PhD Dissertation

International Doctorate School in Information and Communication Technologies

DIT - University of Trento

INTERACTING WITH SERVICE COMPOSITIONS

Alexander Lazovik

Advisor:

Prof. Marco Aiello

University of Trento

Co-Advisor:

Prof. Mike Papazoglou

INFOLAB, Tilburg University

December 2006

Abstract

Interaction with web service enabled marketplaces would be greatly facilitated if users were given a high level service request language to express their goals in complex business domains. This could be achieved by using a planning framework which monitors the execution of planned goals against predefined standard business processes and interacts with the user to achieve goal satisfaction.

The thesis addresses the problem of service composition executions managed by interacting with the client. The planning architecture accepts high level requests, expressed in XSRL (XML Service Request Language). The planning framework is based on the principle of interleaving planning and execution. This is accomplished on the basis of refinement and revision as new service-related information is gathered from service registries and web services instances, and as execution circumstances necessitate change. The system interacts with the user whenever confirmation or verification is needed.

The work is primarily concerned with the problems of composition and monitoring of business processes based on process reference models. XML Service Request Language (XSRL) was proposed to address the issue of service composition by giving the users an explicit control over process executions by describing desired service attributes and functionalities, including temporal and non-temporal constraints between services. Reference model instantiation is planned according to the goals and preferences specified by the user, and an appropriate plan is executed. The algorithms behind are based on the idea of interleaving planning and execution. These algorithms are based on model checking and constraint programming. Research in process monitoring was brought to the definition of a framework where business rules are defined by assertion statements. Assertions are published by the partners involved in the business process. The business process is executed with respect to the specified assertions and, by that, up-to-date business objectives, constraints and new market situations.

Algorithms for interleaving planning, monitoring and execution have been implemented by using Java programming language. The implementation uses constraint programming system to satisfy user goals and preferences against reference business processes. The choice for constraint solvers was motivated by the fact that in a web service scenario, users may wish to know why certain solutions are preferred to others. Explanation-based constraint programming is a viable approach to tackling such issues. As a constraint solver external system (Choco) is used. Choco is a Java library for constraint satisfaction problems, constraint programming and explanation-based constraint solving.

The evaluation of the language and service composition framework focused on showing the feasibility of the approach. There was implemented a domain generator allowing the tuning of the following parameters: number of states, branching factor, non-determinism rate, maximum directed cycle length, and if the service domain corresponds to a dag or a not. Then there were made several experimentations with different XSRL requests with increasing complexity on different domains. The evaluation showed that the system is able to deal with domains which are considerably larger than the biggest ones seen in practice.

Keywords
Service request languages, service composition, service-oriented computing, AI planning, constraint programming

Acknowledgements

First and foremost, I wish to thank my supervisor Marco Aiello for inspiration, insight, and time. The original content and ideas for this thesis are largely a result of collaborative efforts with him and Mike Papazoglou. I would also like to thank Rosella Gennari for her contributions and for sharing her deep knowledge in constraint programming.

I am thankful to Heiko Ludwig for giving the possibility to be a summer intern at IBM TJ Watson Research Center. The months spent there gave me a better understanding of what industrial research is and how science and industry can successfully co-exist and benefit from each other. I would also thank other people whom I met there: Chris Ward, Melissa Buco and David Loewenstern for their insights and ideas in the field of business process customizations.

I also thank all my friends and colleagues who have contributed to this thesis with their support and advice. They have created a friendly and joyful atmosphere, where it has been a great pleasure for me to work.

I am very grateful to my external thesis committee members, Barbara Pernici, Schahram Dustdar and Heiko Ludwig, for the time they have spent in reviewing and nitpicking my thesis.

I would also thank my colleagues from my previous place of work, Sergey Gvardeitsev and Peter Burak. Working with them gave me better understanding on what are open issues in distributed and service-oriented computing.

I wish to thank my university professors from Belarusian State University, Yuri Syroid and Anatoly Zmitrovich for giving me solid formal and mathematical background. Without it, I would never be able to make my thesis technically and mathematically sound.

My parents have provided years of support and encouragement. I appreciate everything you have done for me, and I love you both very much. Without you I would not be where I am today.

Finally, I want to thank my wife Elena for supporting and inspiring me, even though I spend way too many hours working. I love you very much.

Contents

1 Introduction **1**

1.1 An example in the travel marketplace 9

1.2 Published material 12

1.3 Thesis organization 14

2 Related work **17**

2.1 Service-oriented computing 17

2.1.1 Service composition 21

2.1.2 Monitoring of service compositions 22

2.2 Automated composition using AI Planning 24

2.2.1 Automated planning as a part of AI 24

2.2.2 Planning as model checking 32

2.3 Constraint satisfaction 35

2.4 Semantic Web 38

3 Interleaving planning and execution **41**

3.1 SOA business processes 41

3.2 Expressing goals and preferences: XSRL 45

3.3 Service planning problem 49

3.4 Formal semantics of XSRL 51

3.5 Algorithms for interleaving planning and execution 53

4 Service planning as constraint satisfaction **59**
4.1 Encoding of a planning domain 61
4.2 Encoding of goals and preferences 66
4.3 Encoding the example 69

5 Monitoring service-oriented business processes **73**
5.1 Business processes 74
5.2 Assertions 75
5.3 Monitoring framework 81
5.3.1 Service assertion language 85
5.3.2 Service assertion problem 88
5.3.3 A domain instance 90
5.3.4 Planning and monitoring algorithms 92
5.4 Monitoring a sample business process 95
5.5 Discussion: expressing QoS properties 97
5.5.1 Availability, accessibility, performance, and reliability . 98
5.5.2 Security 99
5.5.3 Transactionality 100
5.5.4 Regulatory 102

6 Implementation **103**
6.1 Implementation details 105
6.1.1 Encoding algorithms implementation 109
6.2 Examples of encoding 118
6.2.1 Supply chain 118
6.2.2 Purchase order 122

7 Conclusions **141**
7.1 XSRL and its supporting framework 141
7.2 Use of constraint programming 143

7.3 Monitoring of service compositions 144

Bibliography **147**

A XSRL and XSAL in BNF notation **165**

B Service planning problem using planning as model checking **169**

B.1 Completeness and correctness of planning as model checking with booleanization 175

Chapter 1

Introduction

The Internet and the Web provide great opportunities for companies who would like their customers to have fast, easy and cheap access to company's services by publishing them on-line. Nowadays anyone can buy books, reserve hotels and tickets, check latest news and meteorological forecasts from any device connected to the Internet. However, even if the Internet gives a good potential for business-to-customer interaction, its infrastructure has not been fully exploited for business-to-business solutions. One of the main research topics on the Web today is the service-enabled marketplaces where different parties work together by sharing their business processes. Service-oriented computing tries to solve this problem by introducing a concept of service and framework for service publishing, discovery, binding and composition.

Services are self-describing, open components that support rapid, low-cost composition of distributed applications. Services are offered by service providers – organizations that procure the service implementations, supply their service descriptions, and provide related technical and business support. Since services may be offered by different enterprises on the Internet, they provide a distributed computing infrastructure for both intra and cross-enterprise application integration and collaboration [39].

An important aspect of services is the separation between the interface and the implementation. The interface part defines the functionality visible to the external world and the way it is accessed. The service describes its own interface characteristics, i.e., the operations available, the parameters, data-typing and the access protocols, in a way that other software modules can determine what it does, how to invoke its functionality, and what result to expect in return. In a typical interaction a service client uses the service's interface description to bind to the service provider and invoke its functionalities.

The implementation realizes the interface and the implementation details are hidden from the users of the service. Different service providers using any programming language of their choice may implement the same interface. One service implementation might provide the direct functionality itself, while another service implementation might use a combination of other services to provide the same functionality [108]. A group of services interacting in the above manner forms the service application in a service-oriented computing (SOC) architecture.

Nowadays service-oriented computing is rapidly becoming the prominent paradigm for distributed computing and electronic business applications. SOC allows for service providers and service application developers to construct value-added services by combining existing services that are resident on the Web. To achieve this, firstly, services must be described in terms of the standard service definition language, published in the service registry, and subsequently must be inter-linked to express how collections of web services work jointly to realize more complex functionalities typified by business processes. Business processes described in this way model the actual behavior of a participant in a business interaction as well as the visible message exchange behavior of each of the parties involved in the business protocol. A service business process is defined "in the abstract" by referencing and inter-linking service interfaces involved in a process. A business process is a reusable definition that can be deployed in different ways and in different scenarios, while maintaining a uniform application-level behavior across all of them. For web services, BPEL [23] is used for describing the business processes.

One of the most challenging areas in SOC is the composition of services. Efforts have been made aiming at developing techniques to automatically compose different services in order to achieve complex business goals. In many situations it is desirable to allow a user to gain explicit control over the execution of BPEL expressions and dynamically change the nature of the web service interactions conducted with a particular business partner depending on the state of the process. Consider for example the case of a traveller deciding to change his hotel reservation to take advantage of an unexpectedly low priced weekend offer. Users may need to change message property values in the midst of a computation, e.g., update their holiday budget based on a ticket, hotel prices and availability, evaluate different behavioral alternatives or scenarios during a computation and change their course of action dynamically, or revisit different execution paths based on non-deterministic message property values that result from the invocation of the services involved in a process. This implies that process execution must be made adaptable at run-time to meet the changing needs of users and businesses. Obviously, BPEL specifications do not allow for the required flexibility to react swiftly to unforeseen circumstances or opportunities as choices are predefined and statically bound in BPEL programs. To meet such requirements serious re-coding efforts are needed every time when there is a need for even a slight deviation.

Such advanced functionality can only be supported by a service request language and its appropriate run-time support environment to allow users to express their needs on the basis of the characteristics and functionality of standard business processes whose services are found in service registries. A service request language provides for a formal means of describing desired service attributes and functionality, including temporal and non-temporal constraints between services, service scheduling preferences, alternative options and so on.

In the thesis we introduced an XML Service Request Language (XSRL) and its supporting framework to address the above issues. XSRL expresses a request and executes it according to the user preferences. The framework, that takes XSRL request as input, returns a product as the result of the request, e.g., constructs an end-to-end holiday packages (documents) comprising a number of flight and accommodation choices. XSRL is equipped with constructs for expressing quantitative requests, such as, booking a room for two nights, spending between 100 and 200 euro, etc., but also qualitative operations for sequencing goals, such as, contacting the hotel only after having booked a plane, for stating preferences, e.g., flying rather than taking a train to a destination, for stating the maintaining of a condition during execution, such as, keeping the budget below 500 euro. Loosely speaking, the response documents can be perceived as a series of plans that potentially satisfy a request. In expressing an XSRL request it is important that a user is enabled to specify the way that the request needs to be satisfied.

Traditional approaches in service-oriented computing assume that the person who initiates an interaction with the business process has complete knowledge of the underlying process semantics. However, in practice the requester and the process owner are different people or organizations and the requirement for the user to be aware of the process semantics is somewhat strong. The situation, when the user has precise knowledge about process variables but has limited or no knowledge about the process itself is more typical. Consider an example from the travel domain. An average user is aware of process variables such as price, time, tickets cost, but has only general knowledge of how the particular travel agency processes his request. Taking an example from another domain as buying a computer, one may see that the situation is the same: user usually restricts price, built time, and some additional preferences like particular memory or processor, but he has no constraints on how the computer is assembled as far as he is guaranteed to finally receive it. In the above scenarios, the user formulates his request to a travel agency or a computer selling company in a high level way, basically constraining the crucial process variables. The XML Service Request Language deals with the scenarios above and gives the final user the ability to express his constraints in a high-level intuitive way without requiring him to know the details of the underlying process. Moreover, this brings additional freedom to the process providers, as they can easier make changes to the process, as far as it provides the necessary functionality, without being afraid to violate the agreement between the requester and the process itself.

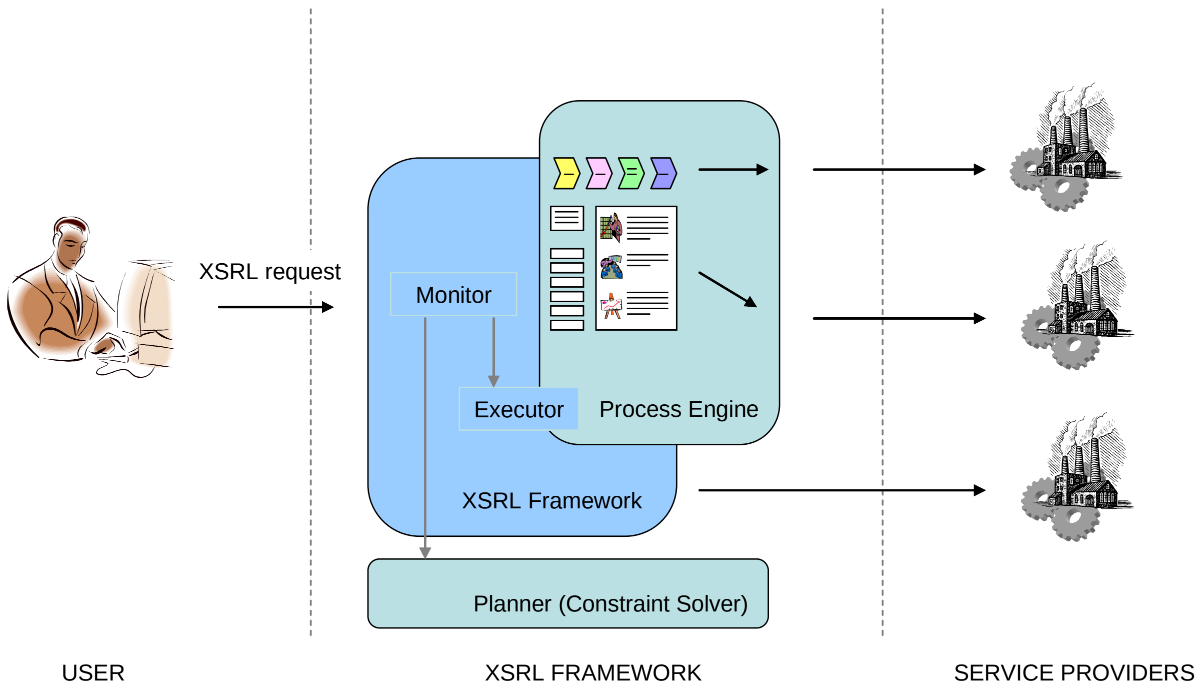


Figure 1.1: XSRL Framework.

For the request language to work, we developed a supporting framework based on the interleaving planning and execution. The high-level view on the framework is shown in Figure 1.1. When user issues a request, the planner finds the execution that potentially satisfy the request. Planning is done by encoding the process and the request into constraint satisfaction problem. The executor, built on top of the process engine, processes the execution generated by the planner until the request is satisfied or re-planning is needed. Re-planning is performed when new information if gathered such as hotel price or room availability that is available only at run-time. The monitor performs interleaving between planning and execution and search for new service providers if needed.

In general, composition in web service based environments poses an interesting and potentially rewarding challenge to the AI community as there is a need for tools and techniques to deal with the composition of loosely-coupled autonomous components, to deal with uncertainty and incomplete information deriving from the absence of a centralized control, to deal with potentially infinite possible executions. On the other hand, making the challenge feasible to approach, unlike other open environments such as those of robot planning, web service environments have the advantage of being highly structured. Operations to be invoked are syntactically specified, registries holding information on active services are available for querying, and so on.

If the planning is done using constraint programming techniques, as we propose, one can take advantage of reuse of the previously generated plans in the replanning phases. In fact, a new value returned by a service invocation may simply result in the addition of a constraint to the solution space and no further replanning is necessary. As a positive side effect of our investigation and proposal for using constraints, we highlight a set of challenges which are of general interest to constraint programming.

Additionally to the algorithms based on the encoding to constraint programming, we have developed an approach based on planning as model checking. It works with booleanized business process domain since originally planning as model checking was developed for planning domains with only boolean propositions.

Service interfaces within a business process are usually distributed between different parties (organization which may play different roles) that can make their changes in different portions of the process. A business process definition language can guarantee the consistency of service interfaces, message ordering and message invocations, but it can not be used to check process runtime properties. Safe execution of the business process can only be ensured by a monitoring mechanism that checks the runtime properties of business process and possibly recovers from assertion violations. Our framework is also able to deal handle this. The monitoring of the business process based on the assertions violations is performed in the following way. At first, assertions are published by the party who wants his assertions to be applied to business processes and monitored during execution. When the business process is executed, the framework allows only those executions to proceed where published assertions are satisfied. If an assertion is violated then the system tries to find an alternative execution path in the business process that does not violate the assertion, if any. Assertions are published on different levels: business process, role or provider. During execution, assertions defined on the business process level are always taken into account; assertions defined by roles are checked only if operations for that role are invoked; provider level assertions are considered if an action of the particular provider is necessary.

More precisely, *monitoring* is a mechanism that ensures the execution of a process is consistent with respect to business rules and user specified requests. As a business process spans several organizations, all of them expect their business rules to be taken into account when the process is executed. Business rules are supplied by service providers and are enforced on business processes that are associated with such rules during their execution. The key point of the approach is to give more control over the business process execution to end users and service providers, as schematically illustrated in Figure 1.2. In the monitoring framework service providers are allowed to issue assertions, and, by that, they have more control on how their service is accessed in the context of the business process execution.

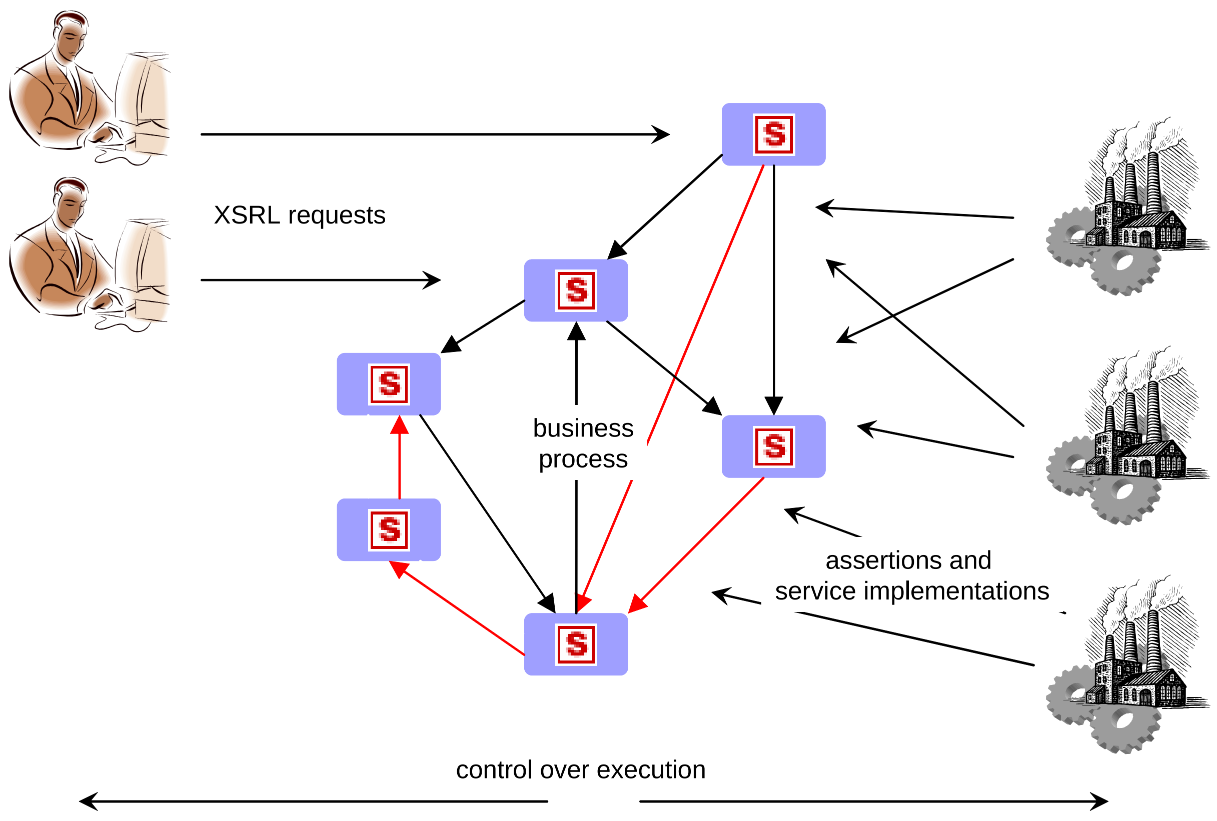


Figure 1.2: XSRL/XSAL Framework: giving more control to involved parties.

Let us now summarize the main contributions present in the thesis:

- identification of an adequate goal language over composed services with expressive power enough for the user, but, at the same time, computationally tractable;
- development of a service composition framework;
- design and implementation of a supporting framework for goal language;
- design and implementation of a supporting framework for service composition;
- implementation and evaluation of the language and service composition framework;
- development of a service monitoring framework with business rules expressed via assertions;
- discussion of applicability of the proposed assertion language for expressing and monitoring QoS properties;

1.1 An example in the travel marketplace

Let us now give the details of an example for planning a trip, that we use through the thesis for demonstrating purposes.

Consider a user requesting a trip to Nowhereland and having a number of additional requirements regarding such a trip, e.g., that the total price of the trip be lower than 300 euro, the prices of the hotel below 200 euro, avoiding using the train, and so on. To be satisfied such a request involves the interaction with various autonomous service providers, including a travel agency, a hotel company and a flight carrier. The services reside in the same travel marketplace and must follow a standard business process for the domain such as the one

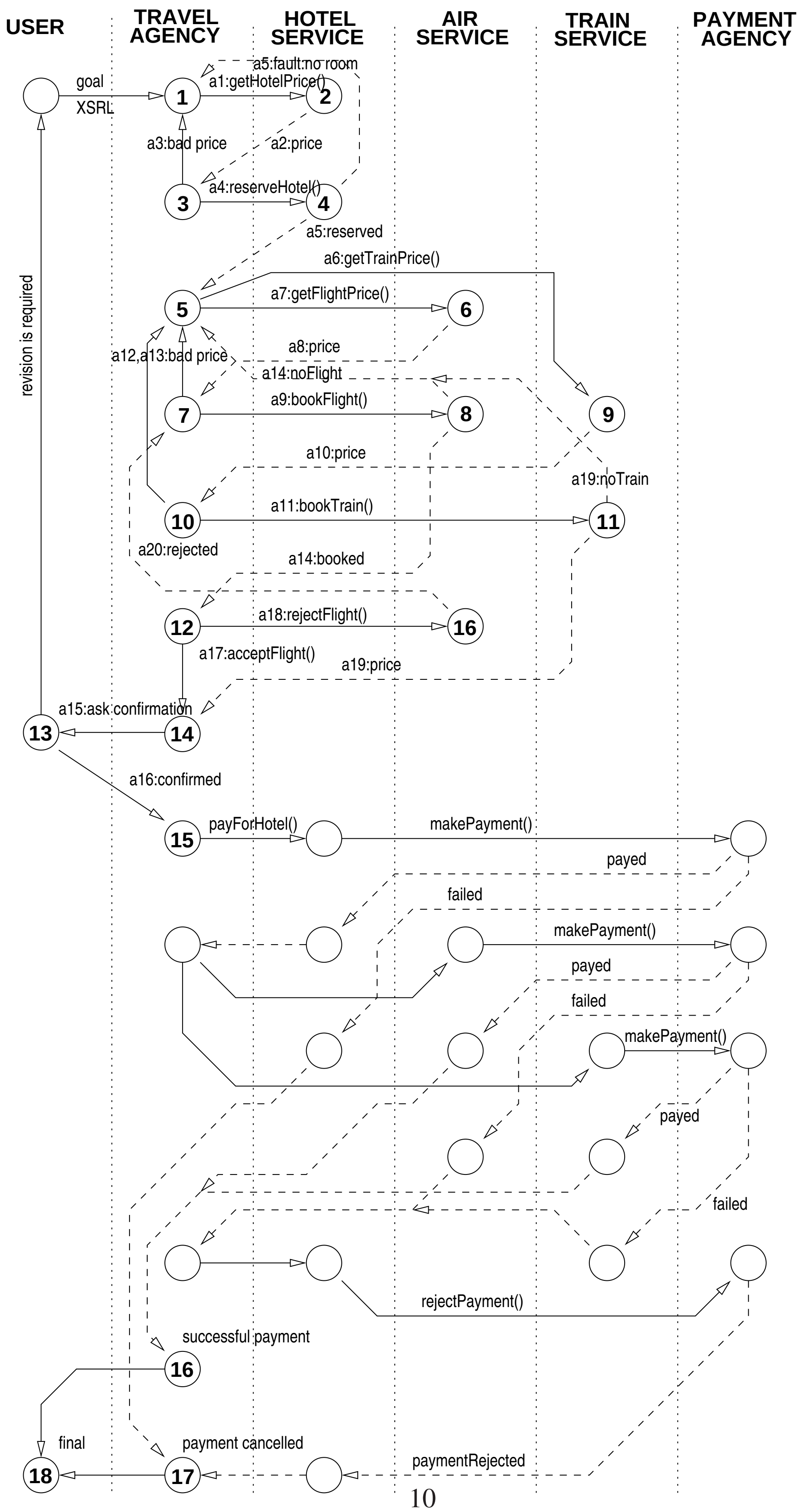


Figure 1.3: A travel business process.

exemplified in Figure 1.3. This process is modeled as a state transition diagram, that is, every node represents a state in which the process can be, while labeled arcs indicate how the process changes state. Actors involved in the process are shown at the top of the diagram. The actors include the user, a travel agency, a hotel service, an air service, a train service and a payment service.

The process is initiated by the user contacting a travel agency, hence, (1) is the initial state. The state is then changed to (2) by requesting a quote from an hotel (action a_1). The dashed arcs represent web service responses, in particular arc a_2 brings the system in the state (3). The execution continues along these lines by traversing the paths in the state transition diagram until we reach state (14). In this state a confirmation of an hotel and of a flight or train is given by the travel agency and the user is prompted for acceptance of the travel package (13).

The state transition diagram is non-deterministic. This is illustrated, for instance, in state (4). In this state the user has accepted the hotel room price however is faced with two possible outcomes, one that a room is not available (where the system transits back to state (1)) and the other one where a room reservation can be made (state (5)). The actual path will be determined at runtime when appropriate services will provide information regarding the availability for the hotel providers.

The lower part of the business process models the payment of the travel package just booked as an atomic action. This means the entire trip is payment atomic.

Services intervening in the process above may have additional requirements and business rules that need to be followed. A particular travel carrier may require advanced payment, a travel agency may want to always have explicit user's approval before committing to a package. At a higher level, different marketplaces may implement the same process but with different rules. For instance, one may additionally require that all air carriers use a specific commu-

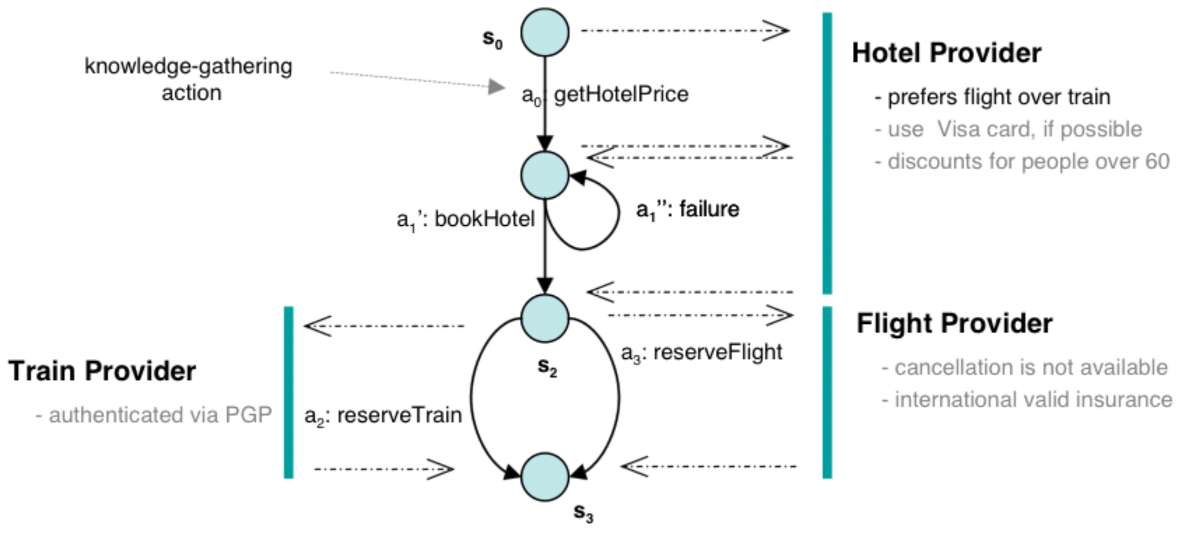

knowledge-gathering action
s_0
a_0 getHotelPrice
a_1': bookHotel
a_1'': failure
s_2
a_2: reserveTrain
a_3: reserveFlight
s_3
Hotel Provider
- prefers flight over train
- use Visa card, if possible
- discounts for people over 60
Flight Provider
- cancellation is not available
- international valid insurance
Train Provider
- authenticated via PGP

Figure 1.4: A travel service domain.

nication protocol. There also could be a set of Quality of Service requirements expressed in the same manner. For example, one of the involved parties may require additional level of security, the overall process could assert the participating services to support the transactional behavior, etc. This sort of additional business rules are what we called assertions.

However, the provided example may result too verbose to use for demonstrating concepts through the thesis. Thus, whenever convenient, we consider the simplified version limiting ourselves to hotel and ticket reservation as shown in Figure 1.4.

1.2 Published material

The work has been developed in collaboration with various people (as the publications indicate) and in particular with Marco Aiello, Mike Papazoglou and Rosella Gennari.

The work presented in the thesis is primarily concerned with the problems of composition and monitoring of the business processes based on the process reference models. We have proposed an XML Service Request Language (XSRL) [81, 84, 82] to address the issue of service composition and service monitoring. The algorithms behind the XSRL supporting framework are based on the idea of interleaving planning and execution. The planning algorithms extend ideas from planning as model checking [81, 84] and constraint programming [79, 1, 80].

Research [83, 2] in process monitoring was brought to the definition of a framework where business rules are defined by assertion statements. Assertions are published by the partners involved in the business process. The business process is executed with respect to the specified assertions and. Extensions towards monitoring of business entities constraints have been proposed in [77].

[1] M. Aiello A. Lazovik and R. Gennari. Choreographies: using constraints to satisfy service requests. In *IEEE Web Services-based Systems and Applications (WEBSA at ICIW)*, 2006.

[2] M. Aiello and A. Lazovik. Associating assertions with business processes and monitoring their execution. *International Journal of Cooperative Information Systems*, 2006.

[77] A. Lazovik. Monitoring of document-oriented business processes. In *1st European Young Researchers Workshop on Service Oriented Computing (YRSOC-05)*, 2005.

[79] A. Lazovik, M. Aiello, and R. Gennari. Encoding requests to web service compositions as constraints. In *Principles and Practice of Constraint Programming (CP-05)*, LNCS 3709, pages 782–786. Springer, 2005.

[80] A. Lazovik, M. Aiello, and R. Gennari. Encoding requests to web service compositions as constraints. Technical Report DIT-05-40, Univ. of Trento, 2005.

[81] A. Lazovik, M. Aiello, and M. Papazoglou. Planning and monitoring the execution of web service requests. In *Conf. on Service-Oriented Computing (ICSOC-03)*, Lecture Notes in Computer Sciences 2910, pages 335–350.

Springer, 2003.

[82] A. Lazovik, M. Aiello, and M. Papazoglou. Planning and monitoring the execution of web service requests. Technical Report DIT-03-049, University of Trento, 2003.

[83] A. Lazovik, M. Aiello, and M. Papazoglou. Associating assertions with business processes and monitoring their execution. In M. Aiello, M. Aoyama, F. Curbera, and M. Papazoglou, editors, *Conf. on Service-Oriented Computing (ICSOC-04)*, pages 94–104. ACM Press, 2004.

[84] A. Lazovik, M. Aiello, and M. Papazoglou. Planning and monitoring the execution of web service requests. *Journal on Digital Libraries*, 2005.

1.3 Thesis organization

The thesis is organized in the following way. In Chapter 2 the overview of currently available work on service composition is provided. Then in Chapter 3 the framework of interleaving planning and execution is introduced. In Chapter 4 planning algorithm based on encoding of the service planning as constraint problem is introduced. Monitoring of service compositions is discussed in Chapter 5. The reference implementation is overviewed in Chapter 6. In the following, the detailed description of each chapter is provided.

Chapter 2 provides the state of the art in service-oriented computing (SOC) and service composition in particular. Section 2.1 describes an overview of SOC and introduces the main issues and research directions in service composition and monitoring. Section 2.2 provides an overview of automated composition problems and concentrates on algorithms based on various AI Planning techniques. Application of constraint satisfaction to web services is described in Section 2.3. The Semantic Web view on service composition and service interface matching, in particular, is introduced in Section 2.4.

Chapter 3 is devoted to the framework for interleaving planning and execution. First, in Section 3.1 the view on service business processed as formal state-transition systems is introduced. The XML Service Request Language (XSRL) is introduced for issuing requests to business processes in Section 3.2. The problem of service planning is defined in Section 3.3. Formal semantics of XSRL is described in Section 3.4. In Section 3.5 algorithms for interleaving planning and execution are provided.

In Chapter 4 a particular planning algorithm based on encoding the service problem to constraint programming is developed. The approach consists of two parts: encoding of the domain (introduced in Section 4.1), and encoding of the service request (introduced in Section 4.2). A sample encoding of travel domain that was introduced in Section 1.1 is provided in Section 4.3.

In Chapter 5 the framework of service process monitoring is introduced. Business rules in the framework are defined by assertion statements, that are formally defined in Section 5.2. The monitoring framework itself is discussed in Section 5.3. The monitoring of the sample process is provided in Section 5.4. Discussion on how assertions can be applied to express and monitor QoS properties is given in Section 5.5.

A reference implementation of framework for interleaving planning and execution is discussed in Chapter 6. In Section 6.1 detailed view on implemented algorithms is provided. Two example snippets from Supply Chain and Purchase Order domains are provided in Sections 6.2.1 and 6.2.2.

Chapter 7 summarizes the thesis work and provides an overview of new research directions, that are opened by the presented work.

In Appendix A formal BNF notation for XSRL is provided. In Appendix B an approach based on planning as model checking is presented. Its correctness and completeness properties are discussed in Section B.1.

Chapter 2

Related work

2.1 Service-oriented computing

The Internet and the Web provide great opportunities for companies who would like their customers to have fast, easy and cheap access to company's services by publishing them on-line. Nowadays anyone can buy books, reserve hotels and tickets, check latest news and meteorological data from any device connected to the Internet. However even if the Internet gives a good potential for business-to-customer model, its infrastructure is not enough for business-to-business solutions. One of the challenging research topics on the Web today is the service-enabled marketplaces where different parties works together by sharing their business processes. *Service-oriented computing* tries to solve this problem by introducing a concept of service and framework for service publishing, discovery, binding and composition.

Services are autonomous platform-independent computational elements that can be described, published, discovered, orchestrated and programmed for the purpose of developing distributed inter-operable applications [108].

Services are self-describing, open components that support rapid, low-cost composition of distributed applications. Services are offered by service providers-organizations that procure the service implementations, supply their service descriptions, and provide related technical and business support. Since services may be offered by different enterprises on the Internet, they provide a distributed computing infrastructure for both intra and cross-enterprise application integration and collaboration [39].

An important aspect of services is the separation between the interface and the implementation part. The interface part defines the functionality visible to the external world and the way it is accessed. The service describes its own interface characteristics, e.g., the operations available, the parameters, data-typing and the access protocols, in a way that other software modules can determine what it does, how to invoke its functionality, and which result to expect in return. In a typical interaction a service client uses the service interface description to bind to the service provider and invoke its functionalities.

The implementation realizes the interface and the implementation details are hidden from the users of the service. Different service providers using any programming language of their choice may implement the same interface. One service implementation might provide the direct functionality itself, while another service implementation might use a combination of other services to provide the same functionality [108, 43].

Nowadays service-oriented computing is rapidly becoming the prominent paradigm for distributed computing and electronic business applications. Web services are one of the most important example of service-oriented computing. SOC allows for service providers and service application developers to construct value-added services by combining existing services that are resident on the Web. To achieve this, firstly, web services must be described in terms of the standard web service definition language WSDL [146], published in UDDI registry [139], and subsequently must be inter-linked to express how collections of web services work jointly to realize more complex functionalities typified by business processes. Figure 2.1 [85] shows typical interactions in a service-oriented architecture. A new web service is defined in terms of compositions of existing (constituent) services on the basis of the standard Business Process Ex-

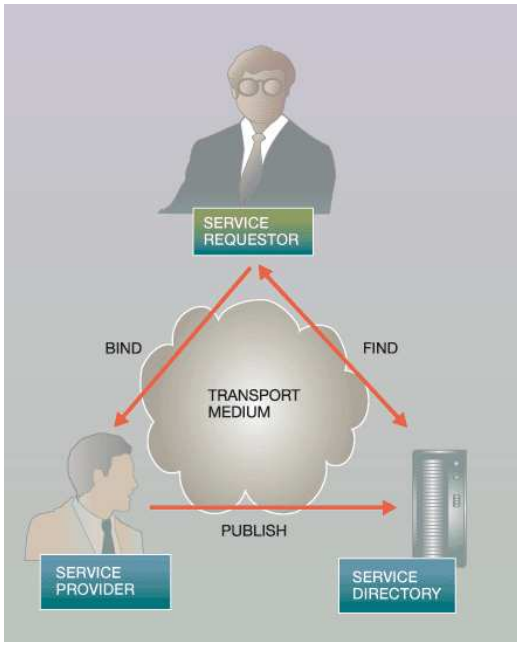


Figure 2.1: Service publishing, discovery and binding.

ecution Language for Web Services (BPEL4WS or BPEL for short [23]). BPEL models the actual behavior of a participant in a business interaction as well as the visible message exchange behavior of each of the parties involved in the business protocol. A BPEL process is defined "in the abstract" by referencing and inter-linking `portTypes` specified in the WSDL definitions of the web services involved in a process. A BPEL process is a reusable definition that can be deployed in different ways and in different scenarios, while maintaining a uniform application-level behavior across all of them. Service compositions in BPEL are described in such a way (e.g., WSDL over UDDI) that allows automatic discovery and offers request matching on service descriptions.

Service-oriented computing was mainly derived from distributed systems concepts (see, e.g., [38]). It was primarily enabled after the announcement of a set of XML protocols for platform-independent message exchange, service description and service discovery. The concept of SOC platform-independent framework is not new: most of the ideas were developed in CORBA [37]. The

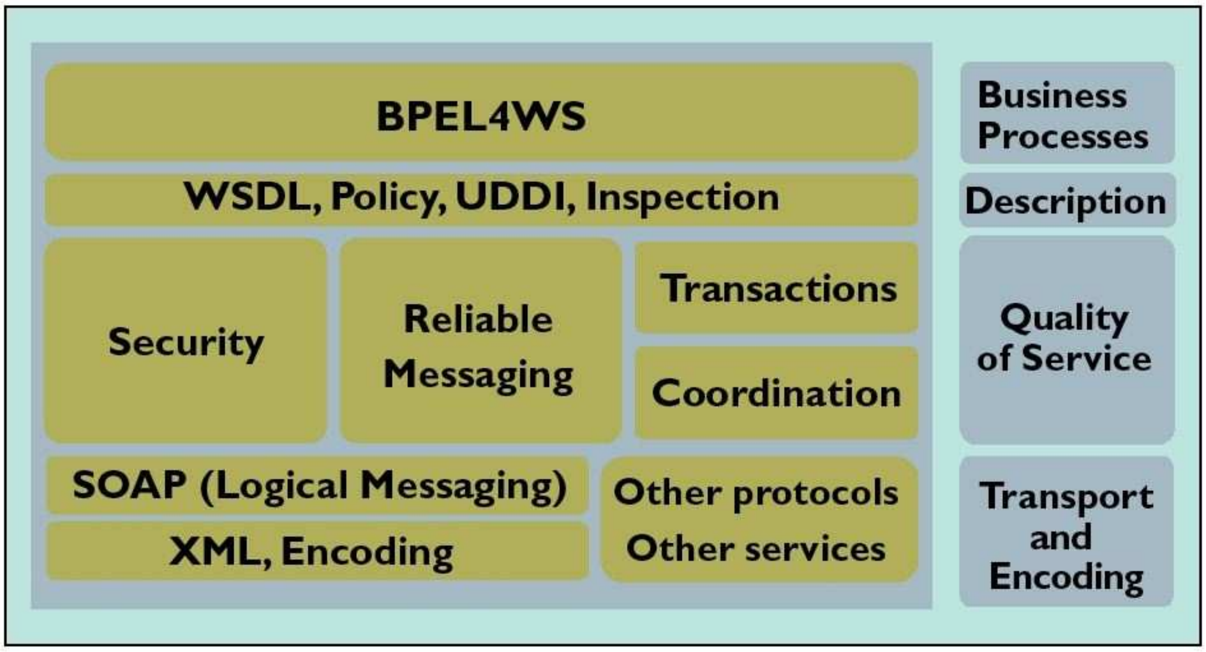


Figure 2.2: Web services protocol stack.

key difference between web service protocol stack and CORBA is that enabling protocols for web services are light-weight, allow for fast implementation, and are easy to be used in legacy applications.

Nowadays a lot of efforts are made to provide transaction and coordination behavior [36, 86, 123], service semantics [149], security mechanisms [63, 64], service discovery [91]. The web service protocol stack is shown in Figure 2.2 [39]. On the lower level of the stack one finds transport and encoding layers, in the middle protocols for the service description, security, transaction and coordination are located, and, finally, on the top level the protocol stack has business process composition layer. A good introduction to service-enabling protocol stack can be found in [109, 111].

2.1.1 Service composition

Service composition is the cornerstone challenge to success of service-oriented computing. Several initiatives have been proposed to enable integration between heterogeneous systems. In particular, as shown in Figure 2.2 the web service protocol stack [39] includes the Web Service Description Language [146], the Simple Object Access Protocol [131], Universal Description, Discovery and Integration [139] that allows platform- and language-independent service publishing, discovery and invocation. Business Process Execution Language for Web Services [23] is focused on representation of web service executions, where composition is known in advance. Choreography Description Language defines, from a global viewpoint, observable inter-enterprise behavior, where ordered message exchanges result in accomplishing a common business goal [67].

Despite all the efforts, service composition is still an extremely complicated task. Complexity comes from different places:

- the number of services and partners available on the Web is high and steadily increasing, making it difficult to choose the right service to find and invoke;
- in a true service-oriented architecture, there is no single owner of the business process, that is, every change to a process has to be approved by all involved parties. Therefore, having consistent and stable business processes that satisfy business goals of all participants and ensure correctness at runtime is hard to be achieved;
- the execution of a business process depends on the behavior of involved partners that is not known when the process is designed, thus, designer of the process has to take into account all possible service behaviors.

That is why having a mechanism for automatic or semi-automatic service composition is crucial for successful enterprise application integration. Several approaches have been proposed to achieve these issues. Service composition is somewhat similar to composition of workflows [28, 42, 140] and techniques developed for workflows can be reused for composition of services. For example, in [30] it is proposed a configurable approach to service composition. However, workflow composition frameworks do not take into account issues specific to service-oriented computing: dynamic binding, highly heterogeneous environments, absence of a single ownership and control over process execution. Service orchestration based on object-oriented data models is presented in [49]. In [104] service composition rules are used for governing the business process construction. In [6] authors model the service composition problem as a mixed integer linear problem where both local constraints and global constraints can be specified. A local constraint allows selection of a Web service according to a desired characteristic. Global constraints are constraints over the whole composite service execution. There have been proposed several approaches for service compositions, e.g., knowledge-based semantic web service composition [31], service discovery and composition based on semantic matching [106], semi-automatic composition of web services based on semantic descriptions [127]. All these approaches work under the assumption of having available rich semantic service description and run-time information. In contrast to this, in a pure service-oriented environment, there is little semantic description and, on the other hand, one deals with incomplete knowledge about service behavior and required information is gathered and analyzed during execution.

2.1.2 Monitoring of service compositions

In the web service literature there are several approaches dealing with the monitoring of the assertions over service-enabled business processes. The WS-Policy framework [144] provides a general purpose model for describing a broad range of service requirements, preferences, and capabilities. Typically it is used when the provider describes the set of conditions the requester should satisfy before invoking the service. RuleML [58] is a powerful technique for expressing business rules over semantically annotated service. On the negative side is the lack of any support for runtime monitoring of the business rules. Extensions of RuleML for distributed service-oriented environments are proposed in [100]. The eFlow [29] is a system that supports composite services for highly dynamic business environment. However, most of the previous work is stressed on the verification of business processes [141, 119], rather than on tuning and re-composition of the process on-the-fly if the process is failed to satisfy all assertions, as it is done in this work.

A *service-level agreement (SLA)* is a contract between providers and clients that specifies, usually in measurable terms, that a service provider and a service client agree before starting to interact. Service-level agreement is often described as series of *Quality of Service (QoS)* expressions. QoS includes both functional and non-functional service quality attributes, such as service cost, service performance metrics (e.g., response time and availability), security and transactional attributes, scalability and reliability [109]. The WS-Policy [144] provides a language that may be used to describe a broad range of service requirements and preference capabilities.

In a distributed service-oriented computing environment, service consumers like to obtain the guarantees related to services they use, often connected to quality of service. However, quality of service and other guarantees that depend on actual resource usage cannot be advertised as an invariant property of a service. Instead, the service consumer must obtain state-dependent guarantees from the service provider, represented as an agreement on the service and the associated guarantees. Additionally, the guarantees on service quality should be monitored and service consumers may be notified of a failure to meet these guarantees. WS-Agreement [89, 90] defines a language and a protocol for advertising the capabilities of service providers and creating agreements based on creational offers, and for monitoring agreement compliance at runtime. In [88]

Cremona: An Architecture and Library for Creation and Monitoring of WS-Agreements is introduced. Extensions to WS-Agreement that allows for run-time monitoring of running service-level agreements is proposed in [50].

2.2 Automated composition using AI Planning

One of the most challenging areas in SOC is the composition of business processes. Many researchers efforts aim at creating techniques to automatically compose different services in order to achieve complex business goals. It is shown (e.g., [132]) that problems in automatic business process composition and service request processing and monitoring are similar to those in planning. Temporally extended goals, i.e., goals expressing not only desired states to achieve but also conditions on how these are to be reached, are on expressive way of defining complex business goals.

2.2.1 Automated planning as a part of AI

One of the closest areas in AI that deals with the problems similar to automatic service composition is *automated planning*. Planning can be seen as a process of reasoning over a set of goals that have to be achieved by organizing a given set of possible actions in a way that their execution satisfies the goals. Actions behavior is usually described via effects. Actions may also have preconditions, denoting that some particular action can only be executed if its preconditions are satisfied. Planning is known to be a complex problem [126], therefore it is generally looking for good feasible plans rather than optimal ones. Since there are various types of actions, there are also various types of planning: robot motion and navigation [68, 75], scheduling [51, 76, 128], space applications [15, 116], web service composition [94, 127, 132].

There are two main kind of planning approaches: domain-dependent and domain-independent planning. A natural, and often more effective, approach is to use domain-dependent problem representation with algorithms and techniques adapted for the specific problem. However, there are many commonalities between all the planning models and algorithms used. And, since it is much more costly to address every new arising planning problem with a specific approach, it is wise to use domain-independent approaches whenever possible.

To be able to deal with various types of service composition problems, that are possibly defined in different ways and standards, we are going to consider only domain-independent planning, tackling web service specifics in an abstract, domain-independent way. This allows us to deal with a wide range of business domains without spending too much time to add domain-specific knowledge.

Let us begin by formalizing the general planning problem. We refer to this model throughout the rest of the work. The planning problem representation given here is adapted from the conceptual planning model provided in [101].

Most of the planning approaches rely on a general model of a *state-transition system* (also called *discrete-event system*). Formally, it is defined as follows:

Definition 1 (State-transition system). *A* state-transition system *is a tuple* $\Sigma = \langle S, A, E, \gamma \rangle$*, where:*

- $S = \{s_1, \ldots, s_n\}$ *is a finite or recursively enumerable set of states;*
- $A = \{a_1, \ldots, a_m\}$ *is a finite or recursively enumerable set of actions;*
- $E = \{e_1, \ldots, e_l\}$ *is a finite or recursively enumerable set of events;*
- $\gamma : S \times A \times E \to 2^S$ *is a state-transition function.*

A state-transition system can be represented as a directed graph with nodes in S with edges from s' to s'' if there exists a and e such that $\gamma(s', a, e) = s''$.

Often it is convenient to define two state-transition functions instead of one:

- $\gamma : S \times A \to 2^S$ is a state-transition function that denotes transitions caused by actions;

- $\gamma : S \times E \rightarrow 2^S$ is a state-transition function that denotes transitions caused by events.

Both events and actions can cause a state transition. However, there is an important difference between them. Actions are transitions that are controlled by the plan executor. That is, if $\gamma(s, a)$ is not empty then invoking action a in a state s results in one of the states $\gamma(s, a)$. Events represent *non-controlled* transitions that may *possibly* bring the system from state s to one of the states $\gamma(s, e)$ if $\gamma(s, e)$ is non empty.

Given a state-transition system Σ, the planning problem is to find which actions to apply in which states in order to achieve some goal. A *plan* is a structure that contains the information for the executor regarding which action to execute at the following step. The goal may be defined in several forms [101]:

- as a set of *goal states* S_g. In this case, the goal is satisfied if the planner builds a plan, which, when executed, brings the system to one of the goal states;

- as a set of conditions over sequence of states visited by the system to be satisfied. For example, one might require to avoid some states, or define states that must be visited during the execution, or prefer one sequence of states to another;

- as an utility function that is specified for each state with penalties and rewards, if a particular state is visited. A goal is either to optimize the utility function or achieve some desired value.

Typically planning is performed off-line, and then the synthesized plan is executed. Sometimes the planning system may want to react according to new sensed information from execution by re-planning or tuning the originally synthesized plan. In this case, the conceptual planning model is extended with an *executor* (also known as *controller*). The resulting model is shown in Figure 2.3, and it consists of three important parts:

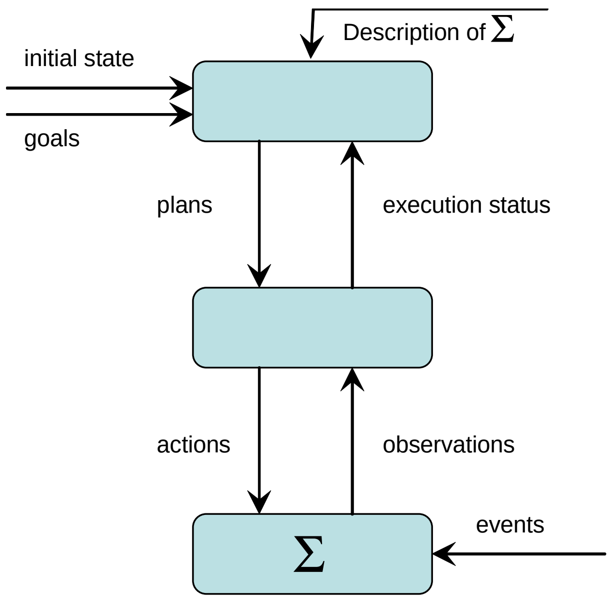


Figure 2.3: Conceptual planning model (adapted from [101]).

- *planner*: given a description of the state-transition system, its initial state and the goal, it works off-line to produce the plans;
- *executor*: given as an input the current state of the state-transition system, it executes actions according to a synthesized plan. The executor may try to sense the current state of the system. In general, it may have only partial knowledge of the state of the system. Partial knowledge can be modeled as an observation function $\eta : O \rightarrow 2^S$ that maps observations $O = \{o_1, \dots, o_k\}$ to possible current states of the system. An executor interacts on-line with the system;
- *state-transition system* Σ: represents the planning environment. It evolves according to invoked actions and external non-controlled events.

The problem of automated planning is known to have high complexity [26, 87, 118]. That is why the first planning algorithms were applied to the restricted version of the general conceptual model. The restricted version of planning is known as *classical planning*, or *STRIPS planning*, referring to STRIPS, one of the first planners [48]. In STRIPS, each action has a precondition, add and delete lists, and these may contain first-order logic expressions. However, providing a well-defined semantics for this formulation has proved to be hard; and, in a subsequent work, preconditions and effects are allowed to contain only atomic propositions [101].

Classical planning considers the following assumptions when dealing with the planning problem [101]:

A0 (Finite Σ) The state-transition system Σ has a finite set of states;

A1 (Fully observable Σ) The state-transition system is *fully observable*, i.e., one has full knowledge about the current state of the system. In this case the observation function η is a one-to-one function;

A2 (Deterministic Σ) The state-transition system Σ is *deterministic* if for every state s, and for every action or event u, every transition has only one possible destination state: $|\gamma(s, u)| \leq 1$;

A3 (Static Σ) The state-transition system Σ is static, i.e., the set of non-controlled events E is empty. The systems remains in the same state until the executor invokes an action;

A4 (Restricted goals) The goal is restricted if it is represented via a set of explicitly defined goal states. Extended goals such as states to be avoided or constraints over trajectories or utility functions are not handled;

A5 (Sequential plans) A solution plan to a planning problem is an ordered finite sequence of actions;

A6 (Implicit time) Actions and events have no duration, they are instantaneous state transitions;

A7 (Off-line planning) The planner is not concerned with any change that may occur in Σ while it is planning. It plans for the given initial state and a goal ignoring the current dynamics of the system.

Traditional search-based algorithms were not always successful with tackling the classical planning problem [101]. In the late 90s were investigated several alternative approaches to classical planning algorithms known to be a neo-classical planning algorithms, including, but not limited to: graph-plan techniques [20], encoding of the planning problem to SAT [65, 66] or CSP [133], application of linear integer programming [142]. Most of the new created algorithm outperform the traditional approaches [101]. However, the main advantage of new approaches is the following: combining with other fields, like integer programming or CSP, it allows to use the benefits of the using fields, for instance, CSP allows natively to use numeric variables, integer programming allows to use optimal planning, etc. Potentially all this allows some relaxation of the restrictive assumptions, that usually allows for more expressive or more compact planning problem representation.

In [69] a form of template planning based on hierarchical template networks and constraint satisfaction is introduced. The framework provides the infrastructure to rapidly construct new applications that extract information from multiple Web sources and interactively integrate the data using a dynamic, hierarchical constraint network. In [137, 135, 136] it is proposed data integration techniques to support dynamic integration of data from web services and support dynamic composition of web services from existing web services.

In [13] services are modeled as execution trees represented by deterministic finite state machines (FSM). The desired service (the goal) is also represented by deterministic FSM and may be partially specified [14].

In [95] the Golog planner is used to automatically compose semantically described services. Actions in [95] are described as Situation Calculus actions [56], and encoded in OWL-S. The approach is exploiting an OWL-S ontology of services by automatically instantiating the user specification with services contained in such an ontology. When the outcome of service is unknown, the successor state is defined by executing knowledge-gathering services.

However, classical planning could not tackle some problems where only partial knowledge of the domain is available, goals are not expressive enough, there are no non-controlled events. Unfortunately, these issues are important in the web service environment: information about service implementation is not known before executing some of its operations, most of the service operations are non-deterministic, goals constraint numeric variables as price or resource quantity. There were several proposals for service composition planning with relaxed classical planning assumptions.

In *Hierarchical Task Network (HTN)* approach for planning is done by problem reduction: the planner recursively decomposes tasks into subtasks, stopping when it reaches primitive tasks that can be performed directly by planning operators. In order to tell the planner how to decompose nonprimitive tasks into subtasks, it needs to have a set of methods, where each method is a schema for decomposing a particular kind of task into a set of subtasks. The HTN planner SHOP2 was applied for web service composition in [148]. Extension of HTN planner SHOP2 to gather information from the web in order to do web-service composition is provided in [72]. In [8] an extension of SHOP2 is introduced to do web service composition in environments when the world is changing while the planning is going on.

Classical planning problem based on plan-graph techniques are extended to deal with uncertainty in [129] and with limited form of partial observability in [143].

Planning as model checking approach is used for domains under uncertainty with non-determinism, partial observability, and extended goals. It is discussed in details in Section 2.2.2.

Another example of planning under uncertainty and non-determinism is an approach based on Markov-decision processes (MDP) [101, 22]. MDP takes a very different view on planning comparing to other planning techniques. The domain is define as a state-transition system, with transitions labeled with probabilities. Goal is represented in terms of state rewards (or action costs). The planning problem is seen as an optimization problem. A utility function can easily express preferences on states and actions, and many applications require such preferences, as for example, web services. On the other hand, sometimes it is not easy to formulate goals in terms of state rewards, and transition probabilities are not always easy to define for many applications. Probabilities in most cases are statistical estimates, they are not always accurate and often not available, what is typical, for example, for web service composition problems. There were made several attempts to use MDP planning for a particular service composition issues. In [9, 10] the goal for MDP planning is expressed using temporal logic.

In [41] it is proposed an approach based on MDPs to model workflow composition. To account for the uncertainty for dynamic environments, authors interleave MDP-based workflow generation and Bayesian model learning. In [150], authors present a hierarchical approach for composing web processes that may be nested – some of the components of the process may be web processes themselves. They model the composition problem using a semi-Markov decision process (SMDP) that generalizes MDPs by allowing actions to be temporally extended. Authors use these actions to represent the invocation of lower level processes whose execution times are uncertain and different from simple service invocations. In [53] it is discussed a method for dynamic web service composition, which is based on Markov Decision Processes (MDP). It is defined on the base of QoS description and addresses the issue of selecting web services for the purpose of their composition. Web service composition patterns including sequential, conditional, parallel and iterative are modeled in MDP.

2.2.2 Planning as model checking

Planning by model checking (where model checking is an automatic technique for verifying finite state concurrent systems [34], see Section 2.2.2, is an approach to planning under uncertainty that deals with non-determinism, partial observability, and extended goals. The proof in model checking can be seen as a plan in planning [101]. The planning as model checking approach is based on the following concepts:

- a planning domain is a non-deterministic state-transition system, where an action from a single state may lead to several different states. The planner does not know the outcomes that will actually take place, when the action will be executed;
- formulas in temporal logic express reachability goals, i.e., set of final desired states, as well as conditions on the entire plan execution paths. They can express requirements of different strength that takes into account non-determinism;
- plans result in conditional and iterative behaviors, and, in general, they are more expressive than simply mapping between states and actions to be executed;
- given a state-transition system and a goal expressed as temporal formula, planning by model checking generates plans that control the evolution of the system so that all the system's behavior makes the temporal formula true. The plan validation process can be formulated as a model checking problem;
- planning as model checking can use symbolic model checking techniques. The set of states can be represented as propositional formulas and searching through the states is performed by doing logical transformations over propositional formulas.

Planning as model checking for extended goals was first introduced in [73, 113] and is implemented in a planner called MBP [16]. Extensions toward interleaving planning and execution in the above context are reported in [17]. The latter work emphasizes on the state explosion problem rather than information gathering. There are several attempts to apply the approach to solve web service composition problem [115]. However, the traditional planning based on model checking does not handle numeric values, which is an important issue for web service environments, where numeric resources are used, e.g., price and time. Another critical issue is that the planner MBP [16] is designed to work off-line, and, by that, it has to take into account all possible service outcomes, while it would probably more efficient to do knowledge gathering at run-time, and then perform re-planning according to new available knowledge.

Our work can be seen as a further extension of ideas developed in planning as model checking. However, there are important differences that are crucial for services-oriented computing. Together with preserving the high expressiveness of EaGLe [73], we extend them by introduction of preferences, optional goals and numeric constraints. That required us to encode the planning problem as a constraint satisfaction to deal with numeric constraints efficiently. Another difference is in the way we deal with incomplete knowledge: we introduce the framework of interleaving planning and execution, that gathers new knowledge from services at run-time, and performs re-planning if needed.

Another implementation of the planning as model checking is MIPS [44], which is based on BDDs [24]. The main strength of MIPS is its pre-compilation phase that leads to a reduction of the state description length. However, MIPS is limited to deterministic domains.

Using temporal formulas in planning is not limited to planning as model checking. Several authors [12, 40] used them for guiding the search, in case planning is domain-specific and some knowledge about domain is known in advance. In contrast, in the thesis we focus on the general planning algorithms assuming we have no knowledge about the business domain we work with.

Model checking

Hardware and software systems inevitably grow in scale and functionality. Because of this increase in complexity, the likelihood of subtle errors is much greater. Moreover, some of these errors may cause catastrophic loss of money, time, or even human life. A major goal of software engineering is to enable developers to construct systems that operate reliably despite their complexity. One way of achieving this goal is by using formal methods, which are mathematically based languages, techniques, and tools for specifying and verifying such systems. Use of formal methods does not a priori guarantee correctness. However, they can greatly increase our understanding of a system by revealing inconsistencies, ambiguities, and incompleteness that might otherwise go undetected [33].

Model checking is a set of formal techniques that is used to verify systems against its specifications. Model checking includes various set of problems: from infinite to finite systems, from linear to branching time structures. In this work we assume that the system is described using the Kripke structure, as it is defined in [34] and the specification is expressed by the temporal formula either linear or branching [45]. A Kripke structure is a type of nondeterministic finite state machine used to represent the behavior of a system. It is basically a graph whose nodes represent the reachable states of the system and whose edges represent state transitions. A labeling function maps each node to a set of properties that hold in the corresponding state. Temporal logics are traditionally interpreted in terms of Kripke structures [34].

There are several research efforts made in the formal verification of the service compositions, in particular, for BPEL. In [122] BPEL is translated to a Petri net model and then some correctness criteria are verified by further translation of the model to a model checking problem for alternating temporal logics.

Analysis of BPEL interactions on the data level is introduced in [52].

However, verifications are usually done off-line, when there is no or little information about service is available. Non-deterministic nature of services also reduces the use of formal verification techniques in service-oriented computing: the number of possible failures is usually large and not static. It is unfeasible to describe such heterogeneous dynamic system with up-to-date formal model to allow complete formal verification. That is why, even verified systems require careful monitoring of web service execution, especially monitoring of composed services.

2.3 Constraint satisfaction

Constraint programming is the study of computational systems based on constraints. The idea of constraint programming is to solve problems by stating constraints (requirements) about the problem area and, consequently, finding solution satisfying all the constraints [11]. In constraint programming, constraints are usually defined in forms of logical relations among several variables, each taking a value in a given domain. By that, the constraints restrict the possible values that variables can take.

A *constraint satisfaction problem (CSP)* is defined as [138]:

- a set of variables X = $\{x_1, \ldots, x_n\}$;
- for each variable x_i, a finite set D_i of possible values (its domain), and
- a set of constraints restricting the values that the variables can simultaneously take.

A *solution* to CSP is an assignment to the set of variables such that all its constraints are satisfied. One may want to find an *optimal* solution, if some objective function is given over CSP variables [138].

There are several approaches developed to solve constraint satisfaction problems. Search with *backtracking* [102] is a method of solving CSP by incrementally extending a partial solution that specifies consistent values for some of the variables, towards a complete solution, by repeatedly choosing a value for another variable consistent with the values in the current partial solution. Another approach, forming consistency techniques, is based on removing inconsistent values from variables domains until the solution is found. These methods are often incomplete [70, 92] and used together with search. For consistency techniques it is convenient to represent the constraint satisfaction problem (in a reduced binary form [11]) as a graph, where nodes are variables and edges are labeled by constraints [97]. The most widely used technique is *arc consistency*. It removes the values from the variable domains that are inconsistent with the corresponding binary constraint.

The combination of both – search and consistency – is usually used to find the solution for the constraint satisfaction problem. For example, *look back* techniques (*backjumping* [54] and *backchecking* [60]) use consistency checks among already instantiated variables. *Look ahead* schemas are used to prevent the conflicts in the future, e.g., *forward checking* [99].

As an alternative to complete search algorithms, generalized stochastic searches can be applied to find the solution for the constraint satisfaction problem: e.g., *hill-climbing* [102], *random-walks* [124], and *tabu-search* [57].

There are several research initiatives in the area of constraint satisfaction towards service-oriented computing. For example, the authors of [46] introduce the framework of open constraint satisfaction problems, that are common to this work. As they pinpoint, for one party in the service it is often impractical to provide the other parties with full information on its own constraints. However, the framework proposed in this work encompasses this view in that it separates the constraint reasoner, the executor and the monitor: so, in our framework, it is not the constraint reasoner who is responsible for requesting and controlling the flow of information.

In [125] it is proposed a framework for handling workflows based on constraints, and a language for describing the workflow. The methodology adopted is similar to ours; e.g., their workflow language is high-level and is based on constraints. However, in the thesis we tackle non-deterministic processes, while [125] does not; our high-level language and constraint modeling allow for optional requests, preferences, execution constraints, and not only sequencing, price, task constraints.

To our knowledge, few constraint systems allow users to directly express complex soft constraints; most of them only allow users to express preferences on basic constraints via reification [4]. On the other hand, there is a considerable amount of theoretical work in the soft constraint literature; e.g., see the work [18, 19, 21] for an introduction to soft constraints and inference algorithms, and [5] for more recent work on soft constraint reasoning at the interface with game theory. Despite solid theoretical work done, to our knowledge there is no stable widely available implementation of constraint solver that reasons over preference constraints. That is why we use a general purpose constraint solver [32] and deal with preferences by adopting reification techniques [4, 121].

Another track of the constraint literature is focused on the resource allocation problem in a distributed scenario and its optimization (e.g., [59]); however this is not an issue in this work, as we abstract from this problem. In fact, in this work we are concerned with web service-oriented business processes and interactions with them, and we propose a modeling as constraint-based problems. Instead of having external constraints as, for instance, in the open constraint setting of [46], we introduce two sorts of variables in our model: variables the constraint system is free to choose values for, and variables that only external parties can choose values for. Accordingly, the latter are called *non-controlled* variables, and any solution to the problem should be independent of their possible values.

2.4 Semantic Web

Semantic Web adds a machine-interpretable information to web contents (and to web services in particular) in order to provide capabilities for automatic discovery, composition, invocation and interoperation [74]. There are several formal languages [35, 130] proposed to support rich declarative specification at a wide variety of information about web services [112] and agents in the multi-agent environments [62, 134] in order to allow involved parties for automatic discovery and interaction. This languages are based on knowledge representation languages and ontologies [96, 105, 106]. However, most efforts in Semantic Web research on service composition is concentrated around discovery of new services that match each other and the initial service request. These approaches usually ignore complexity of service implementations, focusing on service interface and service ontology matching [71, 117, 127].

Despite of its obvious promise, Semantic Web has a number of unresolved issues to be directly applied for service composition problem. Only OWL-S [35] defines composition at a semantic level but its model does not support dynamic service selection. The heterogeneity of services is a vital issue to be resolved yet data and process mediation is only mentioned in the WSMO [47] initiative. Web service interoperation in Semantic Web does not have a clear semantic model, choreographies are ambiguous when interpreted in OWL-S [35], and WSMO [47, 120] does include choreography as a part of its conceptual framework. Significantly, no initiative has yet considered for other web services requirements, such as transactionality, security, trust and execution monitoring [103] (solely QoS issues which have been well defined in Meteor-S [112]). Some of the issues may be solved by enriching the available semantic definitions, that brings large set of data needed to be provided to describe the specific service. The drawback of this approach is that it is difficult to maintain the semantic descriptions up-to-date, making altering the service implementations a very difficult process. In a web services world it is even more difficult, since one has to ensure that all services share the same view on the data semantics and ontology used. Several authors addressed this issue by introducing a data mediator to bridge different vocabularies [25, 120]. Such mediators are able to translate the output of one service to a suitable input of another.

In contrast, our work, as most of the work done in web service composition (see, for example, [132]), does not require all the complexity of rich semantic ontology-based definitions limiting the semantic description to the definition of effects and preconditions of specific services. Using this approach, other important information about service is usually (as it is done in this work) gathered at run-time by intercommunicating with specific service providers. Within weak semantic descriptions, the issue of data semantic heterogeneity between services is open, but as real world experience demonstrates, it is reasonable to assume that involved parties can agree on the shared view on the meaning of the data passed between services.

Chapter 3

Interleaving planning and execution

3.1 SOA business processes

Service-oriented computing allows service providers and service application developers to construct value-added services by combining existing services that are resident on the Web. To achieve this, firstly, services must be described in terms of a standard definition language (e.g., in case of web services in WSDL [146]) and subsequently must be inter-linked to express how collections of services work jointly to realize more complex functionalities typified by business processes. A new service can be defined in terms of composition of existing services on the basis of the standard business process languages, e.g., in BPEL [23]. These process definitions model the actual behavior of participants in a business interaction as well as the visible message exchange behavior of each of the parties involved in the business protocol. Such reusable definitions can be deployed in different ways and in different scenarios, while maintaining a uniform application-level behavior across all of them.

In this thesis, we consider business processes as a means to represent the control flow of business logic and applications. This is achieved by introducing the notion of a state and an action. A *state* represents the state of the process execution. An *action* represents a business activity, which is modeled as a transition between given states. Each action is executed on behalf of a role. A *role* represents a set of business operations that relate to the same party, e.g., a travel agency. Each role has a number of providers associated with it. Providers are found by interacting with service registries. A *provider* is the actual party that implements a role, e.g., a specific travel agency. It is convenient to define the notion of a *process variable*, which is a variable associated with a process changing values, e.g., travel packages, hotel reservations, and so on, as the process progresses through its execution path and its states change. The use of process variables guarantees that the execution of a business process can be monitored during execution as the process traverses a set of states where constraints may need to be applied to these variables. Constraints on the variables represent user goals and preferences.

The business process defined in this way is very similar to an AI planning domain. However classical AI planning approaches, as described in Section 2.2, cannot be directly applied since it is limited by the assumption of determinism: it is assumed that the exact outcomes of the actions are known in advance, that is, for any given plan and an initial state, the world will evolve towards a single fully predictable state.

However, service-oriented business processes are typically non-deterministic. Non-determinism is caused by several reasons, both functional and non-functional. A bank service may reject the payment because some business rules are violated as well as because of service internal technical problem. That is why, when modeling the business process, a more realistic assumption is that the world is non-deterministic: an action may have several possible outcomes, with no advance knowledge on which one will occur. The second source of uncertainty in web services scenarios is the service implementations themselves: the values for most of process variables representing service capabilities (e.g., ticket price) are not known before execution. The system must invoke the service operations to get this knowledge available. This type of uncertainty in the planning literature is usually referred to as planning under *partial observability* [101].

In the following, we adapt the general definition of planning domain and goal provided in Section 2.2. The planning domain represents the possible system evolution, while the goal describes the set of conditions that the system is desired to satisfy after the execution. In the context of this work, service planning domain is extracted from business process definitions. As for a goal language, an XML Service Request Language is introduced.

The *service domain* is the core formalization of the business process that is expressed as a state-transition system with, possibly, non-deterministic actions. The goal language is later introduced to express goals and preferences from the user request. Formally, the service planning domain is defined as follows:

Definition 2 (Service domain). *A* service domain *is a tuple* $\mathcal{D} = \langle \mathcal{S}, \mathcal{V}, \mathcal{A}, \mathcal{R}, r, \mathcal{T}, \mathcal{F} \rangle$*, where:*

- $\mathcal{S}$ *is a set of* states*;*
- $\mathcal{V}$ *is a set of* process variables*; each* $\mathcal{V}$ *variable* v *ranges over a* process domain D_v*;*
- $\mathcal{A}$ *is a set of* actions*, representing process operations. Each action is associated with some role from* R *by the mapping function* $r : \mathcal{A} \to \mathcal{R}$*;*
- $\mathcal{T} : \mathcal{S} \times \mathcal{A} \to 2^{\mathcal{S}}$ *is a* transition *function that represents the control flow of the business process; if* $\mathcal{T}(s, a)$ *contains at most one state then* a *is* deterministic *in* s*, else it is* non-deterministic *in* s*;*
- *for each* v *in* $\mathcal{V}$*,* $f_v \in \mathcal{F}$ *is an effect function* $f_v : \mathcal{S} \times \mathcal{A} \times 2^{D_v} \to 2^{D_v}$*, where* D_v *is the domain of the process variable* v*;* $f_v(s, a, D'_v)$*, with* $D'_v \subseteq D_v$*, is the* effect *of* a *on the variable* v *in* $s \in \mathcal{S}$*.*

To account for all the states the system can enter after the execution of an action (in particular, of a non-deterministic action), we introduce the notion of action outcome:

Definition 3 (action outcome). *Given a service domain $\mathcal{D}$ the triple (s, a, s') is an* outcome *of a in s if $s' \in \mathcal{T}(s, a)$. If the set of $\mathcal{T}(s, a)$ is non empty, then one of its elements is called* normal.

The *normal* outcome of an action represents the "correct" execution of the business process, whereas the other outcomes, if present, represent exceptional conditions. For example, the result of booking a room operation may be either successful reservation representing the "correct" behavior, or exceptional one in case no room is available. The exceptional outcome may also arise from non-functional source, e.g., due to a server crash. In the web services world exceptional behavior is represented by the failures that are defined in WSDL definitions.

Summarizing, the service domain represents the logic of a business process; at every step in its execution, the process is in a state from which a number of actions can be executed; such an action can also be non-deterministic, i.e., it can lead the process to different outcomes. Roles, which represent service interfaces, are associated to actions and implemented by service providers, e.g., the `Hotel` role may be implemented by particular service providers such as `Hilton Hotel` and `Astoria`.

We distinguish a special type of action that acquires information from a particular web service implementation by executing this action, e.g., getting a hotel price from a particular hotel provider. We name this action *knowledge-gathering*. Invocation of knowledge-gathering actions updates the system internal view on the actual world. In this work, it additionally triggers the invalidation of the current plan and results in replanning.

A variable v is said to be *unaffected* by an action a for a particular outcome o if the action does not affect the variable for all possible values x: $\forall x : f_v(a, o, x) = x$.

The concept behind the presented formalization of the service planning domain is that a given business process is, at any instant of its execution, in a state from which a number of actions can be performed to move to a new state, possibly non-deterministically. Roles, which represent service interfaces, are associated to actions and implemented by service providers, e.g., `Hotel` role that may be implemented by particular service providers `Hilton Hotel` and `Astoria`.

3.2 Expressing goals and preferences: XSRL

In many situations it is desirable to let the user gain explicit control over the execution of a service business process and dynamically change the nature of the web service interactions conducted with a particular business partner depending on user goals and preferences. Consider for example the case of a traveler deciding to change his hotel reservation to take advantage of an unexpectedly lowly priced weekend offer. This implies that process execution must be made adaptable at run-time to meet the changing needs of users and businesses. Obviously, currently available business process specifications do not allow for the required flexibility to react swiftly to unforeseen circumstances or opportunities as choices are predefined and statically bound in process definitions. To meet such requirements serious re-coding efforts are needed every time that there is a need for even a slight deviation.

Such advanced functionality can be better supported by a service request language and its appropriate run-time environment to allow users to express their needs on the basis of the characteristics and functionalities of standard business processes. A service request language provides for a formal means of describing desired service attributes and functionalities, including temporal and non-temporal constraints between services, service scheduling preferences, alternative options and so on.

We introduce the XSRL language to express user goals and preferences. It allows us to define both vital and preference constraints on process execution trajectories, i.e. constraints over possible actions in the process execution and intermediate states possibly visited by the execution, as well as constraints that deal with non-deterministic behavior of services, where resulting outcome is not known before the invocation. Vital goals must be satisfied by any process execution, while preference goals express desired constraints, some of which might be more preferable than others.

To express requests for a composition of web services we propose the language XSRL (XML Service Request Language). The goal language is recursively defined as follows.

Definition 4 (goal language). *Basic goals are formed using the following unary operators:* ***vital*** p*,* ***atomic*** p*,* ***vital-maint*** p*,* ***atomic-maint*** p*, where* p *is a proposition. A goal* g *is a basic goal or a combination of goals using the following operators:* ***achieve-all*** $g_1, \ldots, g_n$*,**optional*** g*,* ***before*** g_1 ***then*** g_2*,* ***prefer*** g_1 ***to*** g_2*.*

The intuitive semantics of XSRL constructs is presented in Table 3.1. The BNF notation is provided in Appendix A.

The atomic objects of the language are propositions, that is, boolean combination of linear inequalities and boolean propositions. These can be either true or not in any given state.

Before providing a detailed explanation of XSRL constructs, let us consider an intuitive example of an XSRL request for a travel domain introduced in Section 1.1. Suppose a user is planning a one night trip to Paris and is interested in a number of possibilities in connection with this trip. These include making a hotel reservation in Paris, avoiding to travel by train, if possible, and spending an overall amount not greater than 300 euros for the whole package. Further, the user prefers to spend less than 100 euros for a hotel room but, if this is not possible, he may be willing to spend up to 200 euros for that room. The user wants to pay under the condition that he receives a confirmation for the entire package. Of course, the user would also need to specify dates for his trip and

Goal	Where satisfied	Type of goal
vital p	In a state where p holds to which there is a path from the initial state modulo failures	reachability
atomic p	In a state where p holds to which there is a path from the initial state despite failures	reachability
vital-maint p	In a state to which there is a path from the initial state modulo failures, p must hold in all states along the path	maintainability
atomic-maint p	In a state to which there is a path from the initial state despite failures, p must hold in all states along the path	maintainability
prefer g_1 **to** g_2	In states where g_1 is satisfied, otherwise the satisfiability of g_2 is checked	preference
optional g	States where g is satisfied are checked first, otherwise the goal is ignored	preference
before g_1 **then** g_2	In states, to which there is a path from the initial state, such that, states along these path where g_1 is satisfied preceding those where g_2 is satisfied	sequencing
achieve-all $g_1, \ldots, g_n$	In states, to which there is a path from the initial state, such that, there are states along the path where g_i are satisfied	composition

Table 3.1: Goal language constructs.

night stay in Paris. This will not be considered in this example as it provides no additional explanation of the ideas behind the presented system. Omitting XML tags, the sample request is defined as follows:

```
achieve-all
  before
    achieve-all
      prefer vital-maint $hotelPrice < 100$ to
               vital-maint $hotelPrice < 200$
      optional-maint $\neg trainBooked$
      vital $confirmed \wedge$
            $location = ``Paris'' \wedge$
            $hotelReserved$
  then
    atomic $final$
  vital-maint $price < 300$
```

A number of operators take propositions as arguments. These are used to express ‘how’ to satisfy the propositions. The **vital** p is satisfied if there exists an execution, ignoring non-determinism, that reaches the state satisfying proposition p, it fails otherwise. The **atomic** p means that p have to be reached from the current state despite non-determinism of the domain. If there is no such path to a satisfaction state, it fails. Note the requirements of this operator are stronger than the **vital**. The **vital** operator does not guarantee satisfaction of the goal if the execution of the plan is non-deterministically “takes the wrong path”, this means that non-deterministic action invocation brings the system in a state different from the one in which the final goal is achieved.

Reachability and maintainability goals are further combined by the modality operators to form composite, sequencing, and preference goals. These operators are: **achieve-all**, **before**-**then**, **prefer**-**to**, and **optional**. The goal **achieve-all** $g_1, \ldots, g_n$ succeeds when all subgoals $g_1, \ldots, g_n$ are satisfied, it fails if at least one goal cannot be satisfied. The goal **before** g_1 **then** g_2 is satisfied, if g_1 is satisfied and, starting from the state where g_1 is satisfied, g_2 is also satisfied, it fails otherwise. The goal **prefer** g_1 **to** g_2 succeeds if g_1 is satisfiable, if not, then it succeeds if g_2 is satisfiable, it fails if both g_1 and g_2 are unsatisfiable. Note that by nesting preference statements, one may give a total order over any number of sub-goals. The goal **optional** g is always satisfied as a goal. Its meaning is that, if there exists a plan that satisfies g then this plan must be executed, otherwise the goal is ignored. Summarizing, **achieve-all** provides a way of collecting goals that have all to be satisfied, the goal **before**-**then** is a way of sequencing goals, while **prefer**-**to** and **optional** enable the user to express user preferences over goals.

3.3 Service planning problem

We are now ready to provide a general definition of what a planning problem is in our setting, that is, with respect to our service planning domain from Definition 2 and goal from Definition 4:

Definition 5 (Service planning problem). *A* service planning problem *is a tuple $\mathcal{P} = \langle \mathcal{D}, s_0, g, \mathcal{I}, im \rangle$, where*

- *$\mathcal{D}$ is a domain from Definition 2;*
- *s_0 is an initial state;*
- *g is a goal from Definition 4. All variables constrained by g are required to be process variables;*
- *$\mathcal{I}$ is the set of service providers;*
- *$im : \mathcal{R} \rightarrow 2^{\mathcal{P}}$ is a function that associates service providers with roles.*

Given a service problem, a plan is a policy prescribing the actions to be executed in any given state. To record the states traversed while executing non-deterministic actions, the notion of a plan history is introduced. A plan in our setting is thus formalized as follows.

Definition 6 (plan). *A* plan *for a service problem $\mathcal{P} = \langle \mathcal{D}, s_0, g \rangle$ is a tuple $\pi = \langle \mathcal{H}_\pi, \mathcal{A}_\pi \rangle$, where:*

- *$\mathcal{H}_\pi$, the plan* history, *is a finite set of sequences of* outcomes *$o_1, \ldots, o_m$, $m \in \mathbb{N}$, with the following properties:*
  - *$o_1 = (s_0, a, s)$ for some a and s;*
  - *if $o_i = (s, a, s')$ then $o_{i+1} = (s', a', s'')$;*
- *$\mathcal{A}_\pi : \mathcal{S} \times \mathcal{H}_\pi \to \mathcal{A}$ is the* enabling action *in $s \in \mathcal{S}$ depending on the given state and history.*

Intuitively, the execution of a plan is a particular instantiation of the plan. Formally, it is defined as follows:

Definition 7 (Plan execution). *An* execution *of the plan π is $\sigma = \langle s_0, \ldots, s_n, h \rangle$ where*

- *$s_0, \ldots, s_n$ is a sequence of states, $s \in \mathcal{S}$, with s_0 the initial state of the service problem;*
- *$h := o_1, \ldots, o_n$ is from $\mathcal{H}_\pi$.*

The execution σ is valid *if, for all $i = 1 \ldots (n-1)$, $o_i = (s, \mathcal{A}_\pi(s_i, o_1, \ldots, o_i), s')$. The* normal *executions of π are the executions of π which only contain normal outcomes.*

From here on, we refer only to valid executions, unless stated otherwise. If an execution σ_1 of π repeats another execution σ_2 of π until some state, σ_1 is called a sub-execution of σ_2. Formally, an execution $\sigma_1 = \left\langle s_0^1, \ldots, s_n^1; o_1^1, \ldots, o_n^1 \right\rangle$ of π is a *sub-execution* of the execution $\sigma_2 = \left\langle s_0^2, \ldots, s_m^2; o_1^2, \ldots, o_n^2 \right\rangle$ of π, written as $\pi_1 \subseteq \pi_2$, if $n \leq m$ and,

- $\forall i \in \{0, \ldots, n\} : s_i^1 = s_i^2$,

- $\forall i \in \{0, \ldots, n\} : o_i^1 = o_i^2$.

Intuitively, a solution to a service planning problem is such a plan, whose executions, starting at the initial state, satisfy the goal extracted from the user request. But to define it formally, first we have to define the formal semantics of the service request language.

3.4 Formal semantics of XSRL

Let us now consider the formal semantics of XSRL. The goal of the formal semantics is to define the meaning of XSRL expressions with mathematical rigor. A rigorous formal semantics clarifies the intended meaning of the service request language introduced in Section 3.2, and ensures that no corner cases are left out, and provides a further reference for implementation. We also define the solution to the service planning problem in terms of XSRL goal satisfaction.

At the initial state, most of the domain variables are assigned with initial values. When the corresponding process is executed (e.g., according to pre-synthesized plan), variables are changed according to effect functions of the invoked actions. Since variables are being modified from state to state in the execution, the goal propositions, that contains changing variables might be satisfied in some particular states and are false in all others. We say that the proposition p is satisfied in state s by execution σ started at the initial state s_0, if the effect functions of the enabling actions, that form the execution, satisfy the proposition. Formally, we have the following definition:

Definition 8 (Goal proposition set). *Goal proposition set $\mathcal{P}$ for goal g and set of variables $\mathcal{V}$ is formed according to the two rules below:*

- *all boolean variables in g are in $\mathcal{P}$. A boolean variable p is satisfied if the corresponding variable in $\mathcal{V}$ is true;*

- *all linear constraints appearing in g are added as boolean propositions in $\mathcal{P}$. A formed proposition p is satisfied if variables from $\mathcal{V}$ satisfy the corresponding linear constraint.*

Let us define the notion of a proposition satisfaction, that we use later for introducing the formal semantics of XSRL:

Definition 9 (Proposition satisfaction). *Proposition p is satisfied by a valid execution $\sigma = \langle s_0, \dots, s_n, h\rangle$ for plan π in a domain $\mathcal{D}$ with enabling actions a_i if the proposition p is satisfied after applying effects of the enabling actions: $\overline{f}(a_n, o_n, \overline{f}(a_{n-1}, o_{n-1}, \overline{f}(\dots \overline{f}(a_1, o_1, \overline{D}_0))))$, where $\overline{D}_0$ is a set of initial values of variables. If σ satisfies p, we write:*

$$\sigma \models p$$

The formal semantics of XSRL is introduced via goal satisfaction $\pi \models g$. A goal satisfaction is defined in terms of the set of plan executions and their satisfaction of goal propositions. Formally, the formal semantics of XSRL is defined as follows:

Definition 10 (Formal semantics of XSRL). *Let π be a plan for a domain $\mathcal{D}$, p is a proposition formed from goal g according to Definition 8. Then, the* *formal interpretation of the goal g is defined as follows:*

$\pi \models p$	**iff**	$\emptyset \models p$, *where* $\emptyset$ *is an empty execution*
$\pi \models \neg p, p_1 \wedge p_2, p_1 \vee p_1$	**iff**	$\emptyset \models \neg p, p_1 \wedge p_2, p_1 \vee p_1$ *correspondingly*
$\pi \models$ ***vital*** p	**iff**	$\exists$ *normal* $\sigma \in \pi : \sigma \models p$
$\pi \models$ ***vital-maint*** p	**iff**	$\forall$ *normal* $\sigma \in \pi : \sigma \models p$
$\pi \models$ ***atomic*** p	**iff**	$\forall h \in \mathcal{H}_\pi \exists \sigma = \langle \ldots, h \rangle \in \pi : \sigma \models p$
$\pi \models$ ***atomic-maint*** p	**iff**	$\forall \sigma \in \pi : \sigma \models p$
$\pi \models$ ***achieve-all*** $g_1, \ldots, g_n$	**iff**	$\forall i : \pi \models g_i$
$\pi \models$ ***before*** g_1 ***then*** g_2	**iff**	$\exists \sigma_1, \sigma_2 \in \pi : \sigma_1 \models g_1, \sigma_2 \models g_2, \sigma_1 \subseteq \sigma_2$
$\pi \models$ ***prefer*** g_1 ***to*** g_2	**iff**	$\pi \models g_1$ **or** $\pi \models g_2$ *and* $\nexists \pi' \neq \pi : \pi' \models g_1$
$\pi \models$ ***optional*** g	**iff**	$\pi \models$ ***prefer*** g ***to*** $\top$

The satisfaction of a goal is thus defined in terms of whether a particular plan satisfies the goal or not.

A *solution* to an XSRL request is defined in terms of the plan that satisfies the user goal. Formally, the solution is defined as follows:

Definition 11 (Solution). *A* solution *for a domain D with respect to a goal g from state s_0 is a plan π that satisfies the goal:*

$$\pi \models g$$

Therefore, a *problem* of interleaving planning and execution is the finding of a solution, i.e. a plan, for a given domain, goal and initial state.

3.5 Algorithms for interleaving planning and execution

Two types of uncertainty for the transitions between business process states may arise: non-deterministic failures and unknown outcomes from actions. Non-deterministic failure occurs when an action has several possible outcomes which

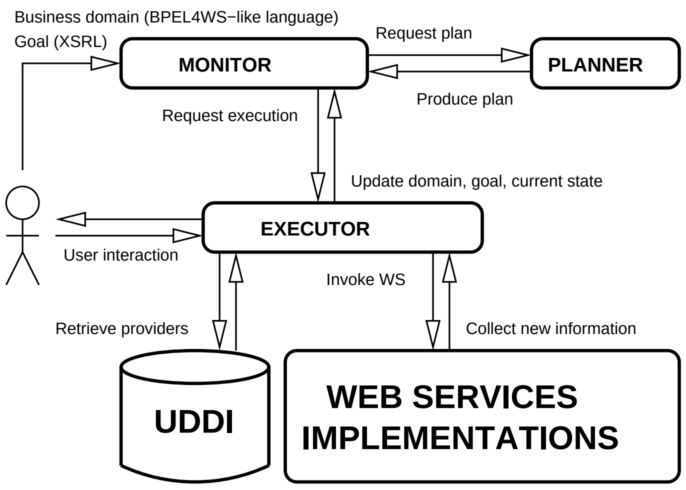


Figure 3.1: High-level XSRL architecture.

are not known before invocation. The list of possible outcomes is known a priori and thus modeled in the domain. The second type of uncertainty requires additional processing before applying the planning techniques. Unknown outcomes of action invocations can be properly handled only at run-time, therefore planning must be interleaved with execution. In a framework based on the interleaving of planning and execution, information on the outcome of action invocation is gathered at run-time and used to replan consistently with the original goal. This idea leads to a planning framework that is based on the notion of interleaving planning and execution.

We propose a planning architecture which works in the following way. The framework receives a request from the user and tries to fulfill it against a standard business process, assuming that it is syntactically correct. The framework returns a failure if the request cannot be satisfied in the given business process under the current run-time circumstances, e.g., ticket dates or hotel prices are not available. During execution the system interacts with the service registry to find suitable service providers, in a web service enabled marketplace, and with the user to ask confirmation or request additional information, if necessary.

The architecture presented in Figure 3.1 divides the framework into three main functional units: a monitor, a planner and an executor. In this section we provide two algorithms for the monitor and the executor, leaving algorithms for the planner for the following Chapter 4.

Algorithm 1 monitor(domain d, state s, goal g)

```
$\pi$ = plan($d, s, g$)
if $\pi$ = $\emptyset$ then
    return success
else
    if $\pi$ = failure then
        if chooseNewProvider($provider$) then
            $d'$ = updateDomain($d$)
            return monitor ($d', s, g'$)
        else
            $g'$ = generate-rollback-goal()
            monitor($d$, $s$, $g'$)
            return failure
        end if
    end if
    $(d', s', g')$ = execute($\pi, d, s, g$)
    return monitor ($d', s', g'$)
end if
```

The *monitor* (Algorithm 1) is responsible for invoking the planner, recovering from failure and invoking the execution of plans. Starting with a domain, an initial state and an XSRL goal, it invokes the planner requesting the synthesis of a plan. Then monitor analyzes the plan. An empty plan means that the goal has been reached and the request has been successfully met. If the planner returns failure, i.e., the goal cannot be satisfied under the current execution context, then it attempts to change a provider. `chooseNewProvider` contacts the executor module which has a list of possible providers for services and keeps track of which providers have been considered during the execution of the plan. If a new provider can be assigned, the execution proceeds, otherwise the monitor tries to rollback all changes to a domain and returns failure. Finally, if a non-empty plan has been produced, the plan is passed on to the executor by invoking the `execute` function. This function returns an updated domain, current state and the new XSRL goal for which one needs to continue the monitoring.

Note that after the execution phase the original goal can be updated. This is necessary for reachability goals only (goals that are not part of any maintainability goal). The idea behind is simple: if one reserves a hotel he/she does not need to look for plans that reserves hotels in the following iterations. We eliminate such subgoals when they are satisfied.

Algorithm 2 execute(plan π, domain d, state s, goal g)

```
repeat
  $a$ = firstAction($\pi$)
  $\pi$ = $\pi - a$
  if webServiceAction(a) then
    $role$ = Rol$_{Act}$($a$)
    if noProviderForRole($role$) then
      $providersList$ = contactUDDI($role$)
      $provider$ = chooseProvider($providersList$)
    else
      $provider$ = previouslyChosenProvider($role$)
    end if
    message = invoke($a, provider$)
  end if
  ($d', s', g'$) = update($d, s, g, a$,message)
  if isKnowledgeGathering($a$) $\vee$ goalFailed($g$) then
    return ($d', s', g'$)
  end if
until $\pi$ = $\emptyset$
return ($d', s', g'$)
```

The *executor* (Algorithm 2) starts with a plan, a domain, an initial state and an XSRL goal. It iterates by attempting the execution of all the actions of the input plan. The `firstAction` of the plan is stored in the variable a and then removed from the plan. If this action requires an interaction with a web service, then one needs to seek for a provider for that action. The construct $role$ stores the role associated with the current action. If the executor has not assigned a provider for that role during the execution so far, then the UDDI is contacted to ask for providers for the given role. A provider is chosen from the list of possible providers using some heuristic function (the first provider who has good references, etc.). If, on the other hand, a provider has already been assigned to a role, then we must continue executing the following actions assigned to the role with the same provider. Once the provider has been identified, the provider is invoked with action a and the possible return messages are stored in the $message$ variable. The next step is that of updating the domain, the current state and the goal by the effects of having executed the action. This step is necessary as the execution of the action may have brought the system into a new state, it may have changed the values of some variables and it may have satisfied the subgoals of the current goal. If the action has been a knowledge gathering action, we have acquired new information and return the current status to the monitor in order to perform re-planning, otherwise we reiterate the cycle by looking at the following action of the plan.

The algorithms for interleaving planning and execution, provided in this chapter assume that the function `plan` in Algorithm 1 finds the plan according with Definition 11. In the following chapter we present an approach based on constraint satisfaction to find a solution to a service planning problem. In Appendix B it is presented an approach based on planning as model checking techniques that deals efficiently with domains with a limited number of numeric constraints.

Chapter 4

Service planning as constraint satisfaction

In this chapter we provide an approach for finding a solution to the service planning problem defined in Chapter 3 by reformulating it in terms of constraints. The encoding is performed in two phases: in phase (i) the service planning domain is encoded; in phase (ii) the goal is added to the encoding. After the constraint problem is solved the plan is extracted from the assignment of the variables.

We use constraint programming because it can be relatively easy extended to support planning for service compositions dealing with uncertainty, non-determinism, and numeric values. By using reification techniques[5, 121] it can also handle user preferences. In service enabled environments values and execution conditions are unknown until the actual service invocation. Using constraints programming techniques, one can take advantage of reuse of the previously generated plans in the replanning phases. In fact, a new value returned by a service invocation may simply result in the addition of a constraint to the solution space and no further replanning is necessary.

In this work, when we talk about non-deterministic actions we refer to their outcomes (that is, states to which the action can take the system) which can be different; yet, once an action is invoked, we assume that its outcome will be always the same. In other words, any of its future invocations in a given execution will produce the same outcome for the same provider.

Our goal is to model a service domain, goal and assertions as a set of constraints over controlled and non-controlled variables. The constraints have the following form:

$$[\forall \xi_i :]\ \overline{c_v}\ \bowtie\ value, \tag{4.1}$$

- $value$ is a value from the domain of the variable v,
- $\overline{c_v}$ is a vector of expressions of the form $\sum \beta_i[\xi_i]a_{i,k}$ with $\beta_i, \xi_i \in \{0,1\}$,
- the ξ_i are non-controlled variables and the β_i are controlled variables,
- $a_{i,k}$ is the effect function of the action a_i for the outcome k,
- $\bowtie$ is either $<$,$>$,$\geq$, $\leq$ or $=$,
- and $[\cdot]$ denote that the expression is optionally present in the constraint.

There are two types of Boolean variables defined: *controlled* variables, denoted by β_i, and *non-controlled* variables, denoted by ξ_i. Controlled variables represent the action effects that are applied if the corresponding service is to be invoked. Non-controlled variables represent non-deterministic action outcomes. The underlying idea is that the constraint solver is not necessarily free to choose a specific value for a non-controlled variable, thus a solution to the problem may be such regardless of the values assigned to the non-controlled variables.

The ratio behind the proposed constraint form defined by Equation 4.1 is the following: $\overline{c_v}$ represents the evolution of the system for some particular execution. If the basic goal g (one of the **vital**, **vital-maint**, **atomic**, **atomic-maint**) restricts the variable v_i, then corresponding constraint c_v is added to a constraint problem. After solving the problem the desired plan is extracted according to the values of βs: if some β_i is instantiated to 1 then its corresponding action a_i is added to the resulting plan. Complex goals (**achieve-all**, **before-then**, **prefer-to**) do not add any new constraints but rather interrelate the βs for basic goals.

For example, for **achieve-all** goal it is done to ensure that all sub-goals constraints choose the same actions in all branching points, for **before** g_1-**then** g_2 it is guaranteed that actions chosen by g_1 must also be chosen by g_2. The described encoding intuition is further described in details.

Let us now define the *service constraint problem* formally:

Definition 12 (service constraint problem). *A* service constraint problem *is a tuple $\mathcal{CP} = \langle \beta, \mathcal{N}, \xi, \mathcal{C} \rangle$, where:*

- β *is a set of* controlled *boolean variables;*
- $\mathcal{N}$ *is a set of* controlled *variables ranging over natural numbers;*
- ξ *is a set of* non-controlled *boolean variables;*
- $\mathcal{C}$ *is a set of constraints, as in Equation 4.1, in which (i) if a non-controlled variable occurs then it is universally quantified, (ii) otherwise a value is available and substituted for the variable.*

A solution *to a service constraint problem is an assignment to controlled variables such that all constraints are satisfied.*

To arrive at the encoding of the service interaction problem as a set of constraints of the form of Equation 4.1, we follow a two phase process. In the first phase, one encodes the service planning domain, while in the second phase, one encodes the goal.

4.1 Encoding of a planning domain

During Phase 1 the service domain is encoded. Starting from a service domain as in Definition 2, we arrive at a set of expressions c_v as in Equation (4.1) plus a set of linear constraints of the form $\sum \beta_i \leq 1$. In the following, we adopt the notation of Equation (4.1); in addition, n is a natural number that specifies how

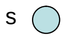
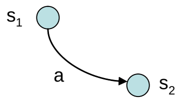
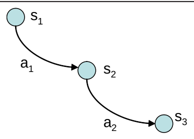
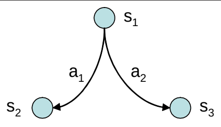
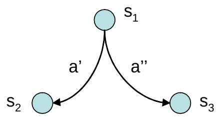
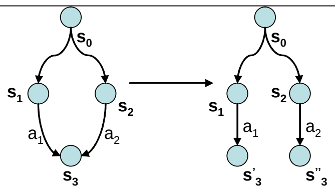
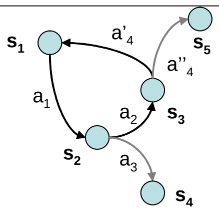
	Domain	Type of action	Encoding
(A)		No action	$\emptyset$
(B)		Single deterministic action	βa
(C)		Sequence of actions	$\beta_1(a_1 + \beta_2 a_2)$
(D)		Deterministic branch point	$\beta_1 a_1 + \beta_2 a_2$ $\beta_1 + \beta_2 \leq 1$
(E)		Non-deterministic branch point	$\beta(\xi_1 a' + \xi_2 a'')$ $\xi_1 + \xi_2 = 1$
(F)		Cycle: state splitting	$\emptyset$
(G)		Cycle: directed cycle	$n\xi(a_1 + a_2 + a'_4)$

Table 4.1: Encoding examples of the service domain.

many times a cycle is followed, while a_i is overloaded to represent not only the action, but also its effects. We only consider linear effect functions.

The encoding is generated following Algorithm 3 that recursively visits the service domain $\mathcal{D}$, separately keeping track of cycles, and returns a set of constraints. Intuitive pictorial explanations of the encoding are shown in Table 4.1.

Algorithm 3 model(state s, domain $\mathcal{D}$, path P): constraint

```
set visited $s$
$P = P \cup s$
**if** degree($s$) == 0 **then**
  // recursion base case
  **return** $\emptyset$
**end if**
**if** $s$ is visited $\wedge$ $s \in P$ **then**
  // visited state: cycle is added, base case
  put(s,P,add-cycle($s$, $\mathcal{D}$, $P$),$cycle$)
  **return** $\emptyset$
**end if**
**if** $s$ is visited $\wedge$ $s \notin P$ **then**
  // visited state: join point
  update-cycles(P, s)
  **return** encoding(s)
**end if**
// branch point
c = $\sum^{\text{degree}(s)} \beta_i$ add-single-action($s$, $\mathcal{D}$, $a_i$, $P$)+get($s$,$P$,$cycle$)
$\sum^{\text{degree}(s)} \beta_i \leq 1, \beta_i \in \{0, 1\}$
$P = P \backslash s$
**return** $c$
```

The algorithm starts from the initial state of the domain and recursively works on outgoing edges. If it reaches a state with no outgoing edges it adds a 0 (base case). If it finds an edge representing a single deterministic action a, then the modeling is βa, where β is a controlled boolean variable. Then $\beta = 1$ means that action a must be in the resulting plan. The sequencing of deterministic actions a_1, a_2 is modeled as follows: $\beta_1(a_1 + \beta_2 a_2)$. If $\beta_1 = 1$ then action a_1 is added to the plan, and if also $\beta_2 = 1$, then a_2 is added to the plan right after a_1. If $\beta_1 = 0$ then neither action a_1 nor a_2 are added.

When two or more actions start from the same state, there are two cases (Table 4.1): (a) different actions or (b) the same action which is non-deterministic. In the first case, $\beta_1 a_1 + \beta_2 a_2$, where $\beta_1 + \beta_2 \leq 1$, means that at most one action can be chosen. In the second case, the non-controlled variables ξ are introduced. Then the modeling (for a non-deterministic action with two possible outcomes) is the following: $\xi_1 a' + \xi_2 a''$, where $\xi_1 + \xi_2 = 1$. Since ξ_1 and ξ_2 are both from $\{0, 1\}$, the constraint $\xi_1 + \xi_2 = 1$ defines that only one non-deterministic outcome is returned from the corresponding action.

In case there is a directed cycle then a variable n ranging over integers is introduced. The variable n denotes the possible number of iterations through a cycle. The right-hand figure shows an example of a cycle and its modeling starting from s_1 with a deterministic branch at s_2 and a non-deterministic one at s_3.

Algorithm 4 add-single-action(state s, domain $\mathcal{D}$, action a, path P): constraint

```
if $a$ is non-deterministic then
    // non-deterministic branch point
    return $\sum^{\texttt{outcomes}(a)} \xi_i$ add-single-action($s$, $\mathcal{D}$, $a'_i$, $P$) $\bigcup \sum^{\texttt{outcomes}(a)} \xi_i = 1$,
    $\xi_i \in \{0, 1\}$
end if
// single deterministic action outcome
$s_{next}$ = get-next-state($s$, $\mathcal{D}$, $a$)
return $a +$ model($s_{next}$, $\mathcal{D}$, $P$)
```

The whole process may be summarized as follows:

(A) Base case. If the degree of the arcs leaving the state s is 0, then there is no constraint to be returned. Table 4.1.(A) illustrates this situation. Also the case of the directed cycle, which is presented below, is a base case.

Algorithm 5 add-cycle(state s, domain $\mathcal{D}$, path P): constraint

```
// identify last cycle in the path
$P = \langle \ldots s, t_1, \ldots, t_r, s \rangle$;
// let $a_i$ be the action going from state $t_i$ to $t_{i+1}$
$\forall$ non-deterministic actions ad a $\xi_j$
return $n \prod \xi_j \cdot \sum^{r+1} a_i$
```

(B) Single deterministic action a is encoded as βa, where β is a controlled boolean variable. $\beta = 1$ means that action a must be in the resulting plan.

(C) Sequence of actions. This rule is applied to consecutive actions as follows (for two deterministic actions): $\beta_1(a_1 + \beta_2 a_2)$. If $\beta_1 = 1$ then action a_1 is added to the plan, and if also $\beta_2 = 1$, then a_2 is added to the plan right after a_1. If $\beta_1 = 0$ then neither action a_1 nor a_2 are added.

(D) Deterministic branching point. If there are several outgoing actions from the state s and the system is supposed to choose only one of them to add to the plan, then this situation (for two actions) is encoded as follows: $\beta_1 a_1 + \beta_2 a_2$, where $\beta_1 + \beta_2 \leq 1$ means that at most one action can be chosen.

(E) Non-deterministic branching point. This rule takes care of non-determinism. The encoding is similar to the case (D), but non-controlled variables ξ_i are used to represent non-deterministic behavior.

(F) Cycle: state splitting. This rule is applied to undirected cycles. To proceed we need to duplicate the state s already visited by creating state s' and recursively encode the duplicated state. There is no other encoding for this case.

(G) Cycle: directed cycle. This rule is applied to directed cycles. Table 4.1.(F) exemplifies a cycle situation in which there are simple deterministic actions (a_1), actions taking out of the cycle (a_3), and non-deterministic ac-

Goal / Assertion	**Encoding**
vital p	$\xi = \xi^0$: $c_v \bowtie v_0$
atomic p	$\forall \xi$: $c_v \bowtie v_0$
vital-maint p	$\xi = \xi^0$: $c_v(t_i) \bowtie v_0$, for all encoding steps t_i
atomic-maint p	$\forall \xi$: $c_v(t_i) \bowtie v_0$, for all encoding steps t_i
prefer g_1**to** g_2	All variables in g_1 are instantiated before those in g_2
optional g	encoded as **prefer** g **to** $\top$
before g_1**then** g_2	for all $g^i \in G_1$, $g^j \in G_2$, steps t_k: $u_k(g^i) \neq 0 \Rightarrow u_k(g^i) = u_k(g^j)$
achieve-all $g_1, \ldots, g_n$	for all $g^i \in G_i$, $g^j \in G_j$, $i \neq j$, steps t_k: $u^k(g_i) \neq 0 \land u^k(g_j) \neq 0 \Rightarrow u^k(g_i) = u^k(g_j)$

Table 4.2: Goal and assertion language encodings.

tions that 'might' lead out of the cycle (a_4''). Variable n in the encoding denotes the number of times the cycle is going to be executed.

4.2 Encoding of goals and preferences

During the second phase of the encoding of the service planning problem, one takes a goal or assertion and the encoding of the service domain, and produces a set of constraints which represent how to achieve the corresponding goal or constraint. The goal/assertion is expressed in the goal and assertion language of Definition 4. The Algorithm 6 parses the goal recursively distinguishing the cases of the various operators and updating the set of constraints. Every time a new basic goal constraint is added, a new set of controlled variables is introduced.

Incidentally, notice that these sets of variables represent the same variable in the original encoding. Choices in different variable sets, if made, are forced to be the same by the relations defined by **achieve-all** and **before-then**.

(A) vital $v \bowtie v_0$. If the goal is **vital** with respect to the variable v constrained by

Algorithm 6 encode-goal(constraint c, goal g): goalset

```
// reachability goals
if $g$ is ‘vital $v \bowtie v_0$’ then
   $c \bigcup \xi_v = \xi_v^0, c_v \bowtie v_0$
   return $\{g\}$
end if
if $g$ is ‘atomic $v \bowtie v_0$’ then
   $c \bigcup \forall \xi \; c_v \bowtie v_0$
   return $\{g\}$
end if
// maintainability goals
if $g$ is ‘vital-maint $v \bowtie v_0$’ then
   for all steps $t_i$ do
      $c \bigcup \xi_v = \xi_v^0, c_v(t_i) \bowtie v_0$
   end for
   return $\{g\}$
end if
if $g$ is ‘atomic-maint $v \bowtie v_0$’ then
   for all steps $t_i$ do
      $c \bigcup \forall \xi \; c_v(t_i) \bowtie v_0$
   end for
   return $\{g\}$
end if
if $g$ is ‘achieve-all $g_1, \ldots, g_n$’ then
   $G_i = \texttt{encode-goal}(c, g_i)$
   for all $g^i \in G_i$, $g^j \in G_j$, $i \neq j$, steps $t_k$ do
      $c \bigcup u^k(g_i) \neq 0 \wedge u^k(g_j) \neq 0 \Rightarrow u^k(g_i) = u^k(g_j)$
   end for
   return $\{G_1, \ldots, G_n\}$
end if
if $g$ is ‘before $g_1$then $g_2$’ then
   $G_i = \texttt{encode-goal}(c, g_i), i \in \{0, 1\}$
   for all $g^1 \in G_1$, $g^2 \in G_2$, steps $t_i$ do
      $c \bigcup u_i(g^1) \neq 0 \Rightarrow u_i(g^1) = u_i(g^2)$
   end for
   return $\{G_1, G_2\}$
end if
if $g$ is ‘prefer $g_1$to $g_2$’ then
   $G_i = \texttt{encode-goal}(c, g_i), i \in \{0, 1\}$
   raise-priority$(G_1, G_2)$
   return $\{G_1, G_2\}$
end if
```

the $\bowtie$ operator on the v_0 value, we restrict the constraint c to what concerns variable v, denoting it by c_v, and we add $c_v \bowtie v_0$ to the constraints set. Since the goal is **vital** we also set all variables ξ associated with c_v to ξ^0, by which we mean that the normal execution is followed, in place of the non-deterministic failure ones.

(B) atomic $v \bowtie v_0$. This case is analogous to the vital one, with the difference that all non-deterministic executions must be considered, thus coming to a universal quantification over the non-deterministic variables ξ.

(C) vital-maint $v \bowtie v_0$. For maintainability goals we need to keep track of all the states visited during a plan execution. Thus, we quantify over the execution steps and we repeat the constraint as for the vital case above for each step.

(D) atomic-maint $v \bowtie v_0$. This case is analogous to the maintainability vital one, with the difference that all non-deterministic executions must be considered, therefore we come to a universal quantification over the non-deterministic variables ξ.

Now we consider the operators which aggregate basic sub-goals.

(E) achieve-all $g_1 \ldots, g_n$. First, recursion is called for all sub-goals $g_1, \ldots, g_n$. Second, one considers all pairs of basic goals coming from the recursive call and all execution steps (as done for the maintainability goals). In all these cases, if during the execution some choices have been made for the same branch point among different sub-goals, these choices have to be the same. Therefore, we add to the set of constraints, expressions forcing the choices for the execution of any subgoals to be the same. The expressions introduce the execution choice variable u. Suppose that $u^j, j \in \{1, 2\}$ denotes the branch that has been chosen by the procedure that tries to satisfy an j-th goal, $u^j = 0$ defines that there were no choice made and $u = i$ denotes that the corresponding β_i is set to 1 for the step under consideration. Then the following constraints ar added: $u^1 \neq 0 \wedge u^2 \neq 0 \Rightarrow u^1 = u^2$.

(F) **before** g_1 **then** g_2. The principle behind the before-then operator is similar to that of the **achieve-all**, with the difference that one forces the ordering of the satisfaction of the subgoals. Again, first we recur on the subgoals, then we introduce the execution choice variables u. The second subgoal g_2 should repeat the path of the first subgoal g_1, until the first is satisfied, and only then the second expression is checked. This is ensured by expressions of the form: $u^1 \neq 0 \Rightarrow u^1 = u^2$ which are added to the set of constraints.

(G) **prefer** g_1 **to** g_2. Preferences are handled not as additional constraints, but rather appropriately instantiating the variables. The first step is to recur on the two subgoals g_1 and g_2. Then the goals g_1 and g_2 are placed in a disjunction. When constraints are checked for satisfiability, variables are assigned in preference order. Optional goals are a sub-case of **prefer-to** goal, in which g_2 is simply true.

The encoding is summarized in Table 4.2.

4.3 Encoding the example

To illustrate how the encoding and framework work, let us introduce an example that is a snippet of the travel process definition (Figure 4.1) defined in Section 1.1. When deciding on a trip, the requester may first want to book the hotel of the final destination and then book a carrier to reach the location of the hotel. The first action a_0: `getHotelPrice` retrieves the hotel price. The next action is that of reserving a hotel (state s_1). This action may non-deterministically result in the successful booking of the room (state s_2) or in a failure (return to state s_1). Finally, there are two ways to reach the state s_3 in which a carrier to arrive

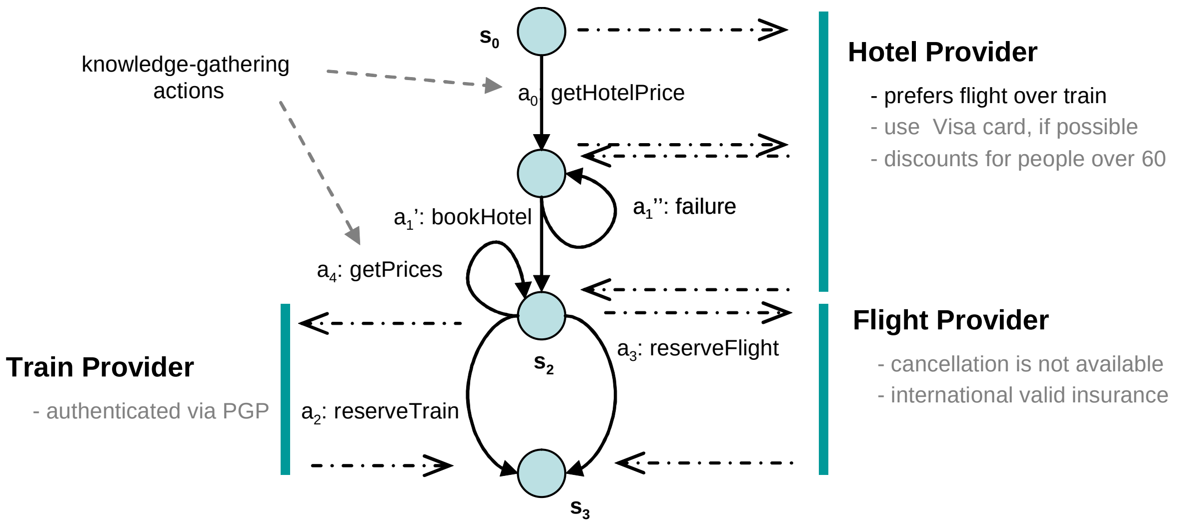

s0
knowledge-gathering actions
a0 getHotelPrice
a1': bookHotel
a1'': failure
a4: getPrices
s2
a3: reserveFlight
a2: reserveTrain
s3
Hotel Provider
- prefers flight over train
- use Visa card, if possible
- discounts for people over 60
Flight Provider
- cancellation is not available
- international valid insurance
Train Provider
- authenticated via PGP

Figure 4.1: A part of a travel business process.

at the site of the hotel is booked. One may choose to fly or to take a train. This is achieved by choosing one of the two actions `reserveTrain` or `reserveFlight`. There are two knowledge-gathering actions introduced: a_0: `getHotelPrice` and a_4: `getPrices`, that retrieve train and flight price from providers. The process variables are: $hotelBooked$, $trainReserved$, $flightReserved$, which are boolean, and $hotelPrice$, $trainPrice$, $flightPrice$, $price$, which are numeric.

The framework works in the following way. At first, the domain is encoded, that is: $\beta_0(a_0 + \beta_1(\xi_1 na_1^{fail} + \xi_2(a_1^{ok} + ma_4 + \beta_2 a_2 + \beta_3 a_3)))$, which represents the paths from state s_1 to s_3 with n and m being the number of times the cycle is followed. Additionally, the constraints on the choice variables $\beta_0, \beta_1, \beta_2, \beta_3 \in \{0, 1\}$, $\beta_2 + \beta_3 \leq 1$, and the constraints on the non-controlled variables $\xi_1, \xi_2 \in \{0, 1\}$, $\xi_1 + \xi_2 = 1$. ξ_1, ξ_2 are introduced.

Say the requester provides the following goal

achieve-all
    vital $hotelBooked$
    vital $trainBooked \;\vee\; flightBooked$
    atomic-maint $price$ < 100

The goal is encoded as follows. Every goal creates its own subset of controlled variables, which are correlated by the **achieve-all** constraint, according to Table 4.2. The first subgoal to be parsed is **vital** $hotelBooked$. Since the $hotelBooked$ variable is influenced only by a_1^{ok} outcome, that is +1, then the encoding is $\beta_1'\xi_2 = 1$ and the non-controlled variables are assigned to normal execution, i.e., $\xi_2 = 1, \xi_1 = 0$, thus $\beta_1' = 1$. By the same reasoning, one has the encoding for the flight and the train: $\beta_0'''\beta_1'''(\beta_2''' + \beta_3''') = 1$.

The atomic goal **atomic-maint** $price < 100$ is slightly different as $price < 100$ has to be checked for each state. Since it is an atomic goal, we have to introduce a universal quantification over the non-controlled variables. We assume that $a_1^{fail} = 0$, that is, no fee is paid for non-successful reservation. Having in mind that $a_0 = 0$ and $a_4 = 0$ since getting price information comes at no cost, only three actions affect the price (`bookHotel`, `reserveFlight`, and `reserveTrain`), and they just add its price to the overall one. Thus, the encoding of the goal is $\forall\xi_2 : s_0 : 0 < 100\ s_2 : \beta_0''\beta_1''\xi_2 hotelPrice < 100, s_3 : \beta_0''\beta_1''\xi_2(hotelPrice + \beta_3'' flightPrice + \beta_2'' trainPrice) < 100$.

Applying **achieve-all** adds the following constraints: $\forall i, j \in \{0, \ldots, 3\}, i \neq j, \beta_2^i + \beta_3^i = 1 \wedge \beta_2^j + \beta_3^j = 1 \Rightarrow \beta_2^i = \beta_2^j \wedge \beta_3^i = \beta_3^j$.

Actions `getHotelPrice` and `getPrices` are knowledge-gathering, and, therefore, must be executed before other actions use the corresponding variables: $\beta_1^j \neq 0 \Rightarrow \beta_0^j \neq 0$ and $\beta_2^j + \beta_3^j \neq 0 \Rightarrow m^j \neq 0$.

The resulting constraint set is $\forall\xi_1, \xi_2 \in \{0, 1\}$:

- $\beta_1' = 1$
- $\beta_0'''\beta_1'''(\beta_2''' + \beta_3''') = 1$
- $\beta_0''\beta_1''\xi_2 hotelPrice < 100$
- $\beta_0''\beta_1''\xi_2(hotelPrice + \beta_3'' flightPrice + \beta_2'' trainPrice) < 100$

and $\forall i, j \in \{1, 2, 3\}, i \neq j$:

- $\beta_1^{(j)} \leq 1 \wedge \beta_2^{(j)} + \beta_3^{(j)} \leq 1$

- $\beta_2^i + \beta_3^i = 1 \wedge \beta_2^j + \beta_3^j = 1 \Rightarrow \beta_2^i = \beta_2^j \wedge \beta_3^i = \beta_3^j$
- $\beta_1^j \neq 0 \Rightarrow \beta_0^j \neq 0 \wedge \beta_2^j + \beta_3^j \neq 0 \Rightarrow m^j \neq 0$

A solution is, for instance, $\beta_0 = \beta_1 = \beta_3 = m = 1, \beta_2 = 0$, that is, `getHotelPrice`, `reserveHotel`, `getPrices`, `reserveFlight`. The executor invokes action a_0 and updates $hotelPrice$. The hotel provider insists on satisfying its assertions, that is, **prefer vital** $flightBooked$ **to vital** $trainBooked$. The following constraint is added: $\beta_0^{iv}\beta_1^{iv}\xi_2\beta_3^{iv} = 1 \vee \beta_0^{v}\beta_1^{v}\xi_2\beta_2^{v} = 1$, with all variables in the first part with higher priority during variable instantiation. Constraints on the right side of the constraint set above include new βs. When checking this constraints, the constraint solver checks if a flight is available and, only if it fails, the train is chosen. If the system fails to find a solution other hotel providers are selected and, if all of them fail, then the framework returns an overall failure.

Assume that the constraints are consistent with the current plan and the executor continues the plan execution with the actions `bookHotel` and `getPrices`. The latter action retrieves prices for the flight and train and updates the corresponding process variables. The constraint system checks if the updated constraint set still has a solution. If, for example, the price for the flight is too high and it violates the constraints, then trains are checked, and, possibly, a new plan is generated from the state s_2, containing one action `reserveTrain`. By executing this action the framework satisfies the initial request and returns a success.

Chapter 5

Monitoring service-oriented business processes

We propose the use of an approach based on interleaving planning and execution in the context of non-deterministic domains to deal with assertions and user expressed requests against standard business processes that result in initiating and executing business processes from diverse organizations. The execution of these business processes in the proposed framework is governed by assertions, which are business rules applied to processes. The framework we propose deals with non-deterministic domains, where it tries to satisfy a user request by taking into account how assertions that appear at different levels, e.g., business process, role, and provider level, are applied during business process execution. The framework focuses, in particular, on the application of business rules that are associated with choreographies. The application of process *choreography assertions* usually results in activating only selected business process segments in different organizations. These are the business process segments that satisfy the process constraints and consequently can be involved in the result of a user request. In addition, the execution path of business processes is monitored to make certain that environmental conditions, i.e., web service supplied information, conform to the choreography assertions and user request requirements. The proposed framework deals with three kinds of assertions depending on their operational context and complexity: simple assertions, where simple reachability conditions are checked; preservation assertions, where maintenance of some condition needs to be satisfied throughout a path comprising a set of states traversed by the process during execution time; and business entity assertions, where the evolution sequence of a particular variable is monitored for correctness. In this thesis we are not concerned with the effect that choreography assertions have on orchestration assertions (assertions that apply in the local context of an organization). We henceforth use the term assertion to mean choreography assertions. As final contribution, we illustrate how the language we propose for expressing assertions can talk about functional as well as non-functional properties of services and their compositions.

5.1 Business processes

For the business process we mostly reuse the definition of the business process from Definition 2. A process is a possibly infinite ordering of activities with a beginning and an end; it has inputs (in terms of resources, materials and information) and a specified output. We may thus define a process as any sequence of steps that is initiated by an event, transforms information, materials, or business commitments, and produces an output [61]. In this thesis, we consider business processes as a means to represent the control flow of business logic and applications. This is achieved by introducing the notion of a state and an action. A *state* represents the state of the process execution. An *action* represents a business activity, which is modeled as a transition between given states. Each action is executed on behalf of a role. A *role* represents a set of business operations that relate to the same party, e.g., a travel agency. Each role has a number of providers associated with it. The providers can be found by interacting with service registries. A *provider* is the actual party that implements a role, e.g., a specific travel agency. It is convenient to also define the notion of a *process* *variable*, which is a variable associated with a process, e.g., travel packages, hotel reservations, as the process progresses through its execution path and its states change. The use of process variables guarantees that the execution of a business process can be monitored during execution as the process traverses a set of states where constraints may need to be applied to these variables. Constraints on the variables may represent user request or business rules.

5.2 Assertions

Actions within a business process are usually distributed between different parties (organization which may play different roles) that can make their changes in different portions of the process. A choreography language can guarantee the consistency of service interfaces, message ordering and message invocations, but it can not be used to check process runtime properties. Safe execution of the business process can only be ensured by a monitoring mechanism that checks the runtime properties of business process and possibly recovers from assertion violations. The monitoring of the business process based on the assertions violations is performed in the following way. First, assertions are published by the party who wants his assertions to be applied to business processes and monitored during execution. When executing the business process, the framework allows only those executions to proceed where published assertions are satisfied. If an assertion is violated then the system tries to find an alternative execution path in the business process that does not violate the assertion, if any. Assertions are published on different levels: business process, role or provider. During execution, assertions defined on the business process level are always taken into account; assertions defined by roles are checked only if operations for that role are invoked; provider level assertions are considered if an action of the particular provider is necessary.

More precisely, *monitoring* is a mechanism that ensures the execution of

Table 5.1: Assertions classification.

Assertion	**Where satisfied**
simple	in a state, where assertion condition is satisfied
preservation	for all states along the process execution
entity lifecycle	specified entity must preserve evolution specified in assertion

a process is consistent with respect to choreography business rules and user specified requests. As a business process spans several organizations, all of them expect that their business rules are taken into account when executing the process. Business rules are supplied by service providers and are enforced on business processes that are associated with such rules during their execution.

Business rules are expressed in the context of a process by assertions. Next we provide a definition of assertions.

Definition 13 (Assertion). *An* assertion *is a condition that applies to the execution of a business process.*

We use the term assertion and business rule interchangeably. An assertion may be satisfied or not during the execution of a business process, more formally:

Definition 14 (Assertion satisfaction). *Given a business process and one of its states, we say that an assertion is* satisfied *if the assertion is true in the specified state and in all future states visited during process execution.*

We classify assertions according to two different dimensions: (i) *operational assertions:* on the basis of the operational context and complexity of the assertion; (ii) *actor assertions:* on the basis of the ownership of the assertion. Operational assertions can be further classified into three categories (Table 5.1):

Simple assertion. A simple assertion is a condition to be satisfied in a *given state* or a *specific set of states* in order to reach a state where the condition is satisfied. Simple assertions are also named *reachability* assertions.

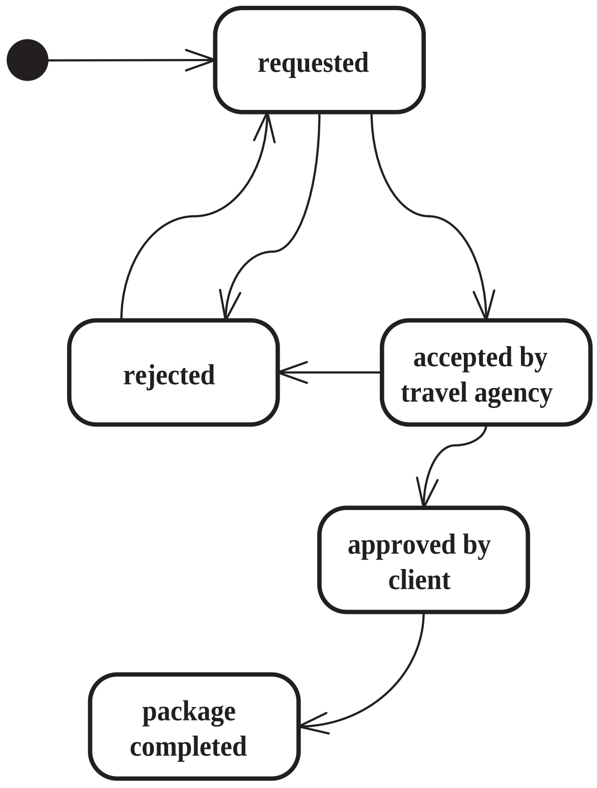


Figure 5.1: A travel package business entity assertion.

An example of such an assertion in the context of a travel domain is the requirement of having a medical insurance if the period of being abroad is more than two weeks. To comply with this assertion we must ensure that if the client requests a travel package with duration beyond two weeks then a medical insurance must be subscribed before the business process progresses successfully.

Preservation assertion. A preservation assertion is a condition to be maintained throughout *all states touched* during the execution of a business process. Preservation assertions are also named *maintainability* assertions. In the same travel example as above, consider a situation in which special offers exist for clients who hold a frequent flyer loyalty card, e.g., OneWorld. An assertion for the use of such card would require that all invoked services accept the card to provide discount or points. To comply with this assertion the execution of the business process will attempt to maintain the execution on those paths where services adhering to the loyalty program are available.

Business entity assertion. A business entity assertion is a property that applies to the evolution sequence of a process variable during process execution. For instance, a business entity assertion can be associated with the status of a travel package, as shown in Figure 5.1. Initially, the “status” variable assumes the value ‘requested’ when the travel package operation is started. From this state, the request can be ‘rejected’, if the travel agency fails to satisfy it and, eventually, return in a ‘requested’ status. Alternatively, the status variable can be ‘accepted by travel agency’ and subsequently be ‘approved by client’ and finally become a ‘package completed’. To comply with this assertion the execution of the business process must ensure that the states of the travel package variable are reached in the prescribed sequence and only change value according to the valid states of the business

Table 5.2: Assertion levels.

Assertion level	Where stored	Usage
business process	domain description	concatenated with user request
role	service description	applied if action of the role is invoked
provider	service registry	applied if provider action is invoked

entity assertion described above.

Assertions are not only classified on the basis of their operational dimension but also on the basis of ownership. Based on the ownership criterion, we may distinguish between three types of assertions (Table 5.2):

Business process-level. The *business process* level assertions are applied to the whole business process. The business process execution environment verifies these assertions during all executions and for all used services. Assertions of this type are maintained by the party who defines the choreography message sequences. These assertions are stored together with the business process itself. The business entity assertion defined in Figure 5.1 is an example of business process level assertion. It defines the possible evolutions of the status of travel package for all executions in the business process. Another example is the following. Usually business processes have an assertion of always reaching the final state despite of the non-determinism inherent in dealing with web service implementations, e.g., purchase a travel package. This assertion ensures process consistency with organization rules and policies.

Role-level. *Role*-level assertions are employed for all the providers implementing a specific role. Typically these assertions represent the constraints defined by the standardizing organizations, government, etc. For example, due to governmental laws all travel agencies may require that together with a flight ticket also a medical insurance is purchased, whenever the

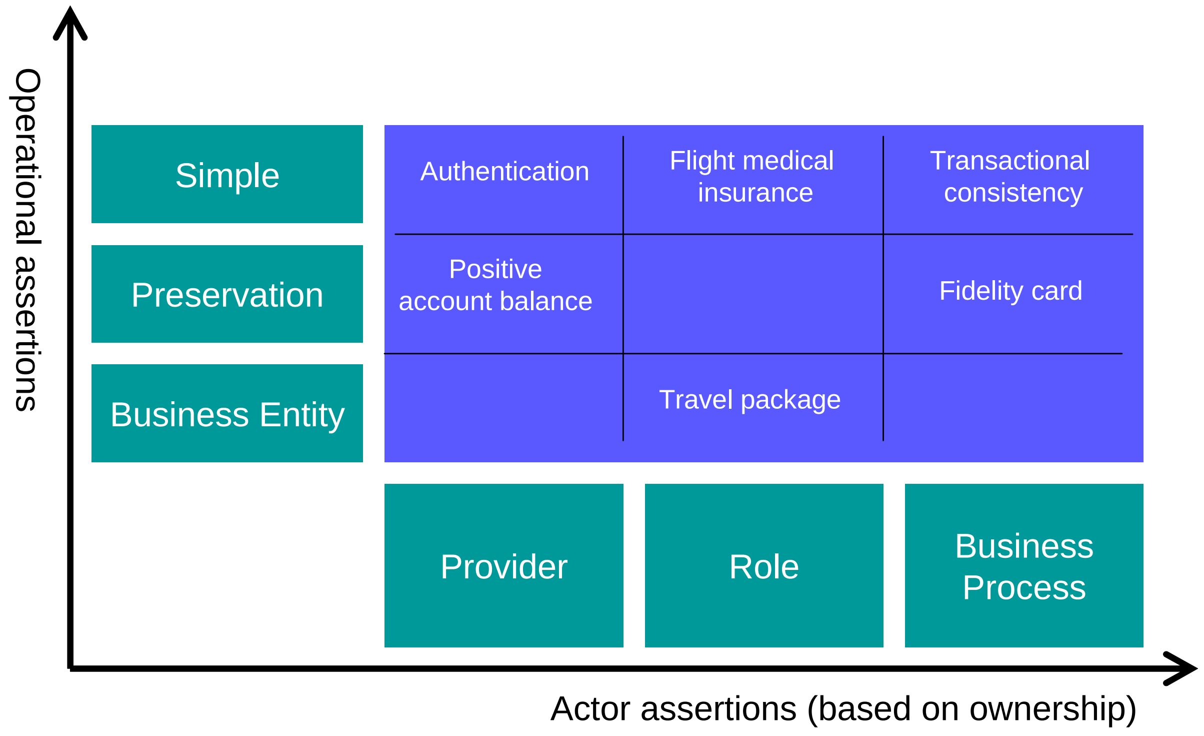


Figure 5.2: Two-dimensional classification of assertions, with examples.

final destination is in a particular set of locations where health risk exist. These assertions are defined together with the service interfaces and stored together with the service descriptions.

Provider-level. At the lowest granularity level assertions are published by a particular service provider. These assertions are stored in service registries together with service implementations. *Provider*-level assertions are used when a particular provider wants to enforce consistency of the business process and its business rules at runtime. For instance, provider role assertions may involve payment service providers having additional constraints, such as, protocol communication preferences, organization licensing, authentication, etc.

Assertions are classified on both the operational and the ownership dimension. The examples provided during the current presentation are summarized in the matrix in Figure 5.2. An example of a simple provider assertion is a requirement of user authentication before using the asserted service. A specific bank provider could require a preservation of a positive amount on the account. This is an example of the preservation/provider assertion. For role-based assertions, examples are the requirement for all travelers to have valid medical insurance or, for travel agencies, that the travel package must be processed with respect to the package entity assertion. Global business process assertion may include, for example, transactional consistency requirement. The users of the business process might have a special fidelity card, that gives them some advantages along the whole execution of the business process.

5.3 Monitoring framework

In Chapter 3 we focused on developing a service request language for web services in service-marketplaces that contains a set of appropriate constructs for expressing requests and constraints over requests as well as scheduling operators. This language, named XSRL for XML Service Request Language [3, 107], enables a user to formulate complex requests against standard business processes. These standard processes are provided by a *market maker* (a consortium of organizations) that brings the suppliers and vendors together. The market maker assumes the responsibility of creating a service-marketplace administration and performs maintenance tasks to ensure the administration is open for business and, in general, provides facilities for the design and delivery of business processes that meet specific business needs and conforms to industry standards [111]. Standard business processes are described in a choreography language such as Web Services Choreography Description Language (WS-CDL) [67]. WS-CDL specifies the common observable behavior of all participants engaged in business collaboration. Each participant could be implemented by completely different languages such as web services applications,

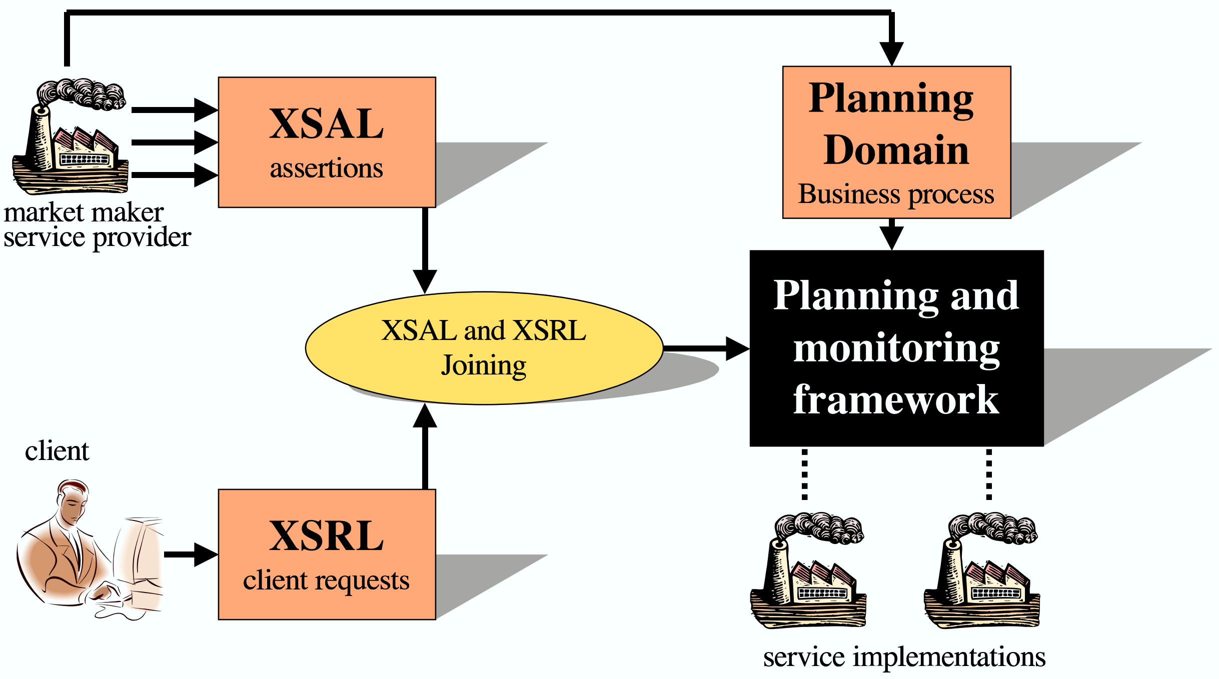


Figure 5.3: Handling of XSAL and XSRL requests.

whose implementation is based on executable business process languages like BPEL, XPDL and BPML.

XSRL and its supporting framework are a powerful tool for enabling a user to formulate requests against business processes but it lacks support for choreography assertions supplied by service providers and/or market makers that can be associated with the execution of a choreographed process. Assertions are essential means for the actors delivering the services and market makers to apply enterprise/marketplace policies and conditions. This limitation of XSRL is addressed by explaining how it is extended by means of an assertion language, which we name XSAL (XML Service Assertion Language).

XSRL and XSAL work in tandem during the planning and monitoring of business processes in order to satisfy the user requests in conjunction with applying service provider and marketplace maker supplied assertions. Figure 5.3 illustrates marketplace makers and actual service providers involved in the mar-

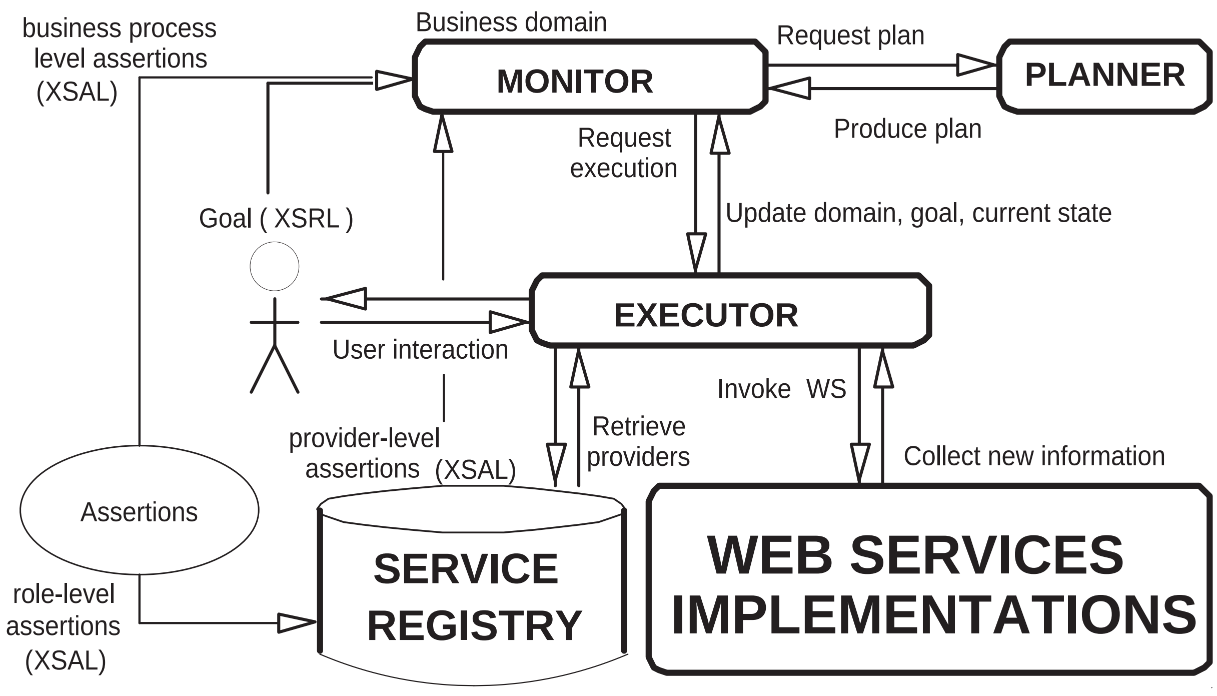


Figure 5.4: Planning and monitoring framework.

ketplace. These are seen to provide a set of assertions in XSAL which govern the behavior and execution of standard business processes. Assertions are associated with the standard business processes against which requests are specified. In Figure 5.3, a user or client states his requests in XSRL. These are combined with the appropriate XSAL assertions and then forwarded to the planning and monitoring framework presented in Figure 5.4. The planning and monitoring framework interacts with the actual implementations of the services in the service marketplace.

To deal with assertions and user requests we extend a system based on the interleaving of planning and execution. The proposed framework, shown in Figure 5.4, consists of four components: monitor, planner, executor and runtime support environment and can be seen as an extension of the monitoring framework introduced in Section 3.5.

The *monitor* manages the overall process of interleaving planning and execution. It takes user requests, the business process, the business process level assertions and starts interacting with the planner. The *planner* synthesizes a plan and returns it to the monitor. The plan is a sequence of actions to be executed. The planner returns a failure if there was no possible execution satisfying the user request in the given domain without violating the assertions. In case of a failure, the monitor eliminates eventual optional goals and assertions or it tries to change service providers. For example, if the business process fails to satisfy the assertion published by one hotel service provider, the framework can try to switch to another hotel service provider whose assertions are less strict. If the planner fails for all possible combinations then the overall execution of the business process fails. Assume that a correct plan exists and therefore it is synthesized. Then the monitor passes it to the executor. The *executor* is responsible for executing the plan. While executing each action of the plan, the executor may gather new information from the service registry or from the service implementations. Whenever new information is obtained, replanning is potentially needed and the domain updated with the just gathered information is returned back to the monitor. The framework works iteratively until the request is satisfied under the given assertions or there is no satisfying execution.

User requests and assertions are modeled as goals, the business process is a domain and the web services and service registries are the environment. After receiving a user request the system builds a plan based on the request, on the assertions and on the business process. The framework is developed to work with a state-based business processes where service invocations are represented as transitions between states. When executing a process the system should respect different constraints that are of two types: user requests and assertions. Assertions are defined on different levels: business process, role and provider. The latter two are stored in service registries. The system should satisfy role and provider assertions only if it intends to use services of corresponding role or provider, respectively. The framework returns success if it satisfies the requested goal without violating the assertions, it returns failure otherwise. When executing a business process the system interacts with the service registry to bind the process services, with web service implementations by invoking services.

5.3.1 Service assertion language

In Section 5.2, we showed that the business rules can be expressed using assertions. The assertions are defined as statements that either true or false in any of the given state. They are classified into simple, preservation, and business entity assertions. The assertions need to be stated in a uniform and unambiguous way by the parties involved in the business process. XSAL (XML Service Assertion Language) serves this purpose. The syntax of XSAL using BNF notation is provided in Appendix A.

One may observe the similarity between XSAL and XSRL. In fact, these two languages share the same expressive power and interpretation, though their intended use is quite different as XSAL is used for expressing assertions while XSRL is used for expressing user requests. Before assessing the formal connection among these two languages we shall first provide the intuitive meaning behind XSAL expressions.

The atomic objects of XSAL are propositions, that is, boolean combination of linear inequalities and boolean propositions. These can be either true or false in any given state. Propositions are further combined by sequencing operators to form assertions. The sequencing operators are: **achieve-all**, **before**-**then**, **prefer**-**to**, **optional**. The goal **achieve-all** succeeds when all nested assertions are satisfied, it fails otherwise. The construct **before**-**then** is satisfied when the first statement is satisfied and, from the state where the first statement is satisfied, the second is also satisfied. It fails otherwise. The construct **prefer**-**to** succeeds if the first statement is satisfiable, if not then it succeeds if the second statement is satisfiable, it fails if both statements are unsatisfiable. The last operator (**optional**) is the least strict constraint and demands the satisfaction of the assertion if possible, if not the assertion is ignored.

The operational assertions can be expressed using the XSAL language. All of the following operators take propositions as arguments. The *simple*, or *reachability*, assertions are expressed by XSAL reachability constructs. Formally, reachability constraints require satisfaction of some proposition before execution of the service that has reachability assertion. But strictness of the satisfaction depends on the particular operator. There are two corresponding XSAL operators: **atomic** and **vital**. The **atomic** operator is used when an assertion is strictly important for the party that specifies it and it must be satisfied regardless of any form of non-determinism. More formally, before executing a service that has this type of assertion, constrained propositions must be true. If there is no such execution then the execution fails immediately. The **vital** operator is used when less strict assertions need to be applied. It tries to find a successful execution to satisfy the constrained proposition. It executes until it has a chance to reach the successful state and fails otherwise.

The *preservation*, or *maintainability*, assertions are expressed by XSAL maintainability constructs. This constructs are used when preservation of some value is needed not only in a single state but during a whole execution sequence. When executing a service with such type of assertions only execution that preserves the constrained value can be followed. Retractable actions must be handled with care. In fact, if such an action is invoked and later retracted all associated assertions are ignored. As in the case with simple assertions, maintainability assertions can be of different types from the point of their strictness. We define two types: **atomic-maint** and **vital-maint**. The first one is used when the proposition value must be preserved along the whole execution regardless of the non-determinism. The second (**vital-maint**) is used when the maintenance assertion must be checked only along the non-exceptional execution. The system in this case should always intend to preserve the asserted proposition but if it fails then the execution stops and return failure.

The **entity** expression is used to form *business entity* assertions. This expression begins by relating to a particular variable. It specifies its starting value in the `start-from` statement and it is continued by any number of `follows` statements which specify the possible evolutions of the variable. Assertions of this type are always strict.

The semantics of XSAL can be defined following two trajectories: (i) considering formal semantic definition based on execution structures over planning domains; (ii) providing translation rules for transforming XSAL expressions into XSRL and combining them with XSRL expressions. Recalling that the semantics of XSRL has been defined we purse the second trajectory as it is more intuitive and better shows the relation occurring between XSAL and XSRL. As a point of notation, we add a `.t` postfix to denote the XSAL expression translated into XSRL, `(...)` to denote the passing of a parameter to a rule, e.g., `start-from(var)` and `follows (var)` takes `var` as a parameter. Expressions where the translation is omitted are propagated unchanged. The symbol '*' in the reduction rule denotes the usual Kleene star.

```
xsal   <- '<XSAL>' statement   '</XSAL>'
xsrl.t =  '<XSRL>' statement.t '</XSRL>'

entity <- '<ENTITY VARIABLE = ' var '>'
                     start-from (var)
                     follows (var)*
             '</ENTITY >'

entity.t = start-from.t +
             '<THEN>'
                 '<ACHIEVE-ALL>'
                     follows.t*
                 '</ACHIEVE-ALL>'
             '</THEN>'

start-from (var) <- '<START-FROM>' proposition '</START-FROM>'
```

```
start-from.t = '<BEFORE>' var proposition '</BEFORE>'


follows (var)   <- '<FOLLOWS>' proposition1   '</FOLLOWS>'
             '<BY>'      proposition2   '</BY>'
follows.t =  '<BEFORE>'
                 '<EQUAL>' var proposition1 '</EQUAL>'
             '</BEFORE>'
             '<THEN>'
                 '<EQUAL>' var proposition2 '</EQUAL>'
             '<THEN>'
```

From the translation one notes that the constructs on propositions, on sequencing and preference statements are the same in both languages XSAL and XSRL. The XSAL business entity assertion construct is not present in XSRL and is translated into the sequencing operators **before-then** binding the business entity variable to propositions.

5.3.2 Service assertion problem

Using a framework based on interleaved planning and execution demands a formal specification of the business process in terms of planning domains. None of the existing business process definition languages can be straightforwardly used as a domain description for our framework. For example, WS-CDL lacks monitoring mechanisms, BPEL lacks choreography protocol support. However, one can devise extensions and modification to these protocols in order to use them as domain descriptions. The domain representation that we adopt is a state-transition system introduced in Section 3.1. It is able to represent non-deterministic actions and potentially incomplete knowledge about the environment. Information that is unknown in advance is gathered at runtime by invocations of web services and by contacting the service registry to obtain web service generated information, e.g., current balances, debt histories, etc.

As a service assertion domain we use the service planning domain, presented in Chapter 3.3. To deal with assertions we rather adopt the definition of service planning problem from Definition 5. Formally, the service monitoring problem is defined as follows:

Definition 15 (Service monitoring problem). *A* service monitoring problem *is defined as tuple $\mathcal{P}_A = \langle \mathcal{P}, \mathcal{Q}, q \rangle$, where:*

- *$\mathcal{P}$ is a service planning problem from Definition 5;*
- *$\mathcal{Q}$ is a set of assertions that can be potentially published;*
- *$q : \{\mathcal{D}\} \cup \mathcal{R} \cup \mathcal{I} \to 2^{\mathcal{Q}}$ is a function that defines the set of assertions, associated with the business process ($\{\mathcal{D}\}$), roles ($\mathcal{R}$), or providers ($\mathcal{I}$).*

The satisfaction of a assertion is defined in terms of whether a particular plan satisfies the assertion or not with respect of whether associated roles and providers have been used during execution of a plan.

A *solution* to a service assertion problem is defined in terms of the plan that satisfies the combined user goal and published assertions. Formally, solution to service assertion problem is adopted from Definition 11.

Definition 16 (Assertion problem solution). *A* solution *for a problem $\mathcal{P}_A s$ is a plan π that satisfies the combined goal and assertions:*

$$\pi \models \textbf{\textit{achieve-all}} \ \{g, q(\{\mathcal{D}\}), \{q(r) | r \in \mathcal{R}_\pi\}, \{q(i) | i \in \mathcal{I}_\pi\}\},$$

where

- *g is a goal extracted from the user request from Definition 4;*
- *$q(\{\mathcal{D}\})$ is an assertion associated with the business process;*
- *$\{q(r) | r \in \mathcal{R}_\pi\}$ are assertions associated with roles. $\mathcal{R}_\pi$ denotes those roles that are used in the execution of the plan;*

- $\{q(i)|r \in \mathcal{I}_\pi\}$ *are assertions associated with providers.* $\mathcal{I}_\pi$ *denotes those providers that are used during the execution of the plan;*

Therefore, a *problem* of monitoring of assertions is the finding of a solution, i.e. plan, for given domain, goal, initial state and assertions associated with involved parties.

Note that the problem of monitoring is more complex than the service planning problem: in monitoring the goal to be satisfied is updated on the fly, depending on the execution chosen.

5.3.3 A domain instance

Let us revisit the example in the travel domain introduced in Figure 1.3 to explain the XSAL use and constructs. Next we present it according to the formal definition of a domain D presented in Section 5.3.2.

There are fourteen states $\mathcal{S} = \{1, 2, \ldots, 14\}$ in the upper half of the figure. The set of variables Var is $\{hotelReserved$, $hotelPrice$, $location$, $trainBooked$, $trainPrice$, $flightBooked$, $flightPrice$, $confirmed$, $money\}$, among which one distinguishes the boolean variables ($hotelReserved$, $trainBooked$, $flightBooked$, $confirmed$), from the real variables ($hotelPrice$, $trainPrice$, $flightPrice$, $money$), and a variable representing location names ($location$). In the set of variables a subset is defined to be of knowledge variables. In the example, we define $hotelPrice$, $trainPrice$, $flightPrice$ to be knowledge variables. There are also nineteen actions that can be performed in the domain $Act = \{a_1, \ldots, a_{19}\}$.

Several roles are involved in the travel business process, that is, $\mathcal{R} = \{$`user`, `hotel`, `air`, `train`, `payment`, `insurance`$\}$. The `user` role represents the requesting party. Typically it is a human user, but it could also be any application software utilizing the business process. The set of actual providers for the roles $\mathcal{R}$ are stored in the service registry.

Arrows in Figure 1.3 are the process actions. For example, states (3) and (4) are connected by the action `reserveHotel` of the `hotel` role. It has two outcomes: normal, where the variable `hotelReserved` is set to true and exception, where the hotel remains unreserved. This action is an example of a non-deterministic action. The two arrows from the state (4) represent different outcomes for this action. Other examples of actions are `bookFlight` for the `air` role and `getTrainPrice` for `train` role.

Assertions work in conjunction with the travel business process and are defined in XSAL. The business process level assertion that ensures that the process always reaches the `final` state is expressed in the following way: **atomic** $final$. Here and in the following we omit XML tags for brevity. An examples of a role-level assertion is the requirement for insurance in case of prolonged stay abroad: **vital** $(healthRisk \rightarrow insuranceTaken)$, where $\rightarrow$ represents logical implication and is expressed using the `<NOT>` and `<OR>` XSAL expressions, as usual.

At the provider level, the hotel provider may prefer, for example, a specific credit card type for payment: **optional** $cardType =$ VISA.

The maintenance assertion for customers of loyalty services that was introduced in Section 5.3.2 is encoded as follows: **optional**$(loyaltyCard \rightarrow (roleType = acceptsLoyaltyCard))$.

In the following we use XSAL to codify the business entity assertion that was presented in Figure 5.1. The XSAL syntax for this assertion is:

entity *travelPackage*
start-from *requested*
 follows *requested* **by** *rejected* $\vee$ *accepted by travel agency*
 follows *rejected* **by** *requested*
 follows *accepted by travel agency* **by**
  rejected $\vee$ *approved by client*
 follows *approved by client* **by** *package completed*

Additional details like precise hotel information, seats type, payment numbers, etc. can be easily integrated in the above example. To do so, one should add corresponding variables and modify the semantic functions of the actions to take into account the introduced variables. Here we omit such additional details to improve readability.

5.3.4 Planning and monitoring algorithms

Having introduced a planning and monitoring framework and the assertion language XSAL, we present the algorithms which handle XSAL assertions together with XSRL requests. Referring to Figure 5.4, we recall that the framework consists of three main components, that is, a monitoring, a executor and a planner. We present algorithms for these components separately.

Algorithm 7 monitor(domain d, state s, goal g)

```
$\pi$ = assert-plan($d, s, g$)
if $\pi$ = $\emptyset$ then
    return success
else
    if $\pi$ = failure then
        if chooseNewProvider($provider$) then
            $d'$ = updateDomain($d$)
            $assert_{provider}$ = extractAssertions($provider$)
            $g'$ = updateGoal($g, assert_{provider}$)
            return monitor ($d', s, g'$)
        else
            $g'$ = generate-rollback-goal()
            monitor($d$, $s$, $g'$)
            return failure
        end if
    end if
    ($d', s', g'$) = execute($\pi, d, s, g$)
    return monitor ($d', s', g'$)
end if
```

The monitor takes a domain d, that is built on the basis of the business process, an initial state s and a goal g. The initial request of the user to the system is combined together with business process assertions, thus, the monitoring algorithm is invoked initially with the following goal: $\textbf{achieve-all}(request, assert_{bp})$, where $request$ is the user request and $assert_{bp}$ is the set of business process level assertions.

The *monitor* (Algorithm 7) is the core of the interleaved planning and execution process. It invokes the planner and the executor in order to satisfy the user requests and the assertions, and it recovers from failures. The algorithm is an extension of the monitoring algorithm presented in [84], where the most notable difference is the updating of the goal to take into account the provider level assertions. When a new provider is chosen then the goal is modified in the following way. First, assertions that are associated with the previously assigned provider being de-assigned are eliminated from the goal. Second, assertions of the new provider are added to a goal by using the **achieve-all** operator. The modification of the goal to take assertions into account is performed by the `extractAssertions` and `updateGoal` functions.

The *executor* (Algorithm 8) takes a plan and executes it in the marketplace. It contacts the service registry when a service implementation for a given role is necessary, it executes actions of the plan and it checks whether replanning is required. When a new provider is requested from the service registry, its assertions are added to the goal g in the following way $\textbf{achieve-all}(g, assert_{provider})$. This is achieved via the `extractAssertions` and `updateGoal` functions.

During execution of the plan π, runtime information is gathered and new assertions are taken into consideration. The plan π must be either compliant with the updated information, or, if it is violated, replanning is performed from the current state.

The function `plan` is the one presented in Chapter 4.

Algorithm 8 execute(plan π, domain d, state s, goal g)

```
repeat
  $a$ = firstAction($\pi$)
  $\pi$ = $\pi - a$
  if webServiceAction(a) then
    if noProviderForRole($role_a$) then
      $providersList$ = contactServiceRegistry($role_a$)
      $provider$ = chooseProvider($providersList$)
      $assert_{provider}$ = extractAssertions($provider$)
      $g'$ = updateGoal($g, assert_{provider}$)
      return $(d', s', g')$
    else
      $provider$ = previouslyChosenProvider($role_a$)
    end if
    message = invoke($a, provider$)
  end if
  $(d', s', g')$ = update($d, s, g, a$,message)
  if isKnowledgeGathering($a$) then
    return $(d', s', g')$
  end if
until $\pi$ = $\emptyset$
return $(d', s', g')$
```

Algorithm 9 assert-plan(domain d, state s, goal g)

```
$\pi$ = plan($d$, $s$, $g$)
if $\pi \neq$ failure then
    $\{assert_{a_1}, \ldots, assert_{a_n}\}$ = extractAssertions($\pi$)
    $g'$ = updateGoal($g$, $\{assert_{a_1}, \ldots, assert_{a_n}\}$)
    if $g' = g$ then
        return $\pi$
    else
        return assert-plan($d$, $s$, $g'$)
    end if
else
    $g'$ = checkViolatedActions($g$, $d$)
    if $g' = g$ then
        return failure
    else
        return assert-plan($d$, $s$, $g'$)
    end if
end if
```

5.4 Monitoring a sample business process

To illustrate the application of the algorithms just presented in the context of the planning and monitoring framework, we use the example presented in Section 1.1 and formalized in Section 5.3.3. Suppose a user is planning a trip to Nowhereland and is interested in a number of possibilities in connection with this trip. These include making a hotel reservation, avoiding to travel by train, if possible, and spending an overall amount not greater than 300 euro for the whole package. Further, the user prefers to spend less than 100 euro for a hotel room but, if this is not possible, may be willing to spend no more than 200 euro for that room. This would be expressed by the following XSRL request:

```
achieve-all
    achieve-all
        prefer vital-maint $hotelPrice < 100$
            to vital-maint $hotelPrice < 200$
```

optional-maint $\neg trainBooked$
vital $confirmed \wedge$
$location = ``Nowhereland'' \wedge$
$hotelReserved$
vital-maint $price < 300$

In addition, suppose that two XSAL business process level assertion such as **atomic** $final$ and the business entity assertion of Figure 5.1 were published. The system starts by combining the user request with the business process assertions in an **achieve-all** construct. The monitor invokes the assert-planner which in turn invokes the planner. The first actions of the initial plan provided by the planner, given the above goal, the business process assertion and the domain as shown in Figure 1.3, is the following sequence of actions: `getHotelPrice`, `reserveHotel`.

The monitor then sends the plan to the executor to start interacting with web service implementations. By these invocation a travel agency and a hotel provider are selected and a room is reserved. Suppose that the government considers Nowhereland to be a health risky location. Then the role level assertion **vital** $(healthRisk \rightarrow insuranceTaken)$ coming from the service registry together with the travel agency role is considered. At this point, the executor returns control to the monitor which in turn requests a new plan from the assert-planner taking into account the given role-level assertion. The new plan generated will now comprise an action bringing the process in the `obtained a medical insurance` state.

Suppose further that the selected hotel is "MyHotel" which comes with the provider level assertion **optional** $cardType =$ VISA. Then, when the executor runs the request payment from the user the $cardType$ is asked to be VISA. If the user refuses such option, execution nevertheless proceeds. Note that if the assertion was **vital** $cardType =$ VISA then the user's refusal would result in a assertion violation and thus a plan failure.

As for a maintainability assertion, suppose that the travel agency is asked by the client to provide services complying with a given loyalty card. Therefore, the travel agency publishes the following assertion: $\mathbf{optional}(loyaltyCard \rightarrow (roleType = acceptsLoyaltyCard))$. This is taken into account by the assert-planner as soon as the user has specified the card in his request.

As for the business entity assertion requiring a travel package to be assembled following specific rules (Figure 5.1), this assertion is always taken into account by the assert-planner when providing new plans to the monitor. Finally, the execution proceeds until the travel package is completed and the user approval is requested. At this point the business level assertion $\mathbf{atomic}\ final$ is the last to be satisfied. This is achieved by a plan going to the `final` state of the business process.

5.5 Discussion: expressing QoS properties

In Section 5.3.1 we introduced the XML Service Assertion Language and showed how it can be used to express objectives and preferences of the parties involved in the execution of a business process. These objectives may be exposed in a service-level agreement (SLA). But SLAs are not limited to functional requirements, often service providers want to expose to the users of their services various quality of service features. These agreements, often in the form of legal contracts, define what services the provider offers and define the quality of service or QoS that they offer. Because of the formal nature of SLAs, the quality of service needs to be specified in measurable terms, such as the guaranteed uptime of the service, the guaranteed maximum and average response times of the service, etc [98]. Various non-functional properties of services are the object of SLAs, most notably: availability, accessibility, performance, reliability, security, transactionality, and regulatory. There are several specification proposals to address QoS and SLAs, for example, WS-Policy [144] or Web Service Level

Agreement [147].

Interestingly, XSAL is able to capture most of these qualitative and quantitative QoS properties in its assertions. Next we show examples of how XSAL expression are used to express quality of service properties and be therefore used as fundamental blocks of SLAs. The advantage of using XSAL for this purpose is twofold. On the one hand, it has the appropriate expressive power to express non-functional properties during the agreement negotiation, on the other hand, it comes with a monitoring framework which serves the purpose of checking at runtime that the SLA terms are not violated. At runtime, instead of rejecting the violated service, the system tries to satisfy the failed assertion or, if that fails too, checks if there are any other business process executions that satisfy the original goal and preferences. For example, let us imagine that the business process failed to present valid credentials to the bank service. The framework first tries to check if there are any activities in the business process that can possibly provide the necessary credentials. If that fails, then the system tries either to ignore the service or select a different bank provider, if there are more available.

Let us now consider what types of service-level agreements can be captured by XSAL expressions. The most relevant categories for QoS requirements in the context of web services are: availability, accessibility, performance, reliability, regulatory, security and transactional behavior [93]. Let us consider them individually.

5.5.1 Availability, accessibility, performance, and reliability

Availability is the quality of whether a web service is available and ready to be invoked. It is defined as a probability of service availability. Sometimes, the time to recover is also added to availability terms, defining the time it takes to repair a temporally non-available service. *Accessibility* is expressed as a probability measure to define whether the service is able to perform a given operation. The service could be available but not accessible if, for instance, the hosting server is overloaded. *Performance* shows how fast the server processes the requests and how many requests are served in the time unit. *Reliability* represents the service degree of being capable of maintaining service quality.

All these properties define the ability of a service to process requests efficiently. These kind of quality aspects are useful for mission-critical business processes requiring high levels of availability and excellent performance.

An example of using XSAL to express availability assertion in the banking domain follows. Suppose one service desires to get the latest financial information in real time. In this case, the business process contains at least two services: a requester and a service providing the necessary data. High performance requirements are expressed in the requester assertion: **vital** $dataFetched \wedge latency \leq 20ms$. Having this assertion the framework checks all the services that provide the financial data (that is those services satisfying the variable $dataFetched$) and selects only the services with the response time lower than $20ms$. The information about the latency time is taken from the service-level agreement of the provider service. The same schema can be applied to check accessibility and reliability quality aspects.

5.5.2 Security

Security is a paramount aspect of service-oriented architectures in its various facets [63, 7, 55], such as message encryption, authentication, and access control. Message encryption is usually handled at the platform level, therefore, XSAL's use is limited to simple encryption requirement expressions, e.g., **vital** $encryptio$
$128bit$.

More interesting is the case of authentication. Service-level agreements contain an XSAL assertion that defines the required security information. For example, the bank provider asks for a particular credential to be provided, e.g., **atomic** $login = \texttt{true} \wedge provider = \text{“}Visa''$. When the framework processes this assertion, it tries to satisfy it by looking if there is an execution in the business process that invokes the service operation that satisfies the variable $login$ and sets the security provider $provider$ to “Visa”. If that fails, then the framework tries to satisfy the initial request in a different way, not using the bank or, if that is not possible, tries different bank providers. The key point of using XSAL assertions is that the framework is delegated to adjusting the execution of the business process according to the provider assertions.

Access control service-level agreements can also be expressed in XSAL. Imagine a situation in which a bank exposes several service implementations. Then, the particular implementation is unknown until instantiation of the providers: every service implementation may contain different requirements, assertions, and preferences based on user access rights, therefore, the future executions of the business process strongly depend on the exposed constraints. In other words, the behavior of the business process depends on the access control rights. For example, say a travel agency considers two different types of users: $normal$ and $loyal$. The implementation for the second might contain the following assertion: **optional** $loyalpartner = \texttt{true}$. This assertion requires the system to prefer providers that are partners of the travel agency, that might offer special discounts, finally allowing to provide better services to the agency’s client.

5.5.3 Transactionality

The loosely coupled and stateless nature of initial web service proposals has posed new challenges for the execution of sequences of service operations which needed to be treated, for instance, atomically. Transactionality of service invocation demands different solutions from traditional database style transactions [110].

Often sequences of service invocations have to support atomic behavior, when, if some service fails, all intermediate changes have to be rolled back. The question such as whether transactions are applicable in the web service environment or what kind of transactions need to be supported (e.g., atomic or long-lived with compensation) are out of the scope of the present thesis. However, XSAL with its **atomic** assertion guarantees some form of transactionality.

Consider the traditional transactional model, using two-phase commit with satisfaction of all ACID properties. WS-AtomicTransaction [145] is a standard that deals with this kind of transactions. In this model, transactions are always consistent and atomic. However, this is only achieved if all of the participants support the corresponding transactional agreement. Sometimes this is weakened and services control their desirable transactional behavior by publishing corresponding attributes. For instance, a Java EJB specification may contain the following attributes: `notSupported`, `supports`, `required`, `requiresNew`, `mandatory`, `never`, and a bank provider exposing some of its data might ask for all the invoked services to support transactionality. An XSAL assertion to achieve such a guarantee is the following: **atomic-maint** $attr \neq$ `notSupported`. The framework takes the assertion and allows the publishing service to participate in the transactions with all participants who support the assertion. In the same way, transaction isolation levels could be set according with specific service quality requirements.

For the long-lived transactions the situation is different. Special attention has to be payed to consistency and atomicity, as transactions based on compensation do not guarantee them. XSAL can express such requirements. First, one could check the consistency of data lifecycle by using the **entity** assertion (e.g., Figure 5.1). In general, to ensure consistency and atomicity the following two operators are used: **atomic** and **atomic-maint**. They ensure that execution satisfies the assertion despite any possible non-deterministic failures. For example, $consistent$ is a variable that is true in all so-called consistent states and false in all other. The assertion **atomic** $consistent$ guarantees that the execution terminates in one of the "consistent" states. That is, before executing a transaction, the framework checks whether all possible executions end up in states that do not violate the assertion. This is a different notion from that of atomicity in ACID transactions as no roll-back is involved, nevertheless is a form of guarantee which starts a sequence of service invocations only if it is possible to arrive to a final state despite any form of non-determinism.

5.5.4 Regulatory

In [93] it is defined the *regulatory* quality of service aspect which represents the conformance of services to specified standards. This type of service-level agreement is usually processed at the level of underlying platform, since it is truly non-functional property. This kind of non-functional property sis beyond the scope of the presented XSAL framework.

Chapter 6

Implementation

In Chapter 3 we introduced the framework for interleaving planning and execution, here we consider an implementation of it. The framework request language XSRL is developed to empower users with explicit control over process executions by describing desired service attributes and functionalities, including temporal and non-temporal constraints between services. The reference model instantiation is planned according to the goals and preferences specified by the user and an appropriate plan is executed. The algorithms underlying it are based on the idea of interleaving planning and execution. These algorithms are based on model checking and on constraint programming.

The reference implementation of the XSRL framework is written in Java 1.5 using the Eclipse (`www.eclipse.org`) programming environment. The framework constraint solver is based on the Choco constraint solver [32]. Choco is a Java library for constraint satisfaction problems (CSP), constraint programming (CP) and explanation-based constraint solving (e-CP). In a Web Service scenario, users may wish to know why certain solutions are preferred to others. Explanation-based constraint programming is a viable approach to tackling such issues. This is one of the reasons that lead us to Choco, as Palm is an explanation-based constraint system built on top of it.

Constraint programming languages allow us to model Boolean requests, numeric requests, symbolic requests, preferential and optional requests *directly and compactly*, without modeling them into propositional formulas (as it would be if we adopted a SAT based encoding [66]); this is already beneficial. Constraint programming aims at building and producing solutions in an interactive manner, optimal or not: most constraint solvers are in fact *incremental*, which allows us to add constraints at run-time. In Web Service scenarios this is a critical feature as some information becomes available only at run-time, and it is totally unrealistic to assume the contrary.

SAT-based solvers do not usually provide off-the-shelf facilities to easily program specific variable orderings, search and optimization strategies; SAT solvers are highly tuned for dealing with decision problems, and the 'quality' of the solution, whether optimal or close 'enough' to optimal, is not of primary concern. Whereas constraint programming provides these facilities. This smoothes the integration of a constraint programming reasoning system into our framework for interleaving planning and execution, see Figure 3.1. For instance search algorithms such as branch-and-bound are implemented in most constraint programming systems; in general, implementations of constraint programming algorithms provide optimality parameters which allow us to tune our framework in a real-time Web Service environment.

We implement the modeling of the service problem into the service constraint problem, and we run some preliminary tests. In addition to the provided algorithms in Section 3.5, we implement a domain generator allowing the tuning of the following parameters: number of states, branching factor, nondeterminism rate, maximum directed cycle length, and if the service domain corresponds to a directed acyclic graph or a not. We then experiment with different XSRL requests of increasing complexity on different domains keeping the whole constraint set in the 64Mb of main memory. We do not provide a full evaluation of the system here, as we are interested in a preliminary feasibility study, but we mention basic facts of implementation:

- the system handles simple reachability requests on domains of up to 2000 states with low levels of branching and non-determinism on an average of 0,947 seconds;
- if we also allow for cycles the average execution time grows up to 1,737 seconds;
- increasing the complexity of the requests by using vital requests, the average execution time increases to 4,29 seconds and the maximum number of state goes down to about 1000;
- further combining reachability and maintainability operators the average execution time reaches 10,97 seconds.

The above values have no pretense of being an evaluation of the system (for more information, please refer to XSRL Reference Implementation web page [78]), but they do show the feasibility of the approach. It should be also noticed that the number of maximum states reached depends on the limited use of memory and the fact that the whole search space is kept into RAM. Nevertheless, we are to deal with domains which are considerably larger than the biggest ones seen in practice (smaller than 200 states: H. Ludwig, IBM TJ Watson. Personal communication. 2006).

6.1 Implementation details

The typical processing of the request from the low-level implementation view is done in the following way. First, the business process is loaded by the framework. This task is accomplished by an instance of the `FreeBPCompiler` class: this class calls a parser, implemented in SAXFreeBPHandler, that uses the XML SAX libraries in order to build the internal data structure. Once the domain is read, the application creates a monitor that has the task to handle the execution of the plans and the request searching for providers if needed. The next step is the definition of the problem itself: the application takes the user XSRL request: it is then converted in a Goal object, which is used along with the business domain to build the problem to solve. At this point the monitor takes the control of the execution; in particular it manages the creation of a planner and an executor for the planning problem. The monitor starts iteratively invoking the planner (to synthesize a plan) and executor (which has to process the actions defined in the plan), checking in the mean time whether the new provider has to be added to the problem. As a default planner implementation, Choco constraint programming library is used.

Let us now go through implementation details of the framework java packages.

The `org.xsrl.domain` package contains the elements that are used in order to represent the domain internally to the application. The most important classes are:

- `Problem`: represents the problem as a whole, containing a domain and a goal;

- `Domain`: represents the domain, composed of states, transitions between states, variables, the roles implemented by the providers, the actions defined for the transitions;

- `Transition`: a transition represents one transition in the control flow of the business process. Depending on the action result, it may lead to different states;

- `State`: it represents the state of the process;

- `Variable`: a variable defined in the business process. Can be defined on different spaces: integer, reals and booleans. Additionally, variable may be in a “not yet defined” state to represent the unknown not yet observed information;

- `Action`: it represents an operation of an arbitrary web service. An action must refer to some role, and can have different results, that are defined by `ActionResult`s. represents a specific result from the execution of an action. The nature of this class lets the domain to distinguish between normal results and faulty ones;

- `ActionEffect`: this class represents a particular effect, given by the execution of an action with a specific returned result;

- `DomainObject`: the base class extended by all the other in the package. Represents a generic object composing the domain;

Figure 6.1 shows the UML class diagram for the `org.xsrl.domain` package.

The `org.xsrl.domain.goal` package contains the definitions of the possible goals that can be instantiated. All the classes implements `Goal`. `BasicGoal` keeps the information about its nature, such as if it is a MAINTAINABILITY or REACHABILITY goal, and whether it is a VITAL or ATOMIC. Complex goals are presented by classes `AchieveAllGoal`, `BeforeThenGoal`, and `PreferToGoal` that represent **achieve-all**, **before-then**, and **prefer-to** goals correspondingly.

In the `org.xsrl.domain.spaces` package classed for the supported types of variable spaces in which a process variable, defined in the domain, can belong to. Boolean, real and integer spaces are supported by the system.

The `org.xsrl.framework` package contains the class definition for the main parts of the framework: monitor, executor, and abstract planner are defined. The monitor has a crucial role in this implementation, in fact it has to accomplish various tasks, including:

- the creation of the problem related to the underlying constraint engine (in our case a Choco);

- the responsibility to call the classes (in particular it calls `ChocoSolver` methods via the `CPSolver`) that encode the user's requests and the domain;
- the retrieval of the solutions, in terms of values for the β variables, for the problem filled with the constraints given by the last step;
- the responsibility to call the executor in order to execute the produced plan.

Executor processes the plan by invoking corresponding actions and updating the domain internal representation with the new observed information from the results of action execution. To assists executor, an arbitrary class `ActionInvoker` is provided to implement a dynamic invocation of a particular action on a particular provider.

The `org.xsrl.framework.registry` package contains classes to work with registries of services. It is used by the executor and the monitor, when provider is needed for executing an action.

Abstract planner defined in `org.xsrl.framework` package provides general, abstract view on planning operations. Actual encoding of the problem to a constraint solver and retrieving the solution in terms of plans and actions is done by classes in `org.xsrl.framework.cp` package. Figure 6.1 shows the UML view on the package. It accomplishes the following tasks:

- representing the business domain in such a way that the algorithms can easily handle it, using the elements in package `org.xsrl.domain`. This task is mainly accomplished by the `Encoding` class.;
- implementing the algorithms to encode the domain and the requests, as seen in Chapter 3. The implementation for this task is divided into several classes: `ChocoSolver` encodes the domain (it gives the internal representation for the business domain), detects the presence of cycles and calls the methods that encode the user's goal. For each type of possible goal, there is a class devoted to the encoding of that particular goal(e.g.,

AchieveAllEncoder, BeforeThenEncoder, VitalMaintEncoder, VitalGoalEncoder). All these classes uses a registry containing the variables defined for the domain (the instance of the class CPVarName); every basic goal class has an internal registry for the variables defined while encoding a particular goal. The complex goals, e.g., AchieveAllEncoder, relates the variables from different subgoal encoders to ensure that all subgoals variables are properly inter-related. The encoding of the cycles is a task demanded to the CycleEncoder class, that provides methods to encode a list of cycles starting from the same state.

The following sections provide an overview of an implementation for the algorithms provided in Section 3.5.

6.1.1 Encoding algorithms implementation

The encoding algorithms provided in this section is based on Choco constraint programming system. Choco is an open-source constraint-solving library written in Java, developed under the BSD license, that can be easily used to express and solve numeric constraints. Once the problem is solved, the constrained variables are assigned to the values that satisfy the conditions. It is possible to get all the solutions, or the first one encountered by the solver. We use the latter possibility since the framework guarantees the satisfaction of request expressed in XSRL without checking whether the solution is optimal.

Internal representation of the domain

Inside the application, the representation of the business domain is made by the class Encoding; it contains a reference to the state of the business domain encoded, keeps the information about the cycles starting from the represented state and the encodings of the states reachable from it. These information are kept in a HashMap that has a transition as a key and another HashMap as value.

This second map represents the relation between every possible result (the key) for the transition with the encoding of the reached state (the value). This last introduced encoding contains an instance of the class `Encoding` and an instance of the class `CPVar`, that will be used to identify the mathematical expressions, defined using the choco library, used for choose between alternative paths in the graph. It is important to notice that the class `CPEncoding` contains an instance of the class Encoding, while this one contains a number of CPEncoding inside the "double-layer map" called *nestedEncodings*.

The UML schema for the classes involved in representing the internal view of the domain can be seen in Figure 6.2, where it is represented the `org.xsrl.framework.cp` package.

Recognizing and handling cycles

A sequence of transitions can generate a cycle; self-transitions can also be present. This situation is recognized while the symbolic representation of the graph is built; this is done recursively by the methods defined in the class `ChocoSolver` from `org.xsrl.framework` package; in order to recognize and keep track of the needed information describing the cycles, four stacks are used: one containing the states, representing the path; one for the encodings for the states in the path, one for the transitions followed and one for the results of these transitions. These info are kept in the `Encoding` class, representing the state from which the cycle starts. It is possible to have more than one cycle originating from the same state, so we have to save the info in another `HashMap`: given the counter for the cycle (n_i, the key) we get a set of lists, starting from which we can build the mathematical expression for the cycle.

The code presented below is able to recognize the cycle and save the relevant information related to it; first of all it checks if the state is contained in the stack representing the path followed (starting from the initial state) and if the state has already been visited; if that is true, a new cycle starting from the current state has been found and then the info about it have to be collected and stored. If the current state has been visited, but it is not in the current path, then we have found a converging point. The encoding has to be done separately for every single transition coming into this state, so we simply forget we have already visited this state and then (recursively) proceed. Finally, we come to the core of this method: first, we set the current state as visited and we push the state in the path stack; then, we check if the current state has no exiting transitions, in which case we do nothing and return an empty encoding; otherwise we create a new encoding for that state and, for every different transition that goes outside of the state, we push the encoding and the transition in the appropriate stack and then recursively build the encodings for the reached states. These actions affect the encoding just pushed in the stack, filling the *nestedEncodings* map for the current encoding. After that a transition has been taken into account we check if the current state has been inserted in the list that contains all the "cycle starting" states. If this is true, we remove all the occurrences of the current state from the list and then add the list containing the information for all the cycles we have found till now, starting from this state, to the encoding representing the state. After this check we call the pop method on the encoding and on the transition stacks. After that all the transitions for the current state have been considered, we can safely pop the current state from the path and return the encoding filled up with all the relevant information for the popped state.

```
encodeDomain(State state, Domain d)
      if(state.isVisited() and path.contains(state))
           collectCycleInfo(state);
           return Encoding.emptyEncoding;

      else if(state.isVisited())
           state.setVisited(false);
           encodeDomain(state, d);

      else
```

```
          state.setVisited(true);
          path.push(state);
          if(state.isTerminating())
               path.pop();
               return Encoding.emptyEncoding;

          Encoding stateEncoding = new Encoding(state);
          for every transition t leaving from state
               encodingStack.push(stateEncoding);
               transitionStack.push(t);
               encodeAction(stateEncoding, t, state, d);
               if(statesWithCycles.contains(state))
                    do
                        statesWithCycles.remove(state);
                    while(statesWithCycles.contains(state));
                    stateEncoding.addCycles(cycleMap.get(state));

               encodingStack.pop();
               transitionStack.pop();

          path.pop();
          return stateEncoding;



collectCycleInfo(State state)
     statesWithCycles.add(state);
     for every state in the cycle
         collect info from the stacks

     cycle = new Cycle(infoCollected);
     cycleList = cycleMap.get(state);
     cycleList.add(cycle);
```

The UML schema for class `CycleContents` can be found in Figure 6.2: it is noticeable that this class contains an instance of the `CPVariable` class. This instance represents the variable that counts the number of times the cycle, described by the `CycleContents` class, is followed. Moreover, it is the key to get a particular cycle description from the map inside the `Encoding` class.

The code shown below builds the mathematical expression for a list of cycles, starting from the information retrieved from an instance of the `Encoding` class; the encodings of the cycles starting from the same state have to be summed up, so we keep them all in a list. For every cycle we have to consider the encodings of the states involved, the transitions that keep the execution in the cycle and the results for these transitions. These three lists contain all the information needed for every single step of the cycle; if the result is not normal and is anyway non-null (this is an internal definition), then we have to pick the ξ associated with the result and store it in a multiplier variable; in any case we sum up the effects. In the end, if the summation for the effects is non-null, we have to keep the encoding of the cycle and store it in the list mentioned above, otherwise we forget the expression for the cycle with no effects.

```
encodeCycleList(cycleInfoList list)
    for every element c in list
         encodeCycle(c);

    return sum(cycleExp);



encodeCycle(cycleInfo)
    encodingList = cycleInfo.getEncodings();
    transitionList = cycleInfo.getTransitions();
    actionList = cycleInfo.getActionResults();
    for every transition t in transitionList
        currentActionResult = actionList.getNext();
        currentEncoding = encodingList.getNext();
        currentCPE = currentEncoding.
                  nested.get(t).get(currentActionResult);
        if(not(currentActionResult.isNormal()
                  and not(currentActonResult.isNullAction())
             cpVar = currentCPE.getCPVar();
             var = put(cpVar);
```

```
                multiplier = multiplier * var;

            effect = chooseTheRightOne(getEffects(t,
                                           currentActionResult));
            cycleEnc = cycleEnc + effect;

        multiplier = multiplier * n;
        cycleEnc = cycleEnc * multiplier;
        cycleExp.add(cycleEnc);
```

Implementing the vital constraint

The algorithm encoding the **vital** goal takes into account iteratively every possible transition and then encodes recursively the destination. Every transition is associated with a multiplier (the boolean variable, controlled or non controlled) and with a result, if the effects of the action modify the state of the variable defined in the goal.

```
Expression encodeVital(Goal goal, Encoding encoding,Variable v)
    Expression expression = new Expression(goal);
    for every Transition t reachable from encoding
        for every ActionResult result for t
            CPEncoding cpEnc = encoding.nested.get(t).get(result);
            mult = boolean variable associated with the result;
            if (result is null)
                nested = encodeVital(goal,cpEnc.getEncoding(),v);
                if(nested is not empty)
                    expression += nested * mult;



            else
                effect = effect associated with the result for v;
                nested = encodeVital(goal,cpEnc.getEncoding(),v);
                if(nested is empty)
                    if (effect is not null)
                        expression += mult * (effect);
                else
                    expression += mult * (effect + nested);




    return expression;
```

Implementing the vital-maint constraint

The algorithm that encodes the **vital-maint** constraint, operates recursively on the internal representation of the domain, which is an instance of the `Encoding` class. Every state visited is associated to a personal representation that encodes the local choices, made among the leaving transitions. For encoding this particular constraint we have to take into account the effects related to the "future" choices. For this reason the mathematical expressions related to the reachable states, built in the recursive calls, are inserted into a list. Every element of this list has to be put in relation with the expression of the state currently visited. Moreover, we have to take into account the possibility of having cycles starting from the current state. Generally, when we have an expression *se* coming from the reachable states, we have to compose the current elements with it in this way: $mult\,(effect + se) + cycleEnc$
where *mult* is the variable that expresses the choice among different paths that leads to the creation of the expression *se*, *effect* is the action result returned by the execution of the action related to the transition and *cycleEnc* is the expression for the cycles starting from the current state.

```
encodeVitalMaint(Encoding encoding, List history)
     State currentState = encoding.getState();
     if(encoding.hasCycles)
          cycleEnc = encodeCycleList(encoding.getCycleList());
```

```
for every Transition t leaving currentState
     for every ActionResult result for t
         List effects = getEffects(t, currentActionResult));
         if(!effects.isEmpty())
              mult = boolean variable associated with the
                     result;
              effect = effect associated with the result;
              update myPersonalEncode;

         CPEncoding nested = encoding.nested.
                                  get(t).get(result);
         CPVar var = nested.getCPVar();
         List historySon = new List();
         encodeVitalMaint(nested.getEncoding(),historySon);
         for every Expression se in historySon
              Expression pe = history.get(se.index);
              pe = pe + mult * (se + effect);



if(myPersonalEncode != null)
     history.add(myPersonalEncode);

if(cycleEnc != null)
     for every Expression exp in history
         exp = exp + cycleEnc;


sumOfDeterm <= 1;
sumOfNondet = 1;
return history;
```

Implementing the achieve-all constraint

To encode this constraint we take the subgoals in pairs from a list. We then check the nature of these goals: if both of them are vital goals, then we have to ensure that, if a choice is made in a deterministic branch point (this fact is expressed checking the values of the *sum* variables), this has to be the same for the two goals. If one is a vital-maint goal, then we have to ensure that the same choice is made also for non-deterministic transitions. It is important to remark that the constraint that imposes the selection of the same choice is triggered if and only if a choice has to be made to achieve a goal.

```
encAchieveAll(goalList, Encoding encoding)
     for every pair gi, gj in goalList, with j > i
         encodeAchieveAll(gi, gj, encoding);

encodeAchieveAll(Goal gi, Goal gj, Encoding encoding)
     if gi and gj are of type vital
         if(encoding.isDeterm() and (encoding is a choice-point))
              for every possible choice cpv in encoding
                    IntVar varGi = getVar(cpv, gi);
                    IntVar varGj = getVar(cpv, gj);
                    sumi = sumi + varGi;
                    sumj = sumj + varGj;
                    eq = makeAnd(eq,makeEq(varGi == varGj));

         makeIfThen(makeAnd(sumi = 1, sumj = 1) , eq);

         for every nested encoding ne in encoding
              encodeAchieveAll(gi, gj, ne);


     else if one is vital and the other is vital-maint
         for every possible choice cpv in encoding
              IntVar varGV = getVar(cpv, gVital);
              IntVar varGVM = getVar(cpv, gVitalMaint);
              sum = sum + varGV;
              eq = makeAnd(eq, makeEq(varGV == varGVM));

         makeIfThen(sum = 1, eq);
         for every nested encoding ne in encoding
              encodeAchieveAll(gi, gj, ne);
```

Implementing the before-then constraint

The code for this constraint is very similar to the one for the achieve-all goal. In this case the operator is binary, so we have to check only two goals, g_1 and g_2; for this goal all the choices made along the path in order to achieving g_1, have to be the same to the ones made for achieving g_2, even if they are related to non-deterministic actions. In this way we express the fact that there is a temporal sequence to be followed trying to achieve the two goals.

```
encodeBeforeThen(Goal g1, Goal g2, Encoding encoding)
      for every possible choice cpv in encoding
            IntVar varG1 = getVar(cpv, g1);
            IntVar varG2 = getVar(cpv, g2);
            sum = sum + varG1;
            makeEq(varG1, varG2);

      makeIfThen(sum = 1, eq);
      for every nested encoding ne in encoding
            encodeBeforeThen(g1, g2, ne);
```

6.2 Examples of encoding

6.2.1 Supply chain

Let us consider a supply chain example described as follows. A set of assemblers and element-vendors put together their business, giving users the possibility to book online a computer. We consider the simple business process snippet adapted from [27]. We suppose that assemblers can always perform their task, while vendors could have some problems giving their services, for example they could run out of particular elements, or they could be overwhelmed by the current jobs and not being able to satisfy any other request.

The domain shown in Figure 6.2.1 is defined as follows: there are four states; the transition *t1* stands for the action related to contact the assembler, while *t2* and *t3* represent the possible alternatives for the element-vendors (say *t2* for AMD and *t3* for Intel); *t4* stands for concluding the transaction, for simplicity we suppose no exceptional results for this transition.

An arbitrary user wants to buy a computer and he does not want to have total cost more than 1000 euro. Following the definitions given in Section 3.2, he expresses this request as follows:

achieve-all
  vital-maint $\mathit{price} \;\leq\; 1000$
  vital $\mathit{concluded} \;=\; \mathit{true}$

In this case we suppose that in the domain the variable named “price” identifies the total price and that the variable “concluded”, if true, identifies the fact that the transaction has been concluded successfully. With the first request the user says that he wants the price to be maintained always less or equal to 1000 euro: this means that, if for a particular state the price goes beyond this threshold and then goes down 1000 again (effect given by a discount, for example), this request is not satisfied.

Encoding the domain

The result of the encoding, starting from state s_1, is represented by the expression $\beta_1(a1 + \beta_2(\xi_1(a3 + \beta_4(a4)) + \xi_2(f3)) + \beta_3(\xi_3(a2 + \beta_5(a4)) + \xi_4(f2)) + n_1(\xi_2)(a1 + f3) + n_2(\xi_4)(a1 + f2))$, where a_i stands for the effects of corresponding actions.

The expressions identified by n_i represent the two cycles that are present in the domain. Expressions $\xi_1 + \xi_2 = 1$, $\xi_3 + \xi_4 = 1$ and $\beta_2 + \beta_3 \leq 1$ are branching point constraints and define that only one branch of the process can be executed.

For the other states we proceed in the same way: for state s_2 we get $\beta_2(\xi_1(a3+\beta_4(a4)) + \xi_2(f3)) + \beta_3(\xi_3(a2 + \beta_5(a4)) + \xi_4(f2))$; for state s_3 we get $\beta_4(a4)$, and for its clone, created by the converging point, we get $\beta_5(a4)$, and finally, for state s_4 we get an empty encoding.

Encoding the request

Let us now take into account the user request. It is made of an achieve-all goal containing a vital and a vital-maint goal. For simplicity we say that the a_i stand for the result of the action that touches the variable mentioned in the goal. To differentiate the boolean variables encoded in different goals, we put a "v" (for vital) and a "vm" (for vital-maint) at the foot of their names. For the vital goal, supposing that the a_i stand for the modifications that involve the variable *AMD-selected*, the expression becomes: $\beta_{1v}(a1 + \beta_{2v}(\xi_{1v}(a3 + \beta_{4v}(a4)) + \xi_{2v}(f3)) + \beta_{3v}(\xi_{3v}(a2 + \beta_{5v}(a4)) + \xi_{4v}(f2)) + n_1(\xi_{2v})(a1 + f3) + n_2(\xi_{4v})(a1 + f2)) = 1$. We get the constraints that assure the normal path is followed: $\xi_{2v} = 0$, $\xi_{4v} = 0$. The vital-maint involves the encodings of all the states in the path, so we get a list of constraints, where the a_i stand for the modifications to the variable *price*; since the actions related to the faulty transitions (the ones controlled by ξ_2 and ξ_4) in this example do not modify this variable at all, and no sub-encodings are present for them, they are not inserted in the expressions:

- $0.0 + \beta_{1vm} * (\beta_{2vm} * (\xi_{1vm} * (\beta_{4vm} * (a4) + a3)) + \beta_{3vm} * (\xi_{3vm} * (\beta_{5vm} * (a4) + a2)) + a1) + n_1 * (\xi_{2vm} * (a1)) + n_2 * (\xi_{4vm} * (a1)) <= 1000.0$

- $0.0 + \beta_{1vm} * (\beta_{2vm} * (\xi_{1vm} * (a3)) + \beta_{3vm} * (\xi_{3vm} * (a2)) + a1) + n_1 * (\xi_{2vm} * (a1)) + n_2 * (\xi_{4vm} * (a1)) <= 1000.0$

- $0.0 + \beta_{1vm} * (a1) + n_1 * (\xi_{2vm} * (a1)) + n_2 * (\xi_{4vm} * (a1)) <= 1000.0$

- $0.0 <= 1000.0$

We get the constraints on the nature of the path: $\xi_{2vm} = 0$, $\xi_{4vm} = 0$. For the encoding of the achieve-all goal, we get the constraints that make the solutions for the single goals follow the same path. A vital-maint goal is present, so the choices for the variables must be the same even for the non controlled variables. We get:

- $if(\beta_{1v} = 1)then(\beta_{1v} == \beta_{1vm})$
- $if(\xi_{1v} + \xi_{2v} = 1)then((\xi_{1v} = \xi_{1vm})and(\xi_{2v} = \xi_{2vm}))$
- $if(\xi_{3v} + \xi_{4v} = 1)then((\xi_{3v} = \xi_{3vm})and(\xi_{4v} = \xi_{4vm}))$
- $if(\beta_{2v} + \beta_{3v} = 1)then((\beta_{2v} = \beta_{2vm})and(\beta_{3v} = \beta_{3vm}))$
- $if(\beta_{4v} = 1)then(\beta_{4vm} = \beta_{4v})$
- $if(\beta_{5v} = 1)then(\beta_{5vm} = \beta_{5v})$

In this case the constraints containing the ξ variables must always be true, since, encoding the domain, we imposed exactly the expression in the *"if"* statement.

Solution to the constraint problem

Suppose that the cost of the assembling process (a_1) is to increase the *price* value of 15 units; let us assume also that the cost of the computer given by $a2$ is of 800 units and that the cost of the one given by $a3$ is of 700. The shipping cost, given by $a4$, is 250 units. Under these conditions only one solution exists. the vital goal imposes that the state s_4 has to be reached. After having followed the first transition (that has only one possible result), we have to make a choice; there is only one transition that respects the conditions given by the vital-maint basic goal: it is the one controlled by the β_{2vm} variable; the normal result must be given, and this is expressed by the ξ_{1vm} variable. The last transition, following this path, is controlled by the β_{4vm} variable. Cycles must not be followed.

Here are the values for the variables controlled by the system; values held for both goals are shown only once:

- $n_2 = 0$
- $n_1 = 0$
- $\beta_1 = 1$
- $\beta_2 = 1$
- $\beta_3 = 0$
- $\beta_5 = 0$
- $\beta_4 = 1$

Execution based on these values reach the final state, therefore the vital goal is satisfied. The values for the variable *price* along the execution are the following: $\{0, 15, 715, 975\}$, so the vital-maint goal is satisfied as well.

6.2.2 Purchase order

Let us consider another example regarding a purchase order scenario in which there are five actors: an user, two different sellers, a shipping service and a banking service. The representation of this domain as a graph can be seen in Figure 6.4.

The two types of sellers considered in the business domain (A and B) offer similar goods; they both refer to an external service for shipment. The user can review the process results if he does not like something, or he can confirm the order before it is passed to the bank for processing the transaction. At this point something can go wrong (if the transaction fails for some reason), and the user is asked to proceed with a new confirmation: this is an example of non-determinism. When the transaction successfully finishes, the final state is reached and the process terminates.

Business process is described in terms of state transition system in self-describing XML format:

```
<business-process name="TestCase Domain">
  <states>
    <state name = "s1"/>
    <state name = "s2"/>
    <state name = "s3"/>
    <state name = "s4"/>
    <state name = "s5"/>
    <state name = "s6"/>
    <state name = "s7"/>
  </states>
  <roles>
    <role name = "role"  interface = "role-interface"/>
  </roles>
  <variables>
    <variable name="price"         type="int"     value="0" />
    <variable name="totPrice"      type="int"     value="150" />
    <variable name="contactPrice"  type="int"     value="1" />
    <variable name="contactBPrice" type="int"     value="5" />
    <variable name="ASel"         type="boolean" value="false" />
    <variable name="ShipSel"      type="boolean" value="false" />
    <variable name="BSel"         type="boolean" value="false" />
    <variable name="TrConcluded"  type="boolean" value="false" />
  </variables>
```

Actions and action effects:

```
  <actions>
    <action name     = "contactA" role     = "role"
           activity = "invoke"   type     = "0">
      <effects>
        <effect operator="="  result="ASel" operand1="true" />
        <effect operator="+="  result="price" operand1="contactAPrice" />
      </effects>
    </action>
    <action name     = "contactB" role     = "role"
            activity = "invoke"   type     = "0">
      <effects>
        <effect operator="="   result="BSel" operand1="true" />
        <effect operator="+="  result="price" operand1="contactPrice" />
      </effects>
    </action>
    <action name     = "shippingNeeded" role     = "role"
            activity = "invoke"         type     = "0">
      <effects>
        <effect operator="="  result="ShipSel" operand1="true" />
        <effect operator="+="  result="price" operand1="contactPrice" />
      </effects>
    </action>
    <action name     = "prepareTerms" role     = "role"
            activity = "invoke"       type     = "0">
      <effects>
        <effect operator="+="  result="price" operand1="totPrice" />
      </effects>
    </action>
    <action name     = "revision" role     = "role"
            activity = "invoke"   type     = "0">
      <effects>
        <effect operator="="  result="review" operand1="true" />
      </effects>
    </action>
    <action name     = "accept" role     = "role"
            activity = "invoke" type     = "0">
      <effects>
        <effect operator="+="  result="price" operand1="contactPrice" />
      </effects>
    </action>
    <action name     = "transaction" role     = "role"
            activity = "invoke"      type     = "0">
      <effects>
      </effects>
      <fault name = "failed">
      </fault>
    </action>
  </actions>
```

Transitions representing the business process control flow:

```
<transitions>
  <transition name = "s1->s2: contactA"
    action      = "contactA"
    state-from  = "s1"
    state-to    = "s2" >
  </transition>
  <transition name = "s1->s3: contactB"
    action      = "contactB"
    state-from  = "s1"
    state-to    = "s3" >
  </transition>
  <transition name = "s2->s4: contactSh"
    action      = "shippingNeeded"
    state-from  = "s2"
    state-to    = "s4" >
  </transition>
  <transition name = "s3->s4: contactSh"
    action      = "shippingNeeded"
    state-from  = "s3"
    state-to    = "s4" >
  </transition>
  <transition name = "s4->s5: prepareTerms"
    action      = "prepareTerms"
    state-from  = "s4"
    state-to    = "s5" >
  </transition>
  <transition name    = "s5->s1: revise"
    action      = "revision"
    state-from  = "s5"
    state-to    = "s1" >
  </transition>
  <transition name    = "s5->s1: proceed"
    action      = "accept"
```

```
      state-from  = "s5"
      state-to    = "s6" >
    </transition>
    <transition name    = "s6->s7: finish"
      action      = "transaction"
      state-from  = "s6"
      state-to    = "s7" >
     <failed faultName = "transactionFailed" state = "s5" />
    </transition>
  </transitions>
</business-process>
```

A user may want the system to maintain the price below 155, being sure that the transaction is successful, but a seller of type A must be selected. This request is encoded as follows:

achieve-all
  vital-maint $price \leq 155$
  vital $ASel = true$
  vital $TrConcluded = true$

Encoding the domain

All transitions in the presented business domain are deterministic, apart from the ones starting from the state s6, that are all transitions representing the result of the transaction. This result is given by a set of factors (concurrent access, for example) that cannot be controlled directly by the system: this is the reason why these transitions are considered non-deterministic, while the others in the domain follow a sequential and predefined behavior. The fact that the two kinds of sellers rely upon the same service type for shipping, leads to the presence of a converging point, while the possibility for the user to review the results of the first part of the process creates a set of cycles: [s1,s2,s4,s5,s1] and [s1,s3,s4,s5,s1]. Going further, the faulty transition coming from the failure of the transaction creates again a set of cycles ([s5,s6,s5]). The encoding of the domain gives the following set of expressions:

- s1: $\beta_1(t_1+\beta_4(t_3+\beta_5(t_5+\beta_6(t_6)+\beta_7(t_7+\beta_8(\xi_2(t_8)+\xi_1(fault_1))))))+\beta_2(t_2+\beta_3(t_4+\beta_5(t_5+\beta_6(t_6)+\beta_7(t_7+\beta_8(\xi_2(t_8)+\xi_1(fault_1))))))+n_1(t_2+t_3+t_5+t_6)+n_2(t_1+t_4+t_5+t_6)$
- s2: $\beta_3(t_4+\beta_5(t_5+\beta_6(t_6)+\beta_7(t_7+\beta_8(\xi_2(t_8)+\xi_1(fault_1)))))$
- s3: $\beta_4(t_3+\beta_5(t_5+\beta_6(t_6)+\beta_7(t_7+\beta_8(\xi_2(t_8)+\xi_1(fault_1)))))$
- s4: $\beta_5(t_5+\beta_6(t_6)+\beta_7(t_7+\beta_8(\xi_2(t_8)+\xi_1(fault_1))))$
- s5: $\beta_6(t_6)+\beta_7(t_7+\beta_8(\xi_2(t_8)+\xi_1(fault_1)))+n_3(\xi_1)(t_7+fault_1)$
- s6: $\beta_8(\xi_2(t_8)+\xi_1(fault_1))$
- s7: $\emptyset$

Encoding the request

The encoding of the **achieve-all** encoding is composed of four steps: three for encoding the basic goals, and one for considering the relations between the domain variables. The results are presented as the application processes them, thus the indexes for the variables are expressed in a different way: the first expresses the state in which the variable is defined, the second expresses the index of the clone of the state (result of the encoding of a converging point), while the third index expresses different possibilities in a single decision point.

The encoding of the **vital-maint** goal is represented by the following list of constraints:

-
  ```
  0.0 + beta_s1_0_0 *
                 *(beta_s3_1_0 *
                        *(beta_s4_2_0 *
                               *(beta_s5_3_0 *
                                      *(beta_s6_4_0 *
                                             *(xi_s6_5_0 * (1.0)) +
                                          + 1.0) +
                                 + n_1*(xi_s6_5_1 *
  ```

```
                                    *( contactPrice[1] )) +
                       + 150.0) +
                + 1.0) +
          + 5.0) +
+ beta_s1_0_1 *
       *(beta_s2_9_0 *
              *(beta_s4_10_0 *
                     *(beta_s5_11_0 *
                            *(beta_s6_12_0 *
                                   *(xi_s6_13_0 * (1.0)) +
                              + 1.0) +
                       + n_3*(xi_s6_13_1 *
                                    *( contactPrice[1] )) +
                       + 150.0) +
                + 1.0) +
          + 1.0)
  + n_2*(( contactBPrice[5] +
              + contactPrice[1] +
              + totPrice[150] ))
  + n_4*(( contactPrice[1] +
              + contactPrice[1] +
              + totPrice[150] )) <= 155.0
```

```
● 0.0 + beta_s1_0_0 *
             *(beta_s3_1_0 *
                    *(beta_s4_2_0 *
                           *(beta_s5_3_0 * (1.0) +
                             + n_1*(xi_s6_5_1 *
                                          *( contactPrice[1] )) +
                             + 150.0)
                      + 1.0)
               + 5.0)
      + beta_s1_0_1 *
             *(beta_s2_9_0 *
                    *(beta_s4_10_0 *
                           *(beta_s5_11_0 * (1.0) +
                             + n_3*(xi_s6_13_1 *
                                          *( contactPrice[1] )) +
                             + 150.0)
```

```
            + 1.0)
         + 1.0)
     + n_2*(( contactBPrice[5] +
           + contactPrice[1] +
           + totPrice[150] ))
     + n_4*(( contactPrice[1] +
           + contactPrice[1] +
           + totPrice[150] )) <= 155.0
```

-
  ```
  0.0 + beta_s1_0_0 *
        *(beta_s3_1_0 *
           *(beta_s4_2_0 * (150.0)
            + 1.0)
         + 5.0)
    + beta_s1_0_1 *
        *(beta_s2_9_0 *
           *(beta_s4_10_0 * (150.0)
            + 1.0)
         + 1.0)
     + n_2*(( contactBPrice[5] +
           + contactPrice[1] +
           + totPrice[150] ))
     + n_4*(( contactPrice[1] +
           + contactPrice[1] +
           + totPrice[150] )) <= 155.0
  ```

-
  ```
  0.0 + beta_s1_0_0 *
        *(beta_s3_1_0 * (1.0)
         + 5.0)
    + beta_s1_0_1 *
        *(beta_s2_9_0 * (1.0)
         + 1.0)
    + n_2*(( contactBPrice[5] +
          + contactPrice[1] +
          + totPrice[150] ))
    + n_4*(( contactPrice[1] +
          + contactPrice[1] +
          + totPrice[150] )) <= 155.0
  ```

-
  ```
  0.0 + beta_s1_0_0 * (5.0)
  ```

```
        + beta_s1_0_1 * (1.0)
        + n_2*(( contactBPrice[5] +
                  + contactPrice[1] +
                  + totPrice[150] ))
        + n_4*(( contactPrice[1] +
                  + contactPrice[1] +
                  + totPrice[150] )) <= 155.0
```

- `0.0 <= 155.0`

The initial value for *price* is 0. Contacting a service costs one, let say, euro. A particular case is represented by the services of type B, for which the contact costs 5 euro. The total cost is given by 150 (the basic cost) plus five times the number of contacts to a service of type B, plus the number of contacts that the system performs against another type of service. In the encoding of this goal it is possible to see that there are four expressions for the cycles, while in the graph representation it is easy to see that there are only three cycles. This is not an error, but the result of the encoding of the converging point. In this case n_1, n_3 are the counters for the cycle(s) formed by the faulty transition in state s7, while n_2, n_4 are the counters for the cycles formed by the request of a revision.

The encoding of the **vital** $ASel = true$ goal gives the following constraint:

```
false | beta_s1_0_0 & (beta_s3_1_0 &
                         &(beta_s4_2_0 &
                              &(beta_s5_3_1 & false))) |
      | beta_s1_0_1 & (beta_s2_9_0 &
                         &(beta_s4_10_0 &
                              &(beta_s5_11_1 & false)) |
                      | true)
```

In this expression the only way to get the variable *ASel* being set to *true* is to follow the part controlled by the second β variable in state s_1: this variable, in fact, multiplies a subexpression that is put in an *or* relation with *true*.

The encoding of the goal **vital** $TrConcluded = true$ is expressed by the following constraint:

```
false | beta_s1_0_0 &
            &(beta_s3_1_0 &
                  &(beta_s4_2_0 &(beta_s5_3_0 &
                                    &(beta_s6_4_0 &
                                          &(xi_s6_5_0 & true))))) |
| beta_s1_0_1 &
           &(beta_s2_9_0 &
                 &(beta_s4_10_0 & (beta_s5_11_0 &
                                     &(beta_s6_12_0 &
                                          &(xi_s6_13_0 & true)))))
```

The variable *TrConcluded*, set to false at the beginning, is modified only by the last transition, if the result is normal. This point in the system is reachable following two different paths (contacting A or B), so the final expression is an *or* between these two.

It is noticeable how only the effects that modify the state of the variable defined in the goal are reported, while, if a nested expression modifies the state of that variable, all the betas and xis regarding the nested expressions are considered.

This is the list of constraints given by the encoding of the branching points. The converging point doubles the number of branching point after it, so there are five constraints. These constraints are inserted (posted) three times, one for every basic goal encoded: these conditions are related to the nature of the domain itself, and not on the nature of the requests of the user.

```
- (s1): beta_s1_0_0 + beta_s1_0_1 <= 1

- (s5): beta_s5_11_0 + beta_s5_11_1 <= 1

- (s5): beta_s5_3_0 + beta_s5_3_1 <= 1

- (s6): xi_s6_5_0 + xi_s6_5_1 = 1

- (s6): xi_s6_13_0 + xi_s6_13_1 = 1
```

The encoding of the **achieve-all** goal results in a list of implications. Also in this case the number of constraints is increased by the presence of the join point.

```
- if   (vital_1_beta_s1_0_0 + vital_1_beta_s1_0_1 == 1)
  then [ (vital_1_beta_s1_0_0 == vital-maint_0_beta_s1_0_0)
         and (vital_1_beta_s1_0_1 == vital-maint_0_beta_s1_0_1) ]

- if   (vital_1_beta_s3_1_0 == 1)
  then [ (vital_1_beta_s3_1_0 == vital-maint_0_beta_s3_1_0) ]

- if   (vital_1_beta_s4_2_0 == 1)
  then [ (vital_1_beta_s4_2_0 == vital-maint_0_beta_s4_2_0) ]

- if   (vital_1_beta_s5_3_0 + vital_1_beta_s5_3_1 == 1)
  then [ (vital_1_beta_s5_3_0 == vital-maint_0_beta_s5_3_0)
         and (vital_1_beta_s5_3_1 == vital-maint_0_beta_s5_3_1) ]

- if   (vital_1_beta_s6_4_0 == 1)
  then [ (vital_1_beta_s6_4_0 == vital-maint_0_beta_s6_4_0) ]

- if   (vital_1_xi_s6_5_0 + vital_1_xi_s6_5_1 == 1)
  then [ (vital_1_xi_s6_5_0 == vital-maint_0_xi_s6_5_0)
         and (vital_1_xi_s6_5_1 == vital-maint_0_xi_s6_5_1) ]

- if   (vital_1_beta_s2_9_0 == 1)
  then [ (vital_1_beta_s2_9_0 == vital-maint_0_beta_s2_9_0) ]

- if   (vital_1_beta_s4_10_0 == 1)
  then [ (vital_1_beta_s4_10_0 == vital-maint_0_beta_s4_10_0) ]

- if   (vital_1_beta_s5_11_0 + vital_1_beta_s5_11_1 == 1)
  then [ (vital_1_beta_s5_11_0 == vital-maint_0_beta_s5_11_0)
         and (vital_1_beta_s5_11_1 == vital-maint_0_beta_s5_11_1) ]

- if   (vital_1_beta_s6_12_0 == 1)
  then [ (vital_1_beta_s6_12_0 == vital-maint_0_beta_s6_12_0) ]

- if   (vital_1_xi_s6_13_0 + vital_1_xi_s6_13_1 == 1)
  then [ (vital_1_xi_s6_13_0 == vital-maint_0_xi_s6_13_0)
         and (vital_1_xi_s6_13_1 == vital-maint_0_xi_s6_13_1) ]
```

```
- if   (vital_2_beta_s1_0_0 + vital_2_beta_s1_0_1 == 1)
  then [ (vital_2_beta_s1_0_0 == vital-maint_0_beta_s1_0_0)
         and (vital_2_beta_s1_0_1 == vital-maint_0_beta_s1_0_1) ]

- if   (vital_2_beta_s3_1_0 == 1)
  then [ (vital_2_beta_s3_1_0 == vital-maint_0_beta_s3_1_0) ]

- if   (vital_2_beta_s4_2_0 == 1)
  then [ (vital_2_beta_s4_2_0 == vital-maint_0_beta_s4_2_0) ]

- if   (vital_2_beta_s5_3_0 + vital_2_beta_s5_3_1 == 1)
  then [ (vital_2_beta_s5_3_0 == vital-maint_0_beta_s5_3_0)
         and (vital_2_beta_s5_3_1 == vital-maint_0_beta_s5_3_1) ]

- if   (vital_2_beta_s6_4_0 == 1)
  then [ (vital_2_beta_s6_4_0 == vital-maint_0_beta_s6_4_0) ]

- if   (vital_2_xi_s6_5_0 + vital_2_xi_s6_5_1 == 1)
  then [ (vital_2_xi_s6_5_0 == vital-maint_0_xi_s6_5_0)
         and (vital_2_xi_s6_5_1 == vital-maint_0_xi_s6_5_1) ]

- if   (vital_2_beta_s2_9_0 == 1)
  then [ (vital_2_beta_s2_9_0 == vital-maint_0_beta_s2_9_0) ]

- if   (vital_2_beta_s4_10_0 == 1)
  then [ (vital_2_beta_s4_10_0 == vital-maint_0_beta_s4_10_0) ]

- if   (vital_2_beta_s5_11_0 + vital_2_beta_s5_11_1 == 1)
  then [ (vital_2_beta_s5_11_0 == vital-maint_0_beta_s5_11_0)
         and (vital_2_beta_s5_11_1 == vital-maint_0_beta_s5_11_1) ]

- if   (vital_2_beta_s6_12_0 == 1)
  then [ (vital_2_beta_s6_12_0 == vital-maint_0_beta_s6_12_0) ]

- if   (vital_2_xi_s6_13_0 + vital_2_xi_s6_13_1 == 1)
  then [ (vital_2_xi_s6_13_0 == vital-maint_0_xi_s6_13_0)
         and (vital_2_xi_s6_13_1 == vital-maint_0_xi_s6_13_1) ]

- if   [ (vital_1_beta_s1_0_0 + vital_1_beta_s1_0_1 == 1)
          and (vital_2_beta_s1_0_0 + vital_2_beta_s1_0_1 == 1) ]
  then [ (vital_1_beta_s1_0_0 = vital_2_beta_s1_0_0)
          and (vital_1_beta_s1_0_1 = vital_2_beta_s1_0_1) ]
```

```
- if   [ (vital_1_beta_s5_3_0 + vital_1_beta_s5_3_1 == 1)
         and (vital_2_beta_s5_3_0 + vital_2_beta_s5_3_1 == 1) ]
  then [ (vital_1_beta_s5_3_0 = vital_2_beta_s5_3_0)
         and (vital_1_beta_s5_3_1 = vital_2_beta_s5_3_1) ]

- if   [ (vital_1_beta_s5_11_0 + vital_1_beta_s5_11_1 == 1)
         and (vital_2_beta_s5_11_0 + vital_2_beta_s5_11_1 == 1) ]
  then [ (vital_1_beta_s5_11_0 = vital_2_beta_s5_11_0)
         and (vital_1_beta_s5_11_1 = vital_2_beta_s5_11_1) ]
```

The branching variables considered in different goals must have the same values, if a choice has been taken to solve these goals. For every decision point there is an expression saying what has been exposed in Chapter 3: the controlled decision points, implemented by β variables, are considered in any case, while the non controlled ones, implemented by ξ variables, are considered only when one of the two goals taken into account is a vital-maint one.

When all the constraints presented have been posted, the system is ready to invoke the choco engine to search for a solution.

Solution to the constraint problem

When the choco engine finds a solution, it is given in output by the application. The solution is composed of a set of values that have to be assigned to the controlled variables and to the variables counting the number of cycle visits. The application gives no value to the non-controlled variables, since this depends on external events, and is therefore determined at run-time. The solutions returned by the engine for the single goals are influenced also by the implications given by the encoding of the achieve-all goal. The solutions, as before, are presented as the application gives them in output. In particular, for the **vital-maint** $price \leq 155$ goal the system gives this list of values.

```
-  n_1 = 0

-  n_2 = 0
-  n_3 = 0

-  n_4 = 0

-  beta_s1_0_0 = 0

-  beta_s1_0_1 = 1

-  beta_s2_9_0 = 1

-  beta_s3_1_0 = 0

-  beta_s4_2_0 = 0

-  beta_s4_10_0 = 1

-  beta_s5_3_0 = 0

-  beta_s5_3_1 = 0

-  beta_s5_11_1 = 0

-  beta_s5_11_0 = 1

-  beta_s6_4_0 = 0

-  beta_s6_12_0 = 1
```

The system, as a result for the *vital TrConcluded* = *true* goal, gives the following list of values:

```
-  beta_s1_0_0 = 0

-  beta_s1_0_1 = 1

-  beta_s2_9_0 = 1

-  beta_s3_1_0 = 0

-  beta_s4_2_0 = 0

-  beta_s4_10_0 = 1

-  beta_s5_3_0 = 0

-  beta_s5_3_1 = 0
```

- `beta_s5_11_0 = 1`
- `beta_s5_11_1 = 0`
- `beta_s6_4_0 = 0`
- `beta_s6_12_0 = 1`

It is important to notice two facts: the solution does not contain any cycle variable; this is due to the fact that there is no cycle defined in the domain that affects the variable specified in the request. The second fact to underline is that the solution has the same values as the one for the vital-maint goal. This is guaranteed by the **achieve-all** semantic.

The application, solving the *vital ASel = true* goal, gives the following list of values:

- `beta_s1_0_0 = 0`
- `beta_s1_0_1 = 1`
- `beta_s2_9_0 = 0`
- `beta_s3_1_0 = 0`
- `beta_s4_2_0 = 0`
- `beta_s4_10_0 = 0`
- `beta_s5_3_0 = 0`
- `beta_s5_3_1 = 0`
- `beta_s5_11_1 = 0`
- `beta_s5_11_0 = 0`
- `beta_s6_4_0 = 0`
- `beta_s6_12_0 = 0`

A choice is made at the first step; no more choices have to be taken, because there is no other action that influences the value of the variable ASel, so the other values are set to zero.

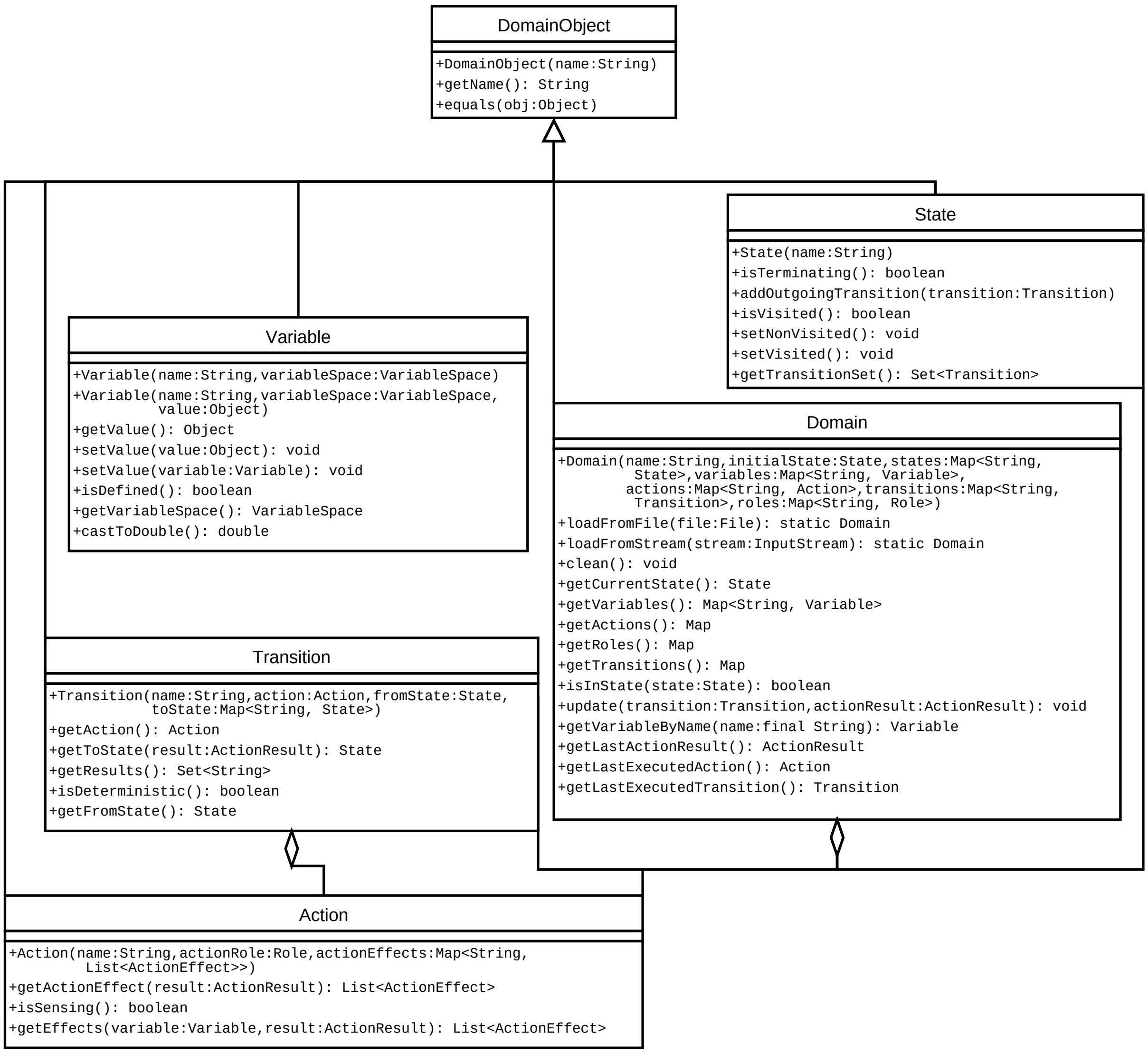

DomainObject
+DomainObject(name:String)
+getName(): String
+equals(obj:Object)
State
+State(name:String)
+isTerminating(): boolean
+addOutgoingTransition(transition:Transition)
+isVisited(): boolean
+setNonVisited(): void
+setVisited(): void
+getTransitionSet(): Set<Transition>
Variable
+Variable(name:String,variableSpace:VariableSpace)
+Variable(name:String,variableSpace:VariableSpace, value:Object)
+getValue(): Object
+setValue(value:Object): void
+setValue(variable:Variable): void
+isDefined(): boolean
+getVariableSpace(): VariableSpace
+castToDouble(): double
Domain
+Domain(name:String,initialState:State,states:Map<String, State>,variables:Map<String, Variable>, actions:Map<String, Action>,transitions:Map<String, Transition>,roles:Map<String, Role>)
+loadFromFile(file:File): static Domain
+loadFromStream(stream:InputStream): static Domain
+clean(): void
+getCurrentState(): State
+getVariables(): Map<String, Variable>
+getActions(): Map
+getRoles(): Map
+getTransitions(): Map
+isInState(state:State): boolean
+update(transition:Transition,actionResult:ActionResult): void
+getVariableByName(name:final String): Variable
+getLastActionResult(): ActionResult
+getLastExecutedAction(): Action
+getLastExecutedTransition(): Transition
Transition
+Transition(name:String,action:Action,fromState:State, toState:Map<String, State>)
+getAction(): Action
+getToState(result:ActionResult): State
+getResults(): Set<String>
+isDeterministic(): boolean
+getFromState(): State
Action
+Action(name:String,actionRole:Role,actionEffects:Map<String, List<ActionEffect>>)
+getActionEffect(result:ActionResult): List<ActionEffect>
+isSensing(): boolean
+getEffects(variable:Variable,result:ActionResult): List<ActionEffect>

Figure 6.1: UML class diagram of the org.xsrl.domain package.

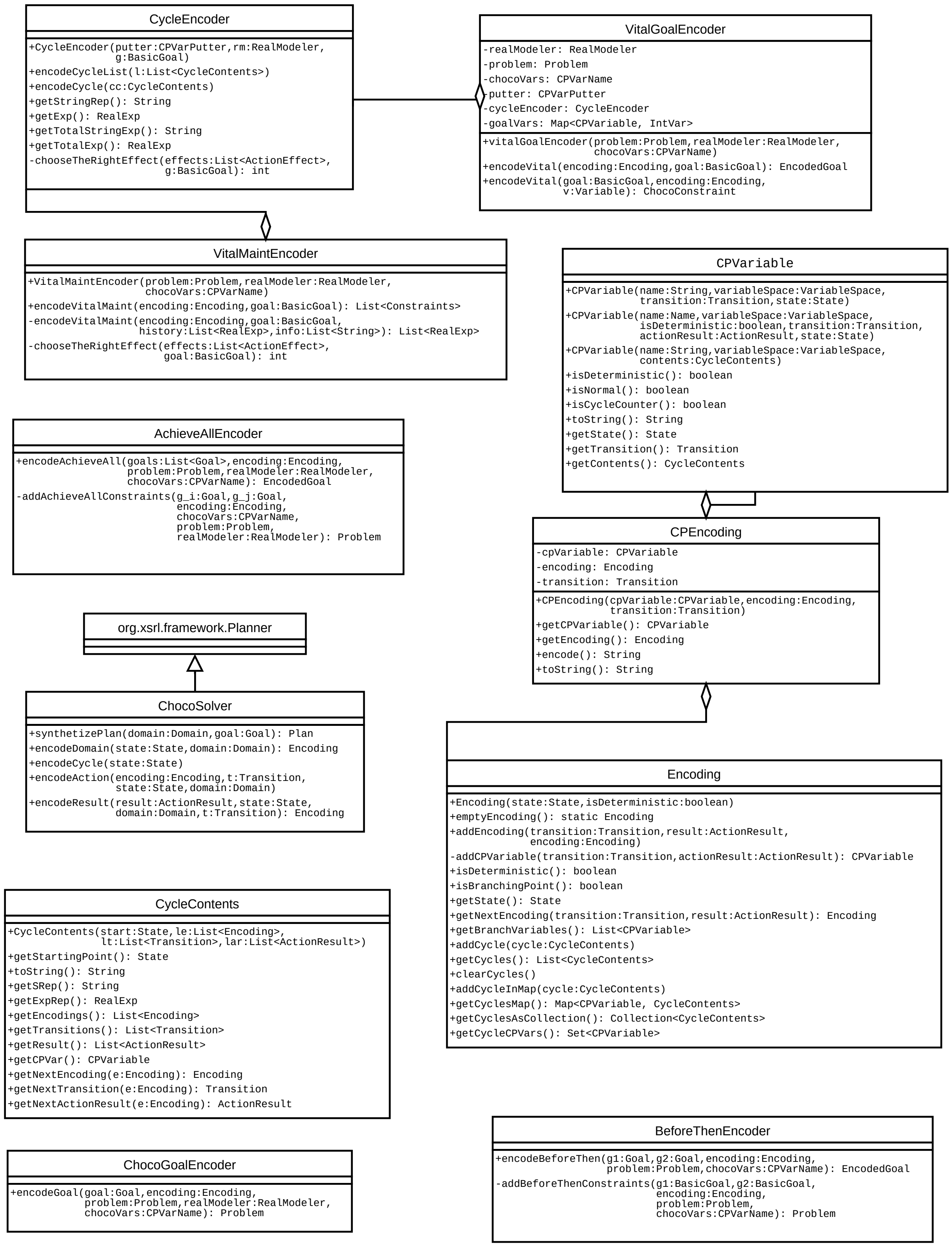

CycleEncoder
+CycleEncoder(putter:CPVarPutter,rm:RealModeler, g:BasicGoal)
+encodeCycleList(l:List<CycleContents>)
+encodeCycle(cc:CycleContents)
+getStringRep(): String
+getExp(): RealExp
+getTotalStringExp(): String
+getTotalExp(): RealExp
-chooseTheRightEffect(effects:List<ActionEffect>, g:BasicGoal): int
VitalGoalEncoder
-realModeler: RealModeler
-problem: Problem
-chocoVars: CPVarName
-putter: CPVarPutter
-cycleEncoder: CycleEncoder
-goalVars: Map<CPVariable, IntVar>
+vitalGoalEncoder(problem:Problem,realModeler:RealModeler, chocoVars:CPVarName)
+encodeVital(encoding:Encoding,goal:BasicGoal): EncodedGoal
+encodeVital(goal:BasicGoal,encoding:Encoding, v:Variable): ChocoConstraint
VitalMaintEncoder
+VitalMaintEncoder(problem:Problem,realModeler:RealModeler, chocoVars:CPVarName)
+encodeVitalMaint(encoding:Encoding,goal:BasicGoal): List<Constraints>
-encodeVitalMaint(encoding:Encoding,goal:BasicGoal, history:List<RealExp>,info:List<String>): List<RealExp>
-chooseTheRightEffect(effects:List<ActionEffect>, goal:BasicGoal): int
CPVariable
+CPVariable(name:String,variableSpace:VariableSpace, transition:Transition,state:State)
+CPVariable(name:Name,variableSpace:VariableSpace, isDeterministic:boolean,transition:Transition, actionResult:ActionResult,state:State)
+CPVariable(name:String,variableSpace:VariableSpace, contents:CycleContents)
+isDeterministic(): boolean
+isNormal(): boolean
+isCycleCounter(): boolean
+toString(): String
+getState(): State
+getTransition(): Transition
+getContents(): CycleContents
AchieveAllEncoder
+encodeAchieveAll(goals:List<Goal>,encoding:Encoding, problem:Problem,realModeler:RealModeler, chocoVars:CPVarName): EncodedGoal
-addAchieveAllConstraints(g_i:Goal,g_j:Goal, encoding:Encoding, chocoVars:CPVarName, problem:Problem, realModeler:RealModeler): Problem
CPEncoding
-cpVariable: CPVariable
-encoding: Encoding
-transition: Transition
+CPEncoding(cpVariable:CPVariable,encoding:Encoding, transition:Transition)
+getCPVariable(): CPVariable
+getEncoding(): Encoding
+encode(): String
+toString(): String
org.xsrl.framework.Planner
ChocoSolver
+synthetizePlan(domain:Domain,goal:Goal): Plan
+encodeDomain(state:State,domain:Domain): Encoding
+encodeCycle(state:State)
+encodeAction(encoding:Encoding,t:Transition, state:State,domain:Domain)
+encodeResult(result:ActionResult,state:State, domain:Domain,t:Transition): Encoding
Encoding
+Encoding(state:State,isDeterministic:boolean)
+emptyEncoding(): static Encoding
+addEncoding(transition:Transition,result:ActionResult, encoding:Encoding)
-addCPVariable(transition:Transition,actionResult:ActionResult): CPVariable
+isDeterministic(): boolean
+isBranchingPoint(): boolean
+getState(): State
+getNextEncoding(transition:Transition,result:ActionResult): Encoding
+getBranchVariables(): List<CPVariable>
+addCycle(cycle:CycleContents)
+getCycles(): List<CycleContents>
+clearCycles()
+addCycleInMap(cycle:CycleContents)
+getCyclesMap(): Map<CPVariable, CycleContents>
+getCyclesAsCollection(): Collection<CycleContents>
+getCycleCPVars(): Set<CPVariable>
CycleContents
+CycleContents(start:State,le:List<Encoding>, lt:List<Transition>,lar:List<ActionResult>)
+getStartingPoint(): State
+toString(): String
+getSRep(): String
+getExpRep(): RealExp
+getEncodings(): List<Encoding>
+getTransitions(): List<Transition>
+getResult(): List<ActionResult>
+getCPVar(): CPVariable
+getNextEncoding(e:Encoding): Encoding
+getNextTransition(e:Encoding): Transition
+getNextActionResult(e:Encoding): ActionResult
BeforeThenEncoder
+encodeBeforeThen(g1:Goal,g2:Goal,encoding:Encoding, problem:Problem,chocoVars:CPVarName): EncodedGoal
-addBeforeThenConstraints(g1:BasicGoal,g2:BasicGoal, encoding:Encoding, problem:Problem, chocoVars:CPVarName): Problem
ChocoGoalEncoder
+encodeGoal(goal:Goal,encoding:Encoding, problem:Problem,realModeler:RealModeler, chocoVars:CPVarName): Problem

Figure 6.2: UML class diagram of the org.xsrl.framework.cp package (only main classes).

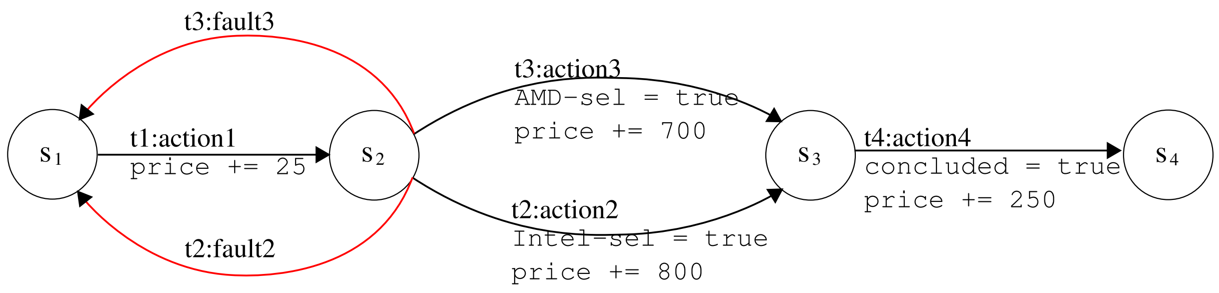


Figure 6.3: Graph representation of the example domain.

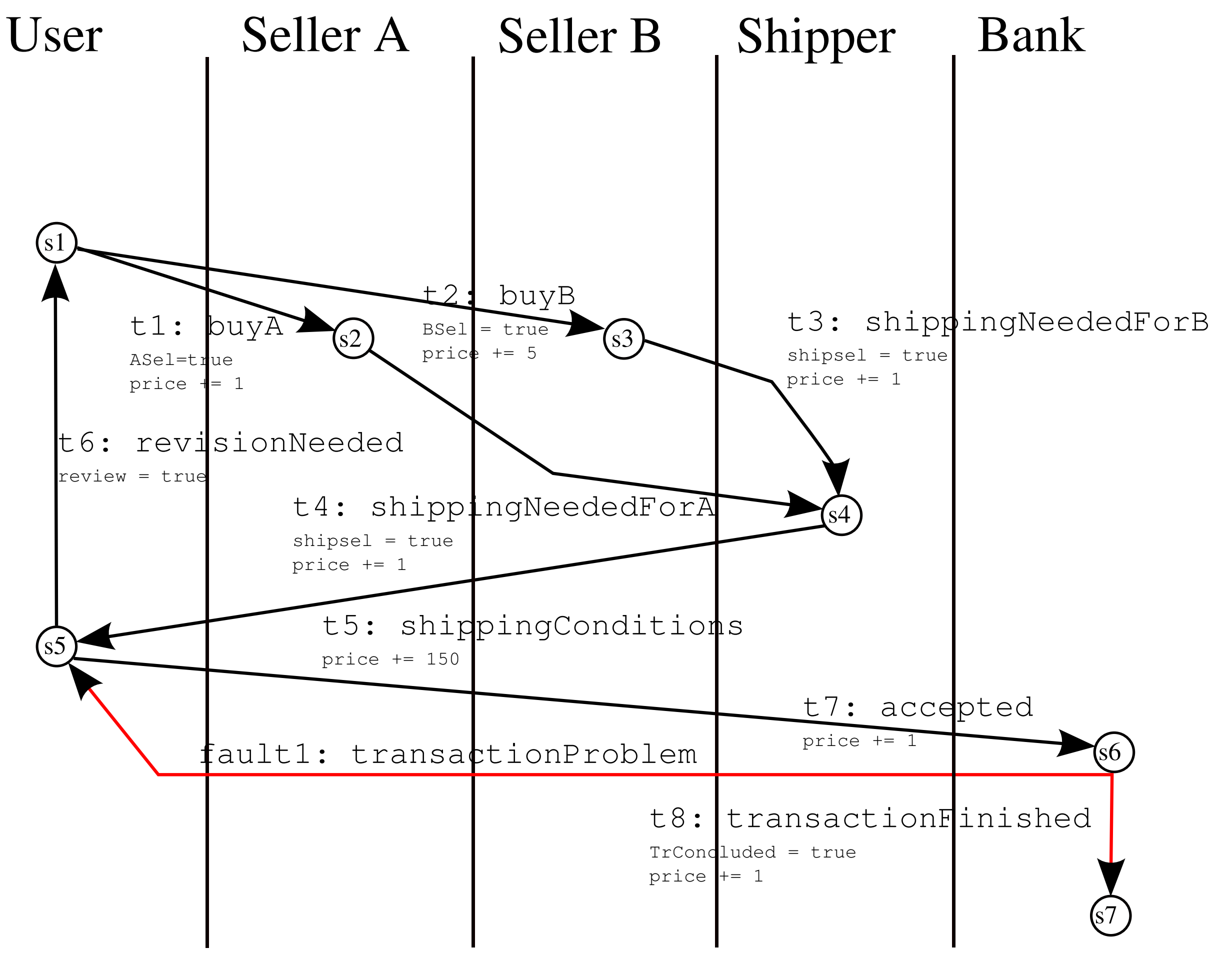


Figure 6.4: Graph representation of the test case.

Chapter 7

Conclusions

7.1 XSRL and its supporting framework

In the thesis we presented an approach for web service interaction based on planning and constraint satisfaction and a service request language (XSRL) developed on the basis of this framework. The planning framework was developed on the basis of a coherent view of the issues arising when planning requests against web services under uncertainty (as plans inevitably do not execute as expected) in dynamic environments where there is the constant need to be able to identify critical decision trade-offs, revise goals and evaluate alternative options. This approach deals with an uncertain and dynamic world such as that of web services a correspondence must be drawn between the actual business process environment and the planers model of it. It then instantiates plans on the basis of the plan model in terms of a user specific request and via interaction with the service registries; and when if necessary, dynamically reconfigures plans on the basis of user interaction. These design considerations are reflected at the level of the request language that generates plans over web services residing in an e-marketplace and its run-time environment.

We defined the full semantics of XSRL in terms of execution structures and we have provided algorithms that satisfy XSRL requests based on service registry supplied information and information gathered from web service interactions.

AI planning provides a sound framework for developing a web services request language and for synthesizing correct plans for it. Based on this premise, we developed a framework for planning and monitoring the execution of web service requests against standardized business processes. The requests taken in the XSRL language are processed by a framework which interleaves planning and execution in order to dynamically adapt to the opportunities offered by available web services and to the preferences of users. The supporting framework based in interleaving planning and execution has been implemented.

Having proposed a framework and having shown its feasibility does now close the problem. In fact, there is a wide space for far more investigations.

One of the most important open issues is to find a proper balance between off-line and on-line planning in the interleaved planning and execution framework. This has a big impact on the efficiency of the approach. In the developed framework, replanning is requested each time new information is acquired from the environment. Instead, it can be useful to produce plans that "know" how to react on the arrival of new information. Another efficiency issue is the enhancement of service registries with better support for provider selection, e.g., based on service quality characteristics or cached web services invocations.

An issue for future investigation is the interaction of the system with service registries. In particular, service registry could be enhanced by providing better support for provider selection, e.g., based on service quality characteristics. From the point of view of planning, there are several aspects that need to be addressed. For example, the current version of the planner does not keep track of previous computations or "remember" history and patterns of interactions.

Finally, all necessary extensions to standard business process languages (e.g., BPEL) and description languages (e.g., WSDL) have to be identified. The extension for BPEL is important because BPEL's main intent is to orchestrate the organizational workflows and it has lack of support of global interorganizational choreography issues in which we are interested in.

7.2 Use of constraint programming

We proposed an approach to planning interactions with web services based on a constraint programming encoding. The key characteristics of the encoding are its dealing with non-determinism, its being unbounded, its representing the possible executions on the domain including traversal properties. We provided algorithms for encoding state representations of the domain together with user requests given in an expressive goal language (XSRL). The proposed encoding is particularly suited to deal with web service marketplaces and gives the user the possibility to satisfy his desires. This is a major improvement with respect to the previous frameworks to deal with web service requests based, for instance, on a model based planner (discussed in Appendix B). In particular, with the proposed approach we deal with numeric values in place of considering boolean conditions coming from the satisfaction of an expression; we handle preference goals by introducing an optimization function; while keeping the desired properties of dealing with non-determinism, having primitives for execution properties (e.g., vital-maint goals) and having a framework to execute the requests. This is an initial proposal to use constraints to encode the satisfaction of a users request with respect to a set of autonomous web services.

The proposed framework is based on constraint programming. Such novel modeling takes full advantage of constraint programming systems, being able to build solutions incrementally, dealing with numeric values and preferences. A number of issues are open for further investigation. Most notably, we have not yet considered issues of efficiency of the proposed algorithms with respect to the minimality of the encoding or of the propagation complexity or execution time. Future research, from the constraint programming point of view, also demands a deeper evaluation of the "average" structure of the business graph and the resulting tightness of the related constraint problem; it is worth studying the "right" balance between expressive power and simplicity of use with respect to languages à la XSRL (e.g., think of web search engines, whose success depends largely on the simplicity of the query languages); one may want to "tune" the constraint modeling (e.g., via global constraints, soft constraints) and the available resolution algorithms to handle service choreographies, but also to be as efficient as possible. Encoding the problem into quantified constraints problems allow us to model non-deterministic actions in a more compact manner. However, to the best of our knowledge, in current constraint programming languages one cannot directly model quantified constraint problems.

7.3 Monitoring of service compositions

We introduced the assertion language XSAL for expressing business rules in the form of assertions over business processes. XSAL is deployable using the framework we propose which is capable of automatically associating business rules with relevant processes involved in a user request. This allows for consistency and conformance to organizational rules and policies when executing a business process. Additionally, it offers runtime control over its execution. We have classified assertions with respect to two process characteristics: operational context and ownership. With respect to the operational context, we distinguish between simple, preservation and business entity assertions. Regarding ownership, we distinguish between business process, role and provider level assertions. We then introduced a framework for planning user requests that comply with assertions and monitoring their execution to recover from violating conditions. Specialized algorithms for planning, monitoring and executing requests and assertions have been proposed for this framework. Finally, we have shown how XSAL has the expressive power to define both functional and non-functional properties in the assertions.

The proposed framework and the XSAL language open interesting research issues. One involves the performance of the framework, in particular, the way providers are selected from the service registry is crucial for the efficiency and effectiveness of the architecture. The current proposal does not address this issue, in other words, providers are chosen randomly. A better solution is to select providers based on provider-level assertions (for instance by comparing active assertions), on reputation and history of previous interactions with the provider, or optimizing some specific QoS parameter (e.g., cost of the service or average latency of the service).

The proposed framework plans for requests and assertions, then monitors the execution of the plans. If there is one possible execution path that can satisfy the request and comply with its associated assertions, this will be found and executed, if not, a failure will be returned. In case that a request succeeds no information is currently provided regarding the quality of the execution. That is, if more possible execution paths complying with the assertions and the user request exist, then only one is guaranteed to be taken. An open issue concerns the comparison of potential solutions (execution trajectories) against optimality metrics, e.g., the shortest plan, the cheapest, the fastest or any other optimality criteria. Solutions to the latter concerns could be addressed resorting to different planning techniques to handle assertions.

Bibliography

[1] M. Aiello A. Lazovik and R. Gennari. Choreographies: using constraints to satisfy service requests. In *IEEE Web Services-based Systems and Applications (WEBSA at ICIW)*, 2006.

[2] M. Aiello and A. Lazovik. Associating assertions with business processes and monitoring their execution. *International Journal of Cooperative Information Systems*, 2006.

[3] M. Aiello, M. Papazoglou, J. Yang, M. Carman, M. Pistore, L. Serafini, and P. Traverso. A request language for web-services based on planning and constraint satisfaction. In *VLDB Workshop on Technologies for E-Services (TES02)*, 2002.

[4] Krzysztof R. Apt. *Principles of Constraint Programming*. Cambdrige U. Press, 2003.

[5] Krzysztof R. Apt, Francesca Rossi, and K. Brent Venable. Cp-nets and nash equilibria. In *Third International Conference on Computational Intelligence, Robotics and Autonomous Systems (CIRAS '05)*. ACM, 2005.

[6] Danilo Ardagna and Barbara Pernici. Global and local qos constraints guarantee in web service selection. In *ICWS*, pages 805–806, 2005.

[7] Bob Atkinson, Giovanni Della-Libera, Satoshi Hada, Maryann Hondo, Phillip Hallam-Baker, Johannes Klein, Brian LaMacchia, Paul Leach,

John Manferdelli, Hiroshi Maruyama, Anthony Nadalin, Nataraj Nagaratnam, Hemma Prafullchandra, John Shewchuk, and Dan Simon. *Web Services Security*. Microsoft, IBM, VeriSign, 1.0 edition, April 2002. available via `http://www-128.ibm.com/developerworks/webservices/library/ws-secure/` on 25/10/2005.

[8] T.-C. Au, U. Kuter, and D. Nau. Web service composition with volatile information. In *ISWC-05*, 2005.

[9] Fahiem Bacchus, Craig Boutilier, and Adam J. Grove. Rewarding behaviors. In *AAAI/IAAI, Vol. 2*, pages 1160–1167, 1996.

[10] Fahiem Bacchus, Craig Boutilier, and Adam J. Grove. Structured solution methods for non-markovian decision processes. In *AAAI/IAAI*, pages 112–117, 1997.

[11] Roman Bartak. Constraint programming: In pursuit of the holy grail. Constraint Programming Survey.

[12] F. Bascchus and F. Kabanza. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence*, 2000.

[13] D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Mecella. Automatic composition of e-services that export their behavior. In *Int. Conference on Service Oriented Computing (ICSOC-03)*, 2003.

[14] D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Mecella. Synthesis of underspecified composite e-services based on automated reasoning. In *Int. Conference on Service Oriented Computing (ICSOC-04)*, 2004.

[15] D. Bernard, E. Gamble, N. Rouquette, B. Smith, Y. Tung, N. Muscettola, G. Dorias, B. Kanefsky, J. Kurien, W. Millar, P. Nayak, and K. Rajan.

Remote agent experiment ds1 technology validation report. Technical report, NASA Ames and JPL report, 1998.

[16] P. Bertoli, A. Cimatti, M. Pistore, M. Roveri, and P. Traverso. MBP: A Model Based Planner. In *Proc. IJCAI'01 Workshop on Planning under Uncertainty and Incomplete Information*, 2001.

[17] P. Bertoli, A. Cimatti, and P. Traverso. Interleaving Execution and Planning via Symbolic Model Checking. In *Proc. of ICAPS'03 Workshop on Planning under Uncertainty and Incomplete Information*, 2003.

[18] S. Bistarelli, U. Montanari, and F. Rossi. Semiring-based constraint solving and optimization. *Journal of ACM*, 44(2):201–236, 1997.

[19] Stefano Bistarelli, Rosella Gennari, and Francesca Rossi. General properties and termination conditions for soft constraint propagation. *Constraints*, 8(1):79–97, 2003.

[20] Avrim Blum and Merrick L. Furst. Fast planning through planning graph analysis. *Artif. Intell.*, 90(1-2):281–300, 1997.

[21] C. Boutilier, R. Brafman, C. Domshlak, H. Hoos, and D. Poole. Preference-based constraint optimization with cp-nets. *Computational Intelligence*, 20:137–157, 2004.

[22] Craig Boutilier, Thomas Dean, and Steve Hanks. Decision-theoretic planning: Structural assumptions and computational leverage. *J. Artif. Intell. Res. (JAIR)*, 11:1–94, 1999.

[23] BPEL4WS. *Business Process Execution Language for Web Services*, May 2003. `http://www-106.ibm.com/developerworks/library/ws-bpel/`.

[24] R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[25] C. Bussler, D. Fensel, and A. Maedche. A conceptual architecture for semantic web enabled web services. *SIGMOD Record*, 31(4):24–29, 2002.

[26] Tom Bylander. The computational complexity of propositional strips planning. *Artif. Intell.*, 69(1-2):165–204, 1994.

[27] Igor Cappello. Encoding web service requests as constraints: an implementation. Master's thesis, University of Trento, 2006.

[28] Fabio Casati, Stefano Ceri, Barbara Pernici, and Giuseppe Pozzi. Conceptual modeling of workflows. In *Advances in Object-Oriented Data Modeling*, pages 281–306, 2000.

[29] Fabio Casati, Ski Ilnicki, LiJie Jin, Vasudev Krishnamoorthy, and Ming-Chien Shan. Adaptive and dynamic service composition in eFlow. In *Conference on Advanced Information Systems Engineering*, pages 13–31, 2000.

[30] Fabio Casati, Mehmet Sayal, and Ming-Chien Shan. Developing e-services for composing e-services. In *Proceedings of 13th Int. Conf. on Advanced Information Systems Engineering (CAiSE)*, 2001.

[31] L. Chen, N.R. Shadbolt, C. Goble, F. Tao, S.J. Cox, C. Puleston, and P. Smart. Towards a knowledge-based approach to semantic service composition. In *2nd Int. Semantic Web Conf. (ISWC2003)*, Lecture Notes in Computer Sciences 2870, pages 319–334. Springer, 2003.

[32] Choco. Constraint programming system. `http://choco.sourceforge.net/`.

[33] E. Clarke and J. Wing. Formal methods: State of the art and future directions, 1996.

[34] E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[35] The DAML Services Coalition. OWL-S 1.0. White Paper, 2003.

[36] Enzo Colombo, Chiara Francalanci, and Barbara Pernici. Modeling business transactions as e-services. In *SEBD*, pages 179–190, 2003.

[37] CORBA. Common object request broker architecture. `www.corba.org`.

[38] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed systems. Concepts and design*. Addison-Wesley, 2001.

[39] F. Curbera, R. Khalaf, N. Mukhi, S. Tai, and S. Weerawarana. The next step in web services. *Commun. ACM*, 46(10):29–34, 2003.

[40] P. Doherty and J. Kvarnstrom. Talplanner: a temporal logic based planner. *AI Magazine*, 22(3):95–102, 2001.

[41] Prashant Doshi, Richard Goodwin, Rama Akkiraju, and Kunal Verma. Dynamic workflow composition: Using markov decision processes. *Int. J. Web Service Res.*, 2(1):1–17, 2005.

[42] S. Dustdar. Web services workflows - composition, coordination, and transactions in service-oriented computing. *Concurrent Engineering: Research and Applications*, pages 237–246, 2004.

[43] S. Dustdar and M. Aiello. Service oriented computing: Service foundations. *Service Oriented Computing, Internationales Begegnungs- und Forschungszentrum (IBFI)*, 05462, 2006.

[44] S. Edelkamp and M. Helmert. *The implementation of Mips*.

[45] E. A. Emerson. Temporal and modal logic. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science: Volume B, Formal Models and Semantics*, pages 995–1072. North-Holland Publishing Company, Amsterdam, The Netherlands, 1990.

[46] Boi Faltings and Santiago Macho-Gonzalez. Open constraint programming. *Artif. Intell.*, 161(1-2):181–208, 2005.

[47] D. Fensel, H. Lausen, A. Polleres, J. de Bruijn, M. Stollberg, D. Roman, and J. Domingue. *Enabling Semantic Web Services: The Web Service Modeling Ontology*. Springer, 2007.

[48] R. Fikes and N. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3–4):189–208, 1971.

[49] I. Fikouras and E. Freiter. Service discovery and orchestration for distributed service repositories. In *Conf. on Service-Oriented Computing (ICSOC-03)*, Lecture Notes in Computer Sciences, pages 59–74. Springer, 2003.

[50] G. Frankova, D. Malfatti, and M. Aiello. Semantics and extensions of ws-agreement. *Journal of Software*, 1(1):23–31, 2006.

[51] S. French. *Sequencing and scheduling: an introduction to the mathematics of the Job Shop*. Horwood, 1982.

[52] X. Fu, T. Bultan, and J. Su. Analysis of interacting bpel web services. In *13th International Conference on World Wide Web (WWW '04)*, pages 621–630. ACM Press, 2004.

[53] A. Gao, D. Yang, S. Tang, and M. Zhang. Web service composition using markov decision processes. In *Advances in web-age information management*, 2005.

[54] J. Gaschnig. Performance measurement and analysis of certain search algorithms. Technical Report CMU-CS-79-124, Carnegie-Mellon University, 1979.

[55] David Geer. Taking steps to secure web services. *COMP*, 36(10):14–16, October 2003.

[56] G. De Giacomo, Y. Lesperance, and H. Levesque. Congolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence*, 1–2(121):109–169, 2000.

[57] F. Glover and M. Laguna. Tabu search. In *Modern Heuristics for Combinatorial Problems*. Blackwell Scientific Publishing, Oxford, 1993.

[58] Benjamin N. Grosof. Representing e-commerce rules via situated courteous logic programs in ruleml*1. *Electronic Commerce: Research and Applications*, 3(1):2–20, 2004.

[59] Y. Hamadi, A.M. Frisch, and I. Miguel. An overview of the gridline project. In *Planning and Scheduling for Web and Grid Services - ICAPS 2004*, 2004.

[60] R. Haralick and G. Elliot. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–314, 1980.

[61] P. Harmon. Analyzing activities. *Business Process Trends*, 1(4), 2003.

[62] I. Horrocks, U. Sattler, and S. Tobies. Practical reasoning for expressive description logics. In *LPAR-99*, pages 161–180. Springer Verlag, 1999.

[63] James B. D. Joshi, Walid G. Aref, Arif Ghafoor, and Eugene H. Spafford. Security models for web-based applications. *Commun. ACM*, 44(2):38–44, 2001.

[64] A. Grode K. Geihs, R. Kalcklosch. Single sign-on in service-oriented computing. In *Conf. on Service-Oriented Computing (ICSOC-03)*, Lecture Notes in Computer Sciences, pages 384–394. Springer, 2003.

[65] Henry A. Kautz, David A. McAllester, and Bart Selman. Encoding plans in propositional logic. In *KR*, pages 374–384, 1996.

[66] Henry A. Kautz and Bart Selman. Planning as satisfiability. In *ECAI*, pages 359–363, 1992.

[67] Kavantzas. *Web Services Choreography Description Language 1.0*, April 2004. `http://lists.w3.org/Archives/Public/www-archive/2004Apr/att-0004/cdl_v1-editors-apr03-2004-pdf.pdf`.

[68] L. Kavraki. Algorithms in robotics: The motion planning perspective. *Frontiers of Engineering Publication*, pages 90–93, 1999.

[69] C. A. Knoblock, S. Minton, J. L. Ambite, M. Muslea, J. Oh, , and M. Frank. Mixed-initiative, multi-source information assistants. In *Proceedings of the World Wide Web Conference*, 2001.

[70] V. Kumar. Algorithms for constraint satisfaction problems: A survey. *AI Magazine*, 13(1):32–44, 1992.

[71] U. Kuter, E. Sirin, D. Nau, B. Parsia, and J. Hendler. Information gathering during planning for web service composition. In *Third International Semantic Web Conference (ISWC2004)*, 2004.

[72] U. Kuter, E. Sirin, D. Nau, B. Parsia, and J. Hendler. Information gathering during planning for web service composition. *Journal of Web Semantics*, 3(2–3):183–205, 2005.

[73] U. Dal Lago, M. Pistore, and P. Traverso. Planning with a language for extended goals. In *18^{th} National Conference on Artificial Intelligence (AAAI-02)*, 2002.

[74] R. Lara, H. Lausen, S. Arroyo, J. de Bruijn, and D. Fensel. Semantic web services: description requirements and current technologies. In *International Workshop on Electronic Commerce, Agents, and Semantic Web Services*, 2003.

[75] J. Latombe. Motion planning: A jorney of robots, molecules, digital actors, and other artifacts. *International Journal of Robotics Research*, 18(11):1119–1128, 1999.

[76] E. Lawler, J. Lenstra, A. Kan, and D.Shmoys. Sequencing and scheduling: algorithms and complexity. *Logistics of Production and Inventory, Handbooks in Operations Research and Management Schience Volume 4*, pages 445–552, 1993.

[77] A. Lazovik. Monitoring of document-oriented business processes. In *1st European Young Researchers Workshop on Service Oriented Computing (YRSOC-05)*, 2005.

[78] A. Lazovik. Xsrl reference implementation. `www.dit.unitn.it/~lazovik/xsrl`, 2006.

[79] A. Lazovik, M. Aiello, and R. Gennari. Encoding requests to web service compositions as constraints. In *Principles and Practice of Constraint Programming (CP-05)*, LNCS 3709, pages 782–786. Springer, 2005.

[80] A. Lazovik, M. Aiello, and R. Gennari. Encoding requests to web service compositions as constraints. Technical Report DIT-05-40, Univ. of Trento, 2005.

[81] A. Lazovik, M. Aiello, and M. Papazoglou. Planning and monitoring the execution of web service requests. In *Conf. on Service-Oriented Computing (ICSOC-03)*, Lecture Notes in Computer Sciences 2910, pages 335–350. Springer, 2003.

[82] A. Lazovik, M. Aiello, and M. Papazoglou. Planning and monitoring the execution of web service requests. Technical Report DIT-03-049, University of Trento, 2003.

[83] A. Lazovik, M. Aiello, and M. Papazoglou. Associating assertions with business processes and monitoring their execution. In M. Aiello, M. Aoyama, F. Curbera, and M. Papazoglou, editors, *Conf. on Service-Oriented Computing (ICSOC-04)*, pages 94–104. ACM Press, 2004.

[84] A. Lazovik, M. Aiello, and M. Papazoglou. Planning and monitoring the execution of web service requests. *Journal on Digital Libraries*, 2005.

[85] F. Leymann, D. Roller, and M.-T. Schmidt. Web services and business process management. *IBM Systems Journal*, 41(2):198–211, 2002.

[86] Mark Little. Transactions and web services. *Commun. ACM*, 46(10):49–54, 2003.

[87] Michael L. Littman. Probabilistic propositional planning: Representations and complexity. In *AAAI/IAAI*, pages 748–754, 1997.

[88] Heiko Ludwig, Asit Dan, and Robert Kearney. Cremona: An architecture and library for creation and monitoring of ws-agreements. In *ICSOC*, pages 65–74, 2004.

[89] Heiko Ludwig, Alexander Keller, Asit Dan, and Richard P. King. A service level agreement language for dynamic electronic services. In *WECWIS*, pages 25–32, 2002.

[90] Heiko Ludwig, Toshiyuki Nakata, Philipp Wieder, Oliver Wldrich, and Wolfgang Ziegler. Reliable orchestration of resources using ws-agreement. Technical Report TR-0050, Institute on Grid Systems, Tools and Environments, CoreGRID - Network of Excellence, July 2006.

[91] P. Obreiter M. Klein, B. Konig-Ries. Stepwise refinable service descriptions: Adapting daml-s to staged service trading. In *Conf. on Service-Oriented Computing (ICSOC-03)*, Lecture Notes in Computer Sciences, pages 178–193. Springer, 2003.

[92] A. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.

[93] A. Mani and A. Nagarayan. Understanding quality of service for web services. `http://www-128.ibm.com/developerworks/webservices/library/ws-quality.html`, 2002.

[94] D. McDermott. Estimated-regression planning for interactions with Web Services. In *6th Int. Conf. on AI Planning and Scheduling*. AAAI Press, 2002.

[95] S. McIlraith and T. C. Son. Adapting Golog for composition of semantic web-services. In D. Fensel, F. Giunchiglia, D. McGuinness, and M. Williams, editors, *Conf. on principles of Knowledge Representation (KR)*, 2002.

[96] B. Medjahed and A. Bouguettaya. A multilevel composability model for semantic web services. *IEEE Transactions on Knowledge and Data Engineering*, 17(7):954–968, 2005.

[97] U. Montanary. Networks of constraints fundamental properties and applications to picture processing. *Information Sciences*, 7, 1974.

[98] M. Mullender and M. Burner. Application architecture: Conceptual view. `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnea/html/eaappconintro.asp`, 2002.

[99] B. Nadel. Tree search and arc consistency in constraint satisfaction algorithms. In *Search in Artificial Intelligence*. Springer-Verlag, New York, 1988.

[100] C. Nagl, F. Rosenberg, and S. Dustdar. Vidre - a distributed service-oriented business rule engine based on ruleml. In *10th IEEE International Enterprise Distributed Object Computing Conference (EDOC'06)*, 2006.

[101] D. Nau, Malik Ghallab, and P. Traverso. *Automated Task Planning. Theory and Practice*. Morgan Kaufmann, 2004.

[102] N. J. Nilsson. *Principles of Artificial IntelligencePalo Alto*. Tioga, Palo Alto, 1980.

[103] L. Nixon and E. Paslaru. State of the art of current semantic web services initiatives, 2004.

[104] B. Orriens, J. Yang, and M. Papazoglou. Model driven service composition. In *Conf. on Service-Oriented Computing (ICSOC-03)*, Lecture Notes in Computer Sciences, pages 75–90. Springer, 2003.

[105] S. Oundhakar, K. Verma, K. Sivashanugam, A. Sheth, and J. Miller. Discovery of web services in a multi-ontology and federated registry environment. *International Journal of Web Services Research (JWSR)*, 2(3):1–32, 2005.

[106] M. Paolucci, T. Kawamura, T. Payne, and K. Sycara. Semantic matching of web services capabilities. In *Proceedings of the International Semantic Web Conference (ISWC)*, pages 333–347, 2002.

[107] M. Papazoglou, M. Aiello, M. Pistore, and J. Yang. Planning for requests against web services. *IEEE Data Engineering Bulletin*, 25(4):41–46, 2002.

[108] M. Papazoglou, V. d'Andrea, D. Plexousakis, P. Grefen, J. Yang, M. Mecella, and P. Plebani. Service-oriented computing manifesto.

[109] M. Papazoglou and J.-J. Dubray. A survey of web service technologies. Technical Report DIT-04-058, University of Trento, June 2004.

[110] M. Papazoglou and W. van den Heuvel. Service oriented architectures. *VLDB Journal*, 2005. To appear.

[111] M. P. Papazoglou and D. Georgakopoulos. Service-oriented computing. *Commun. ACM*, 46(10):24–28, 2003.

[112] Abhijit Patil, Swapna Oundhakar, Amit Sheth, and Kunal Verma. METEOR-S web service annotation framework. In *Proceedings of the International World Wide Web Conference (WWW)*, 2004.

[113] M. Pistore, A. Marconi, P. Bertoli, and P. Traverso. Automated composition of web services by planning at the knowledge level. In *IJCAI-05*, pages 1252–1259, 2005.

[114] M. Pistore and P. Traverso. Planning as model checking for extended goals in non-deterministic domains. In *Proc. 7th International Joint Conference on Artificial Intelligence (IJCAI-01)*, 2001.

[115] Marco Pistore, F. Barbon, Piergiorgio Bertoli, Dmitry Shaparau, and Paolo Traverso. Planning and Monitoring Web Service Composition . In *ICAPS'04 Workshop on Planning and Scheduling for Web and Grid Services*, June 2004.

[116] G. Rabideau, S. Chien, J. Willis, and T. Mann. *Interactive, repair-based planning and scheduling for shuttle payload operations*. Artificial Intelligence, 2003.

[117] J. Rao and X. Su. Toward the composition of semantic web services. LNCS 3033, pp. 760–767, 2004.

[118] J. Rintanen. Complexity of planning with partial observability. In *14th International Conference on Automated Planning and Scheduling*, pages 345–354. AAAI Press, 2004.

[119] William N. Robinson. Monitoring web service requirements. In *Conf. on Requirements Engineering (RE 2003)*, pages 65–74, 2003.

[120] D. Roman, H. Lausen, and U. Keller. Web service modeling ontology standard (WSMO-standard). Working Draft D2v0.2, WSMO, 2004.

[121] Hana Rudova and Keith Murray. University course timetabling with soft constraints. In *Practice And Theory of Automated Timetabling, Selected Revised Papers*, LNCS 2740, pages 310–328. Springer-Verlag, 2003.

[122] B.-H. Schlingloff, A. Martens, and K. Schmidt. Modeling and model checking web services. *Electronic Notes in Theoretical Computer Science: Issue on Logic and Communication in Multi-Agent Systems*, 126:3–26, March 2005.

[123] B. Schmit and S. Dustdar. Model-driven develoment of web service transactions. *International Journal on Enterprise Modelling and Information Systems Architectures*, 1(1), 2005.

[124] B. Selman and H. Kautz. Domain-independent extensions to gsat: Solving large structured satisfiability problems. In *IJCAI-93*, 1993.

[125] Pinar Senkul, Michael Kifer, and Ismail Hakki Toroslu. A logical framework for scheduling workflows under resource allocation constraints. In *VLDB*, pages 694–705, 2002.

[126] H. Simon. *The Sciences of Artificial.* MIT Press, 1996.

[127] E. Sirin, B. Parsia, and J. Hendler. Filtering and selecting semantic web services with interactive composition techniques. *IEEE Intelligent Systems*, 19(4):42–49, 2004.

[128] D. Smith, J. Frank, and A. Jonsson. Bridging the gap between planning and scheduling. *Knowledge Engineering Review*, 15(1):47–83, 2000.

[129] David E. Smith and Daniel S. Weld. Conformant graphplan. In *AAAI/IAAI*, pages 889–896, 1998.

[130] Mike Smith, Christopher Welty, and Deborah McGuinness (eds.). OWL web ontology language guide. Recommendation, W3C, February 10 2004.

[131] SOAP. *Simple Object Access Protocol 1.1*, May 2000.

[132] B. Srivastava and J. Koehler. Web Service Composition - Current Solutions and Open Problems. In *Proceedings of ICAPS'03 Workshop on Planning for Web Services*, Trento, Italy, June 2003.

[133] Mark Stefik. Planning with constraints (molgen: Part 1). *Artif. Intell.*, 16(2):111–140, 1981.

[134] K. Sycara, S. Widoff, M. Klusch, and J. Lu. Larks: Dynamic matchmaking among heterogeneous software agents in cyberspace. *Autonomous Agents and Multi-Agent Systems*, 5(2):173–203, 2002.

[135] S. Thakkar, J.L. Ambite, and C.A. Knoblock. A view integration approach to dynamic composition of web services. In *1st ICAPS International Workshop on Planning for Web Services (P4WS 2003)*, 2003.

[136] S. Thakkar, J.L. Ambite, and C.A. Knoblock. A data integration approach to automatically composing and optimizing web services. In *2nd ICAPS International Workshop on Planning and Scheduling for Web and Grid Services*, 2004.

[137] S. Thakkar, J.L. Ambite, C.A. Knoblock, and C. Shahabi. Dynamically composing web services from on-line sources. In *AAAI Workshop on Intelligent Service Integration*, 2002.

[138] E. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, London, 1995.

[139] UDDI. *Universal Description, Discovery, and Integration*, 2002. `http://www.uddi.org`.

[140] Wil van der Aalst and Kees van Hee. *Workflow Management: Models, Methods, and Systems*. The MIT Press, 2002.

[141] Wil M. P. van der Aalst, H. T. de Beer, and Boudewijn F. van Dongen. Process mining and verification of properties: An approach based on temporal logic. In *OTM Conferences (1)*, pages 130–147, 2005.

[142] Thomas Vossen, Michael Ball, Amnon Lotem, and Dana S. Nau. On the use of integer programming models in ai planning. In *IJCAI*, pages 304–309, 1999.

[143] Daniel S. Weld, Corin R. Anderson, and David E. Smith. Extending graphplan to handle uncertainty & sensing actions. In *AAAI/IAAI*, pages 897–904, 1998.

[144] WS-Policy. *Web Services Policy Framework*, May 2003. `http://www-106.ibm.com/developerworks/library/ws-polfram/`.

[145] WS-Transaction. *Web Services Transaction*, August 2002. `http://www-106.ibm.com/developerworks/webservices/library/ws-transpec/`.

[146] WSDL. *Web Services Description Language 1.1*, March 2001. `http://www.w3.org/TR/wsdl`.

[147] WSLA. *Web Service Level Agreements*, 2003. `http://www.research.ibm.com/wsla`.

[148] D. Wu, B. Parsia, E. Sirin, J. Hendler, and D. Nau. Automating daml-s web services composition using shop2. In *ISWC-03*, 2003.

[149] E. Stroulia Y. Wang. Semantic structure matching for assessing web-service similarity. In *Conf. on Service-Oriented Computing (ICSOC-03)*, Lecture Notes in Computer Sciences, pages 194–207. Springer, 2003.

[150] H. Zhao and P. Doshi. A hierarchical framework for composing nested web processes. In *ICSOC*, 2006.

Appendix A

XSRL and XSAL in BNF notation

Let us first define the notation we use throughout the language definitions. The notation is mostly based on BNF with the following assumptions:

- `+element` denotes multiple elements, at least one,
- `[element]` denotes optional element,
- `<element>` and `</element>` denotes starting and ending XML element tag correspondingly,
- any trailing spaces before or after elements or XML tags are trimmed and ignored.

In the proposed notation XSRL language is defined as follows:

```
xsrl      <- '<XSRL>' goal '</XSRL>'

goal <- achieve-all | then | prefer | optional
        vital | atomic |
        vital-maint | atomic-maint |
        proposition

achieve-all    <-
    '<ACHIEVE-ALL>' +goal '</ACHIEVE-ALL>'
then           <-
    '<BEFORE>' goal '</BEFORE> <THEN>' goal '</THEN>
```

```
prefer          <-
    '<PREFER>' goal '</PREFER> <TO>' goal '</TO>'
optional        <-
    '<OPTIONAL>' goal '</OPTIONAL>'

vital           <-
    '<VITAL>' proposition '</VITAL>'
atomic          <-
    '<ATOMIC>' proposition '</ATOMIC>'

vital-maint     <-
    '<VITAL-MAINT>' proposition '</VITAL-MAINT>'
atomic-maint <-
    '<ATOMIC-MAINT>' proposition '</ATOMIC-MAINT>'

proposition     <- '<CONST ATT="true|false">' | var |
                   '<AND>' +proposition '</AND>' |
                   '<OR>' +proposition '</OR>' |
                   '<NOT>' proposition '</NOT>' |
                   '<GREATER>' var '</GREATER>'
                       '<THAN>' rval '</THAN>' |
                   '<LESS>' var '</LESS>'
                       '<THAN>' rval '</THAN>' |
                   '<EQUAL>' var rval '</EQUAL>'
var             <- a..zA..Z[rval]
rval            <- +a..zA..Z0..9.
```

The XSAL language in the proposed notation is defined as follows:

```
xsal       <- '<XSAL>' assertion '</XSAL>'
assertion <- statement | achieve-all | then | prefer

achieve-all  <- '<ACHIEVE-ALL>' +assertion '</ACHIEVE-ALL>'
then         <- '<BEFORE>' assertion '</BEFORE>'
                '<THEN>'   assertion '</THEN>'
prefer       <- '<PREFER>' assertion '</PREFER>'
                '<TO>'     assertion '</TO>'

statement    <- entity | vital | optional |
                atomic | vital-maint | optional-maint

entity   <- '< ENTITY VARIABLE = ' var '>'
                       start-from
                      follows*
             '</ENTITY>'
start-from <- '<START-FROM>' proposition '</START-FROM>'
follows    <- '<FOLLOWS>'    proposition '</FOLLOWS>'
              '<BY>'         proposition '</BY>'

vital         <- '<VITAL>'    proposition  '</VITAL>'
optional      <- '<OPTIONAL>' proposition  '</OPTIONAL>'
atomic        <- '<ATOMIC>'   proposition  '</ATOMIC>'

vital-maint
    <- '<VITAL-MAINT>'    proposition '</VITAL-MAINT>'
optional-maint
    <- '<OPTIONAL-MAINT>' proposition '</OPTIONAL-MAINT>'

proposition  <- '<CONST ATT="true|false">' |  var  |
                '<AND>' +proposition '</AND>' |
                '<OR>'  +proposition '</OR>'  |
                '<NOT>'  proposition '</NOT>' |
                '<GREATER>' var '</GREATER>'
                  '<THAN>' rval '</THAN>' |
                '<LESS>'    var '</LESS>'
                  '<THAN>' rval '</THAN>' |
                '<EQUAL>'   var rval '</EQUAL>'
var          <-  a..zA..Z[rval]
rval         <- +a..zA..Z0..9.
```

Appendix B

Service planning problem using planning as model checking

In this chapter we show how to adapt the planner based on the planning as model checking algorithms to solve the service planning problem defined in Chapter 3. Planning as model checking supports extended goals over boolean variables [114], but it cannot deal with numeric variables and constraints efficiently. In contrast, XSRL in addition to dealing with boolean variables used in typical goal languages, such as the one proposed in [73], deals with variables that range over domains such as reals, integers, and so on. To allow for this we introduce the notion of 'booleanization'. The idea behind booleanization is that constraints expressed in the goal over domains ranging over variables are treated as boolean propositions. For example, consider the expression $money < 100$ with an integer variable $money$. After booleanization this becomes a boolean proposition that can be either true or false.

Definition 17 (Booleanization). *The booleanization of a domain $\mathcal{D}$ with respect to a goal g is a tuple*
$BD = \langle \mathcal{S}', Prop, Act, R, P, Out, Tr', Role_{Act}, Role_{P} \rangle$ *derived from the original domain $\mathcal{D}$ in the following way. The set of variables Var is replaced by the set of boolean proposition $Prop$ according to the following rules:*

- *all boolean variables in Var are also in P,*
- *all linear constraints appearing in g are added as boolean propositions in P,*
- *all variables in Var that do not appear in g are omitted in P.*

The set of states and transition function are changed to fit the above introduction of boolean propositions.

An execution structure of a plan over a booleanized domain for a given goal, represents the possible ways a plan can be executed and it is essential to determine the reachability of a given goal from a particular state.

Definition 18 (Execution Structure). *The execution structure of plan π in the booleanization of domain D with respect to goal g from state s_0 is the structure $K = \langle S, R, L \rangle$, where*

- *$S = \{(s, c) : action(s, c)$ is defined $\}$ is the set of states of the execution structure,*
- *$R = \{((s, c), (s', c')) :$ if $\exists (s, c) \rightarrow (s', c')$ and $ctxt(s, c, s') = c'\}$ is the relation*
- *$L(s, c) = \{b \in P\}$,*

The execution structure of a plan in a domain represents how the domain is traversed by the plan. Before defining the notion of goal satisfaction, we need to introduce a few elements of notation. We use the symbol σ to denote *finite paths*. S denotes the set of all states in the execution structure K. Given a set Σ of finite paths, the set of minimal paths in Σ is defined as $min\{\Sigma\} = \{\sigma \in \Sigma : \forall \sigma' < \sigma \implies \sigma' \notin \Sigma\}$. Given a goal g, $S_g(s)$ represents the set of finite paths that lead to the satisfaction of goal g from state s, while $F_g(s)$ represents the set of finite paths that lead to a failure. A state s' is said to be *reachable* from the state s if there exists a path starting from s and leading to s'. A plan is denoted by π.

The notion of goal satisfaction $K, s \models g$ is defined in terms of the set of failure states for the goal g on the execution structure K derived from a booleanized domain with starting state s as follows

$$K, s \models g \text{ iff } F_g(s) = \emptyset$$

The set of failure states $F_g(s)$ for a goal g from a state s is defined inductively in the following way:

p

$S(s) = \{(s)\}$, $F(s) = \emptyset$, that is, $p \in L(s)$ for all proposition letters p of the booleanized domain, otherwise $S(s) = \emptyset$, $F(s) = \{(s)\}$

$\neg p, p_1 \wedge p_2, p_1 \vee p_1$

not p, p_1 and p_1, p_1 or p_1

achieve-all $g_1..g_n$

$S(s) = min\{\sigma : \exists\sigma_1 \leq \sigma\ \sigma_1 \in S_{g_1}(s) \wedge \ldots \wedge \exists\sigma_n \leq \sigma\ \sigma_n \in S_{g_n}(s)\}$
$F(s) = min\{F_{g_1}(s) \cup \ldots \cup F_{g_n}(s)\}$

before g_1 **then** g_2

$S(s) = \{\sigma_1; \sigma_2 : \sigma_1 \in S_{g_1}(s) \wedge \sigma_2 \in S_{g_2}(last(\sigma_1))\}$
$F(s) = \{\sigma_1 : \sigma_1 \in F_{g_1}(s)\} \cup \{\sigma_1; \sigma_2 : \sigma_1 \in S_{g_1}(s) \wedge \sigma_2 \in F_{g_2}(last(\sigma_1))\}$

prefer g_1 **to** g_2

$S(s) = \{\sigma_1 : \sigma_1 \in S_{g_1}(s)\} \cup \{\sigma_1; \sigma_2 : \sigma_1 \in F_{g_1}(s) \wedge \sigma_2 \in S_{g_2}(last(\sigma_1))\}$
$F(s) = \{\sigma_1; \sigma_2 : \sigma_1 \in F_{g_1}(s) \wedge \sigma_2 \in F_{g_2}(last(\sigma_1))\}$

atomic p

if there is some infinite path ρ such that $\forall s' \in \rho\ s' \not\models p$ then
$S(s) = \emptyset$, $F(s) = \{s\}$, otherwise:
$S(s) = min\{\sigma : first(\sigma) = s \wedge last(\sigma) \models p\}$, $F(s) = \emptyset$

vital p

$S(s) = min\{\sigma : first(\sigma) = s \wedge last(\sigma) \models p\}$

$F(s) = min\{\sigma : first(\sigma) = s \wedge \forall s' \in \sigma\ s' \not\models p \wedge \forall \sigma' \geq \sigma\ last(\sigma') \not\models p\}$

optional p

- if $\exists \pi :\ \pi, s \models vital\ p$, otherwise
- if $\forall \pi' \neq \pi : \pi', s \not\models vital\ p$

optional-maint p

- if $\exists \pi :\ \pi, s \models vital\ maint\ p$, otherwise
- if $\forall \pi' \neq \pi : \pi', s \not\models vital\ maint\ p$

vital-maint p

if $K, s' \models p$ holds for all states s' reachable from s then

$S(s) = \emptyset$, $F(s) = \emptyset$, otherwise $S(s) = \emptyset$, $F(s) = \{s\}$

The satisfaction of a goal is thus defined in terms of whether a goal may fail or not during execution.

A *solution* to an XSRL request is defined in terms of the plan and one of the possible plan executions. This execution is required to satisfy all XSRL goal propositions. Formally,

Definition 19 (Solution). *A* solution *for a domain D with respect to a goal g from state s_0 is the tuple $\langle \pi, \sigma \rangle$, where:*

π *is a valid plan for domain D and goal g: $K_{D,\pi}, s_0 \models g$*

σ *is one of the possible executions of the plan π, that satisfies the goal g*

A *problem* of interleaving planning and execution is the finding of a solution for given domain, goal and initial state.

Algorithm 10 plan(domain d, state s, goal g)

```
domain_bool = booleanize($d$)
repeat
    goal_bool= booleanize($g$)
    plan = MBPplan(domain_bool,$s$,goal_bool)
    if plan != failure then
        return plan
    else
        if there are untraversed combinations of optional goals then
            modify $g$ accordingly
        else
            return failure
        end if
    end if
until true
return failure
```

The *planner* function (Algorithm 10) is very short as it relies on an existing planner (MBP). MBP is a model based planner which, given a domain description and a goal, synthesizes a plan for the given goal or returns failure if a plan does not exist. Since MBP deals only with domains and goals in which the variables are boolean a preliminary step is necessary in order to adapt MBP to our framework. This reduction, called booleanization, takes all linear constraints over non boolean variables and turns them into boolean propositions which are true, false or undefined in the current state of the domain. The same reduction is necessary for the goal. The planner returns a sequence of actions for 'reaching' the booleanized goal. For brevity, we do not give the full details of booleanization here, but simply explain the basic concept behind it:

(i) The booleanized domain is as the original one except that instead of the set of variables we have a set of proposition letters specified by the rules (i) and (ii).

ii non-bool linear constraints in the goal are transformed into boolean propositions. Note that two distinct propositions (e.g., $price < 10$ and $price > 5$) are introduced to take into account two constraints on the same variable.

(iii) The truth of the propositions is established recursively by starting from the current state, looking at the current values of the variables and moving along the actions using semantic rules to establish the truth of propositions. In case of conflicting values for a proposition in a state (e.g., the case of two actions with different semantic rules entering in the same state), the state is divided into two states and then the propagation proceeds further from each state. If an action enters an already visited state without proposition conflicting value then the booleanization process is complete.

After the booleanization, the domain is passed to a model-based planner. The planner is invoked until the plan is found or all combinations of optional goals are attempted. The algorithm works with optional goals in the following way. First, it processes them as vital and, in case of failure, the planner function iterates through the optional goals, eliminating (or reintroducing) them from a goal until it can synthesize a plan or all combinations of optional goals have been taken into account. For instance, for an optional goal “booking a train, if possible”: first the planner tries to find a plan with “booking a train” as a vital condition and then, in case of failure, it tries to synthesize a plan without any restriction on trains. There is no particular rule on which goals are eliminated first and in which order. The algorithm only ensures to the user a complete search throughout all optional goals combinations. This approach gives us correct but possibly non optimal solution, for instance, the algorithm may find a solution with a hotel price equal to 200, where there may exist hotels with prices equal to 180. This is caused by the non optimality of solutions generated a planner such as MBP. An optimal search would require a higher level of complexity.

Despite the fact that planning as model checking algorithms are not efficient with domains with large amount of numeric data and constraints (that is typical for web services scenario), it can be used for efficient planning for domains with mostly boolean formulas with limited number of action effects over numeric variables, like integers and reals.

However, the algorithms for planning as model checking based on booleanization does preserve completeness in general. In order to be complete, there are number of additional assumptions have to be held, as it is shown in the following section.

B.1 Completeness and correctness of planning as model checking with booleanization

Lemma 1 (Repeatable executions). *Given a domain D, goal g and an initial state s_0, if the assumptions (ii) and (iii) are satisfied, then the execution σ for a plan π is repeatable, that is, the execution σ of the plan π is invariant from the number of times the plan π is executed.*

Repeatable executions. An execution of a plan depends on an environment. More precisely, it depends on the knowledge variables and on actions output types. From assumption (ii) it follows that knowledge-gathering actions return the same values being invoked in the same context. Thus, the environment for all plan executions is the same. By assumption (iii) for the same knowledge variables values actions have a deterministic outcome. It follows that all executions of a plan are the same. □ □

Lemma 2 (Infinite executions). *Given a domain D, goal g and an initial state s_0, if the assumptions (ii) and (iii) are satisfied, then the infinite execution σ for a plan π is always successful, that is, $K_{D,\pi}, s_0 \models g$.*

Infinite executions. The plan consists of finite number of states, contexts and transitions between them, but it can imply executions that have infinitely many action invocations. When plan is executed, Algorithms 8 checks if the goal is failed after every action. Thus, infinite execution is possible only when goal is satisfied after each action, that is, if $K_{D,\pi}, s_0 \models g$. □ □

Theorem 1 (Algorithm soundness and completeness). *Given a domain D, a goal g and an initial state s_0, under assumptions (i)–(vi) Algorithms 7, 8 and 10 are* sound *and* complete*:*

1. *if there exists a non-empty set of solutions Ω, s.t. $\forall \langle \pi, \sigma \rangle \in \Omega : K_{D,\pi}, s_0 \models g$ and $K_{D,\sigma}, s_0 \models g$ then plan π of one of the solutions $\langle \pi, \sigma \rangle$ is found and its successful execution σ is executed by Algorithms 7, 8 and 10.*
2. *if the set of solutions is empty $\Omega = \emptyset$ then Algorithm 7 returns failure*

Algorithm completeness. The proof is split in two parts. First, we prove that if at least one solution $\langle \pi, \sigma \rangle$ exists then Algorithm 7 finds a plan π and executes a successful execution σ. Secondly, completeness property is proven: Algorithm 7 returns a failure if there is no solution for the given input.

Soundness. From [73] it follows that the planner for extended goals based on model checking always synthesizes a valid plan if at least one exists, and returns failure otherwise. A valid plan is the plan that for a given booleanized domain D_{bool} satisfies the goal g: $K_{D_{bool},\pi}, s_0 \models g$. From assumption (iv) it follows that if a valid plan exists for domain D then it also exists for a booleanized one, and, therefore, the model-based planner finds it.

Let us assume that solution $\langle \pi, \sigma \rangle$ exists s.t. $K_{D,\pi}, s_0 \models g$ and $K_{D,\sigma}, s_0 \models g$. From assumption (i) it follows that all actions are retractable. Therefore we can always return to an initial state with the same critical variables values. Thus, without loss of generality, we can assume that at start of every iteration the corresponding compensated actions are executed to return the domain to an initial state.

Let us define the algorithm *iteration* as one pair of planner-executor invocation in Algorithm 7. As it follows from the theorem assumptions (ii) and (iii) the number of algorithm iterations is finite. Therefore either an executor is stuck in an infinite execution or the planner is invoked for all possible combinations of providers. From Lemma 2 it follows that if an executor processes the infinite execution then the execution satisfies the goal. On the other hand, if the planner is invoked for all possible combinations of providers, it should, finally, synthesize a plan yielding a solution. From Lemma 1 it follows that each plan π has a repeatable execution σ, and, therefore a synthesis of solution plan π implies that executor processes the execution σ from a solution pair $\langle \pi, \sigma \rangle$.

Completeness. It is obvious that if the plan π is synthesized and its execution completely processed, they form a solution. The synthesized plan is always satisfies the goal: $K_{D,\pi}, s_0 \models g$, from other point, if the executor processes the goal till the end, then this execution is successful. As follows from Lemma 2 infinite executions are always successful. Therefore, by definition of a solution, a pair $\langle \pi, \sigma \rangle$ is a solution. We have already shown that the number of iterations is finite, therefore, if there is no solution for the problem then Algorithm 7 returns failure. □ □

Finally, we consider the domain integrity property.

Corollary 1 (Domain integrity). *Given a domain D, a goal g and an initial state s_0, under assumptions (i)–(vi) domain integrity is preserved by Algorithms 7, 8 and 10, that is, if Algorithm 7 returns failure then the critical variables remain unchanged.*

Domain integrity. We have already shown that Algorithm 7 returns failure if there is no solution. Before returning a failure, the rollback plan is synthesized and executed. It is always successful according to assumption (i), and, therefore, the algorithm preserves domain integrity. □ □