S2):

$$\frac{\sigma_S = exec, L_A \overset{first}{\rightarrow} a_H \cdot L_{A_T}, \sigma_{a_H} = ready, \sigma_{a_H} \overset{\mu}{\rightarrow} \sigma_{a_H}^{exec}}{\langle L_A, \sigma_S \rangle \overset{sequence}{\rightarrow} \langle L_A, \sigma'_S \rangle}$$

This rule applies when the sequence is already started and the *first* activity in the sequence is executed:
Rule 3 (S3):

$$\mathbf{S3.1}\ \frac{\sigma_S = exec, L_A \overset{first}{\rightarrow} a_H \cdot L_{A_T}, \sigma_{a_H} = exec, \sigma_{a_H}^{exec} \overset{\tau}{\rightarrow} \sigma_{a_H}^{noexec}}{\langle L_A, \sigma_S \rangle \overset{sequence}{\rightarrow} \langle L_{A_T}, \sigma'_S \rangle}$$

$$\mathbf{S3.2}\ \frac{\sigma_S = exec, L_A \overset{first}{\rightarrow} a_H \cdot L_{A_T}, \sigma_{a_H} = exec, \sigma_{a_H}^{exec} \overset{\tau}{\rightarrow} \sigma_{a_H}^{undefined}}{\langle L_A, \sigma_S \rangle \overset{sequence}{\rightarrow} \langle L_A, \sigma'_S = undefined \rangle}$$

In this case the activity a_H is terminated and its state can evolve:

1. from $exec$ to $noexec$, if the termination is normal;
2. from $exec$ to $undefined$, if a fault has occurred.

With this rule the *first* activity processes the activity a_H and the L_A list is pruned of this activity. The remaining activity list is called L_{A_T}. Notice that, in the first case the list of activities to process remains unchanged and only the sequence of states is changed; in the second case the $sequence$ termination is abnormal and its state is $undefined$.

Rule 4 (S4):

$$\frac{\sigma_S = exec, a = \bot, \sigma_S \overset{\tau}{\rightarrow} \sigma_S^{noexec}}{\langle a, \sigma_S \rangle \overset{sequence}{\rightarrow} \langle a, \sigma_S^{noexec} \rangle}$$

This rule applies when the next activity in the sequence to process is $\bot$. In such case the sequence state becomes $noexec$ and the sequence activity ends.

Axiom 1 (SA1)

$$\frac{}{\langle \bot, \sigma_S \rangle \overset{sequence}{\rightarrow} END}$$

this axiom states that if no more activities are in the list of activities to process, the sequence ends.

Axiom 2 (SA2):

$$\frac{\sigma_S = undefined}{\langle L_A, \sigma_S \rangle \overset{sequence}{\rightarrow} END_{undefined}}$$

```
1  S: Sequence
2  end=false;     fault=false;
3  while (!end && ! fault)
4  {
5          if (first (S)==⊤ )
6          {
7                  apply S1;
8          }
9          else if (first(S)==⊥)
10         {
11                 apply S4;
12                 end=true;
13                 apply SA1;
14          }
15         else
16         {
17                 apply S2;
18                 if (fault_occurred)
19                 {
20                         apply S3.2;
21                         fault=true;
22                         apply SA2
23                 }
24                 else
25                 {
26                         apply S3.1;
27                 }
28         }
29
30 }
```

Figure 4.10: pplication of the rules for a sequence construct

This axiom states that the state of $sequence$, σ_S is $undefined$, because a fault has occurred.

The rules for a sequence construct are applied as in Figure 4.10:

S1 is applied to start a sequence; no component activity of the sequence is yet started. **S2** is applied the first time to start the first activity in the sequence. **S3** is applied when the activity executed in the previous step is terminated. If this activity does not complete correctly the **S3.2** and **SA2** rules are applied and the sequence terminates in an *undefined* state, otherwise the other rules can be applied to the new list of activities to process that is the previous one pruned of the first activity. If the current list of activities to process contains only the $\perp$ activity the **S4** and **SA1** rules are applied and the process ends, else the **S2** rule at point is applied on the remaining activities.

Flow

Let be :

$$F = \top \| a_1 \| a_2 \| a_3 \cdots \| a_n \| \bot$$

Since it is no possible to start all activities simultaneously, a way to define an order of activation must be provided. In our definition the order is derived from the value of the index i. Thus the *first* transition can be applied also to flow rules. In addition it is necessary to remember all activities enacted and all activities that need to be started. The activities may terminate their execution only when all of them were started. Anyway activities execution is intended to be concurrent. Likewise for sequence, the state of the flow depends on the state of each component activity:

$$\begin{aligned} \sigma_F &= exec \Leftrightarrow \exists i \in \{1, \cdots, n\} : \sigma_{a_i} = exec \\ \sigma_F &= noexec \Leftrightarrow \forall i \in \{1, \cdots, n\} : \sigma_{a_i} = noexec \end{aligned}$$

If the $flow$ is defined with the $links$, the following semantics rules apply:

Rule 1 (F1):

$$\frac{\sigma_F = ready, L_A \overset{first}{\rightarrow} \top || L_{A_T}, \sigma_F \overset{\mu}{\rightarrow} \sigma_F^{exec}}{\langle L_L, L_A, \sigma_F \rangle \overset{flow}{\rightarrow} \langle L_L, L_A, \sigma_F^{exec} \rangle}$$

This rule applies when the $flow$ is not yet started. In such case the first activity of the $flow$ is the $\top$ activity. The *first* transition put in the L_{A_T} list the remaining $flow$ activities and the state of the $flow$ through the application of the μ becomes $exec$. The new state of the $flow$ is then σ_F^{exec} and the remaining activities to process (L_{A_T}) were pruned of the $\top$ activity.

Rule 2 (F2):

$$\mathbf{F2.1}\ \frac{\sigma_F = exec, \langle L_A, L_L, \sigma_F \rangle \overset{join}{\rightarrow} \left\langle L_A^{New}, L_L^{New}, \sigma_F^{New} = exec \right\rangle}{\langle L_L, L_A, \sigma_F \rangle \overset{flow}{\rightarrow} \left\langle L_A^{New}, L_L^{New}, \sigma_F^{New} \right\rangle}$$

$$\mathbf{F2.2}\ \frac{\sigma_F = exec, \langle L_A, L_L, \sigma_F \rangle \overset{join}{\rightarrow} \left\langle L_A^{New}, L_L^{New}, \sigma_F^{New} = undefined \right\rangle}{\langle L_L, L_A, \sigma_F \rangle \overset{flow}{\rightarrow} \left\langle L_A^{New}, L_L^{New}, \sigma_F^{undefined} \right\rangle}$$

This rules applies when the $flow$ is already started. At this point, the $Join$ transition can be applied. It starts only the activities whose $TransitionCondition$ is evaluated true. The state of $flow$ changes to:

- σ_F^{exec}: this state is the same of the state σ_F except for the state of some activities that become $exec$;
- $\sigma_F^{undefined}$: if a fault has occurred and it not is handled.

Rule 3 (F3):

$$\mathbf{F3.1}\ \frac{\sigma_F = exec, \langle L_A, L_L, \sigma_F \rangle \overset{TauCostruct}{\rightarrow} \left\langle L_A^{New}, L_L^{New}, \sigma_F^{New} = exec \right\rangle}{\langle L_L, L_A, \sigma_F \rangle \overset{flow}{\rightarrow} \left\langle L_A^{New}, L_L^{New}, \sigma_F^{New} \right\rangle}$$

$$\mathbf{F3.2}\ \frac{\sigma_F = exec, \langle L_A, L_L, \sigma_F \rangle \overset{TauCostruct}{\rightarrow} \left\langle L_A^{New}, L_L^{New}, \sigma_F^{New} = undefined \right\rangle}{\langle L_L, L_A, \sigma_F \rangle \overset{flow}{\rightarrow} \left\langle L_A^{New}, L_L^{New}, \sigma_F^{undefined} \right\rangle}$$

This rules applies to terminate at least one of the activities in execution. The $TauCostruct$ must be applied.

Rule 4 (F4):

$$\frac{\sigma_F = exec, \langle L_L, L_A \rangle \overset{AnalisysStatusLink}{\rightarrow} true, \sigma_F \overset{tau}{\rightarrow} \sigma_F^{noexec}}{\langle L_L, L_A, \sigma_F \rangle \overset{flow}{\rightarrow} \langle L_L, L_A, \sigma_F^{noexec} \rangle}$$

Rule 5 (F5):

$$\frac{\sigma_F = exec, \langle L_L, L_A \rangle \overset{AnalisysStatusLink}{\rightarrow} false, \sigma_F \overset{tau}{\rightarrow} \sigma_F^{undefined}}{\langle L_L, L_A, \sigma_F \rangle \overset{flow}{\rightarrow} \left\langle L_L, L_A, \sigma_F^{undefined} \right\rangle}$$

The $AnalisysStatusLink$ checks if the flow can terminate correctly depending on link status and transition conditions. In case of $DeathPathElimination$ application, dead links conditions are propagated in the flow construct.

These rules respectively apply if: F4) all started activities inside the $flow$ are terminated and the $Join$ transition does not start any activity; F5) Errors occur in the flow definition. In this case the state of the flow activity becomes $undefined$.

Rule 6 (F6):

$$\frac{a = \bot, \sigma_F = exec, \sigma_F \overset{tau}{\rightarrow} \sigma_F^{noexec}}{\langle L_L, a, \sigma_F \rangle \overset{flow}{\rightarrow} \langle L_L, \bot, \sigma_F^{noexec} \rangle}$$

This rule applies if the only activity in the flow to start is the $\bot$ activity. In such case the $flow$ state becomes $noexec$ through the τ transition.

Axiom 1(FA1):

$$\frac{}{\langle L_L, \bot, \sigma_F = noexec \rangle \overset{flow}{\rightarrow} END}$$

This axiom states that if no more activities are in the list of activity to process, the $flow$ ends.

Axiom 2(FA2):

$$\frac{}{\langle L_L, L_A, \sigma_F = noexec \rangle \overset{flow}{\rightarrow} END}$$

This axiom states that $flow$ ends correctly.

Axiom 3(FA3):

$$\frac{\sigma_F = undefined}{\langle L_L, L_A \rangle \overset{flow}{\rightarrow} END_{undefined}}$$

This axiom states that the state of $flow$ is σ_F = undefined, because a fault has occurred.
The rules for a flow construct are applied as in Figure 4.11:

```
1  F:Flow
2  end=false; fault= false;
3
4  while (!end && !flow)
5  {
6          if (first(F)==⊤
7                  apply F1;
8          else
9          {
10                 apply F2.1 or F2.2 to all activities ready in F;
11                 apply F3.1 or F3.2 for all activities that can terminate;
12                 if (fault_occurred)
13                 {
14                         fault=true;
15                         apply F5,FA2;
16                 }
17                 else if ( all activities were executed)
18                 {
19                         end= true;
20                         apply F6,FA3;
21                 }
22                 else if (F ended due to Links States Analysis)
23                 {
24                         end=true;
25                         apply F4;FA1;
26                 }
27         }
28 }
```

Figure 4.11: Application Of the rules for the flow construct

F1 is applied to start a flow when no component activity of the Flow is yet started. **F2** (2.1 and 2.2) are then recursively applied when starting new activities in the flow depending on the execution states of the other activities and **F3** (3.1 and 3.2) are recursively applied when terminating the activities in the flow. **F4** is applied when terminating the flow activity in presence of general activities; **FA1** is applied too and the derivation process ends. **F5** is applied when terminating the flow activity incorrectly; **FA2** is applied too and the derivation process ends. **F6** is applied when all activities in the flow were terminated correctly and the $\perp$ activity is examined; the **FA3** is applied and the process ends;

The definition of flow rules with the presence of **links** is more complicated and is omitted for brevity sake.

IF

Let be :

$$IF = \top \rightleftharpoons c_1 \rightleftharpoons c_2 \rightleftharpoons c_3 \cdots \rightleftharpoons c_n \| \bot$$

the representation of a switch activity.

$c_i = [C_{a_i}, a_i], i \in \{1, \cdots, n-1\}$ represents the *case* of index i, where C_{a_i} is the condition that enables the execution of the activity a_i in the case. The last term of the case : $c_n = [C_{else}, a_{Ow}]$ is the else case, that is executed if all conditions of other cases are false.

Likewise for sequence, the state of the 'if' depends on the state of each component activity:

$$\begin{aligned} \sigma_{If} &= exec \Leftrightarrow \{\exists c_i = [C_{a_i}, a_i] : C_{a_i} = true \\ &\quad \wedge \sigma_{If_{a_i}} = exec, i \in \{1, \cdots, n-1\}\} \vee \sigma_{If_{a_{Ow}}} = exec \\ \sigma_{If} &= noexec \Leftrightarrow \forall i \in \{1, \cdots, n\} : \sigma_{If_{a_i}} = noexec \end{aligned}$$

Let be L_C the list of cases c_i to enact. Thus the following rules can be applied for 'if' construct:

Rule 1 (IF1):

$$\frac{\sigma_{If} = noexec, L_C \overset{first}{\rightarrow} \top \rightleftharpoons L'_C, \sigma_{If} \overset{\mu}{\rightarrow} \sigma_{If}^{exec}}{\langle L_C, \sigma_{If} \rangle \overset{If}{\rightarrow} \left\langle L'_C, \sigma_{If}^{exec} \right\rangle}$$

This is similar to **S1** and **F1**.

Rule 2 (If2):

$$\frac{\sigma_{If} = exec, L_C \overset{verifyCond}{\rightarrow} L'_C, \sigma_{If} \overset{\mu}{\rightarrow} \sigma'_{If}}{\langle L_C, \sigma_{If} \rangle \overset{If}{\rightarrow} \left\langle L'_C, \sigma_{If}^{exec} \right\rangle}$$

This rule applies when the 'if' activity has started and when no condition was yet verified. The transition *verify Cond* verifies conditions C_{a_i} in order to choose the activity a_i to start. The state of this activity becomes $exec$ and the global state of the 'if' activity changes by applying the μ. Notice that the L'_C list contains only the a_i activity.

Rule 3 (IF3):
(IF 3.1)

$$\frac{\sigma_{If} = exec, L_C \overset{first}{\rightarrow} a \cdot \bot, \sigma_{If_{a_i}}^{exec} \overset{\tau}{\rightarrow} \sigma_{If_{a_i}}^{noexec}, \sigma_{If}^{exec} \overset{\tau}{\rightarrow} \sigma'_{If}}{\langle L_C, \sigma_{If} \rangle \overset{If}{\rightarrow} \left\langle \bot, \sigma_{If}^{exec} \right\rangle}$$

(IF 3.2)

$$\frac{\sigma_{If} = exec, L_C \overset{first}{\rightarrow} a \cdot \bot, \sigma_{If_{a_i}}^{exec} \overset{\tau}{\rightarrow} \sigma_{If_{a_i}}^{undefined}, \sigma_{Sw}^{exec} \overset{\tau}{\rightarrow} \sigma'_{If}}{\langle L_C, \sigma_{If} \rangle \overset{If}{\rightarrow} \left\langle \bot, \sigma_{If}^{undefined} \right\rangle}$$

This rule terminates the execution of the activity a_i executed in the 'if' by applying the rule **IF2**. The state σ_{If} obviously of the 'if' changes.

In this case the activity a_i is terminated and its state can evolve:

1. from $exec$ to $noexec$, if the termination is normal;
2. from $exec$ to $undefined$, if a fault has occurred.

Notice that, in the second case the if termination is abnormal and its state is $undefined$.

Rule 4 (IF4):

$$\frac{\sigma_{IF} = exec, a = \top, \sigma_I F \xrightarrow{\tau} \sigma_{IF}^{noexec}}{\langle a, \sigma_{IF} \rangle \xrightarrow{IF} \langle top, \sigma_{IF}^{noexec} \rangle}$$

This rule changes the state of the 'if' from $exec$ to $noexec$ if the case activity was terminated and only the $\perp$ activity remains to be executed.

Axiom 1 (IFA1):

$$\frac{}{\langle \perp, \sigma_{IF} \rangle \xrightarrow{IF} END}$$

this axiom states that if no more activities are in the list of activity to process and if the state of the 'if' is $noexec$, the 'if' ends.

This axiom states that if no executed activity has been processed to process, the 'if' ends.

Axiom 2 (IFA2):

$$\frac{\sigma_I F = undefined}{\langle L_A, \sigma_I F \rangle \xrightarrow{IF} END_{undefined}}$$

This axiom states that the state of if, σ_S is $undefined$, because a fault has occurred.

The rules for a 'if' construct are applied as in Figure 4.12:

IF1 is applied to start the 'if'; no component activity of the 'If' is yet started. **IF2** is applied the first time to start the first activity in the 'If'. **IF3** is applied when the activity for which the condition is true is executed in the previous step is terminated. If this activity does not complete correctly the **IF3.2** and **IFA2** rules are applied and the 'if' terminates in an *undefined* state. Otherwise, the current list of activities to process contains only the $\perp$ activity the **IF4** and **IFA1** rules are applied and the process ends.

Pick

The Pick activity is defined through couples $(messages, activity)$. If the pick is active and an external message arrives, the related activity is executed. Is is similar to the if activity but the message is not known when activity starts.

```
1  If: If
2  end=false;     fault=false;
3  while (!end && ! fault)
4  {
5          if (first (If)==⊤ )
6          {
7                  apply IF1;
8          }
9          else if (first(S)==⊥)
10         {
11                 apply IF4;
12                 end=true;
13                 apply IFA1;
14          }
15         else
16         {
17                 apply IF2;
18                 if (fault_occurred)
19                 {
20                         apply IF3.2;
21                         fault=true;
22                         apply IFA2
23                 }
24                 else
25                 {
26                         apply IF3.1;
27                 }
28         }
29
30 }
```

Figure 4.12: Application of the rules for the if construct

Notice that a pick activity may wait infinitely the arrive of a message. At this purpose, an $alarm$ activity may be defined in order to terminate the pick after a given time. When one of the pick activity terminates, the pick terminates too.

The pick semantics is similar to the switch semantics but refers to a couple $e_i = [E_{a_i}, a_i]$ instead of c_i, where E_{a_i} are the events waited by the pick activity.

All the semantic rules were traduced into a prolog prolog program as described in section 4.3.1. We omit the description of this program because of its complexity.

4.3.3 Pattern Analysis of BPEL4WS

In the following we will show results from the analysis of workflow patterns. After a brief description of the workflow pattern, we will show how, describing patterns as rules to prove in the prolog system, several implementations for patterns were discovered.

WP1

Description An activity in a workflow process is activated after the completion of the previous activity in the same process. In Figure 4.13, the sequence of the activities **A** and **B** is depicted. Let we can observe, **A** and **B** are in sequence since there is a control flow edge from **A** to **B** which has no conditions associated with it.

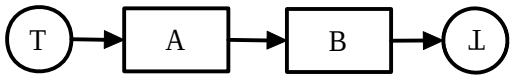


Figure 4.13: Sequence Pattern

Example The flight ticket is issued after the payment transaction is correctly executed.

Motivation The *Sequence* pattern is used to construct a series of consecutive activities which execute in turn one after an other.

Formalization The behavior of the sequence pattern depicted in Figure 4.13 is the following: If the *Sequence* is already started, the activity **A** can be executed. After the termination of the activity **A** the next activity in the *Sequence* e.g. **B** will be executed. *Sequence* ends when all the activities belong to it will terminate.

BPEL Implementation

Many solutions to implement this pattern with BPEL4WS were discovered. In Figure 4.14 three different implementations of this pattern are shown. In Figure 4.14 a) simply the *sequence* activity is used. Inside the sequence, to execute the **A** and **B** web services, two **invoke** activities are used.

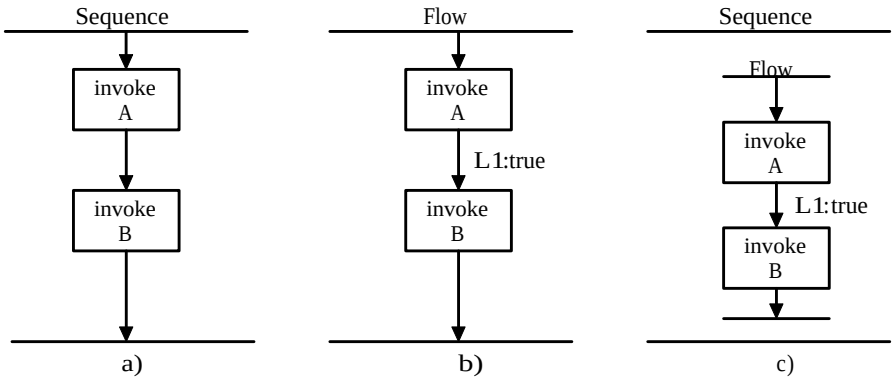
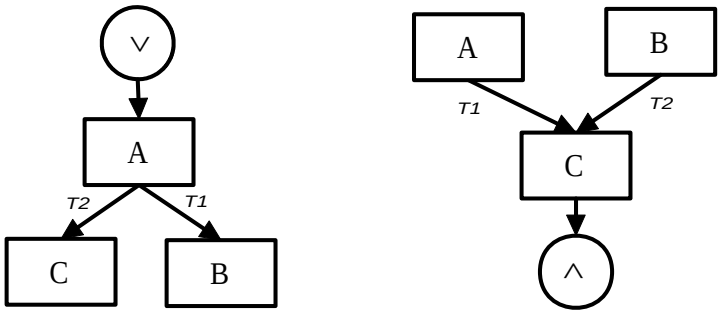


Figure 4.14: WP1 Pattern Implementation

In Figure 4.14 b) and c) alternative implementations for the pattern are shown. The first one is realized by using the *flow* activity and **links** introducing a precedence relation between the **A** and **B** invoke activities. The *invoke A* activity is the SourceLink for the link **L1** (with the transition condition true) while the *invoke B* is the TargetLink for the same link. In this way the activity *invoke B* can be executed only after the

termination of the *invoke A* activity. The implementation of Figure 4.14 c) is simply a sequence activity with only one component activity: the flow depicted in Figure 4.14 b). This is not an efficient implementation but it is reported to show how our methodology is able to find all implementations in a given language of a workflow pattern. The other implementations are not reported here and in the following for brevity sake.

WP2 - Parallel Split



a) Parallel Split b) Synchronization

Figure 4.15: Parallel Split and Synchronization Patterns

Description It is a point in the process where a single control thread is split in two or more threads each of which execute concurrently. This is the case of Figure 4.15a) where, after the completion of the activity **A**, two distinct threads are activated simultaneously and activities **B** and **C** are executed in parallel.

Example Once a login session to the payment system is established, it is possible to control the credit card limits and the credit availability simultaneously.

Motivation The *Parallel Split* pattern allows a single thread of execution to be split into two or more branches which can be executed concurrently. These branches may or may not be re-synchronized at some future time.

Formalization The behavior of the *Parallel Split* pattern depicted in Figure 4.15 a) is the following. When the parallel split is already started, the the activity **A** has been executed and the activities **B** and **C** may be activated(their state is ready). When the activity **A** terminates, the *Split* construct activate two distinct transitions, t_1 and t_2, and then the activities **B** and **C** are executed. The *Split* will terminate, when both the activities **B** and **C** will be ended.

WP3 - Synchronization

Description It is a point in the process where two or more concurrent control flows converge in an unique thread of control. This is the case of Figure 4.15 b), where after the completion of both **A** and **B** activities, the activity **D** is activated.

Example The payment transaction can be completed only if both card limits and credit availability activities are verified.

Motivation *Synchronization* provides a means of reconverging the execution of more concurrent threads. Generally, these threads are created from a *Parallel Split* pattern.

Formalization The behavior of the *Synchronization* pattern depicted in Figure 4.15 b) is the following. When both the activities **A** and **B** have already been executed and the activity **C** is ready to be activated(its state is ready). When both the activities **A** and **B** will terminate, the *Synch* construct will perform the activation of both the transitions t_1 and t_2 and the activity **C** will be executed. If one ncoming branch will fail the construct could give a failure. Notice that, the behavior of the pattern is not defined if an incoming branch is executed more than ones for a given case. Therefore, the semantics of the construct does not consider the possibility to trigger again an incoming branch that has previously been completed. Also, the synchronization will be in deadlock if one or more of incoming branches will not correctly complete.

BPEL Implementation - WP2 and WP3

Solutions for both parallel split and synchronization patterns are depicted in Figure 4.16.

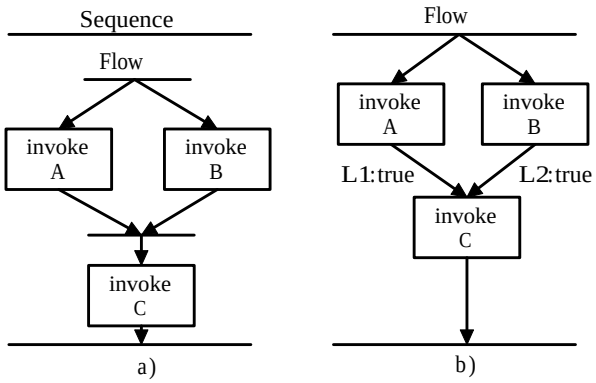
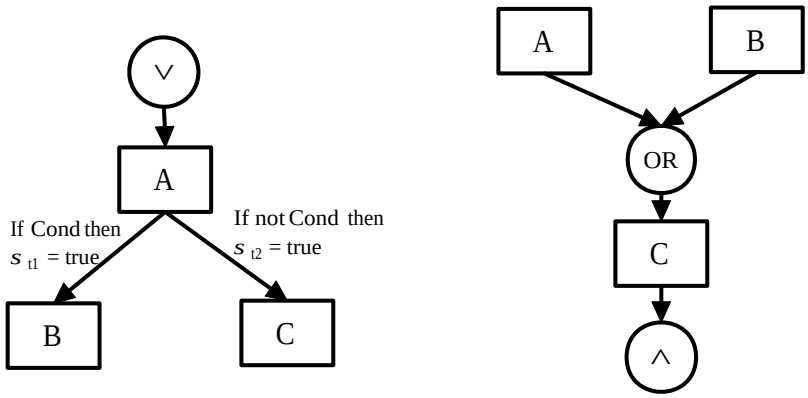


Figure 4.16: WP2-WP3 Pattern Implementation

The parallel split is supported natively by the **flow** construct. Both in Figure 4.16 a) and b), the parallel split of the activity *invoke A* and *invoke B* are implemented by including them in a **flow**. For synchronization, instead, two different implementation are shown. In Figure 4.16 a), the flow is the first activity included in a sequence. When the flow terminates, the activity *invoke C* is executed, implementing the synchronization of the two control flow paths spawned by the flow activity. In Figure 4.16 b), instead, the the synchronization is exploited by defining two links (with transition conditions true) in the flow activity. The *invoke C* activity will be executed only after the termination of the other two invoke activities.

WP4 - Exclusive Choice

Description It is a point in the workflow process where one of more branches in the control flow is activated based on the value of logical condition associated with the branch. This happens, for example, in Figure 5.9 a), where after the termination of the activity **A** the thread of control is passed to one of its



a) Exclusive Choice b) Simple Merge

Figure 4.17: Exclusive Choice and Simple Merge Patterns

outgoing branches, t_1 or t_2, depending on the value of the condition *Cond*, which can only be calculate at run time.

Example After the specification of the Train journey data, the user can choice either to acquire the ticket on line, through the *CreditCard* service, or to pay to the train station directly.

Motivation The *Choice* pattern allows the thread of control to be directed to a specific activity, depending on the outcome of the preceding activity, value of some data elements or the result of a user decision.

Formalization The choice pattern allows the thread of control to be direct on the activity **B** making *exec*, only if the activity **A** is terminated and the condition *Cond* is evaluated true. Besides, if the condition *Cond* is evaluate false the activity **C** is activate.

WP5 - Simple Merge

Description It is a point in the process where two or more alternative branches converge together without synchronization and only one of these branches can be activated at the same time. This is the case of Figure 5.9 b), the *Simple Merge* allows the execution of **C** activity if **A** activity or **B** activity were previously activated and terminated.

Example At the conclusion of either the *CreditCard payment* or *cash payment* activities, a mail of notification is sent.
Motivation The *Simple Merge* patterns makes available of merging several distinct branches into one such that each thread of control received on incoming branches is immediately passed onto the outgoing branch.

Formalization The *Simple Merge* pattern can activate the activity **C** only if either the activity **B** or the activity **A** is terminated.

WP6 - Multiple Choice

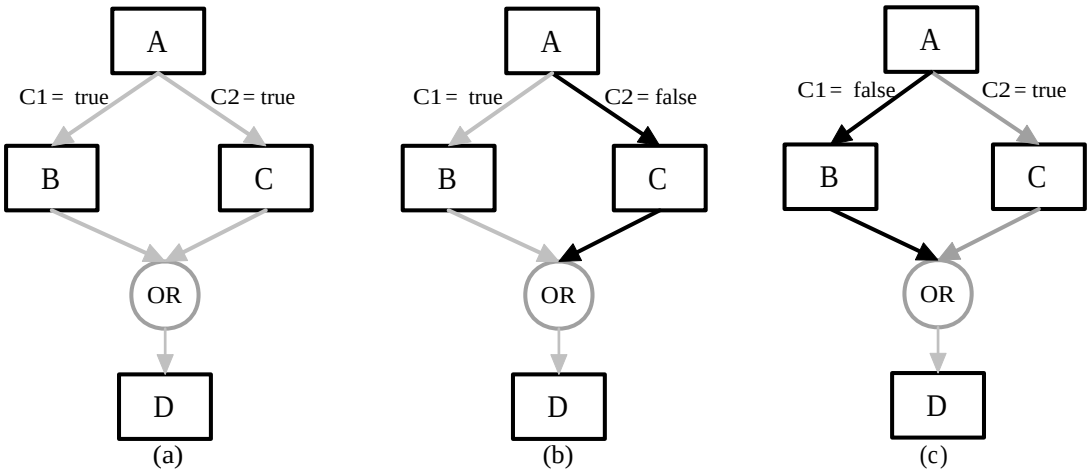


Figure 4.18: Multi Choice and Synchronizing Merge Patterns

Description It is a point in the workflow process where one or more branches in the control flow may be activated based on the value of logical condition associated with the branches. The number of branches to be activated can be greater than one, differently from the **Exclusive Choice** pattern. This happens, for example, in Figure 4.18, where after the termination of the activity **A** the thread of control is passed to one or both of its outgoing branches, t_1 or t_2, depending on the value of the conditions *C1* and *C2*, which can only be calculate at run time.

Example Depending on the medical insurance selected by the user(health, dental and so on), one or more insurance companies are called in order to calculate the best price.

Motivation The *Choice* pattern allows the thread of control to be diverged into several concurrent threads. The decision of which thread to start is made at run time and depends on control data. If no one of the branches is activated, the workflow could go to deadlock.

Formalization The behavior of the multiple choice pattern depicted in Figure 5.9 a), when both the conditions *C1* and *C2* are evaluated true is the following. The multiple choice pattern allows the thread of control to be direct on the activity **B** and *C* making *exec* their states respectively, only if the activity **A** is terminated and the conditions *C1* and *C2* are evaluated true. If only one condition (*C1* or*C2*) is true the semantics of the pattern is the same as **exclusive choice** pattern and is depicted in Figure 5.9 b) and c).

WP7 - Synchronizing Merge

Description It is a point in the process where two or more branches converge into a single thread. Some of these paths may be *active* and some not. If there exists more active paths, the following thread is activated only when all other active paths complete their executions. It is the case of Figure 4.18 when, for example,only the **B** and **C** activity paths are active. The execution of the **D** activity will start only after the completion of both **B** and **C** activities.

Example After receiving the responses of all invoked insurance companies, the better prices can be calculated.

Motivation The *Simple Merge* patterns makes available of merging several distinct branches into one such that each thread of control received on incoming branches is immediately passed onto the outgoing branch.

Formalization The *Simple Merge* pattern can activate the activity **D** only if either or both the activity **B** and **C** are terminated. Then, the activity **D** will be executed.

Bpel Implementations - WP4-WP5-WP6-WP7

These pattern implementations are showed in figure 4.19.

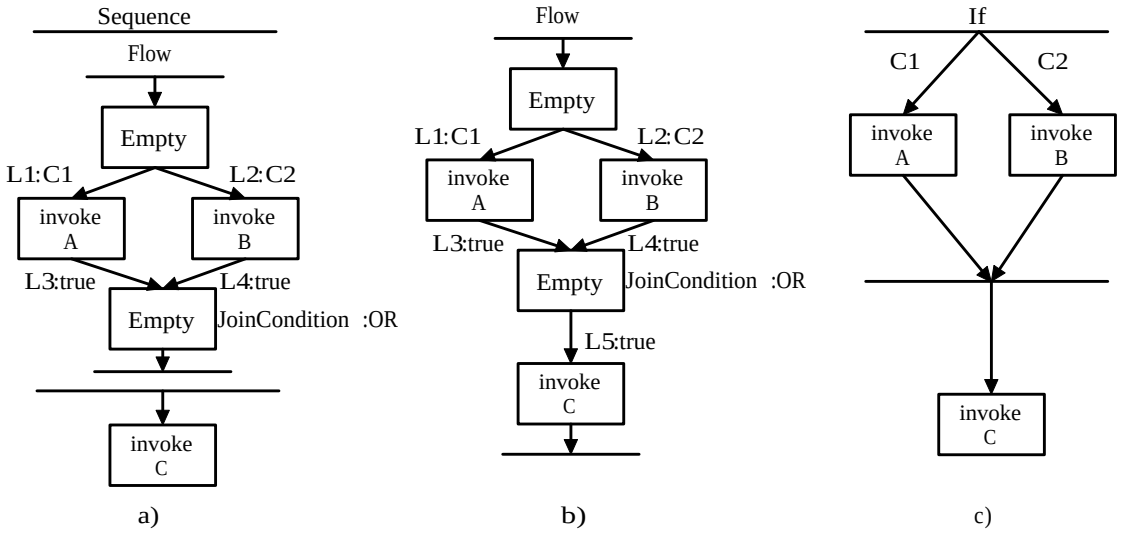


Figure 4.19: WP4-WP5-WP6-WP7 Pattern Implementation

Exclusive choice pattern can be implemented by using the flow activity (at the top of Figure 4.19 a) and b)) or by using the *switch* construct (at the top of Figure 4.19 c)). In the first case, two links (L1 and L2) have to be defined and two mutual exclusive conditions (C1 and C2) have to be associated to them. Since no activity have to be executed before the choice, in the first case the SourceLink activity for the two links is an empty one. In the second case, simply a *switch* activity is used that natively supports this pattern.
The simple merge pattern is achieved natively by using the switch construct that merges all path (the bottom of Figure 4.19 c)) or by defining inside the flows two other links (L3 and L4 in Figure 4.19 a) and b)) having another *empty* activity as TargetLink. In addition on this activity a particular *join condition* has to be defined. It is called **OR join condition** and allows the execution of the TargetLink activity when, given more concurrent execution threads in the process, the first execution branch reaches the synchronization point. A deeper description of this condition is reported in [8]. After the synchronization the next activity may be executed by using the same consideration made for the patterns previously described.
The Multi-Choice and the Synchronizing Merge patterns can be implemented in the same way of the exclusive choice and simple merge patterns depicted in Figure 4.19 a) and b) if the conditions C1 and C2 are

not mutually exclusive.

WP8 - Multi Merge

Description A point in the process where two or more branches riconverge into a single thread without synchronization. If more than one branch gets activated, possibly concurrently, the activity following the merge is started for every action of every incoming branch. Let we consider the pattern in Figure 4.18, if the **B** and **C** activity paths are active and the synchronization is implemented through a **Multi Merge** pattern the activity **D** will be executed twice.

Example Sometimes two or more branches share the same ending. Two activities audit application and process applications are running in parallel which should both be followed by an activity close case, which should be executed twice if the activities audit application and process applications are both executed.

Motivation The *Multi-Merge* pattern provides a means of merging distinct branches in a process into a single branch. Although several execution paths are merged, there is no synchronization of control flow and each thread of control which is currently active in any of the preceding branches will flow unimpeded into the merged branch.

Formalization The *Multi Merge* pattern will activate the activity **D** every times the activities **B** or **C** will be terminated. Then, the activity **D** became **exec**.

BPEL Implementation Bpel does not offer a direct support for this patters. In fact, it does not allow for two active threads following the same path without creating new instances of another process [89].

WP9 - Discriminator

Description At this point the process waits for the first incoming active branches, activating the following one. The process then waits for the termination of all the other active branches, without activating other instance of the following branch. When all active incoming branches terminate, it can accept other *discriminating* pattern. This is the case of Figure 4.20 when all **A**, **B**, and **C** activities are active. If **B** is the first to end, **D** activity is executed. The process than wait for the termination of the other two activities, without executing **D** anymore.

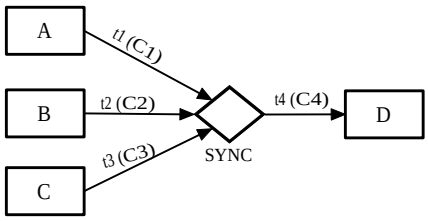


Figure 4.20: WP9 Pattern

Example When handling a cardiac arrest, the check breathing and check pulse activities run in parallel. Once the first of these has completed, the triage activity is commenced. Completion of the other activity is

ignored and does not result in a second instance of the triage activity.

Bpel Implementation This pattern is not directly supported in BPEL4WS.

In fact, there is not a structured activity construct which can be used for implementing it, nor can links be used for capturing it. The reason for not being able to use the link construct with an or joinCondition, is the fact that a joinCondition is evaluated when the status of all incoming links are determined and not, as required in this case, when the first positive link is determined.

WP10 - Arbitrary Cycles

Definition This pattern is used to describe cycles without imposing restrictions on the number of execution needed for the loop, without previously defining at which point in the process the next iteration will start and with the opportunity of nesting other loops in it. This is the example of Figure 4.21 d) when at each iteration it is not known if the loop will restart executing the **A** or the **B** activity.

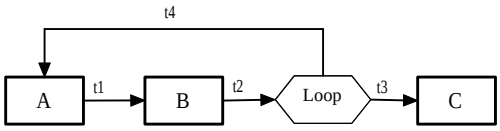


Figure 4.21: Arbitrary Cycles Pattern

Bpel implementation This pattern is not supported in BPEL4WS. Although the *while* and *repeat until* activities allow for structured cycles, it is not possible to jump back to arbitrary parts of the process, i.e. only loops with one entry point and one exit point are allowed. The restriction made that links can not cross the boundaries of a loop and that links may not create a cycle.

WP11 - Implicit Termination

Definition A given subprocess is terminated when there is nothing else to do without defining an explicit termination activity.

Bpel implementation This pattern is natively supported by the *terminate* activity. The structured activities, except for *flow* with *links*, do not support this pattern, since they complete only if all their outermost activity complete(explicit termination). Besides, using the *flow* with *links*, a subprocess can have multiple sink activities (i.e., activities not being a source of any link) without requiring one unique termination activity.

WP12 - Multi Instances without Synchronization

Definition This pattern allows the creation of multiple instances of an activity spawning a new thread of control for each created activity. For example in Figure 4.20 this pattern allows the creation of a new instance of the **D** activity whenever the incoming activities terminate.

Example When booking a trip, the activity book flight is executed multiple times if the trip involves multiple flights.

Bpel Implementation This pattern is implemented by using a *while* activity as depicted in figure 4.22. The while activity creates new process instances that run in parallel without synchronization.

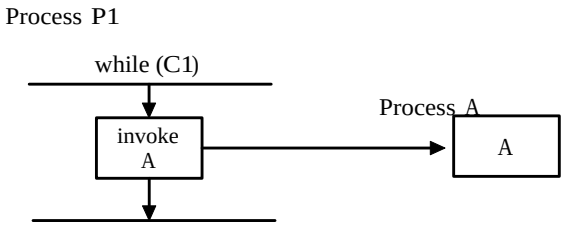


Figure 4.22: WP12 Pattern Implementation

WP13 - WP15 Multi Instances with Synchronization

Definition These three patterns allow in a point of the workflow process the creation of a certain number of instances of a given activity. The instances are later synchronized. In the **WP13** pattern the number of instances is known at design time; in the **WP14** the number of instances is known during the run time but before the first initiation of the activity; in the **WP15** the number of instances to create is known after the first initiation of the activity: new instances are created on demand until no more instances are required.

Example WP13 The Annual Report must be signed by all six of the Directors before it can be issued.

Bpel Implementation - WP13

This pattern can be implemented by using the *flow* construct and placing in it the needed number of activity replica. For example in Figure 4.23, three instances of the activity **A** are activated concurrently.

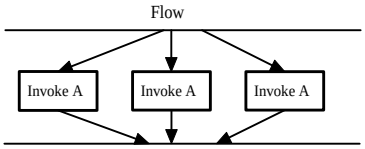


Figure 4.23: WP13 Pattern Implementation

Example WP14 In the review process for a journal paper submitted to a journal, the review paper activity is executed several times depending on the content of the paper, the availability of referees and the credentials of the authors. The review process can only continue when all reviews have been returned

Example WP15 When booking a trip, the activity book flight is executed multiple times if the trip involves multiple flights. Once all bookings are made, an invoice is sent to the client. How many bookings are made is only known at runtime through interaction with the user.

Bpel Implementation - WP14 - WP15

These patterns can be implemented by using a *while* activity as depicted in Figure 4.24 a).

Inside the while, the **A** web service is invoked. Then a *reply* activity waits for some data that are used to update the value of **C1** by using an *assign* activity. The value of C1 is not known during the while and

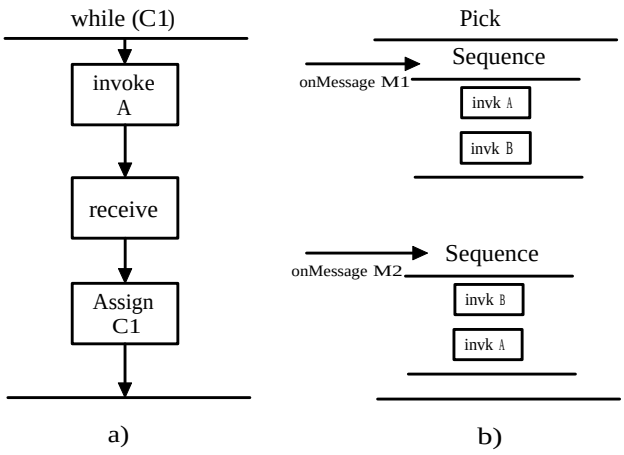


Figure 4.24: WP14-WP15-WP17 Pattern Implementation

an arbitrary number of instances of the activity *A* may be created.

WP16 - Deferred Choice

Definition This pattern allows the execution of different branches of a workflow process depending on information which is not necessary available when the point of choice is reached.

Example Once a customer requests an airbag shipment, it is either picked up by the postman or a courier driver depending on which is available to visit the customer site first.

Bpel Implementation

This pattern can be implemented by using a *pick* activity as depicted in Figure **??**. The deferred choice is associated to messages associated to the external events. Inside the *pick*, based on the first event that occurs (*postman* or *CorierDriver*) the activity **A** or **B** will be invoked

WP17 - Interleaved Parallel Routing

Description This pattern represents a set of activities executed in an arbitrary order. The activities are executed only once and their execution order is known at run time.

Example At the end of each year, a bank executes two activities for each account: add interest and charge credit card costs. These activities can be executed in any order. However, since they both update the account, they cannot be executed at the same time.

Bpel Implementation

A possible implementation of this pattern is depicted in Figure 4.24 b): a pick activity is used with sequences used as component activities. In each sequence activity, the needed order of the internal activities to execute is defined. External events select the needed sequence of activities.

WP18 - Milestones

Definition A milestone is a point in a workflow process where a given activity *A* has finished its execution and its subsequent activity *B* is not yet started. The pattern allows the instantiation of a given activity only if a (declared) milestone was reached.

Example Most budget airlines allow the routing of a booking to be changed providing the ticket has not been issued.

Bpel Implementation This pattern is not supported by the BPEL language.

WP19 - Cancel Activity

Definition An enabled activity is withdrawn prior to it commencing execution. If the activity has started, it is disabled and, where possible, the currently running instance is halted and removed.

Example The purchaser can cancel their building inspection activity at any time before it commences.

WP20 - Cancel Case

Description A complete process instance is removed. This includes currently executing activities, those which may execute at some future time and all sub-processes. The process instance is recorded as having completed unsuccessfully.

Example A customer withdraws his/her order.

Bpel Implementation WP19 - WP20

WP20 is solved with the *terminate* activity. Besides, WP19 is dealt with using fault and compensation handlers, specifying the course of action in cases of faults and cancellations.

4.4 Physical Composition

The WF graph generated in the previous phase has to be translated in a BPEL executable process. The translation is composed on two sub-phases: generation of the control flow, and I/O mapping. Finally, a verification of the executable process to avoid faults that can happen in presence of particular inputs or conditions.

BPEL Executable Process Generation

The BPEL generator produces a (concrete) BPEL workflow that can be deployed onto a runtime infrastructure, to realize the composite service. We first generate the WSDL description for the composite service. It provides the name and interface of the composite service and describes the port types for stitching together the component services. Once the WSDL has been generated, partner link types are defined, linking the component services. The next step is the generation of the BPEL flow. Components are invoked in the manner described by the pattern tree.

The composite service accepts inputs from the user that is fed to the first component service and sends an output from the last component service back to the user. We introduce variables that capture the output

of one service and provide it as input to the next. Specific details for each component service are obtained using the WSDL description for the corresponding instance.

I/O Mapping - Data Flow

After the data flow has been constructed, the BPEL generated might still not be readily deployable on a workflow engine. This is due to the fact that the code for messaging between components services needs to handle issues like(input/output) type matching and transformation, mismatch in invocation protocols that are used, ordering of parameters etc. At this step, this problems are manually resolved by the designer.

4.4.1 Verification of the Executable Process

This phase aims at the *formal verification of BPEL executable processes*, by defining a syntax-driven operational semantics for BPEL. The goal is to provide the developer of BPEL processes with a light, practical means to debug BPEL compositions against the most common semantic errors. Its current version can perform the following types of analysis:

- detection of wrong usage of BPEL constructs;
- detection of unexpected behaviors of BPEL processes due, for example, to bad **links** use or bad constructs combinations;
- detection of faults due to undiscovered semantics errors;
- detection of wrong usage of fault handlers.
- end state reachability: this allows to prevent deadlocks or livelocks.

The designer can use these information both to repair semantic errors or to implement fault compensation strategies before the deployment of the process.

Of course, BPEL differs from conventional programming languages, since typical elementary objects and operations are fairly abstract. Our formalization focuses on *executable processes* and mainly covers BPEL structured activities and control links. Event and exception handling as well as the communication between BPEL processes are the objects of future works. Nevertheless, a formal semantics of fault handlers and event handlers is proposed in [79], compensation handlers are considered in [27], and communication between BPEL processes is formalized in [46]. Although the above mentioned restrictions, the proposed formalization and tool helps to develop BPEL implementations whose behavior is deterministically defined and independent by the BPEL engine enacting the process.

BPEL control flow analysis

Faults could happen during the execution of a BPEL process even if the process definition is syntactically correct. An off-line analysis of the control flow of a BPEL process definition, driven by the semantics of the BPEL constructs, can help to discover faults before the process is executed. Some faults can happen only in presence of particular inputs or conditions: this makes difficult their detection. An off-line control flow analysis of the process definition driven by semantics of each constructs can help to discover some faults before the Some faults can happen only in presence of some particular inputs or conditions: this make difficult the detection of these faults. An exhaustive analysis can detect the errors leading to faults before the process is deployed.

In particular our approach is based on the derivation of semantics rules of each BPEL construct. This approach has the following advantages:

- It provides an exhaustive analysis of all faults that may occur during a process execution;
- it allows to retrieve the cause of the fault, making possible the definition of compensation actions if needed;
- it promotes the usage of formal rules to describe BPEL language constructs: this allows BPEL process engine implementation to refer to these rules to define language constructs behaviors.
- it promotes the usage of formal rules to describe BPEL language constructs: the implementation of a BPEL engine may refer to these rules to define the behavior of the language constructs.

The analysis based on the proposed approach allows to detect:

- wrong usage of BPEL constructs;
- undesirable behaviors of BPEL processes due, for example, to bad **links** use or bad constructs combinations;
- faults due to undiscovered semantics errors;
- wrong usage of fault handlers.
- end state reachability problems.

In order to allows these types of analysis, the following steps must be performed:

1. the operational semantic of BPEL constructs must be provided;
2. a set of derivation rules based on these definitions must be implemented;
3. a framework based on these derivation rules must be developed in order to detect automatically faults and retrieve their causes.

4.5 Discussion

The proposed framework (YAAWSC) is a system for end to end composition of web services. Our approach uses domain ontologies to describe operations, data and services, and aims at producing an executable process expressed by a standard workflow language that can be formally verified and validated. A *Service Description* contains the information about the services available in the *Domain*.

Despite currently available composition methodologies, the proposed approach allows utilizing not only web services but also web applications. The only constrain is that the considered services/applications have to be described in the Domain. The approach makes use of both Functional and Non-Functional requirements of the services in order to automatically compose them. The service functionalities are described formally, using a domain-specific terminology that is defined in the *Knowledge Base*. In particular, the *Knowledge Base* contains the domain specifications expressed by a *domain ontology* and two main Upper Ontologies for defining concepts related to the entities “Service” (e.g. concepts related to security, authentication, fault tolerance, etc.) and “Data” (type, casting, relationships among types, etc.).

Moreover to evaluate the feasibility and the correctness of the transformation, we have formally defined a workflow language through operational semantics. That semantics is contained into *Semantics Rules* knowledge base.

The technique used for the composition is *rules-based*. The process is realized in terms of the following phases:

1) *Logical Composition*. This phase provides a functional composition of service operations to create a new functionality that is currently not available.

2) *Transformation Feasibility*. This phase verifies the feasibility and the correctness of the transformation of the new functionality into a executable process expressed by a standard workflow language.

3) *Physical Composition*. This phase aims at producing an executable process, which is formally verified and validate.

When a new service has to be created, the user has to specify the *functional requirements* of the requested service using the concepts defined into the *Knowledge Base* and the workflow language to use to describe the executable process. Driven by the user request specification, the *Logical Composition* module first synthesizes an *operational flow graph (OF)* by reasoning on the facts contained in the *knowledge base* and applying the *inference rules* (*IR*). Then, the *operation flow graph* is modified by inserting opportune service wrappers in order to resolve Input/Output mismatches. Finally, a graph transformation technique is applied to the *operation flow graph* in order to identify the workflow patterns which realize the composite service. The output of the first phase is a *pattern tree (PT)*, which will be consumed by the *Transformation Feasibility* module in the second phase. The *Transformation feasibility* module checks by inferencing on the *semantic rules*, if the patterns (belonging to the *pattern tree*) can be realized by composing the language constructs. Generally, one or more construct combinations that realize the pattern tree are provided. The output of this phase is a representation of the pattern tree through the language constructs. If the pattern tree cannot be implemented, the user can choose either to repeat the process with more relaxed requests or to select another available composition language to implement the executable process. Finally, in order to turn the abstract process into an executable process, the *Physical Composition* module generates the code by analyzing the PT and recursively associating proper construct skeletons to PT nodes. The output of this phase is an executable process, which can be verified before being enacted to detect syntax or semantic errors in the code. Such composition happens at planning time.

Since a formal approach is used to generate the model during the logical phase, the generated workflow model is formally correct. We also evaluate the feasibility of the transformation and the formal definition of the derived BPEL process. The expressiveness of the approach at the logical level is defined by all possible workflow patterns. Based on the composition language used to implement the physical process, such expressiveness can be reduced. In fact using BPEL we are limited by the workflow patterns that are currently supported by that language. Note however that the choice of BPEL has been done since BPEL is currently the standard for web service composition, and because it allows reusing the composite services in order to create more complex orchestrations. Further reusability can be performed for the workflow model generated at the logical level. This model can also be used by operational based approaches.

The evaluation of the coding effort is small since the developer does not have to know the BPEL language syntax. The only manual process that still needs to be performed is the IO mapping using assign constructs. A comparison of the proposed approach with the ones described in chapter 2 is summarized in Table 4.2.

Table 4.2: Summary of the considered dimensions for the web service composition methods analysis. (+) means the dimension (i.e. functionality) is provided, (-) means the dimension is not provided. These marks do not vehicle any ”positive” or ”negative” information about the approaches except the presence or the absence of the considered dimension.

		Self-Serv	eFlow	Synthy	Meteor-s	Composer	Astro	YAAWSC
Service Discovery								
Which	Web Service	+	+	+	+	+	+	+
	Legacy/Web Application	+	-	-	-	-	-	+
What	Functional Requirements	-	-	+	+	+	+	+
	Non-Functional Requirements	+	+	+	-	+	-	+
When	Design Time	-	-	-	-	+	-	-
	Planning Time	-	-	+	+	-	+	+
	Execution Time	+	+	-	-	-	-	-
Composition Technique								
Composition Technique	Operational Based	+	+	-	-	-	-	-
	Rule Based	-	-	+	+	+	+	+
Composition Approach	Manual	-	-	-	-	-	-	-
	Semi-Automatic	+	+	+	-	+	-	-
	Automatic	-	-	-	+	-	+	+
Formalization	Logic Phase	-	-	+	+	-	+	+
	Physical Phase	-	-	-	-	-	+	+
	Whole Process	+	+	-	-	-	-	+
Expressiveness	Sequence	+	+	+	+	+	+	+
	Parallel Split	+	+	+	+	+	+	+
	Synchronization	-	+	-	-	-	-	+
	Exclusive Choice	+	+	+	-	-	+/-	+
	Simple Merge	+	+	+	+	-	-	+
	Multi Choice	-	-	-	-	-	-	+
	Multi Merge	-	-	-	-	-	-	+
	Discriminator	-	-	-	-	-	-	+
	Loop	-	-	-	+	-	-	+
	Transformation Feasibility	-	-	-	-	-	+/-	+
Composition Framework Characteristics								
Coding Effort	Considerable Effort	+	+	-	-	-	-	-
	Medium effort	-	-	+	-	-	-	-
	Small Effort	-	-	-	+	+	+	+
Reusing	Generated Executable Process	-	-	+	+	+	+	+
	Workflow Model Adaptaion	-	+	-	-	-	+	+

Chapter 5

Automatic Composition Framework and Case Study

5.1 Automatic Composition Framework

The composition development process described in the previous chapter is supported and automated by the architecture shown in Figure 5.1. Note that the Domain Ontologies have to be analyzed in order to build the $\mathcal{KB}$. The *Ontologies Analyzer* is the component in charge of populating the $\mathcal{KB}$ with Prolog axioms. The *Request Interpreter* analyzes the user request and produces a description of the required service. The description is defined in terms of IOPE by using the concepts in the $\mathcal{KB}$. This module translates the required service description in the Prolog query $PQ_{\mathcal{W}}$.

The *OF Generator* performs the automatic generation of the Operational Flow Graph. It is based on a Prolog inference engine (*Composition Rules Engine*) and it uses the inference *Rules* ($\mathcal{IR}$) described in Chapter 4 in order to build the OF graph while visiting an inference tree. For operation matching purposes, a proper component (*Matcher*) implements the IOPE matching, needed to select the candidate operations as explained in Chapter 4. During the OF graph generation, more than one solution (OFs) can be selected to implement the required service. The OF generator will choose the first OF whose effects completely satisfy the request.

The *Service Wrapping Generator* modifies the OF graph if its services are not compatible in terms of I/O. It introduces in the OF the activation of proper wrapper operations (retrieved from the *Service Catalog* in order to perform I/O types translations for operations if needed). The new OF graph is analyzed by the *SW Graph Builder*. It implements the algorithm for the SW graph building described in Chapter 4. The output of this component is a SW graph containing the composition of operations in terms of workflow patterns. This graph is then analyzed in order to be translated (if possible) in a BPEL executable process by the *Executable BPEL Process Generator*. This is achieved by substituting proper BPEL activities skeletons in place of patterns inside the SW. This last component also uses the *BPEL Semantics rules* (BPEL constructs semantics rules translated into *prolog* rules) in order to establish if a given pattern can be defined

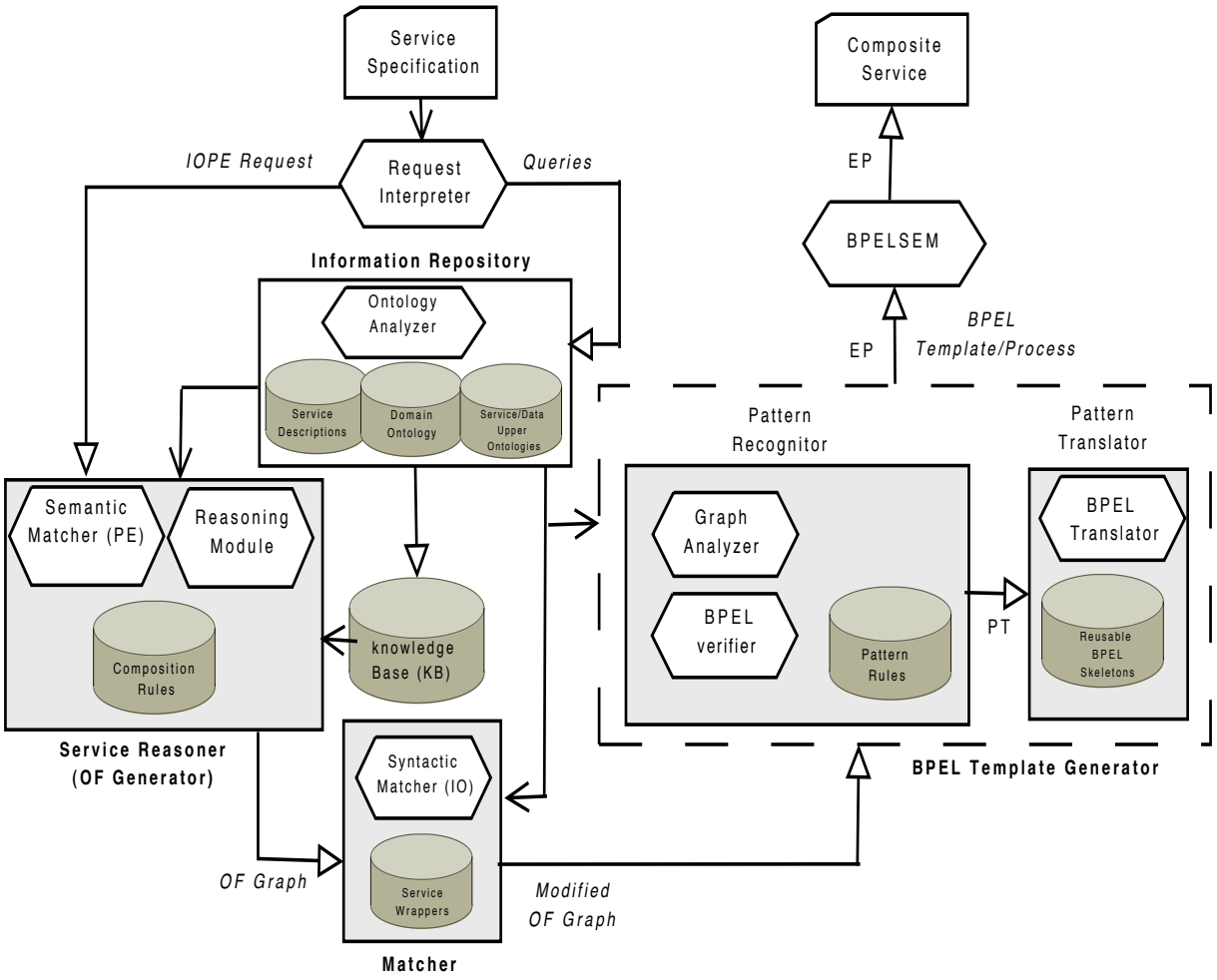


Figure 5.1: Composition process and architecture

in the BPEL language.

BPEL2SEM verifies the semantics correctness of the BPEL executable process, and performs the following analysis: (a) the detection of wrong usage of BPEL constructs; (b) the detection of unexpected behaviors of BPEL processes; (c) the detection of faults due to undiscovered semantics errors; (d) state reachability, allowing us to prevent deadlocks or livelocks. *BPEL2SEM* is composed of a static and dynamic analyzers. Through the *Static Analyzer* this module checks whether: (a) the BPEL definition contains at least one activity able to start the process (the creation of a process instance in BPEL4WS is done by setting the ”createIstance” attribute of some receive or pick activities to ”yes”); (b) the links elements are defined in the flow activity; (c) every link declared within a flow activity has exactly one source activity and one target activity; (d) the source activity is declared for each target activity of a link, and vice versa. Thanks to the *Static Analyzer*, the *Dynamic Analyzer* is able to execute a semantic analysis on a correct process. The *Dynamic Analyzer* aims to explore the full state space of a BPEL process. The analysis is performed by inferencing prolog rules of the *Semantics Rules* knowledge base. The Dynamic Analyzer determines if a BPEL process is correct in the sense that its execution can be performed from the first to the last activity depending on the process definition. Moreover, when the analyzed process is checked to end in an undefined state, the *Dynamic Analyzer* can detect errors and retrieve their causes,

using the *Check Rules*.

The composition architecture may be instantiated by using different techniques and tools. In our current implementation the Upper and Domain Ontologies are expressed by OWL[1], Web Services are described by OWL-S [5] and WSDL specifications. We use Fedora[2] to implement the Information Repository and populate the $\mathcal{KB}$ used by the Service Reasoner. At this aim SPARQL[3] queries are generated by the Request Interpreter. The Reasoning Module uses a Prolog engine, working on the composition rules. The Matcher is implemented by PERL scripts. We have developed the BPEL Translator and the algorithms used by the Graph Analyzer introduced in Section 4.2.4; the Verifier is based on the formalization of BPEL and on our related verification tool BPEL2SEM is described in the section 5.2.4.

5.2 Running Example

In order to describe the details of the developed framework, we present and describe the application of the methodology to a particular example.

Let us consider the case of a national rail wants to realize a booking web service allowing to plan journeys and buy tickets. This scenario allows registered users to book seats and purchase tickets by using credit cards, or to choose to make a reservation deferring the payment. In both cases a notification message is delivered to the user by an e-mail that summarizes the booking data. Suppose that a set of (secure

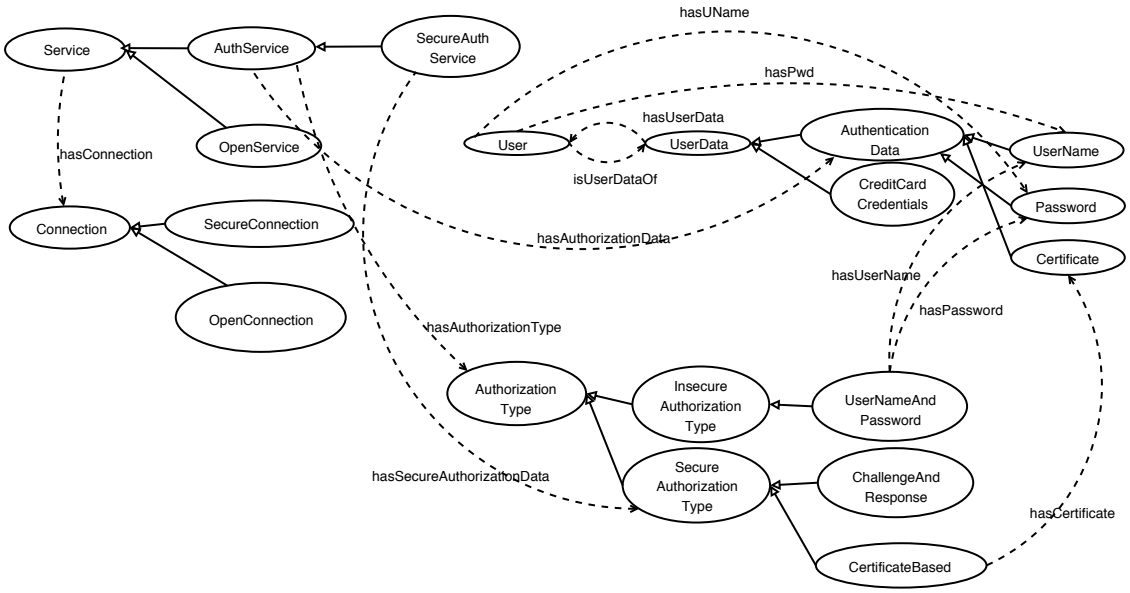


Figure 5.2: Service and User Ontologies

and tested) web services are available to handle reservations (allowing user authentication, searching the timetable, reserving seats, delivering notification messages) and to perform secure credit card transactions (allowing for checking user's credentials and the bank account data). Figure 5.2 shows parts of the Service Upper Ontology and its relationships with a part of the Domain Ontology related to the user of a rail inquire service. Figure 5.3 shows part of the Domain Ontology related to journey, reservation and payment and

[1] http://www.daml.org/
[2] http://www.fedora.info/
[3] http://www.w3.org/TR/rdf-sparql-query/

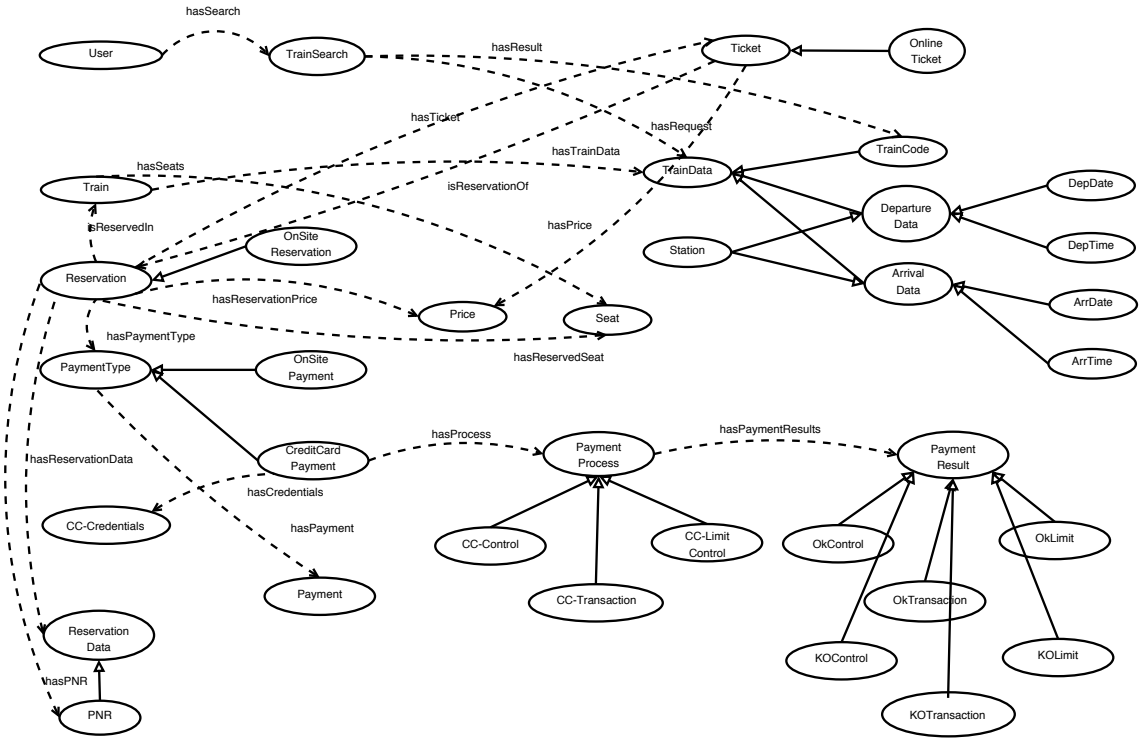


Figure 5.3: Train and Payment Ontologies

relationships among them. The figures report just those parts of ontologies that have been used to develop the example. The concepts are represented by ellipses, the hollow-triangle arrows represent *is-a* relationships and broken-line arrows represent relevant properties, that are described by the associated labels. The instances of the component services are not reported in Figure 5.2 because they will be described in the following. The booking service will satisfy the following PE specification:

- Pre-conditions: *AuthenticationData*, *canChoose(PaymentType)*, *TrainData*
- Effects: *hasPrice(Price), hasPNR(PNR), PNR, NotificationMail, hasReservation(Reservation)*

5.2.1 From User Request To Operation Flow Graph

Driven by the IOPE requirements of the user request, the *OF Generator* first synthesizes the Operational Flow Graph (OF) by reasoning on the facts contained in the Service Ontology (see Figure 5.2) and applying the Inference Rules. For operation matching purposes, a proper component (*Matcher*) implements the IOPE matching, needed to select the candidate operations as explained in Chapter 4. The matching is done by reasoning on the Domain Ontology depicted in Figure 5.3.

During the OF graph generation, more than one solution (OFs) can be selected to implement the required service. The OF generator will chose the first OF whose effects completely satisfy the request.

The selected service operations and their preconditions and effects description are reported in Table 5.1. Notice that both preconditions and effects are described by using concepts and relations reported in Figures 5.2 and 5.3.

The generated operation flow graph is depicted in Figure 5.4. In the figure each rounded box represents a service operation, the notation used is $service\_name :: operation\_name$. The split and join types (AND, OR, XOR) are also indicated. The labels at the top and the bottom of the boxes indicate services preconditions and effects, respectively.

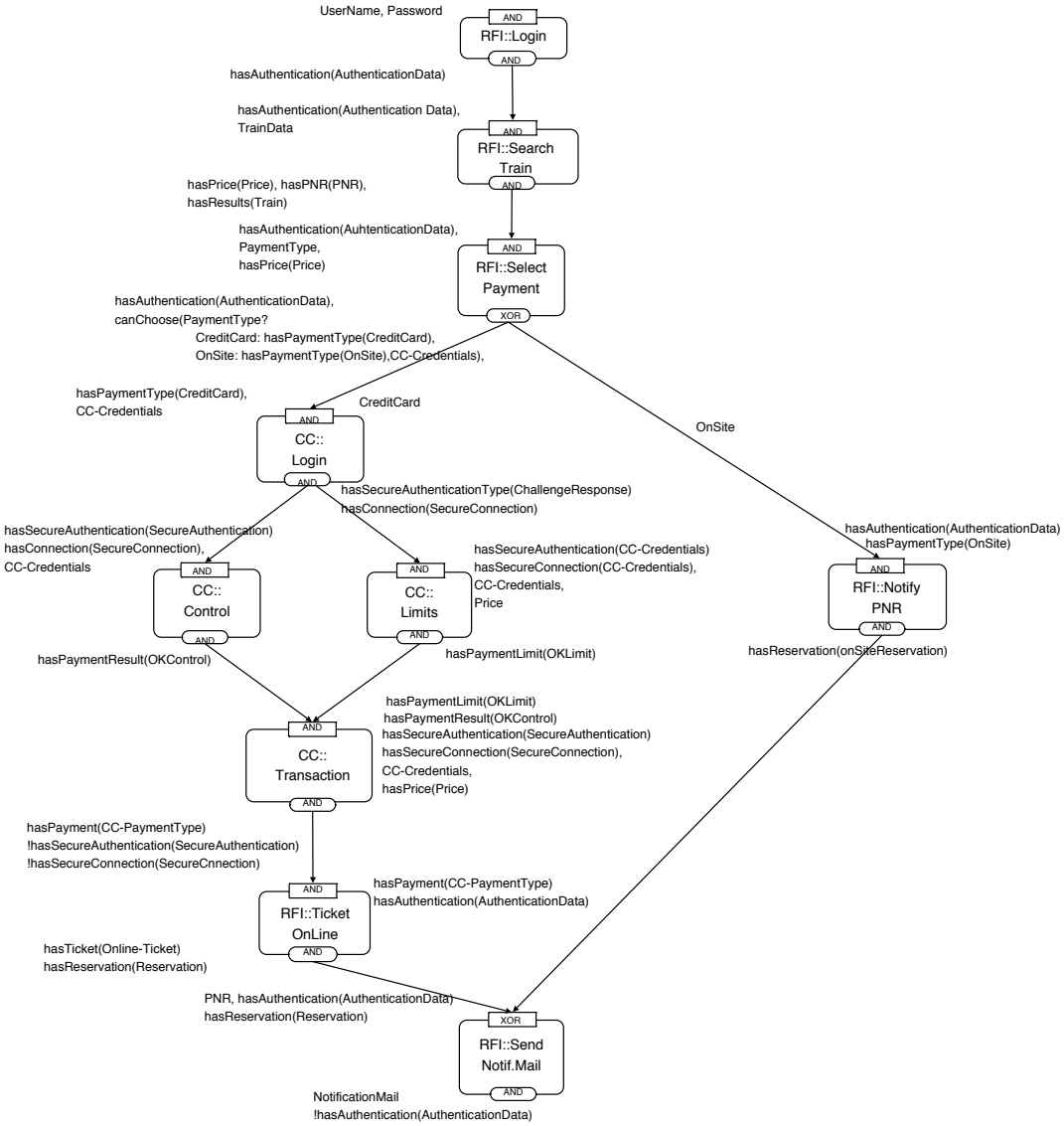


Figure 5.4: OF Graph

The selected service operations and their preconditions and effects description are reported in Table 5.1. Notice that both preconditions and effects are described by using concepts and relations reported in Figures 5.2 and 5.3.

A brief description of the selected services and the obtained flow graph is given in the following. To invoke the booking service, the user has to provide the Authentication Data and the CreditCard Credential if the credit card payment modality is chosen. The flow starts with an User Authentication Request, sent to RFI service. *RFI::Login* needs username and password to allow for user authentication. Once the login is executed with success, the user (to which *AuthorizationData* belong) is authorized on the system (the Effect

remains true until another service cancels it: this is specified by using the *!* symbol before the Effect to cancel). When a user is logged in, it is possible to request a Train journey specifying a *TrainData* containing Departure and Arrival dates, times and stations. The *RFI::SearchTrain* can then be invoked with these data in order to provide the price and the reservation number (PNR) of the selected journey. Then the user can select the payment (through credit card on-line transaction or paying at the train station when leaving). The *RFI::SelectPayment* has the *canChoose(...)* effect, since it allows for the choice of the two payment types. For what concerning Credit Card management services, they all require a secure connection. Once a login session to the payment system is estabilished, it is possible to control the card limits and availability. The payment transaction can be accomplished only if card limits and availability are varified (these tasks can be accomplished by *CC::Login*, *CC::Limits*, *CC::Control* and *CC::Transaction*). In the case of a payment in the station (onSite), the reservation for the user has to be registered (*RFI::NotifyPNR*) and validated only after the payment. Anyway, an e-mail with the operation notification (*RFI::SendNotif.Mail*) must be sent to the user at the end of the process.

Table 5.1: Component Services PEs

Service Name	**Preconditions**	**Effects**
RFI::Login	UserName, Password	hasAuthentication(AuthenticationData)
RFI::SearchTrain	hasAuthentication(AuthenticationData), TrainSearch	hasPrice(Price),hasPNR(PNR) hasResults(Train)
RFI::SelectPayment	hasAuthentication(AuthenticationData), PaymentType, hasPrice(Price)	canChoose(PaymentType)
RFI::NotifyPNR	hasAuthentication(AuthenticationData) hasPaymentType(OnSite)	hasReservation(onSiteReservation)
CC::Login	hasPaymentType(CreditCard) CC-Credential	hasSecureAuthenticationType(ChallengeResponse) hasConnection(SecureConnection)
CC::Control	hasSecureAuthentication(SecureAuthentication) hasSecureConnection(SecureConnection), CC-Credential	hasPaymentResult(OKControl)
CC::Limits	hasSecureAuthentication(SecureAuthentication hasSecureConnection(SecureConnection), CC-Credential,Price	hasPaymentLimit(OkLimit)
CC::Transaction	hasPaymentLimit(OkLimit) hasPaymentResult(OkControl) hasSecureAuthentication(SecureAuthentication) hasSecureConnection(SecureConnection) CC-Credential, hasPrice(Price)	hasPayment(CC-PaymentType) !hasSecureAuthentication(SecureAuthentication) !hasSecureConnection(SecureConnection)
RFI::TicketOnLine	hasPayment(CC-PaymentType) hasAuthentication(AuthenticationData)	hasTicket(OnlineTicket) hasReservation(Reservation)
RFI::SendNotif.Mail	hasAuthentication(AuthenticationData) hasReservation(Reservation)	NotificationMail !hasAuthentication(AuthenticationData)

The generated OF graph describes only the control flow, while the data flow mapping is generated by the *Matcher*. The Input and Output parameters of the selected services are reported in the Table 5.2. The meaning of the Input/Output data are defined by using concepts defined in Figure 5.2; besides their types are

reported in the Table 5.2. Since the data types of Input and Output types of each pair of consecutive services are matching, the parameters are compatible (there is no need for implementing wrappers). However, a manual mapping has to be performed to the final executable process (generated by the *BPEL Translator*) in order to handle issues like mismatch in the invocation protocols and ordering of parameters.

Table 5.2: Component Services Input/Output

Service Name	Input	Output
RFI::Login	UserName - xsd:string Password - xsd:string Certifiate - xsd:complexType	Autentication - xsd:boolean
RFI:: SearchTrain	DepartureDate - xsd:date ArrivalDate - xsd:date DepartureStation - xsd:string ArrivalStation - xsd:string	Price - xsd:double Train - xsd:string
RFI::SelectPayment	PaymentType - xsd:string	CreditCard - xsd:int OnSite - xsd:int
RFI::NotifyPNR	Train - xsd:string Price - xsd:double	PNR - xsd:string Price - xsd:double
CC::Login	CreditCardCredentials - xsd:complexType	Logged - xsd:boolean
CC::Control	CreditCardCredentials - xsd:complexType	Controlled - xsd:boolean
CC::Limits	CreditCardCredentials - xsd:complexType Price - xsd:double	OkLimit - xsd:boolean
CC::Transaction	CreditCardCredentials - xsd:complexType	TransationEnabled - xsd:boolean Credential - xsd:complexType
RFI::TicketOnLine	Credential - xsd:complexType Price - xsd:double Train - xsd:string	PNR - xsd:string PaymentReceipt - xsd:complexType
RFI::SendNotif.Mail	PNR - xsd:string PaymentReceipt - xsd:complexType	MailSent - xsd:boolean

5.2.2 From Operation Flow Graph to Pattern Based Workflow

The graph in Figure 5.4 is managed by the Graph Analyzer in order to identify patterns. In Figure 5.5 the steps required in order to discover patterns in the graph are reported.

First of all (a), the Sequences **S1** and **S2** are identified. On the graph with the two sequences macro-nodes, the Parallel execution pattern **P1** is identified (b). Then (c-d), the Sequence **S3** and the (Exclusive) Choice **C1** are identified. Finally (e-f) the whole graph is reduced to the sequence **S4** containing all the other detected patterns. The output of this phase is the *Pattern Tree* (PT) depicted in Figure 5.6, which contains the nested structure of workflow patterns detected in the OF graph.

5.2.3 From Pattern Tree To BPEL Executable Process

The Pattern Tree is then coupled with a query for the *BPEL Verifier* that will determine if the patterns composition defined by the PT can be implemented in BPEL or not. The Query Generated for the Example appears in the right part of Figure 5.7.

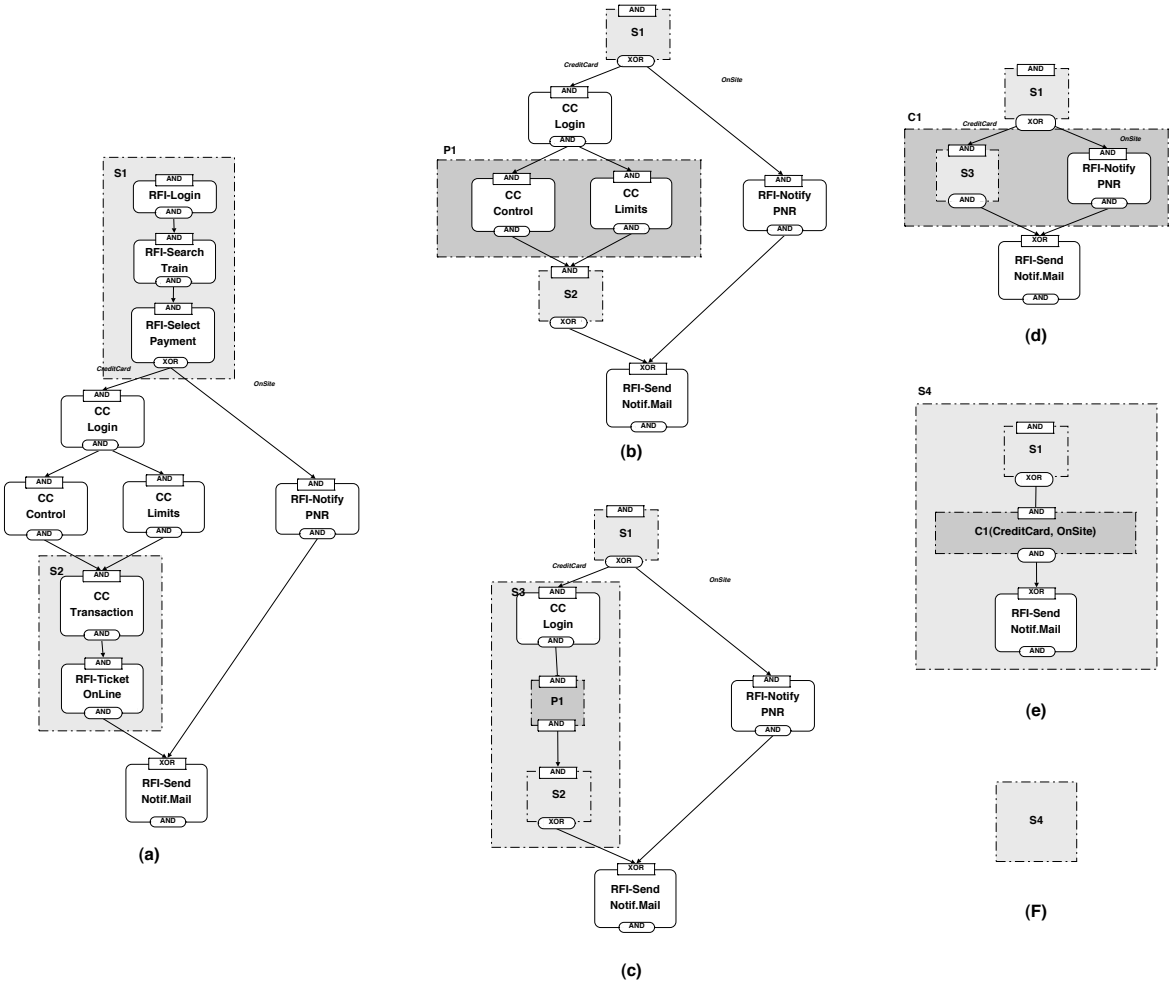


Figure 5.5: Graph Analysis

The *BPEL Verifier* applying the BPEL semantics rules defined in the section 4.3.2, generates a derivation tree and evaluates if an execution path exists to terminate the process, that implies that the transformation is feasible. For example, the output of the *BPEL Verifier* for the *Sequence S3* is depicted in Figure 5.8. The figure shows that all activities inside the Sequence S3 are correctly executed and the sequence terminates with axiom SA1 (no more activities in the list of activities need to be processed).

Since *BPEL Verifier* evaluates that the transformation is feasible, the Pattern Tree (PT) is consumed by *BPEL Translator* in order to produce a BPEL executable process. With reference to the booking service example, Figure 5.7 depicts the main components of the BPEL skeleton.

The BPEL Translator uses the Reusable BPEL Skeleton repository to generate the BPEL process form the PT. The repository contains: 1) the template of the implementation of the skeleton of the workflow patterns in BPEL and 2) the template of the basic BPEL process, constituted of a sequence activity containing the receive and reply activities.

The rules used to define the skeleton of the workflow patterns are described in the table 5.4. Generally several implementations can be provided in BPEL for a given pattern (see section 4.3.1), for example parallel execution with synchronization can be implemented with or without the use of the $link$ BPEL construct. The kind of implementation to be chosen depends mainly on the structure of the patterns tree,

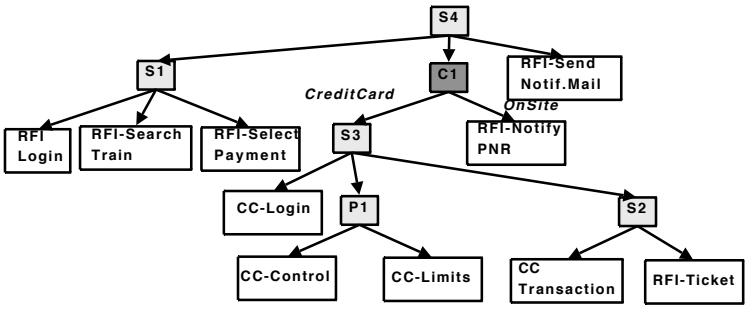


Figure 5.6: Pattern Tree

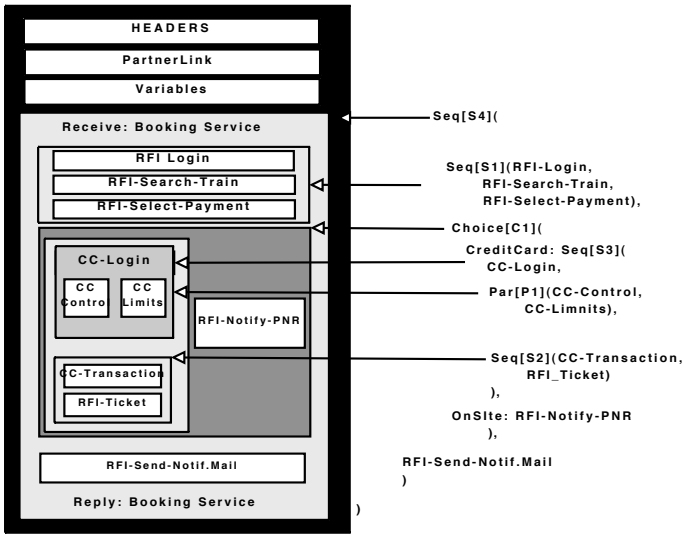


Figure 5.7: Graph Analysis

i.e. on the combination of patterns belonging to the PT. For example, in the left part of Figure 5.9, the translation of the sequence *S3: CreditCard* is depicted; observe that the sequence *S2*, belonging to *S3*, is not translated with a new *Sequence* construct but the two activities are just put in the right order into the *Sequence S3*, since the semantics of the pattern remains unchanged. To optimize the translation of the PT graph all possible implementations in BPEL of the workflow patterns and the BPEL skeletons of the most frequently used compositions are stored in the repository. During the translation phase, the PT graph is recursively managed by the *BPEL Translator* module in order to translate the patterns into the BPEL construct. First of all the *BPEL Translator* gets the template of the basic BPEL process from the repository and sets the parameters of the *receive* and *reply* activities, using the information available in the WSDL of requested service[4]. Then, the PT is recursively visited, and the detected patterns are translated using the associated templates found in the repository *Reusable BPEL Skeleton*. During the translation phase, in order to choose the skeleton that better implements the pattern tree, the *BPEL Translator* first translates each pattern with the proper template and then checks whether there is in the repository a BPEL skeleton having the same patterns tree of the whole PT graph or of the one or more of its branches. After this step, an *invoke* activity is associated to each activity. For each *invoke*, the proper parameters are set (for

[4]Observe that, the WSDL file of the composed services is automatically generated during the first phase of the life cycle, *Processing of the user request*.

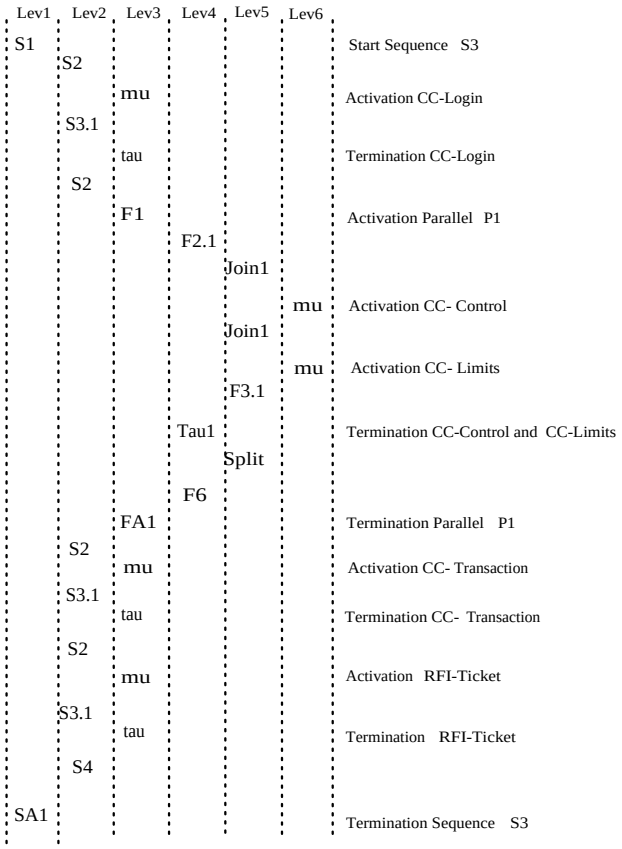


Figure 5.8: Derivation Tree of Sequence S3

example, input and output variables, operations etc.), getting those information from the WSDL files of the associate services. Also, at this step the partner links and the variables are set in the BPEL skeleton. When the patterns translation is complete, some modifications are made to the process in order to implement the right control flows defined in the PT.

With reference to the running example and the situation depicted in Figure 5.7, the chosen pattern has to be implemented in BPEL using the flow with link constructs (see the chosen pattern implementation in Figure 5.9). It results that the target link L1 and the source link L3 are added to the implementation of the sequence *S3* to implement the choice and the synchronization respectively. Finally, the obtained process is verified with *BPEL2SEM* module (see section 5.2.4) in order to verify the syntax and semantics correctness. If the created process is a valid BPEL process, the skeleton is stored in the repository in order to be reused in the future. Observe that the generated BPEL skeleton describes only the control flow between the involved services but does not consider the data flow mapping. Therefore, manual modifications are needed before the deployment of the process in order to manage the data flow, including the appropriate *assign* activities.

5.2.4 BPEL2SEM

In the following we describe the main architecture of the BPEL2SEM verifier. This tool is able to perform the analysis of a BPEL process. The BPEL2SEM architecture is presented in Figure 5.10.

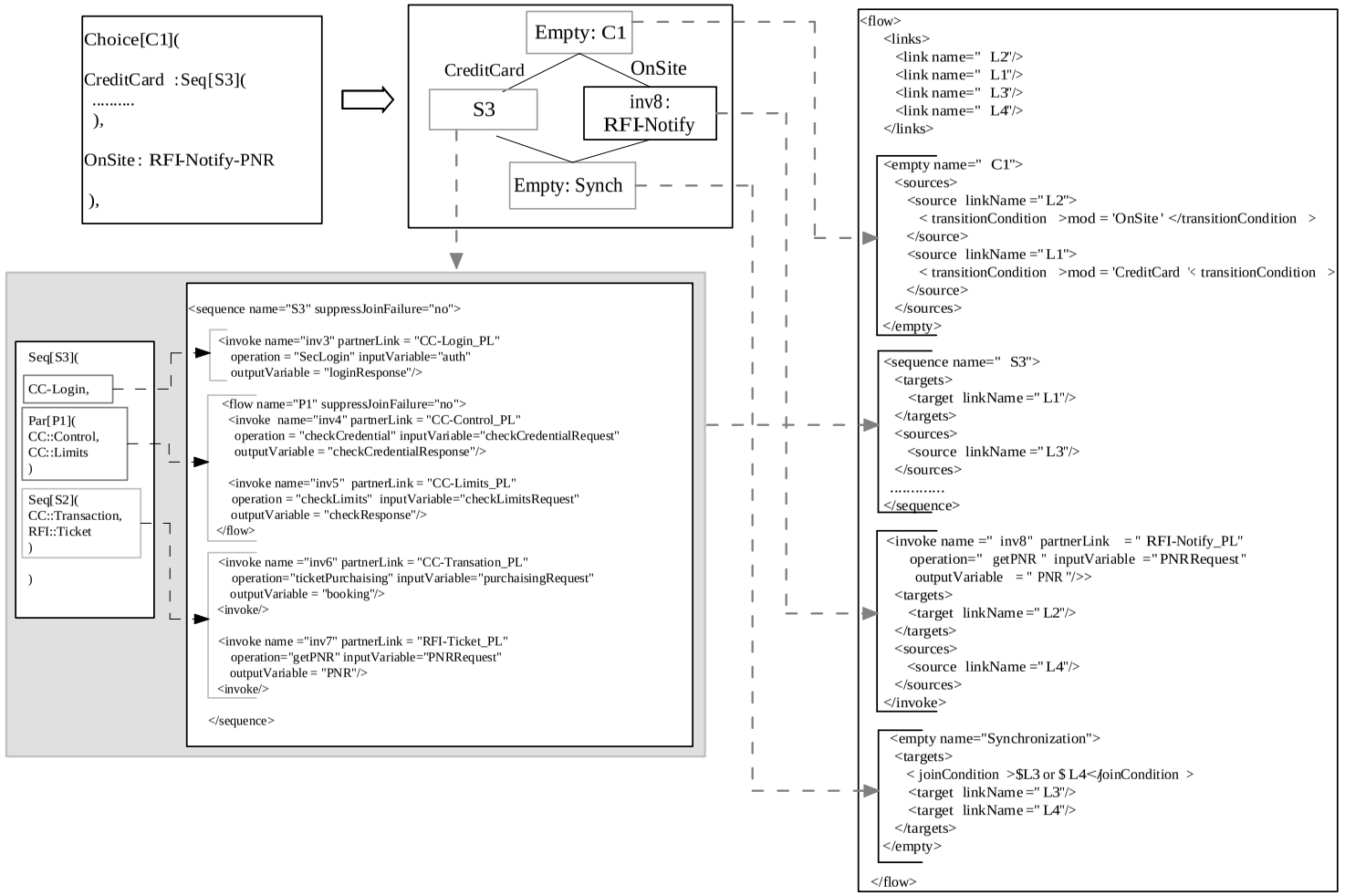


Figure 5.9: Choice 1 Pattern Implementation

Table 5.3: Pattern Implementation in BPEL. (-) means that the conditions do not have to be specified

Pattern	BPEL Construct	Join Implementation	Split Implementation
Sequences	*sequence*	-	-
Parallel Split & Synchronization	*flow*	-	-
	flow& *links*	-	*empty* activity having an *AND Join Condition*
Simple Choice & Simple Merge	*flow* & *links*	*empty* activity having source links with mutually exclusive *Transition Conditions*[b]	*empty* activity having an *OR Join Condition*[a]
	if	-	-
Parallel Split & Discriminator	*flow* & *links*	-	*empty* activity having an *XOR Join Condition*
Exclusive Choice & Simple Merge	*flow* & *links*	*empty* activity having source links with mutually exclusive *Transition Conditions*	*empty* activity having an *XOR Join Condition*
	if	-	-
Multi Choice & Multi Merge	not supported[c]	-	-

[a]The Join Condition is a Boolean expression of the activity's incoming links
[b]The Transition Condition is associated to the activity's outgoing links and its value determines the status of the link.
[c]This pattern is not supported in BPEL since it does not allow multiple (possibly concurrent) activations of an activity following a point where multiple paths converge.

Table 5.4: Pattern Hash Map Examples

Pattern Skeleton	Path
Choice(Seq(Flow),Activity)	$//choice\_squence.xml$

The *Static Analyzer* analyzes a BPEL process and translates it into an internal representation. Furthermore, this module performs different kinds of analysis on the BPEL process:

- it checks if the BPEL definition contains at least one activity able to start the process (the creation of a process instance in BPEL4WS is done by setting the "createIstance" attribute of some receive or pick activity to "yes");
- it checks if the elements links are defined in the flow activity;
- it checks if every link declared within a flow activity has exactly one source activity and one target activity;
- it determines, if, for each target activity of a link,the source activity is declared, and vice versa;

Thanks to the *Static Analyzer*, the *Dynamic Analyzer* can perform a semantic analysis on a correct process.

The *Dynamic Analyzer* aims to explore the full state space of a BPEL process. This analyzer uses the BPEL semantics rules translated into *prolog* rules and stored into a knowledge base (*Prolog Rules*). The *Prolog Rules* contains:

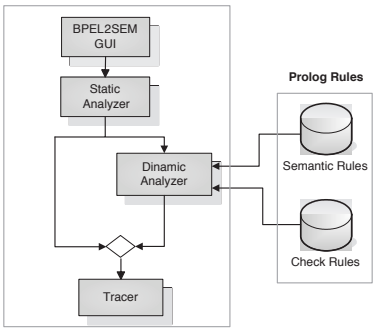


Figure 5.10: BPEL2SEM Architecture

- the rules of the constructs semantics (*Semantics Rules*);
- some rules used to detect errors and retrieve their causes, when the analyzed process is checked to end in an undefined state (*Check Rules*).

The analysis is performed by inferencing prolog rules of the *Semantics Rules* knowledge base. This leads inevitably to an explosion of states to analyze. Proper pruning techniques are implemented into the Dynamic Analyzer in order to cope with the state space explosion problem. The *BPEL2SEM UI* allows user to define proper pruning policies or to analyze the whole state space of the process to analyze. The Dynamic Analyzer determines if a BPEL process is correct in the sense that its execution can be performed from the first to the last activity depending on the process definition.

Finally, the *Tracer* uses information generated during the analysis phases in order to produce information about process execution (traces) both when the semantics incorrectness are present in the process definition or not.

5.2.5 Example

In the following, an example of a BPEL process is presented to show how BPEL2SEM verifier is able to detect anomalies in BPEL process definition. The example is simple enough to explain how semantics rules described in section 4.3.2 are applied by BPEL2SEM verifier in order to state if a given process terminates without faults.

The Figure 5.11 shows a marketplace process. The marketplace provides two type of goods, *TypeA* and *TypeB*; the customer orders one or both types of goods. On receiving the purchase order from the customer, the process initiates two task concurrently: one to select a shipper (ShippingSequence module) and one to schedule good production (ProductionFlow module). The $ProductionFlow$ module verifies if the customer has required one or both types of goods. In the first case, only the left or right part of the module is activate, both otherwise. The $Shipping$ module selects a shipper and defines the shipping price that is required to calculate the final price of the order.
The final price is obtained after shipping and production costs are calculated (PriceCalculation). A response message is sent to customer during the reply activity that follows PriceCalculation.

The process syntax is correct and no errors are detected while compiling and deploying it (The ActiveBPEL [15] engine and designer tool were used for these purposes).

Notice that, when the customer requires only one type of goods (for example only *TypeB*) the process, once deployed and invoked, reveals an undesirable behavior: it terminates with a fault generated by internal

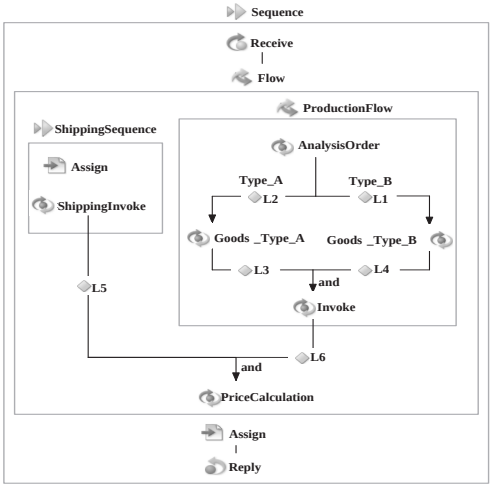


Figure 5.11: BPEL example

timeouts mechanisms since no more activities are enacted after the execution of the $Goods\_TypeB$ invoke activity.

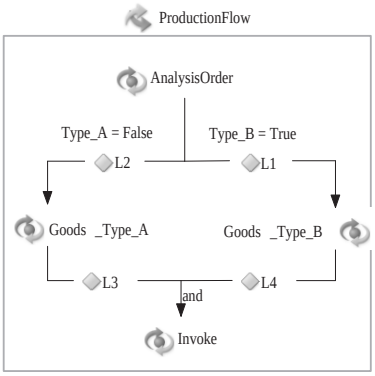


Figure 5.12: Error Example

In this example, the fault can be caused by semantics errors depending by:

- a wrong usage of $flow$ construct;
- a bad **links** use or a bad definition of the $TransitionConditions$ on the links;
- some particular input that produces these values for the $TransitionConditions$.

The BPEL2SEM is able to verify the semantic correctness of the process definition by applying the semantic rules defined in the previous section and verifying if an execution path exists to terminate the process without faults.

Figure 5.13 shows the derivation tree for $ProductionFlow$ module; notice that after the $AnalysisOrder$ termination (Lev5) the *Dynamic Analyzer* starts the $Goods\_TypeB$ invoke activity and applying the *Dead Path Elimination* at the $Goods\_TypeA$ invoke activity forces the $L3$ condition value becomes $false$. When the $Goods\_TypeB$ is ended (Lev7), the $Dynamic\ Analyzer$, applying the $F5$ and $FA3$ $rules$,

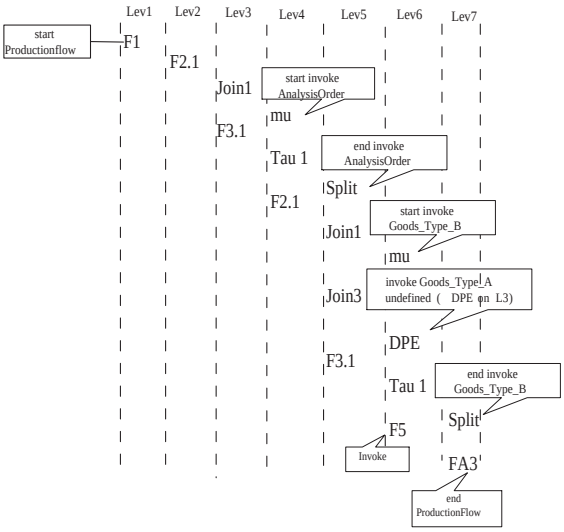


Figure 5.13: Prolog Rules

detects that the flow is ended incorrectly because the $JoinCondition$ on the $Invoke$ is evaluated false. This fault is propagated on $Reply$ activity and the process ends incorrectly.

The main problem of this process execution is that the path of the $Goods\_TypeA$ invoke activity, in the left side of the ProductionFlow module, will never be executed due to the L2 transition condition value (see Figure 5.12). The last invoke activity will never be executed, because it has an *and* join condition on its incoming links.

With a Backward analysis, the BPEL2SEM retrieves the cause of this fault. Since the Invoke activity has an and join condition, the rule that cannot be applied is the termination of the $Goods\_TypeA$ invoke in the flow. Recursively the verifier states that the first rule that denies the activation of the Invoke activity is that of $AnalysisOrder$ invoke termination. This rule allows the execution of $Goods\_TypeB$ invoke path but not the execution of the $Goods\_TypeA$ invoke path. The cause is that the transition condition on L2 link ever evaluates false. The verifier shows this result and ends its execution.

```
1 FA3(ProductionFlow) -> DPE(Invoke) -> Join3(Goods_TypeA) ->
2 F2(Goods_TypeA, Goods_TypeB ) -> split (L2) ->
3 L2 state: negative
4 cause: Type_A=false
```

Figure 5.14: Backward Analysis

The output of BPEL2SEM backward analysis is reported in Figure 5.14. The FA3 rules applied on ProductionFlow states that the flow cannot terminate without a fault. This is due to death path elimination (DPE) performed on *Invoke* activity that cannot be activated. The DPE is needed because the Join3 rule on Goods_TypeA invoke activity does not allow its activation. This is due to the impossibility of execute a split on L2 link after applying the F2 rule on the two invokes path. This is due to the fact that the L2 link state is negative because the Type_A transition condition is false.

Notice that conditions could be evaluated to that values also during the execution of a more complex process. The BPEL2SEM is also able to verify if one (or more) of the possible execution of a process

where several condition values are used in order to choose control flows path during process execution.

Chapter 6

Conclusions

Web service composition is a very active area of research due to the growing interest of public and private organizations in services integration and/or low cost development of value added services. The problem of building an executable web service from a service description has many faces since it involves web services discovery, matching, and integration according to a composition process. The automated composition of web services is a challenge in the field of service oriented architecture and requires an unambiguous description of all the information needed to select and combine existing web services. In this thesis, we have proposed a unified composition development process to the automated composition of web services which is based on the usage of Domain Ontologies for the description of data and services, and on workflow patterns for the generation of executable processes.

6.1 Summary of the contribution

In Chapter 2 we have discussed the state of the art of the several solutions for the automated web service composition. In particular, the analysis has been done comparing the approaches through seven dimensions which cope with several aspects related to service discovery, integration, verification and validation of the composite service. Moreover, a first study regarding the combination of Web 2.0 and SOA approaches has been presented. In particular, the objective of this study is to analyze the richnesses and weaknesses of the Mashup tools. Thus, we have identified the behaviors and characteristics of general Mashup applications and analyze the tools with respect to the key identified aspects from the Mashup applications by focusing on the data level.

In Chapter 3, we have presented the research results that were obtained by adopting a black-box migration approach based on wrapping to migrate functionalities of existing Web applications to Web services. This approach is based on a migration process that relies on black-box reverse engineering techniques for modelling the Web application User Interface. The reverse engineering techniques are supported by a toolkit that allows a semi-automatic and effective generation of the wrapper. The software migration platform and the migration case studies that were performed to validate the proposed approach are also presented in this chapter.

In Chapter 4, we have proposed a formal composition development process to the automated composition of web services, which is based on the usage of Domain Ontologies for the description of data and services, and on workflow patterns for the generation of executable processes. The process is realized in terms of the following phases:1) *Logical Composition*. This phase provides a functional composition of service operations to create a new functionality that is currently not available; 2)*Transformation Feasibility*. This phase verifies the feasibility and the correctness of the transformation of the new functionality into a executable process expressed by a standard workflow language; 3) *Physical Composition*. This phase aims at producing an executable process, which is formally verified and validate.

In Chapter 5, we have presented an architecture and a tool we developed that implements our methodology. Moreover a complex case study is presented to explain the different phases of the methodology.

6.2 Final remarks

The proposed composite service creation environment can be used to generate potential workflows for achieving the desired functionality reusing existing web services. The result is the reduction of the development time of new value added services. At the best of our knowledge in automated service composition background, the research reported in this thesis is the first one tackling the following issues:

1. defining and validating systematic approaches for exporting existing software applications towards the new Service Oriented Architecture.

2. defining a formal composition methodology, which copes with several aspects related to service discovery, integration, verification and validation of the composite service. In fact, an operational semantics is the formal basis of all composition development process phases: it is used:

   - to define the relationships between operations and data;
   - to express the flow of the operations which realize the composition goal;
   - to identify the composition pattern described by the composition flow;
   - to formalize the workflow language constructs;
   - to support the validation of the composition that allows us to formally verify: 1) the correctness of the workflow model generated during the logical phase, 2) the feasibility and the correctness of the transformation and 3) the correctness of the Bpel executable process.

   That formal approach allows for (i) ensuring the correctness of the workflows in terms of the implementability of the control flow and process verification and (ii) enabling workflow reuse.

3. defining a methodology that does not depend on a particular workflow language. In fact, pattern trees obtained after the logical phase can be implemented using different composition languages, and the feasibility of the transformation can be verified using the methodology described in the section 4.3.1.

Bibliography

[1] *Mashup Styles, Part 1: Server-Side Mashups, http://java.sun.com/ developer/ technicalArticles/J2EE/mashup_1/.*

[2] *Mashup Styles, Part 2: Client-Side Mashups, http://java.sun.com/ developer/ technicalArticles/J2EE/mashup_2/.*

[3] *OASIS: Web Services Business Process Execution Language Version 2.0. (2007), http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html.*

[4] *Semantic Annotations for WSDL, http://www.w3.org/TR/sawsdl/.*

[5] *W3C, Semantic Markup for Web Service(DAML-S), http://www.w3.org/Submission/OWL-S/.*

[6] *W3C, Simple Object Acess Protocol (SOAP) - Version 1.3, http://www.w3.org/TR/soap/.*

[7] *W3C, Web Services Description Languages (WSDL) - Version 1.1, http://www.w3.org/TR/wsdl.*

[8] *WS-BPEL, Web Services Business Process Execution Language - Version 2.0, http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html.*

[9] *Workflow Handbook 2001.* Workflow Management Coalition, http://www.wfmc.org/standards/docs.htm, 2001.

[10] Reference model for service oriented architecture 1.0. committee specification 1, http://www.oasis-open.org/committees/download.php/19679/soa-rm-cs.pdf, 2006.

[11] *A Domain-Specific Language for Web APIs and Services Mashups*. Springer, 2007.

[12] Httpunit, http://httpunit.sourceforge.net/, 2008.

[13] Uddi, http://uddi.org/pubs/uddi-v3.0.2-20041019.htm, 2008.

[14] Xml path language (xpath),http://www.w3.org/tr/xpath, 2008.

[15] LLC ActiveBPEL. The open source bpel engine, www.activebpel.org, 2008.

[16] Vikas Agarwal, Girish Chafle, Koustuv Dasgupta, Neeran M. Karnik, Arun Kumar, Sumit Mittal, and Biplav Srivastava. Synthy: A system for end to end composition of web services. *J. Web Sem.*, 3(4):311–339, 2005.

[17] Rohit Aggarwal, Kunal Verma, John A. Miller, and William Milnor. Constraint driven web service composition in meteor-s. In *IEEE SCC*, pages 23–30, 2004.

[18] Gustavo Alonso, Fabio Casati, Harumi A. Kuno, and Vijay Machiraju. *Web Services - Concepts, Architectures and Applications*. Data-Centric Systems and Applications. Springer, 2004.

[19] Mehmet Altinel, Paul Brown, Susan Cline, Rajesh Kartha, Eric Louie, Volker Markl, Louis Mau, Yip-Hing Ng, David Simmen, and Ashutosh Singh. Damia: a data mashup fabric for intranet applications. In *VLDB '07*, pages 1370–1373. VLDB Endowment, 2007.

[20] T. Andrews. Business process execution language for web services (bpel), ftp://www6.software.ibm.com/software/developer/library/wsbpel.pdf, May 2003.

[21] A. Avizienis, J. Laprie, and B. Randell. Fundamental concepts of dependability, research report n01145, laas-cnrs, 2001.

[22] C. Batini, M. Lenzerini, and S. B. Navathe. A comparative analysis of methodologies for database schema integration. *ACM Comput. Surv.*, 18(4):323–364, 1986.

[23] R. Baumgartner, G. Gottlob, M. Herzog, and W. Slany. Interactively adding web service interfaces to existing web applications. In *Int. Symposium on Applications and the Internet*, pages 74–80, 2004.

[24] Boualem Benatallah, Quan Z. Sheng, and Marlon Dumas. The self-serv environment for web services composition. *IEEE Internet Computing*, 7(1):40–48, 2003.

[25] Manish Bhide, Pavan Deolasee, Amol Katkar, Ankur Panchbudhe, Krithi Ramamritham, and Prashant Shenoy. Adaptive push-pull: Disseminating dynamic web data. *IEEE Transactions on Computers*, 51(6):652–668, 2002.

[26] Engin Bozdag, Ali Mesbah, and Arie van Deursen. A comparison of push and pull techniques for ajax, 2007.

[27] Michael J. Butler, Carla Ferreira, and Muan Yong Ng. Precise modelling of compensating business transactions and its application to bpel. *J. UCS*, 11(5):712–743, 2005.

[28] G. Canfora, A.R. Fasolino, G. Frattolillo, and P. Tramontana. Migrating interactive legacy systems to web services. In IEEE CS Press, editor, *European Conference on Software Mainteinance and Reengineering*, pages 23–32, 2006.

[29] G. Canfora, A.R. Fasolino, G. Frattolillo, and P. Tramontana. A wrapping approach for migrating legacy system interactive functionalities to service oriented architectures. *Journal of Systems and Software*, 2007.

[30] Fabio Casati, Ski Ilnicki, Li jie Jin, Vasudev Krishnamoorthy, and Ming-Chien Shan. Adaptive and dynamic service composition in *flow*. In *CAiSE*, pages 13–31, 2000.

[31] Fabio Casati and Ming-Chien Shan. Dynamic and adaptive composition of e-services. *Inf. Syst.*, 26(3):143–163, 2001.

[32] Francisco Curbera, Matthew J. Duftler, Rania Khalaf, and Douglas Lovell. Bite: Workflow composition for the web. In *ICSOC*, pages 94–106, 2007.

[33] Florian Daniel, Jin Yu, Boualem Benatallah, Fabio Casati, Maristella Matera, and Regis Saint-Paul. Understanding ui integration: A survey of problems, technologies, and opportunities. *IEEE Internet Computing*, 11(3):59–66, 2007.

[34] G.A. DiLucca, A.R. Fasolino, and P. Tramontana. Web pages classification using concept analysis. In IEEE CS Press, editor, *International Conference on Software Maintenance*, 2007.

[35] D. Draheim and G. Weber. *Form Oriented Analysis. A New Methodology to Model Form Based Applications*. Springer-Verlag, 2005.

[36] Schahram Dustdar and Wolfgang Schreiner. A survey on web services composition. *Int. J. Web Grid Serv.*, 1(1):1–30, 2005.

[37] Schahram Dustdar and Wolfgang Schreiner. A survey on web services composition. *International Journal of Web and Grid Services*, 1(1):1–30, August 2005.

[38] Rob Ennals and David Gay. User-friendly functional programming for web mashups. In *ICFP '07*, pages 223–234, New York, NY, USA, 2007. ACM.

[39] Robert J. Ennals and Minos N. Garofalakis. Mashmaker: mashups for the masses. In *SIGMOD '07*, pages 1116–1118, New York, NY, USA, 2007. ACM.

[40] F. Estievenart, J. Meurisse, J. Hainaut, and P. Thiran. Semi-automated extraction of targeted data from web pages. In *International Conference on Data Engineering Workshops*, page 48, 2006.

[41] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[42] Roxana Geambasu, Cherie Cheung, Alexander Moshchuk, Steven D. Gribble, and Henry M. Levy. Organizing and sharing distributed personal web-service data. In *WWW*, pages 755–764, 2008.

[43] Francesco Moscato Giusy Di Lorenzo, Nicola Mazzocca and Valeria Vittorini. Automating web service composition: from control-flows to ws-bpel templates using workflow patterns. Technical report, 2008.

[44] H. Guo, C. Guo, F. Chen, and H. Yang. Wrapping client-server application to web services for internet computing. In *International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 366–370, 2005.

[45] Alon Halevy. Why your data won't mix. *Queue*, 3(8):50–58, 2005.

[46] Sebastian Hinz, Karsten Schmidt, and Christian Stahl. Transforming bpel to petri nets. In *Proceedings of the International Conference on Business Process Management (BPM2005), volume 3649 of Lecture Notes in Computer Science*, pages 220–235. Springer-Verlag, 2005.

[47] D. Hollingsworth. The workflow reference model, http://www.wfmc.org/standards/docs/tc003v11.pdf, 2008.

[48] David F. Huynh, David R. Karger, and Robert C. Miller. Exhibit: lightweight structured data publishing. In *WWW '07*, pages 737–746, New York, NY, USA, 2007. ACM.

[49] David F. Huynh, Robert C. Miller, and David R. Karger. Potluck: Data mash-up tool for casual users. In *ISWC/ASWC*, pages 239–252, 2007.

[50] Irvine Richard G.Mathieu Jen-Yao Chung, Kwei-Jay Lin. In *IEEE Computer, Special Issue on Web Services Computing*, pages 36–46, London, UK, 2003. EEE Computer.

[51] Anant Jhingran. Enterprise information mashups: integrating information, simply. In *VLDB '06: Proceedings of the 32nd international conference on Very large data bases*, pages 3–4. VLDB Endowment, 2006.

[52] Y. Jiang and E. Stroulia. Towards reengineering web sites to web-services providers. In IEEE CS Press, editor, *European Conference on Software Mainteinance and Reengineering*, pages 296–305, 2004.

[53] Raman Kazhamiakin and Marco Pistore. A parametric communication model for the verification of bpel4ws compositions. In *EPEW/WS-FM*, pages 318–332, 2005.

[54] B. Kiepuszewski. Expressiveness and suitability of languages for control flow modelling in workflows, 2002.

[55] C.A. Knoblock, K. Lerman, S. Minton, and I. Muslea. Accurately and reliably extracting data from the web: a machine learning approach. *Bulletin of the IEEE CS TC on Data Engineering*, 23(3):33–41, 2000.

[56] N. Kushmerick, D. Weil, and R. Doorenbos. Wrapper induction for information extraction. In *International Joint Conference on Artificial Intelligence*, pages 729–735, 1997.

[57] A. Laender, B. Ribeiro-Neto ad A. Silva, and J. Teixeira. A brief survey of web data extraction tools. *SIGMOD record*, 31(2), 2002.

[58] Xuanzhe Liu, Yi Hui, Wei Sun, and Haiqi Liang. Towards service composition based on mashup. In *IEEE SCW*, pages 332–339, 2007.

[59] Giusy Di Lorenzo, Anna Rita Fasolino, Lorenzo Melcarne, Porfirio Tramontana, and Valeria Vittorini. Turning web applications into web services by wrapping techniques. In *WCRE*, pages 199–208, 2007.

[60] Giusy Di Lorenzo, Hakim Hacid, Hye young Paik, and Boualem Benatallah. Mashups for data integration: An analysis. Technical Report UNSW-CSE-TR-0810, 2008.

[61] Giusy Di Lorenzo, Nicola Mazzocca, Francesco Moscato, and Valeria Vittorini. Towards semantics driven generation of executable web services compositions. *JSW*, 2(5):1–15, 2007.

[62] Giusy Di Lorenzo, Francesco Moscato, Nicola Mazzocca, and Valeria Vittorini. Automatic analysis of control flow inweb services composition processes. In *PDP*, pages 299–306, 2007.

[63] R Hull M Gruninger and S McIlraith. A first-order ontology for semantic web services. In *W3C Workshop on Framework for semantics web services*, 2005.

[64] Annapaola Marconi, Marco Pistore, and Paolo Traverso. Automated composition of web services: the astro approach. *IEEE Data Eng. Bull.*, 31(3):23–26, 2008.

[65] Sheila A. McIlraith, Tran Cao Son, and Honglei Zeng. Semantic web services. *IEEE Intelligent Systems*, 16(2):46–53, 2001.

[66] X. Meng, D. Hu, and C. Li. Schema-guided wrapper maintenance for web-data extraction. In *ACM International Workshop on Web Information and Data Management*, pages 1–8, 2003.

[67] Francesco Moscato, Nicola Mazzocca, Valeria Vittorini, Giusy Di Lorenzo, Paola Mosca, and Massimo Magaldi. Workflow pattern analysis in web services orchestration: The bpel4ws example. In *HPCC*, pages 395–400, 2005.

[68] S. Murugesan. Understanding web 2.0. *IT Professional*, 9(4):34–41, July-Aug. 2007.

[69] Maria E. Orlowska, Sanjiva Weerawarana, Mike P. Papazoglou, and Jian Yang, editors. *Service-Oriented Computing - ICSOC 2003, First International Conference, Trento, Italy, December 15-18, 2003, Proceedings*, volume 2910 of *Lecture Notes in Computer Science*. Springer, 2003.

[70] Maria E. Orlowska, Sanjiva Weerawarana, Mike P. Papazoglou, and Jian Yang, editors. *Service-Oriented Computing - ICSOC 2003, First International Conference, Trento, Italy, December 15-18, 2003, Proceedings*, volume 2910 of *Lecture Notes in Computer Science*. Springer, 2003.

[71] Massimo Paolucci, Takahiro Kawamura, Terry R. Payne, and Katia Sycara. Semantic matching of web services capabilities. pages 333–347. Springer-Verlag, 2002.

[72] Mike P. Papazoglou, Paolo Traverso, Schahram Dustdar, and Frank Leymann. Service-oriented computing: a research roadmap. *Int. J. Cooperative Inf. Syst.*, 17(2):223–255, 2008.

[73] Mike P. Papazoglou and Willem-Jan van den Heuvel. Service oriented architectures: approaches, technologies and research issues. *VLDB J.*, 16(3):389–415, 2007.

[74] Marco Pistore, Paolo Traverso, and Piergiorgio Bertoli. Automated composition of web services by planning in asynchronous domains. In *ICAPS*, pages 2–11, 2005.

[75] Marco Pistore, Paolo Traverso, Piergiorgio Bertoli, and Annapaola Marconi. Automated synthesis of executable web service compositions from bpel4ws processes. In *WWW (Special interest tracks and posters)*, pages 1186–1187, 2005.

[76] Wolfgang Pree. *Design patterns for object-oriented software development*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1995.

[77] Erhard Rahm and Philip A. Bernstein. A survey of approaches to automatic schema matching. *The VLDB Journal*, 10(4):334–350, 2001.

[78] Jinghai Rao and Xiaomeng Su. A survey of automated web service composition methods. In *SWSWPC*, pages 43–54, 2004.

[79] D. Fahland W. Reisig. Asm-based semantics for bpel: The negative control flow. In *Proc. 12th International Workshop on Abstract State Machines*, pages 131–151, 2005.

[80] Dirk Riehle and Heinz Zllighoven. Understanding and using patterns in software development. theory and practice of object systems. In *VCK96 John Vlissides, James O. Coplien and Norm Kerth*, pages 3–13, 1996.

[81] Stuart J. Russell and Peter Norvig. *Artificial intelligence: a modern approach*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.

[82] Christoph Schroth. Web 2.0 versus soa: Converging concepts enabling seamless cross-organizational collaboration. In *CEC/EEE*, pages 47–54, 2007.

[83] Hans Schuster, Dimitrios Georgakopoulos, Andrzej Cichocki, and Donald Baker. Modeling and composing service-based nd reference process-based multi-enterprise processes. In *CAiSE '00: Proceedings of the 12th International Conference on Advanced Information Systems Engineering*, pages 247–263, London, UK, 2000. Springer-Verlag.

[84] M. Sipser. Introduction to the theory of computation. *PWS*, pages 47–63, 1997.

[85] E. Sirin, J. Hendler, and B. Parsia. Semi-automatic composition of web serv-ices using semantic descriptions, 2002.

[86] Evren Sirin, Bijan Parsia, and James A. Hendler. Filtering and selecting semantic web services with interactive composition techniques. *IEEE Intelligent Systems*, 19(4):42–49, 2004.

[87] Wil M. P. van der Aalst, Marlon Dumas, and Arthur H. M. ter Hofstede. Web service composition languages: Old wine in new bottles? In *EUROMICRO*, pages 298–307, 2003.

[88] Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, Bartek Kiepuszewski, and Alistair P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.

[89] Petia Wohed, Wil M. P. van der Aalst, Marlon Dumas, and Arthur H. M. ter Hofstede. Analysis of web services composition languages: The case of bpel4ws. In *ER*, pages 200–215, 2003.

[90] Jeffrey Wong and Jason Hong. Marmite: end-user programming for the web. In *CHI '06*, pages 1541–1546, New York, NY, USA, 2006. ACM.

[91] Zixin Wu, Karthik Gomadam, Ajith Ranabahu, Amit P. Sheth, and John A. Miller. Automatic composition of semantic web services using process mediation. In *ICEIS (4)*, pages 453–462, 2007.

[92] Zixin Wu, Karthik Gomadam, Ajith Ranabahu, Amit P. Sheth, and John A. Miller. Automatic composition of semantic web services using process and data mediation. In *Tecnical report, Kno.e.sis center,Wright State University,*, Febrary 28, 2007.

[93] Jin Yu, Boualem Benatallah, Regis Saint-Paul, Fabio Casati, Florian Daniel, and Maristella Matera. A framework for rapid integration of presentation components. In *WWW '07*, pages 923–932, New York, NY, USA, 2007. ACM.

[94] Z. Zhang and H. Yang. Incubating services in legacy systems for architectural migration. In IEEE CS Press, editor, *Asia-Pacific Software Engineering Conference*, pages 196–203, 2004.

[95] Zhuopeng Zhang and Hongji Yang. Incubating services in legacy systems for architectural migration. In *APSEC*, pages 196–203, 2004.