Encoding Web Services Requests
As Constraints: An Implementation

Codifica Di Richieste A Web Services Come Vincoli:
Un’Implementazione

Relatore: Prof. Marco Aiello
Correlatore: Dott. Alexander Lazovik
Laureando: Igor Cappello

22 febbraio 2006
Contents

1 Introduction 4
  1.1 Organization of the thesis .............................. 5

2 The XSRL Service Request Language 6
  2.1 The XSRL Framework ..................................... 6
    2.1.1 Planning domain .................................. 7
    2.1.2 Transitions, actions and effects ................... 7
    2.1.3 The plan ........................................... 8
    2.1.4 Expressing the request ............................ 8
    2.1.5 Interleaving planning and execution ............... 9
  2.2 A concrete example .................................... 9
    2.2.1 The business domain ............................. 9
    2.2.2 The user’s request ............................... 9

3 Encoding a request as a constraint 11
  3.1 Encoding the domain ................................... 12
  3.2 Encoding the requests .................................. 13
  3.3 The example encoded ................................. 15
    3.3.1 The domain encoded ............................... 15
    3.3.2 The request encoded ............................. 15
    3.3.3 The solution ....................................... 16

4 Implementation 18
  4.1 A brief overview ....................................... 18
    4.1.1 The org.xsrl.domain package ..................... 19
    4.1.2 The org.xsrl.domain.goal package ............... 20
    4.1.3 The org.xsrl.domain.spaces package ............ 20
    4.1.4 The org.xsrl.framework package ................. 20
    4.1.5 The org.xsrl.framework.registry package ...... 21
    4.1.6 The org.xsrl.framework.cp package ............. 21
    4.1.7 The business domain ............................. 22
  4.2 Description of the code ................................. 26
    4.2.1 Internal representation of the domain .......... 26
    4.2.2 Recognizing and handling cycles .................. 27
    4.2.3 Implementing the vital constraint ............... 30
    4.2.4 Implementing the vital-maint constraint ....... 30
    4.2.5 Implementing the achieve-all constraint ....... 32
    4.2.6 Implementing the before-then constraint ...... 33
Chapter 1

Introduction

Service Oriented Computing (SOC) is a new paradigm in distributed programming; this is given by the fact that the focus shifts from the “object” to the “service”: a service is an autonomous entity that exposes some functionalities to an external system. Implementing this abstract definition in a network and enabling the service to exchange XML-based messages in an asynchronous way gives us a web service, that typically concentrates its functionalities on a specific task. We can have for example a web service that handles the information regarding the weather, another one that handles traffic management and so on. Putting together the descriptions of different web services, we obtain a distributed system that can handle complex tasks. The various tasks that arise trying to handle this composition are known as the service composition problem. This comprehends various aspects to take into account: selecting available services, to be composed to achieve a complex goal, and performing this composition automatically are two examples of these aspects. The fact that these tasks are performed in an automatic way opens the way to orchestration and choreography, or, more generally, to the composition of web services. This means we can build services, based on individual and independent services, with added value, in the sense that, cooperating, they are able to perform complex tasks.

The work presented focuses its efforts into enabling the user to express complex requests upon a composition of web services described as a business domain. The proposed approach consists in encoding the business domain and the user’s request into a set of mathematical constraints. Solving these constraints means finding a plan, that contains the sequence of action defined in the domain, whose execution satisfies the user’s request.

The description of a web service is typically given in a WSDL (Web Service Description Language) document. WSDL [20] is a XML-based language made for representing in an abstract and structured way the operations that a web service can execute, defining the format that an external user must follow to submit the requests. The information about the providers that give an implementation of a particular type of web service are kept in a Service registry. UDDI [18], sponsored by OASIS (Organization for the Advancement of Structured Information Standards), is a Service registry that works as a web service that handles information about service providers and the implementations given by these providers, along with meta-data about the service itself. Once the user (client) has got the description of the web service, directly from the web service
or through a UDDI registry, it can send requests to the service. The basic layer to this client - web service structure is represented by the SOAP [1] layer, that defines the way a request to a web service and a respond for the client have to be written. SOAP lays upon HTTP protocol, through which XML documents, representing requests and responds, are exchanged.

1.1 Organization of the thesis

In Chapter 2 the XSRL framework, upon which the application is based, is presented and described; a simple example of a business domain and of a request are given in Section 2.2. The ideas that are behind the encoding phase are presented in Chapter 3, along with the encoding of the example presented in Chapter 2. In Chapter 4 the algorithms, implementing the encoding proposed in Chapter 3, are presented and explained. Chapter 5 presents a case study; the related work is presented in Chapter 6, while the conclusions and the future work are presented in Chapter 7.
Chapter 2

The XSRL Service Request Language

2.1 The XSRL Framework

If we have to interact with a set of web services, one can think of building a plan in order to organize the service invocations for achieving our goal. The plan has to be related to the domain we are considering; this domain tells us how many web services there are, what they can do (the actions they can execute) and how they are connected, what are the variables that are defined in the particular context and how these variables are affected by the results of the actions that the web services can take. Some of this information can be collected from a WSDL document, described in Chapter 1. We want to give the user a way to express the request: this can be expressed using the XSRL language (as defined in [10]). We propose a framework that supports an orchestrated execution of a plan to perform request expressed in XSRL, which schema is represented in Figure 2.1.

![XSRL Framework architecture](image)

Figure 2.1: XSRL Framework architecture
The user submits its own requests to the monitor, that has also received the business domain representation document. With these information, the monitor can ask the planner to build a plan; with a suitable plan prepared, the monitor asks the executor module to start calling the actions defined within the plan. The executor collects the information about the available web services from an UDDI registry (as presented in Chapter 1); once the executor has the information about the available web services, it starts invoking the web services using SOAP messages. The executor must have the capabilities to interact directly with the user, for example to collect run-time needed information not provided with the XSRL request. The business domain can change dynamically, new web services implementations can become available at any time, while others can be no more reachable (the underlying network could go down, or the service itself could crash). For these reasons the executor has to be able to handle these changes, informing the monitor that the domain has been modified, so the available plan is no more up to date. When the monitor is informed of such changes, it has to request a new plan, built taking into account the new information.

2.1.1 Planning domain

Following the definitions given in [11], we recall the formal definition of goal, transition and role. The domain is a tuple composed of:

- a set $S$ of states
- a variable space $V$
- a set $O$ of output results
- a set $A$ of actions that are defined for the web services in the domain
- a set $T$ of Transitions, expressed as functions; $Tr: S \times A \times O \rightarrow S$
- a set $R$ of roles associated with actions, whose task is to define the nature of web services (their interfaces)
- a set of providers identified by their URI
- a function that associates actions to roles; $R_{Act}: A \rightarrow R$
- a function that associates every provider to a role in the process

Initially the domain is in a starting state that we call $s_0$.

2.1.2 Transitions, actions and effects

We model a request invoked against a web service as a transition associated with an action that will change the state in which the system is, as well as a set of variable values. The changes can vary according to the different outcoming results for the action.

In this framework an action is associated to one or more results: every action can follow the normal execution of the action, returning a normal result. In some cases the actions cannot be executed following the normal (or default) execution. In these cases, the action gives an abnormal result. A transition is
said to be nondeterministic if the action corresponding to it can have one or more abnormal results. If a transition is nondeterministic, it must have anyway an action associated with a normal result: in other words, a normal execution has to be possible for every transition. For a nondeterministic transition, the actual path followed can be determined only at run-time.

2.1.3 The plan

In the proposed framework a plan (formally defined in [11]) represents the sequence of actions that the system has to perform to achieve the user’s goal. This plan is prepared by the Planner module, according to the information available when the request, given by the user, has been processed by the system.

2.1.4 Expressing the request

Customers (from now on defined more generally as users) want to define their goals in the most intuitive and natural way; this idea of intuitive and natural for the user corresponds to an abstract and complex way for the system. If the user is given the possibility to do this in a way close to his intuitive way, the framework gives added value to the user’s experience. XSRL gives a set of methods for the user to express his goal and how the task has to be achieved. The given implementation codes a subset (the vital, vital-maint, before-then, achieve-all items) of the ways to express a request introduced by XSRL [10] (Xml Service Request Language) specifications: we have a set of basic requests vital p, vital-maint p, atomic p, vital-atomic p, where p is a goal defined on the variables of the domain; then we can say that a request g is a basic request or one between achieve-all g1 . . . gn, before g1 then g2, prefer g1 to g2, optional g .

- vital p: this constraint expresses that p must hold in a state reachable from s0; we impose that, following a path composed only by transitions associated with normal results, we can reach a state in which p is true. For that reason the nature of this type of request is said to be of reachability.

- vital-maint p: this constraint is different from the previous one because it expresses the fact that p holds in a state if it is verified also in all the states that belong to the path from s0 to this state; this is the reason because this request is said to be of maintainability type: it has to assure that p is true for all actions following the path. Indeed this is a stronger requirement than the one offered by the vital operator

- achieve-all g1 . . . gn: this constraint has the peculiarity that it takes as parameter a list of constraints, assuring the fact that it holds in a state if and only if all the requests g1 . . . gn are satisfied in the same state. More, if, in order to achieve one of the subgoals, a particular path has been chosen, the same choices between deterministic actions must be taken trying to satisfy all the other requests; however, if a request is a vital-maint one, this fact must be valid also for the nondeterministic actions chosen to satisfy that particular request.

- before g1 then g2: this constraint is very similar to the one just presented, but has the peculiarity that it imposes a chronological sequence between
the satisfying of the two request. This is made trying to satisfy the first request, and then achieving the second one following the same path used before (in our case this means that we have to make the same choices for the actions to take, either deterministic or nondeterministic).

- **atomic** p: this request is similar to the one defined for the vital one, but it requires that the preference must be verified even if in the path non-normal results are encountered. In other words, it is a vital goal that must be verified for all the possible outcomes for the non-deterministic transitions.

- **atomic-maint** p: as before, this is very similar to the vital-maint goal, but it also considers all the possible outcomes for the non-deterministic transitions.

- **prefer** $g_1$ to $g_2$: for this request we first try to satisfy request $g_1$; if that it is not feasible, we try to satisfy request $g_2$, independently from the first attempt.

- **optional** g: this is a special case for the request just presented: it can be expressed as a *prefer g to true* request.

### 2.1.5 Interleaving planning and execution

In the framework proposed in [11], planning the actions to satisfy a user’s goal and executing these actions are operations that are interleaved. Taking this approach is necessary because in this framework we suppose we do not operate in a priori fully known environment. We start with a partial knowledge of variable values, then we make a plan and start executing it. Some variables’ values can be discovered at later stages of execution; in this case, we update the domain and eventually prepare a new plan and restart executing it. In this way the application can adapt the execution to domain changes and to the acquisition of new knowledge.

### 2.2 A concrete example

#### 2.2.1 The business domain

Let us consider the supply chain situation described in the following. A set of assemblers and various element-vendors have put together their business, giving users the possibility to book online a computer. 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.

#### 2.2.2 The user’s request

An online user wants to buy a computer and does not want the total cost to be more than 1000 euro. Following the definitions given in Section 2.1.4, he expresses this request as
\textit{achieve-all}(vital-maint (price \leq 1000), vital (concluded = 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 an 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.
Chapter 3

Encoding a request as a constraint

The final goal of our approach to service orchestration is getting a set of constraints that correlate the domain (and then the results of the transitions) with the request expressed by the user. These constraints are in the form $c_v \star val$, where $c_v$ is a mathematical expression, val is a numeric value defined in a request and $\star$ can be one of the following $\{\leq, =\}$. Building the set of expressions involves two different phases: we must first take into account the domain itself, considering the structure of the graph representing the domain: branching points made of deterministic or non-deterministic actions, cycles, converging points are the cases that are treated by the algorithms presented. Once the business domain is encoded, we have to consider the user-defined request, expressed in an XSRL statement, built on the top of the domain’s definition. Now we can define the service constraint problem; it is a tuple CP that contains:

- a set of boolean variables $b_i$
- a set of boolean variables $c_i$
- a set of numeric variables $n_i$
- a set of constraints in the form $c_v \star val, \star \in \{\leq, =\}$

The difference between the two sets of boolean variables lays on the control that the system has on assigning to them different values: $b$ variables are called controlled; this means that the system has the ability to say that a $b$ variable is true or false. This kind of variables will be used when dealing in particular with deterministic branch points, because choosing between possibilities will be equivalent to set to true one variable and to false all the other alternatives for that transition. The other kind of variables, $c$, are called non controlled: the system cannot set a value for such variables; these ones are used when dealing with non-deterministic transitions: this is equivalent to say that the system is given by the execution itself the right values for these variables. Anyway the system must take into account these values encoding the domain and the user’s requests.
3.1 Encoding the domain

First, we need a way to express the domain (as defined in Section 2.1.1) in terms of mathematical expressions in order to use constraint-solving engine. For this purpose we use the two different sets of boolean variables introduced at the beginning of Chapter 3. The values associated to these two types of variables define the choices made between different paths in the graph representing the domain, given that the system is not free to decide whether a non-controlled variable is true or false; in this way the non-determinism, found in a subset of the transitions defined in the domain, can be treated by the framework, which is able to consider the possible non-deterministic outcomes of the transitions, referring to the non-controlled boolean variables. The boolean variables $\beta$ and $\xi$ are used when forming the expressions that will become the basis on which the constraints, that encode the requests to the web services, will be built. This is done visiting recursively the domain. In the following list, presenting the factors that have to be taken into account when encoding a domain, the transitions in a deterministic branch point are related to different actions (denoted with $a_i$, $a_j$); this is not true for a nondeterministic branch point, where different transitions are related to different outcomes for the same action; to identify every transition in a nondeterministic branch point, we use a different notation: if we have two different possible results for a nondeterministic action $a_k$, we identify the first using $a'_k$ and the second using $a''_k$.

- the existence of transitions leaving the state: if there are no transitions, then nothing has to be encoded.

- the nature of these transitions (deterministic or not): if the transitions are nondeterministic, then a variable $\beta_i$ is introduced; this variable multiplies the expression given by the sum $\sum_j (\xi_j \ast a_k^{i-th})$, where $a_k^{i-th}$ represent the $i-th$ possible effect given by the execution of the $k-th$ action, and $\xi_j$ is the variable that expresses the outcome (non controlled by the system) between the different results for the action. In the end we get $\beta_i \ast \sum_j (\xi_j \ast a_j)$ for the mathematical expression. This case corresponds to the nondeterministic branching point in Table 3.1, where there are only two different possible outcomes for the action. In the case of a deterministic set of transitions leaving the state, we do not have to introduce non-controlled variables; their place is taken by the controlled variables, so we get $\sum_j (\beta_j \ast a_k)$. The different $a_k$ represent the outcomes given by the execution of different actions, and the $\beta_j$ represent the variable controlling the $j-th$ choice. This case is expressed in the deterministic branching point in Table 3.1. Actually we obtain more than one of these encodings: given that the boolean variables express a choice, it is natural to impose that, for the controlled variables controlling a deterministic branching point, must hold $\sum \beta_i \leq 1$: a possible choice is represented by all the $\beta$ set to zero, meaning “do nothing”. For the non-controlled variables we impose that exactly one outcome must be taken; this is given by an external entity, but only one path can be followed: $\sum \xi_j = 1$.

- the presence of a cycle starting from the current state

- the nature of the cycle (directed or converging): if we find that a cycle exists, first we check if the current state is part of the path followed. If
not, then we have found a converging point. This case is solved splitting the current state and doing nothing more; this action creates two new twin states that must be encoded separately, as shown in the converging case in table 3.1. If the current state is part of the path, then we have discovered a directed cycle. This situation is handled introducing first a numeric variable $n_i$ that tells us the number of times the cycle is followed during execution, then we must take into account all the non-controlled variables that leave the execution in the cycle, and finally we must look at all the effects that are given by the actions taken in order to remain in the cycle. The result of this encoding is given by the expression $n_j \cdot (\prod_i \xi_i) \cdot (\sum_k a_k)$, as shown in the directed cycle case in table 3.1.

### 3.2 Encoding the requests

The encoding of the user-defined requests works as follows:

- for the **vital** $p$ request we consider the first state of the domain and build the expression for that state (considering the domain encoding for that state), then we create a new constraint taking the operator and the second operand from $p$. In this way we get a constraint containing the boolean variables controlling the different paths, the results affecting the domain variables coming from the execution of the actions, and the user’s request. We then impose that the normal path must be followed with a list of constraints on the values for the $\xi$ variables: this means that only normal results are accepted for non-deterministic actions. In this way the user can say that $p$ must be true in the end, but he does not care about what happens during the execution: $p$ can be false during execution, but it must be true in the end.

- for the **vital-maint** $p$ request we consider all the expressions built following a particular path in a recursive way. The set of constraints encoding a particular state is made of the one built as for the vital case, plus all the constraints for the reachable states, plus the constraint regarding the single state (this is recursively repeated for the reachable states). In this way we get a list of constraints, to which we add the ones that impose that the normal path is followed, as for the vital case. In this way the user can say that $p$ must be true for all the states composing the path from the initial state to a particular state, being sure that only normal results are accepted for non-deterministic actions.

- for the **achieve-all** $g_1 \ldots g_n$ request we consider the subgoals taken in pairs and then look recursively if, for some branching points in the domain, some choices have been made to achieve those goals. If it is true, then we impose that the same choices on the boolean variables that control those branching points have to be made while trying to achieve all the goals; this is true with the specifications shown in Chapter 1: it is always valid for the controlled variables, while it is valid for the non-controlled variables involved in a vital-maint request. This request gives the user the opportunity to define a list of constraints that must be all true at the same time.
<table>
<thead>
<tr>
<th>situation</th>
<th>graph before</th>
<th>graph after</th>
<th>encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>simple transition</td>
<td><img src="image1.png" alt="Diagram" /></td>
<td><img src="image2.png" alt="Diagram" /></td>
<td>$\beta_1 a_1$</td>
</tr>
<tr>
<td>deterministic</td>
<td><img src="image3.png" alt="Diagram" /></td>
<td><img src="image4.png" alt="Diagram" /></td>
<td>$\beta_1 a_1 + \beta_2 a_2$</td>
</tr>
<tr>
<td>branchpoint</td>
<td><img src="image5.png" alt="Diagram" /></td>
<td><img src="image6.png" alt="Diagram" /></td>
<td>$\beta_1 a_1 + \beta_2 a_2$</td>
</tr>
<tr>
<td>non-deterministic</td>
<td><img src="image7.png" alt="Diagram" /></td>
<td><img src="image8.png" alt="Diagram" /></td>
<td>$\beta_1 (\xi_1 a' + \xi_2 a'')$</td>
</tr>
<tr>
<td>branchpoint</td>
<td><img src="image9.png" alt="Diagram" /></td>
<td><img src="image10.png" alt="Diagram" /></td>
<td>$\xi_1 + \xi_2 = 1$</td>
</tr>
<tr>
<td>converging point</td>
<td><img src="image11.png" alt="Diagram" /></td>
<td><img src="image12.png" alt="Diagram" /></td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>cycle</td>
<td><img src="image13.png" alt="Diagram" /></td>
<td><img src="image14.png" alt="Diagram" /></td>
<td>$n_1 (\xi_1 + \xi_3) (a'' + a_2 + a_3)$</td>
</tr>
</tbody>
</table>

Table 3.1: Basic cases considered encoding the domain
for the before $g_1$ then $g_2$ request we consider the path followed achieving the goal $g_1$, then we impose that the choices on the values for the boolean variables made during this phase must be the same while trying to achieve the goal $g_2$. This is true for controlled variables, and for non controlled variables as well: this constraint expresses a temporal sequence between the achieving of two goals, so from the user’s point of view the goal $g_2$ has to be achieved in a situation in which the solution for goal $g_1$ is given “for granted”.

3.3 The example encoded

The encoding for the example given in Section 2.2 is presented next.

3.3.1 The domain encoded

The domain is defined as follows: there are four states; the transition $t_1$ stands for the action related to contact the assembler, while $t_2$ and $t_3$ represent the possible alternatives for the element-vendors (say $t_2$ for AMD and $t_3$ for intel); $t_4$ stands for the conclusion of the transaction, for simplicity we suppose no faulty results are present for this transition. In figure 3.1 it is possible to see a graph representation of this business domain.

![Figure 3.1: the graph representation of the example domain](image)

The result of the encoding, for state $s_1$, is represented by the expression $\beta_1(a_1 + \beta_2(\xi_1(a_3 + \beta_3(a_4)) + \xi_2(f_3))) + \beta_3(\xi_3(a_2 + \beta_3(a_4)) + \xi_4(f_2)) + n_1(\xi_2(a_1 + f_3) + n_2(\xi_4(a_1 + f_2)))$, where the $a_i$ stand for the results of the actions.

The expressions identified by $n_i$ represent the two cycles that are present in the domain. The encoding of the domain does not return this expression only: there are also the expressions given by the controls on the different branching points; these ones must be always true: $\xi_1 + \xi_2 = 1$, $\xi_3 + \xi_4 = 1$ and $\beta_2 + \beta_3 \leq 1$.

For the other states we proceed in the same way: for state $s_2$ we get $\beta_2(\xi_1(a_3 + \beta_4(a_4)) + \xi_2(f_3)) + \beta_3(\xi_3(a_2 + \beta_3(a_4)) + \xi_4(f_2))$; for state $s_3$ we get $\beta_3(a_4)$, and for its clone, created by the converging point, we get $\beta_5(a_4)$, and finally, for state $s_4$ we get an empty encoding.

3.3.2 The request encoded

Let us take into account the user’s request presented in Section 2.2.2. 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 ai stand for the modifications that involve the variable AMD-selected, the expression becomes: \(\beta_{1v}(a1 + \beta_{2v}(\xi_{1v}(a3 + \beta_{3v}(a4) + \xi_{2v}(f3)) + \beta_{2v}(\xi_{3v}(a2 + \beta_{3v}(a4)) + \xi_{4v}(f2)) + n_1(\xi_{2v})(a1 + f3) + n_2(\xi_{4v})(a1 + f2)) = 1\). 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 ai stand for the modifications to the variable price: since the actions related to the faulty transitions (the ones controlled by \(\xi_{2}\) and \(\xi_{4}\)) in this example do not modify this variable at all, and no subencodings 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 \(\xi\) variables must always be true, since, encoding the domain, we imposed exactly the expression in the “if” statement.

### 3.3.3 The solution

Suppose that the cost of the assembling process (a1) 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 s1 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 \(\beta_{2vm}\) variable; the normal result must be
given, and this is expressed by the $\xi_{1vm}$ variable. The last transition, following this path, is controlled by the $\beta_{4vm}$ variable. Cycles must not be followed. Here are reported the values for the variables controlled by the system; these values hold for both goals, so they are reported only once:

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

With these values, the final state is reached, so the vital goal is satisfied. The values for the variable price, along the path, are the following: $\{0, 15, 715, 975\}$, so the vital-maint goal is satisfied as well.
Chapter 4

Implementation

In Chapter 2 we have illustrated a method to encode user’s requests into Constraint Programming. Here we devote our attention on how one can implement such method into a framework for accepting requests and monitoring their execution. We have implemented the algorithms in Java, anyway, for clarity, here are reported pseudocode versions. These algorithms are an extension of a preexisting XSRL reference application developed by Alexander Lazovik. The choice made for the constraint-solver engine has elected choco (choco.sourceforge.net) as the current engine: this is an easy-to-use, open-source engine that is built on an event-based propagation mechanism with backtrackable structures. Moreover it can use explanations to show why, in a solving run, a constraint could not be satisfied. This particular feature, anyway, has not been used in this work, to soften the work of the solving engine as much as possible. Another interesting feature of the choco engine is that constraints can be “freezed” and reactivated dynamically. The possibility to use these interesting features has made us choose the Choco system.

4.1 A brief overview

The process to obtain a possible execution of the plan, built to satisfy user’s requests made against the domain, starts parsing the xml file representing the domain itself. This task is accomplished by an instance of the FreeBPCompiler class: this class calls a parser, implemented in SAXFreeBPHandler, that uses the SAX libraries in order to build the data structure, as presented in Chapter 1, that is delegated to represent the business domain. Once the domain is read, the application creates a monitor that has the task to handle the execution of the plans and the search for needed providers. The next step is the definition of the problem itself: here the application needs the user to specify its own request, expressed as presented in Chapter 2; this request is then convertend 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 of an executor for the problem. The monitor starts calling iteratively the executor, which has to invoke the actions defined in the plan, checking in the mean time if it has to search for new providers. This particular framework implements the one presented in
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 problem)
- 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 $\beta$ 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.

4.1.1 The org.xsrl.domain package

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 the execution of an action. For every possible result, it can lead to a different state; these information are kept in a Map.
- State: it contains a set of outgoing transitions; moreover, it contains boolean variables that work as flags when encoding the domain as exposed in Section 3.1.
- Variable: a variable defined in the domain. Can be defined on different spaces: integer and boolean are the ones supported now.
- Action: this class represents an action that a web service can execute. An action must refer to a specific role, and can have different results.
- ActionResult: this class represents a specific result given by 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.

In figure 4.1 it is possible to see the uml class diagram for the org.xsrl.domain package. All the classes inherit from DomainObject class.
4.1.2 The org.xsrl.domain.goal package

The org.xsrl.domain.goal package contains the definitions of the possible goals that can be instantiated. All the classes extend BasicGoal. In every goal are kept the information about its nature, such as if it is a MAINTAINABILITY or REACHABILITY goal, and if it is a VITAL or ATOMIC (not yet implemented) goal.

4.1.3 The org.xsrl.domain.spaces package

In the org.xsrl.domain.spaces package are defined the supported types of spaces in which a variable, defined in the domain, can live. At this moment only boolean and integer spaces are supported.

4.1.4 The org.xsrl.framework package

The org.xsrl.framework package contains the class definition for the elements of the XSRL schema, as defined in Chapter 2. It is noticeable the presence of the class ActionInvoker, that has the task to invoke a particular action on a
defined provider. The other classes implement the three main elements for the application: the planner, the monitor and the executor. The implementations for these classes are anyway quite simple: they demand all the effort in encoding and solving the problem and the domain to the classes in framework.cp package.

4.1.5 The org.xsrl.framework.registry package

The org.xsrl.framework.registry package contains the definition of the registry, specified in an xml file as the domain, that defines the implementations of the available web services. As for the domain, this xml file is handled by a SAX parser.

4.1.6 The org.xsrl.framework.cp package

The org.xsrl.framework.cp package is very rich; in figure 4.2 it is possible to see the relations between the main classes composing it. The classes contained in it have to:

- represent the business domain in such a way that the algorithms can easily handle it, using the elements in package org.xsr.domain. This task is mainly accomplished by the Encoding class.

- implement the algorithms to encode the domain and the requests, as seen in Chapter 3. The implementation for this complex 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 (AchieveAllEncoder, BeforeThenEncoder, VitalMaintEncoder, VitalGoalEncoder). All these classes need a register containing the variables defined for the domain (the instance of the class CPVarName, called choco-Vars); every class has an internal register for the variables defined while encoding a particular goal. 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.
Figure 4.2: the UML diagram of the org.xsrl.framework.cp package (only main classes)

4.1.7 The business domain

A particular instance of the business domain can be seen as a graph, given that the transitions can be represented by the arcs in the graph and the business states can intuitively be seen as the graph states. An example of this double vision possibility is given now; this example will be the reference for the rest of the thesis. The presented example, shown in Figure 4.3 is not related to any
specific test-case, but it can be easily related to real life: in fact it represents the sequential execution of two actions, in which for the second one more than one implementation is present. The first action (action1), for instance, cannot fail, while action3 and action2 can give faulty results. If the fault is given by action3, then the execution has to restart from the beginning, while if the fault derives from the execution of action2 the system can continue executing action4, reaching the final state (s4) if it has a normal result; if action4 gives a faulty result, then the system has to restart from the beginning.

![Graph Representation of a Simple Business Domain](image)

Figure 4.3: The graph representation of a simple business domain

In Figure 4.4 it is shown how an xml document can express the states, the roles and the variables composing the domain related to the example shown in Figure 4.3. The roles express the interfaces used to call the actions, implemented by the web services. The variables are instantiated with an initial value; in this framework they can only be of type `int` or `boolean`. 

23
<business-process name="BusinessDomain xml example">
    <states>
        <state name = "s1" />
        <state name = "s2" />
        <state name = "s3" />
        <state name = "s4" />
    </states>
    <roles>
        <role name = "role1" interface = "role1-interface"/>
        <role name = "role2" interface = "role2-interface"/>
        <role name = "role3" interface = "role3-interface"/>
    </roles>
    <variables>
        <variable name="intVar1" type="int" value="0" />
        <variable name="intVar2" type="int" value="100" />
        <variable name="boolVar1" type="boolean" value="false" />
        <variable name="boolVar2" type="boolean" value="false" />
        <variable name="boolVar3" type="boolean" value="true" />
    </variables>
    <actions>
        ....
    </actions>
    <transitions>
        ....
    </transitions>
</business-process>

Figure 4.4: xml code to express the variables and the roles

In Figure 4.5 it is shown how the actions can be defined using the xml dialect understood by the application: every action is related to a specific role; in this way it is possible to define an interface for a web service, simply saying that it implements a specific role. Every action can be associated to a list of effects that modify the values of the variables previously defined. For now, only two operators are supported: the assign operator and the increment operator: = , +=. Another element that can be present is the one represented by the fault tag, which represents the possibility for that action to have a faulty outcome.
<actions>
  <action name = "action1" role = "role1"
    activity = "invoke" type = "0">
    <effects>
      <effect operator="=" result="boolVar1" operand1="true" />
      <effect operator="+='" result="intVar1" operand1="intVar2" />
    </effects>
  </action>
  <action name = "action2" role = "role2"
    activity = "invoke" type = "0">
    <effects>
      <effect operator="=" result="boolVar3" operand1="true" />
      <effect operator="+='" result="intVar1" operand1="50" />
    </effects>
    <fault name = "faultAction2">
      <effect operator="=" result="boolVar2" operand1="false" />
    </fault>
  </action>
  <action name = "action3" role = "role3"
    activity = "invoke" type = "0">
    <effects>
      <effect operator="=" result="boolVar2" operand1="true" />
      <effect operator="+='" result="intVar2" operand1="75" />
    </effects>
    <fault name = "faultAction3">
      <effect operator="=" result="boolVar2" operand1="false" />
    </fault>
  </action>
  <action name = "action4" role = "role3"
    activity = "invoke" type = "0">
    <effects>
      <effect operator="=" result="boolVar2" operand1="true" />
    </effects>
    <fault name = "faultAction4">
      <effect operator="=" result="boolVar2" operand1="false" />
    </fault>
  </action>
</actions>

Figure 4.5: the actions defined in the domain

In Figure 4.6 it is shown how the transitions of the business domain can be coded. Every transition must refer a defined action, specifying also its endpoints (the states). If there is a faulty result defined in the implemented action, then the transition must define also its behaviour in this special case.

25
Figure 4.6: the transitions defined in the domain

4.2 Description of the code

4.2.1 Internal representation of the domain

Inside the application, the representation of the business domain is made by the class Encoding; this one 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 4.3, where it is represented the org.xsr.framework.cp package.
4.2.2 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, inserted in the 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 in Figure 4.7 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 \texttt{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);
}

Figure 4.7: code for recognizing cycles

The UML schema for class CycleContents can be found in figure 4.2: 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 in Figure 4.8 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 $\xi$ 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.

```java
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);
}
```

Figure 4.8: encoding a cycle
4.2.3 Implementing the vital constraint

The algorithm to encode the vital constraint has been written by Alexander Lazovik. The algorithm, presented in figure 4.9, 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;
}

Figure 4.9: encoding the vital constraint

4.2.4 Implementing the vital-maint constraint

The algorithm that encodes the vital-maint constraint, shown in Figure 4.10, 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 rela-
tion 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.

```python
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;
}
```

Figure 4.10: encoding the vital-maint constraint
Table 4.1 shows how the effects of the reachable states can be related to the effects of the transition taken to reach them (the transitions $a_2'$ and $a_2''$ are nondeterministic). The encoding presented does not complete $s_1$'s encoding, because we would have to take into account also its representation: $\beta_1 a_1$.

Table 4.1: example of a simple encoding

### 4.2.5 Implementing the achieve-all constraint

To encode this constraint we take the subgoals in pairs, as shown in Figure 4.11, 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 nondeterministic 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);
        }
    }
}

Figure 4.11: encoding the achieve-all constraint

4.2.6 Implementing the before-then constraint

The code for this constraint is very similar to the one for the achieve-all one, as it is possible to see in Figure 4.12. In this case the operator is binary, so we have to check only two goals, g1 and g2; for this goal all the choices made along the path in order to achieving g1, have to be the same to the ones made for achieving g2, even if they are related to nondeterministic actions. In this way we express the fact that there is a temporal sequence to be followed trying to achieve the two goals.
Figure 4.12: encoding the before-then constraint

4.3 Choco engine

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. The constraints are posted, that means “inserted into” in the choco jargon, to a problem. Once the problem is solved, the variables defined to build the constraints have the values that satisfy the conditions. It is possible to get all the solutions, or the first one encountered by the solver. This library is not intended for enterprise usage, since there are no specific optimization algorithms implemented: this means that performance issues can arise, in particular conditions; moreover, the support given is not compatible with enterprise scenarios. At the moment the application uses the latest beta version available of the engine (version: 1.0b003). In this latest version some bugs concerning the handling of some basic logic operators (such as the AND operator applied to a list of constraints) have been corrected.
Chapter 5

Case Study

The presented case study considers a 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 5.1, while the xml code describing the business domain can be found in Figure 5.2 (states, variables and roles), Figure 5.3 (for the actions defined in the domain) and Figure 5.4 (for the transitions). The two types of seller considered in the domain (A and B) offer similar goods; they both refer to an external service for shipping. 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 (the transaction fails for some reason), and the user is asked to proceed with a new confirmation: this is an example of nondeterminism. When the transaction successfully finishes, the final state is reached and the process terminates.

![Graph representation of the test case](image)

Figure 5.1: the graph representation of the test case
<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>
</business-process>

Figure 5.2: States, variables and roles for the test case
Figure 5.3: actions for the test case
Figure 5.4: transitions for the test case

A generic user would want the system to build a plan for him, trying to maintain the price below 155, being sure that the transaction has a nice result, but a seller of type A must be selected. This request can be encoded in this way:

\[
\text{achieve-all( vital-maint (price \leq 155 ), vital (ASel = true ), vital (TrConcluded = true) )}
\]

5.1 Domain encoding

All the transitions in the domain are deterministic, apart from the ones starting from the state s6, that are the 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 nondeterministic, while the others in the domain follow a sequential and predefined behaviour. The fact that the two kinds of seller 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, done following the indications presented in Section 3.1, gives the following set of expressions:

- **s1**: \[\beta_1(t_1 + \beta_4(t_3 + \beta_5(t_5 + \beta_6(t_6)) + \beta_7(t_7 + \beta_8(\xi_2(t_8) + \xi_1(fault_1)))) + \beta_2(t_2 + \beta_3(t_4 + \beta_5(t_5 + \beta_6(t_6)) + \beta_7(t_7 + \beta_8(\xi_2(t_8) + \xi_1(fault_1)))) + n_1(t_2 + t_3 + t_5 + t_6)\]

- **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**: \[\emptyset\]

### 5.2 Request encoding

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 of these steps are presented as the application gives them, so 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 given by the application is composed, as usual, of a list of constraints:

\[
\begin{align*}
0.0 & + \beta_{s1\_0\_0} \times \\
& \times (\beta_{s3\_1\_0} \times \\
& \times (\beta_{s4\_2\_0} \times \\
& \times (\beta_{s5\_3\_0} \times \\
& \times (\beta_{s6\_4\_0} \times \\
& \times (xi_{s6\_5\_0} \times (1.0)) + \\
& + 1.0) + \\
& + n_1* (xi_{s6\_5\_1} \times \\
& + (contactPrice[1]) ) + \\
& + 150.0) + \\
& + 1.0) + \\
& + 5.0) + \\
& + \beta_{s1\_0\_1} \times \\
& \times (\beta_{s2\_9\_0} \times \\
& \times (\beta_{s4\_10\_0} \times \\
& \times (\beta_{s5\_11\_0} \times \\
& \times (\beta_{s6\_12\_0} \times \\
& \times (xi_{s6\_13\_0} \times (1.0)) + \\
& + 1.0) + \\
\end{align*}
\]
\[ 0.0 + \beta_{s1,0,0} \cdot \beta_{s3,1,0} \cdot \beta_{s4,2,0} \cdot \beta_{s5,3,0} \cdot (1.0) + n_1 \cdot \xi_{s6,5,1} \cdot \text{contactPrice}[1] + 150.0 \]

\[ + 1.0 \]

\[ + n_2 \cdot (\text{contactBPrice}[5] + \text{contactPrice}[1] + \text{totPrice}[150]) \]

\[ + n_4 \cdot (\text{contactPrice}[1] + \text{contactPrice}[1] + \text{totPrice}[150]) \leq 155.0 \]

\[ 0.0 + \beta_{s1,0,0} \cdot \beta_{s3,1,0} \cdot \beta_{s4,2,0} \cdot (150.0) \]

\[ + 1.0 \]

\[ + n_2 \cdot ((\text{contactBPrice}[5] + \text{contactPrice}[1] + \text{totPrice}[150]) \]

\[ + n_4 \cdot (\text{contactPrice}[1] + \text{contactPrice}[1] + \text{totPrice}[150]) \leq 155.0 \]

\[ 0.0 + \beta_{s1,0,0} \cdot \beta_{s3,1,0} \cdot \beta_{s4,2,0} \cdot (150.0) \]

\[ + 1.0 \]

\[ + n_2 \cdot ((\text{contactBPrice}[5] + \text{contactPrice}[1] + \text{totPrice}[150]) \]

\[ + n_4 \cdot (\text{contactPrice}[1] + \text{contactPrice}[1] + \text{totPrice}[150]) \leq 155.0 \]
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 unit of money. A particular case is represented by the services of type B, for which the contact costs 5 units of money. 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 this 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 named \( ASel \) set to true is to follow the part controlled by the second \( \beta \) variable in state s1: this variable, in fact, multiplies a subexpression that is put in an or relation with true.

The encoding of the goal vital \( TrConclusion = true \) gives this 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): $beta_{s1_0_0} + beta_{s1_0_1} \leq 1$
- (s5): $beta_{s5_11_0} + beta_{s5_11_1} \leq 1$
- (s5): $beta_{s5_3_0} + beta_{s5_3_1} \leq 1$
- (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 constraint gives a list of implications. Also in this case the number of constraints is increased by the presence of the converging 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)]

43
- 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 \(\beta\) variables, are considered in any case, while the non controlled ones, implemented by \(\xi\) 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.

5.3 The solution

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 Tr\textit{\text{Concluded}} = 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_1 = 0
- beta_s5_11_0 = 1
- 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.

5.4 Performance

To run the test we have used the JUnit plugin for Eclipse. The operating system is GNU-Linux (Gentoo, kernel 2.6.14), while the hardware is composed of a Centrino Sonoma platform, with the CPU operating at a clock of 2 GHz and with 1 GB of ram. The input for the application is represented by the union of the XML file representing the business domain, and an object of type org.xsrl.domain.goal.AchieveAllGoal, described in Section 4.1.2, which represents the user’s request. Both of these input elements are presented and discussed in the introduction of Chapter 5 (in Figures 5.2, 5.3 and 5.4 for the XML representation of the domain). The output of the application is given in Section 5.2 for the encoding of the requests, and in Section 5.3 for the resulting plan, expressed as a set of values for the controlled variables and the counters for the cycles. In order to provide information on the performance of our system, we have used five runs, called run 1, run 2, ..., run 5, of the system, using the same input. An additional run, let us call it run 0, has been used to let the operating system load into memory everything needed. The choco solving engine and the JUnit plugin provide a timing measurement on their own. To pick the time used to encode the domain and the request, two instances of the java.util.Calendar class have been used: the first has been instantiated before the call to the encoding method, while the second has been instantiated just after the return of the function. The difference between the two times (expressed in milliseconds) has been used as the time necessary to perform the operation. It is important to underline that the time taken to encode the domain does not take into account the time to read and parse the XML file describing the domain itself, while it comprehends the time spent to recognize and prepare the descriptions for cycles. The JUnit test time takes into account also the time to check correctness.
assertions; moreover, JUnit tests run in the Eclipse environment, so the times reported for the JUnit test have to be interpreted considering these working conditions. The collected values are reported in Table 5.1, along with the averages and the standard deviations. Every time is expressed in milliseconds. It is possible to see that, in this case, the encoding of the goal has a very little impact on the running time of the whole application: on average, in fact, it represents a 4% of the running time of the JUnit test. The standard deviation for goal encoding, if compared to the average, represents a relatively high value. This fact can be understood if we consider that the underlying system is not static. This can be seen if we consider the fifth run where, for goal encoding, the value is higher than the one given by the bootstrap run (run 0). The time taken to search for a solution (time reported by the choco engine), for this specific set of input values, represents a 40% of the running time. Considering the standard deviation for the times taken to solve the problem, we can see that the values are relatively close to the average, which in turn is very close to the median value. The times taken to encode the domain can be interpreted in the same way: the time given by the run 0 is the highest, but is quite close to the average anyway. If we consider the remaining values for the domain encoding, we can see that there is only one value higher than the average; there is only one value lower than 80 ms, given by the fifth run. In this case the median value is the more representative. The JUnit test is the most affected from the bootstrap run: the difference with the average is 70 ms. If we consider the median value, this difference raises to 80 ms. This fact can be explained if we consider that, as said before, the JUnit test runs in the Eclipse environment. Given that the test case represents a simplified concrete example, the time taken to solve the problem can be considered too high. This is due to the number of constraints posted, and this depends upon the presence of a converging point and to the semantic of the achieve-all operator: the number of subgoals inserted into the achieve-all operator, in fact, has a direct impact on the number of posted constraints.

<table>
<thead>
<tr>
<th>Run</th>
<th>Domain encoding</th>
<th>Goal encoding</th>
<th>Choco solver</th>
<th>JUnit run</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>99</td>
<td>19</td>
<td>218</td>
<td>557</td>
</tr>
<tr>
<td>1</td>
<td>84</td>
<td>19</td>
<td>190</td>
<td>470</td>
</tr>
<tr>
<td>2</td>
<td>83</td>
<td>15</td>
<td>177</td>
<td>454</td>
</tr>
<tr>
<td>3</td>
<td>85</td>
<td>16</td>
<td>198</td>
<td>476</td>
</tr>
<tr>
<td>4</td>
<td>89</td>
<td>16</td>
<td>203</td>
<td>488</td>
</tr>
<tr>
<td>5</td>
<td>77</td>
<td>25</td>
<td>183</td>
<td>468</td>
</tr>
<tr>
<td>Average</td>
<td>86.17</td>
<td>18.33</td>
<td>194.83</td>
<td>485.5</td>
</tr>
<tr>
<td>Median</td>
<td>84.5</td>
<td>17.5</td>
<td>194</td>
<td>473</td>
</tr>
<tr>
<td>Standard deviation</td>
<td>7.39</td>
<td>3.67</td>
<td>14.8</td>
<td>36.74</td>
</tr>
</tbody>
</table>

Table 5.1: times, in milliseconds, for the runs of the test case
Chapter 6

Related Work

In service oriented computing, various initiatives have been proposed to enable integration of heterogeneous systems: in particular, the web service protocol stack includes WSDL (Web Service Description Language) [20], SOAP (Single Object Access Protocol) [1] and UDDI (Universal Description Discovery and Integration) [18]. BPEL-WS (Business Process Execution Language for Web Services) [3] is a language focused on representing web service executions: in this approach the composition is known in advance. With CDL (Choreography Description Language) [9], ordered message exchanges result in accomplishing a common business goal. Complexity in service composition comes basically from three factors. First, the number of web services on the net is steadily increasing; this makes difficult to choose the right service to invoke. The second factor is represented by the fact that, in a business process, there is no single owner: this means that a change to a process has to be evaluated and accepted by all the involved parties. This means that is hard to ensure correctness at run-time, given that is hard to maintain the process correct and stable. The third factor, for composition complexity, is given by the fact that the overall behaviour of the process depends on the execution of the single web services composing the process. The behaviour of these services is not known when the process is designed, so, when designing the process, all possible service behaviours have to be taken into account. This complexity can be faced automatizing the service composition task; several approaches have been proposed to achieve these issues. Service composition can be considered similar to composition of workflows [4] [19], so the techniques developed to achieve this task can be reused: for example, in [4], a configurable approach to service composition is proposed. Some differences between workflows and services, however, arise. These differences are related to specific service-oriented computing issues: heterogeneity of the environment, absence of single ownership and control over process execution, for example. Other approaches, based on the new possibilities given by Semantic Web, work under the assumption of having rich semantic descriptions. For example, in [14] semantic matching is used for service discovery and composition, while in [15] semantic descriptions are used to compose web services in a semi-automatic way. In a pure service-oriented environment, anyway, it is more common to face a situation in which there is incomplete knowledge about service behaviour and little semantic description of these services. Another approach using semantic notations to express business roles is presented in RULE-ML [7]. This approach
has one main drawback: it does not take into account run-time monitoring for checking the business roles. The importance of planning, when facing the problem of automated web service composition, has been addressed by various authors [8, 12, 13, 16]. In [8] the authors focus their attention on information gathering and integration rather than on service composition. In [13], the Golog planner is used to compose in an automatic way semantically described services. The two latter approaches consider the goal as a set of states to reach, without considering the way in which this task is accomplished. The work presented in [16] comprehends a review covering various web services composition techniques.

Planning as encoding requests is not a completely new approach: for example this technique has been used in [6] and, using temporal constraints, in [5].

In [2], the approach considered takes into account constraint satisfaction as the way to get the best plan instance among the set generated by the planner, while an approach considering interleaved planning and execution is presented in [11]. In this work, anyway, the planner module does not use constraint programming to prepare a plan.

This work deals with nondeterminism introducing a new type of variable, called non controlled variable; this makes it different from the other works that deal this situation using “external constraints”, as for example in [17], in which an overview about constraints and constraint satisfaction is given.
Chapter 7

Conclusions

When a business process is made of a set of independent web services, the problems related to the composition of these services arise. One of the most important problems to solve is to automatically prepare a plan, whose execution enables the process to satisfy an user’s request. The presented solution to the planning problem in a domain, built composing several web services, is based upon the encoding of the user’s requests against a standardized business process; this step gives a set of numerical constraints whose solution, given by a solving engine, represents the plan that the system has to follow, in order to achieve the goal defined by the user. This goal has to be expressed following the XSRL specifications. The selected engine used to solve the constraints encoded, at the moment, is choco. As we said in Chapter 4, it is an open-source library that supports basic numerical constraints, built on integer or real values. The solution given by this engine is composed of a set of values for the variables controlling the choices that the system can take, trying to achieve the user’s request. We assume that the system can face nondeterministic behaviours: the result of some transitions can depend on events that are not under the control of the application. In this scenario the application can only be aware of this fact, taking it into account when encoding the constraints. A number of challenges remain open for future work. The work proposed implements a subset of the XSRL definitions; one of the next steps to take is to implement the atomic goals and the preference goals (in development at the moment). Another point that could be extended is the one regarding the operators supported in the request. A performance test, related to the nature of the goal and to the nature of the business domain, should be developed, along with a formal analysis of the algorithms presented. Another important test to be taken could cover different solving engines (different from choco), taking into account also versioning of the engines. A graphical interface to model the business domain and the user’s request could speed up the testing process, and lower the usage difficulty for the final users. The presented application does not consider interleaving of planning and execution, yet; this is a crucial point in a real-life scenario.
Bibliography


Chapter 8

Appendix A: Introduzione

La Computazione Orientata al Servizio è un nuovo paradigma nella programmazione distribuita; ciò è dato dal fatto che la focalizzazione tende a spostarsi dall’“oggetto” al “servizio”: un servizio è un’entità autonoma che espone delle funzionalità verso un sistema esterno. L’implementazione di questa definizione astratta in una rete e la possibilità per i servizi di scambiare messaggi basati su XML in modalità asincrona, dà come risultato un web service, che tipicamente specializza le sue funzionalità in un compito particolare. Per esempio possiamo avere un web service che gestisce le informazioni riguardanti le condizioni metereologiche, un altro che gestisce le condizioni del traffico e così via. Mettendo assieme le descrizioni di una serie di web services, otteniamo la definizione di un sistema distribuito che è in grado di gestire compiti complessi. Le varie problematiche che vanno affrontate nella gestione di queste composizioni di servizi sono conosciute come problema della composizione di servizi. Questo comprende vari aspetti che vanno presi in considerazione: la selezione di servizi disponibili da comporre per raggiungere un obiettivo complesso, e la composizione automatica di questi servizi sono due esempi. Il fatto che questi compiti vengono svolti in maniera automatica permette di ottenere orchestrazione e coreografia o, più in generale, la composizione di servizi. Ciò significa che è possibile costruire servizi, basati su servizi individuali e indipendenti, a valore aggiunto, nel senso che, cooperando, i servizi di base possono gestire compiti complessi. Il lavoro presentato si concentra sul dare all’utente la possibilità di esprimere richieste complesse verso una composizione di web services descritte come un dominio di lavoro. L’approccio proposto consiste nel codificare il dominio di lavoro e la richiesta dell’utente come vincoli matematici. Risolvere questi vincoli significa trovare un piano, contenente la sequenza di azioni definite nel dominio, la cui esecuzione soddisfa la richiesta dell’utente.

La descrizione di un web service è tipicamente data per mezzo di un documento WSDL (Web Service Description Language). WSDL [20] è un linguaggio basato su XML che rappresenta in modo astratto e strutturato le operazioni che un web service può eseguire, definendo nel contempo il formato che un utente esterno deve usare per effettuare le richieste. Le informazioni relative ai provider che forniscono un’implementazione di un particolare tipo di web service sono mantenute in un registro di servizi (un database ordinato ed indicizzato). UDDI (Universal Description, Discovery, Integration) [18], . UDDI, un registro di servizi sponsorizzato da OASIS (Organization for the Advancement
of Structured Information Standards), lavora come un web service che gestisce informazioni riguardanti sia i fornitori di servizi, sia i tipi di implementazioni date da questi, assieme a meta-dati riguardanti il servizio stesso. Una volta che l’utente (il client) ottiene la descrizione del web service, direttamente dal web service o attraverso un registro UDDI, può mandare richieste al web service. Il livello che rappresenta questa struttura client-web service è rappresentata dal livello SOAP [1], il quale definisce il modo in cui una richiesta ad un web service e la risposta da parte del web service devono essere scritte. SOAP si basa a sua volta sul protocollo HTTP, attraverso il quale i documenti XML, che rappresentano richieste e risposte, vengono trasmessi.

8.1 Organizzazione della tesi

Nel Capitolo 2 la struttura XSRL, su cui si basa l’applicazione, viene presentata e descritta; un semplice esempio di un dominio di lavoro e di una richiesta vengono forniti nella Sezione 2.2. Le idee che stanno dietro la fase di codifica sono presentate nel Capitolo 3, assieme alla codifica dell’esempio presentato nel Capitolo 2. Nel Capitolo 4 gli algoritmi, che implementano le codifiche proposte nel Capitolo 3, sono presentati e descritti. Il Capitolo 5 presenta un caso di studio; lo stato dell’arte è presentato nel Capitolo 6, mentre le conclusioni e gli sviluppi futuri sono presentati nel Capitolo 7.
Quando un processo è costituito da un insieme di web services indipendenti, varii problemi relativi alla composizione di questi servizi vengono alla luce. Uno dei problemi più importanti da risolvere è quello che riguarda la preparazione automatica di un piano, la cui esecuzione fa sì che il processo soddisfi la richiesta di un utente. La soluzione proposta, relativa al problema della pianificazione in un dominio composto da svariati web services, si basa sulla codifica delle richieste degli utenti verso un processo standardizzato; questo passo dà come risultato un insieme di vincoli matematici la cui soluzione, data da un motore di risoluzione, rappresenta il piano che il sistema deve seguire per raggiungere il goal definito dall’utente. Questo goal deve essere espresso seguendo le specifiche XSRL. Il motore scelto per risolvere i vincoli dati dalla codifica, al momento, è choco. Come detto nel Capitolo 4, choco è un libreria open-source che supporta vincoli numerici di base, comprendenti variabili intere o reali. La soluzione data dal motore è composta da un insieme di valori per le variabili che controllano le scelte che il sistema è libero di compiere, cercando di soddisfare le richieste dell’utente. Un’assunzione che viene fatta riguarda il fatto che il sistema deve tener conto di comportamenti nondeterministici: i risultati di certe transizioni dipendono da eventi che non sono sotto il controllo dell’applicazione. In questo scenario l’applicazione può solo tener conto di questo fatto, trattandolo in modo particolare nella codifica. Una serie di sfide rimangono aperte per gli sviluppi futuri. Il lavoro presentato implementa un sottoinsieme delle definizioni XSRL; uno dei prossimi passi consiste nell’implementazione dei goal di tipo atomici e di quelli che esprimono una preferenza (al momento in fase di sviluppo). Un altro punto che potrebbe essere esteso è quello riguardante gli operatori matematico-logici supportati nella richiesta. Un test riguardante le prestazioni, che tenga conto della natura del goal e di quella del dominio di lavoro, dovrebbe essere sviluppato, assieme ad un’analisi formale degli algoritmi presentati. Un altro test interessante potrebbe riguardare motori di risoluzione diversi da choco, considerando anche diverse versioni degli stessi. Un’interfaccia grafica che permetta di modellare il dominio di lavoro e la richiesta dell’utente potrebbe accelerare la fase di testing e abbassare il livello di difficoltà nell’uso dell’applicazione per gli utenti finali. L’applicazione presentata non considera ancora l’intervallamento di pianificazione ed esecuzione; questo punto è cruciale in uno scenario reale.