klase

Platform for the composition of Web Services: Experiments

ITC-irst
Università di Trento

Abstract. Testing is a crucial activity to establish the practical applicability of automated composition, verification and monitoring techniques, to validate theoretical results on their behavior, to track down bottlenecks and inspire new solutions. During the course of the project, we tackled a variety of scenarios, some coming from real applications and inspired by industrial contacts, some designed ad-hoc to have specific features. In this deliverable, we describe in turn the various scenarios, the tests we ran over them, and their results.

Document Identifier	Deliverable D7.3
Project	MIUR-FIRB project RBNE0195K5 "Knowledge Level Automated Software Engineering"
Version	v1.0
Date	October 31, 2006
State	Final
Distribution	Public

Acknowledgements.

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

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

Executive Summary

During the course of the Klase project, several advanced techniques have been designed and implemented to support the automated composition, verification and monitoring of processes. Establishing the practical usability of these approaches, identifying their limits, and validating the theoretical claims on their behavior is crucial to the actual enactment of tools that can be successfully integrated within existing SOC architectures.

For these reasons, a thorough experimental evaluation of each of the techniques developed in time is fundamental. In particular, during the course of the four years of the project, we designed and carried out experiments following two main directions:

1. we took inspiration as much as possible from existing real-life scenarios, either commonly known ones, or scenarios provided to us by industrial partners. These scenarios gave us grasp on the actual requirements and issues which arise when composing, verifying and monitoring web services.

2. we designed several "synthetic" scenarios, where specific features are present, affecting the complexity of the composition/verification/monitoring. These scenarios allow us to isolate the way such features reflect on the practical behavior of our techniques, therefore identifying possible bottlenecks and identifying directions for improvement.

This deliverable provides a structured presentation of these two families of adopted scenarios, summarizing the tests that we ran on them, and their results. In particular, for each scenario, an informal description is given, accompanied by a more formal description of its modeling, and by a detailed description of the (composition, verification, monitoring) tests executed on it.

Contents

1 Realistic scenarios **1**

1.1 The P&S scenario 1

1.1.1 Description 1

1.1.2 Experiments: composition 6

1.2 The VTA scenario 16

1.2.1 Description 16

1.2.2 Experiments: composition 18

1.2.3 Experiments: verification 20

1.2.4 Experiments: monitoring 22

1.3 The WMO scenario 23

1.3.1 Description 23

1.3.2 Experiments: verification 24

1.4 The VOS scenario 26

1.4.1 Description 26

1.4.2 Experiments: composition 29

1.4.3 Experiments: monitoring 32

1.4.4 Experiments: verification 34

1.5 The Health-care scenario 35

1.5.1 Description 35

1.5.2 Experiments: verification 39

1.6 The Loan Approval scenario 41

1.6.1 Description 41

1.6.2 Experiments: verification 43

2 Synthetic scenarios **45**

2.1 Approach 45

2.2 Simple services 46

2.2.1 Description 46

2.2.2 Experiments: composition 46

2.3 Binary unbalanced protocols 49

2.3.1 Description 49

2.3.2 Experiments: composition 49

2.4 Binary balanced protocols 51

2.4.1 Description 51
2.4.2 Experiments: composition 52
2.5 N-ary balanced protocols 53
2.5.1 Description 53
2.5.2 Experiments: composition 54
2.5.3 Correlation between results 55

3 History of the Deliverable **57**
3.1 1st Year 57
3.2 2nd Year 57
3.3 3rd Year 58
3.4 4th Year 58

Chapter 1

Realistic scenarios

1.1 The P&S scenario

1.1.1 Description

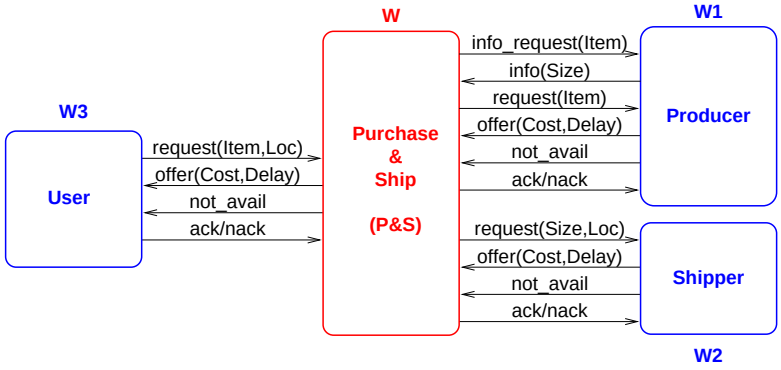


Figure 1.1: The P&S scenario.

The first scenario we consider will be named "P&S", for "Producer and Shipper". We assume that two independent existing services: have been published: a furniture purchase service Producer, and a delivery service Shipper. These services are not atomic, i.e., they cannot be executed in a single request-response step, but they are stateful processes, and they require to follow an interaction protocol that involve different sequential and conditional steps. In our example, the Producer accepts requests for given products. If the requested product is available, it provides information about its size and, if the requester confirm the interest to buy, the Producer makes an offer with a cost and a production time. This offer can be accepted or refused by the requester. In both cases the Producer terminates execution, with success or failure, respectively.

Also the behavior of the Shipper is stateful. Once it receives a request to deliver an

object of a given size to a certain location, the Shipper may refuse to process the request, or may produce an offer where the cost and the time to deliver are specified. In the latter case, the invoker can either confirm the order, or cancel it.

Similarly, the User performs a two-step interaction with the P&S: first it sends his/her request, then it gets either a refusal or an offer, and finally (in the latter case) it either confirms or dis-confirms the request.

This scenario lends very well to composition, where the goal is to implement a composed service for furniture purchasing and delivering, the P&S service, by composing two independent existing services: a furniture purchase service Producer, and a delivery service Shipper. The P&S service should allow a User to ask for desired products that should be delivered at a desired location, see Fig. 1.1.

The goal of the P&S is to sell a product at a destination, as requested by a customer. To achieve this, the P&S has to interact with the customer on the one side and with Producer and Shipper on the other side, trying to reach a situation where the three interactions reach a successful completion, i.e., three final confirmations are obtained. Clearly, the goal of selling a product at destination may be not always satisfiable by the P&S, since its achievement depends on decisions taken by third parties that are out of its control: the Producer and the Shipper may refuse to provide the service for the requested product and location (e.g., since the product is not available or the location is out of the area of service of the shipper), and the customer may refuse the offer by the P&S (e.g., since it is too expensive). If this happens, the P&S should step back from both orders and it should not commit to the customer. Indeed, we do not want the P&S to buy something that cannot be delivered, as well as we do not want it to promise a product at destination that it will not be able to buy.

The order in which the interactions with the different services are interleaved in the implementation of the P&S is critical. For instance, when the P&S gets a request for a given item from a customer, it has to obtain the size of the item from the producer before it can call the shipper. Two offers, from the Producer and from the Shipper are necessary to the P&S in order to make an overall offer to the customer. Moreover, the offers from Producer and Shipper can be accepted by the P&S only after the customer has accepted the offer from the P&S. The necessity to figure out and realize all these constraints makes the implementation of the P&S a complex task, also in very simple scenarios like the one described above.

We assume that the component and composed services are represented using the WS-BPEL language. WS-BPEL (the "Business Process Execution Language") [ACD+03] is one of the emerging standards for describing the stateful behavior of the service. In WS-BPEL, a set of atomic communication operations (i.e., invoke, receive, and reply activities) are combined within a workflow that defines the process implemented by the stateful service. The atomic communications correspond to atomic web service calls, and are defined in a WSDL (Web Service Description Language [CCMW]) specification. WSDL is a standard language for describing operations implemented as Web services along with

the input and output data of these operations.

There are two flavors of WS-BPEL, namely *abstract* WS-BPEL specifications, which are used to publish the interaction protocol with external web services, and *executable* WS-BPEL programs, that are used to implement the process defining a service. Executable WS-BPEL programs can be executed by standard engines, such as the Active BPEL Open Engine or the Oracle BPEL Process Manager [Act, Ora].

The composition problem can therefore be described as follows: given a set of *abstract* WS-BPEL specifications describing the (interactions with) the component services, and given some composition requirements that describes the desired functionalities of a composed service, construct the *executable* WS-BPEL that implements a composed service that, when executed, satisfies the requirements.

Fig.1.2 presents the WSDL specification for the Producer, abridged from technical details irrelevant to our discussion (in particular, we omit name-space management, operation bindings and aliases). The WSDL specification starts with the definition of the data types used in the interactions. In the case of the Producer, they are the requested `Item` and its `Size`, `Cost`, and production `Delay`. The actual definition of these data types is not relevant to our purposes, and is also omitted from the WSDL specification.

The WSDL specification then describes the structure of the messages relevant for the interactions with the Producer. According to the specification, a `requestMsg` message contains the requested article, `art`. The `infoMsg` and `offerMsg` messages contain, respectively, the `size` and the production `cost` and `delay` for an article. The other three messages (`unavailMsg`, `ackMsg`, `nackMsg`) do not carry data values.

Then, the WSDL specification defines the invocation and reply operations provided by the service. Operations are collected in port types, that are associated to different communication channels of the producer service with its partners. In our example, we define two port types, namely `P_PT` for the incoming requests and messages and `PC_PT` for the outgoing messages from the producer to the invoking service.Finally, the WSDL specification defines bi-directional links between the service and its partners. In our case, there is only one of such links, between the Producer and the customer invoking it.

Figure 1.3 shows the abstract WS-BPEL of the Producer. It starts with a declaration of the links with external parties that are exploited within the process. In this case, only one external partner exists, the `client` of the producer. The type of the link is `P_PLT` (see the WSDL file). Then the variables that are used in input/output messages and their data types are declared (see lines 5–13). The rest of the abstract WS-BPEL specification (lines 14-53) describes the interaction flow.

The Producer is activated by a customer request of information for a specified product (see the `receive` instruction at line 15) The item specified by the requester is stored in variable `requestVar`. The `operation="request"` identifies which WSDL operation is performed. The `partnerLink="client"` and `portType="P_PT"` fields serve to identify the link along which the request is received (of course, these fields be-

```
1  ⟨definitions name = "Producer"⟩
2    ⟨types⟩
3      ⟨schema targetNamespace = "http://www.astroproject.org/Producer"⟩
4        ⟨element name = "Item"⟩
5          .......
6        ⟨/element⟩
7        ⟨element name = "Size"⟩
8          .......
9        ⟨/element⟩
10       ⟨element name = "Location"⟩
11         .......
12       ⟨/element⟩
13       ⟨element name = "Cost"⟩
14         .......
15       ⟨/element⟩
16       ⟨element name = "Delay"⟩
17         .......
18       ⟨/element⟩
19     ⟨/schema⟩
20   ⟨/types⟩
22   ⟨message name = "requestMsg"⟩
23     ⟨part name = "art" type = "Item"/⟩
24   ⟨/message⟩
25   ⟨message name = "infoMsg"⟩
26     ⟨part name = "size" type = "Size"/⟩
27   ⟨/message⟩
28   ⟨message name = "offerMsg"⟩
29     ⟨part name = "cost" type = "Cost"/⟩
30     ⟨part name = "delay" type = "Delay"/⟩
31   ⟨/message⟩
32   ⟨message name = "ackMsg"⟩
33   ⟨message name = "nackMsg"⟩
34   ⟨message name = "unavailMsg"⟩
36   ⟨portType name = "P_PT"⟩
37     ⟨operation name = "request"⟩
38       ⟨input message = "requestMsg"/⟩
39     ⟨/operation⟩
40     ⟨operation name = "ack"⟩
41       ⟨input message = "ackMsg"/⟩
42     ⟨/operation⟩
43     ⟨operation name = "nack"⟩
44       ⟨input message = "nackMsg"/⟩
45     ⟨/operation⟩
46   ⟨/portType⟩
48   ⟨portType name = "PC_PT"⟩
49     ⟨operation name = "info"⟩
50       ⟨input message = "infoMsg"/⟩
51     ⟨/operation⟩
52     ⟨operation name = "unavail"⟩
53       ⟨input message = "unavailMsg"/⟩
54     ⟨/operation⟩
55     ⟨operation name = "offer"⟩
56       ⟨input message = "offerMsg"/⟩
57     ⟨/operation⟩
58   ⟨/portType⟩
60   ⟨plnk:partnerLinkType name = "P_PLT"⟩
61     ⟨plnk:role name = "P_Service"⟩
62       ⟨plnk:portType name = "P_PT"/⟩
63     ⟨plnk:role⟩
64     ⟨plnk:role name = "P_Customer"⟩
65       ⟨plnk:portType name = "PC_PT"/⟩
66     ⟨plnk:role⟩
67   ⟨plnk:partnerLinkType⟩
69 ⟨/definitions⟩
```

Figure 1.2: The WSDL for the Producer process.

```
1  ⟨process name = "Producer"⟩
2    ⟨partnerLinks⟩
3      ⟨partnerLink name = "client" partnerLinkType = "P_PLT" myRole = "P_PT" partnerRole = "PC_PT"/⟩
4    ⟨/partnerLinks⟩
5    ⟨variables⟩
6      ⟨variable name = "requestVar" messageType = "requestMsg"/⟩
7      ⟨variable name = "unavailVar" messageType = "unavailMsg"/⟩
8      ⟨variable name = "infoVar" messageType = "infoMsg"/⟩
9      ⟨variable name = "offerVar" messageType = "offerMsg"/⟩
10     ⟨variable name = "ackVar" messageType = "ackMsg"/⟩
11     ⟨variable name = "nackVar" messageType = "nackMsg"/⟩
12     ⟨variable name = "availVar" messageType = "xsd : boolean"/⟩
13   ⟨/variables⟩
14   ⟨sequence name = "main"⟩
15     ⟨receive name = "getRequest" operation = "request" variable = "requestVar" partnerLink = "client" portType = "P_PT"/⟩
16     ⟨assign name = "setAvail"⟩
17       ⟨copy⟩⟨from opaque = "yes"/⟩⟨to variable = "availVar"/⟩⟨/copy⟩
18     ⟨/assign⟩
20     ⟨switch name = "checkAvail"⟩
21       ⟨case condition = "bpws : getVariableData('avail') = xsd : False"⟩
22         ⟨invoke name = "sendUnavail" operation = "unavail" inputVariable = "unavailVar" partnerLink = "client"
23                 portType = "PC_PT"/⟩
24       ⟨/case⟩
25       ⟨otherwise
26         ⟨assign name = "prepareInfo"⟩
27           ⟨copy⟩⟨from opaque = "yes"/⟩⟨to variable = "infoVar" part = "size"/⟩⟨/copy⟩
28         ⟨/assign⟩
29         ⟨invoke name = "sendInfo" operation = "info" inputVariable = "infoVar" partnerLink = "client"
30                 portType = "PC_PT"/⟩
31         ⟨pick⟩
32           ⟨onMessage name = "getNack" operation = "nack" variable = "nackVar" partnerLink = "client"
33                      portType = "P_PT"/⟩
34           ⟨/onMessage⟩
35           ⟨onMessage name = "getAck" operation = "ack" variable = "ackVar" partnerLink = "client" portType = "P_PT"⟩
36             ⟨assign name = "prepareOffer"⟩
37               ⟨copy⟩⟨from opaque = "yes"/⟩⟨to variable = "offerVar" part = "delay"/⟩⟨/copy⟩
38               ⟨copy⟩⟨from opaque = "yes"/⟩⟨to variable = "offerVar" part = "cost"/⟩⟨/copy⟩
39             ⟨/assign⟩
40             ⟨invoke operation = "sendOffer" inputVariable = "offerVar" partnerLink = "client" portType = "PC_PT"⟩
41             ⟨pick⟩
42               ⟨onMessage name = "getAccepted" operation = "ack" variable = "ackVar" partnerLink = "client"
43                          portType = "P_PT"/⟩
44               ⟨/onMessage⟩
45               ⟨onMessage name = "getRefused" operation = "nack" variable = "nackVar" partnerLink = "client"
46                          portType = "P_PT"/⟩
47               ⟨/onMessage⟩
48             ⟨/pick⟩
49           ⟨/onMessage⟩
50         ⟨/pick⟩
51       ⟨/otherwise⟩
52     ⟨/switch⟩
53   ⟨/sequence⟩
54 ⟨/process⟩
```

Figure 1.3: The abstract WS-BPEL for the Producer process.

come useful when there are more than one partners for the WS-BPEL process). At lines 16-18, Producer decides on the availability of the requested product; notice that the internal logic governing this computation is purposely not disclosed in the abstract WS-BPEL, as the source of the data is marked as "`opaque`". Then, in the `switch` activity named `checkAvail`, the service decides on the basis of the availability. If the Producer is not available, it signals this to its partner (see activity `invoke` on line 22) and terminate; otherwise, it prepares and send the information regarding the size of the required item (lines 26-29). The information regarding the size is (internally and opaquely) computed within the `assign` statement named `prepareInfo`. After sending the information, the Producer suspends (instruction `pick` at line 31), waiting for the customer either to acknowledge or to refuse to proceed to buy the specified product. If the customer refuses to proceed (statement `onMessage` on line 32), the Producer terminates the execution. If, otherwise, a message is received that corresponds to operation `ack` (line 35), meaning that the customer confirms its interest in the item, then the Producer prepares and sends and offer to the customer, which contains a cost, and an expected delivery time (lines 35-40). Again, opaqueness is used to hide the actual way in which the production time (line 37) and the cost (line 38) are computed. After sending the offer, the Producer suspends waiting for a final acknowledgment of refusal of the offer by the client. and terminates after receiving it either with success or failure, respectively (lines 41-48).

The WSDL and abstract WS-BPEL specifications for the Shipper are similar to that of the Producer, so we do not report them.

The abstract WS-BPEL of the Producer and of the Shipper are published, and available as input to potential users of the services, as well as to the composition task. In order to be able to complete the composition task, we also need to have as input a description of the interaction protocol that the User will carry out with the composite service: the implementation of the P&S will depend on the protocols of all of the interacting partners, i.e. the Producer, the Shipper and the User.

Fig. 1.4 shows the abstract WS-BPEL of the User. After determining the content of its request (lines 14-17), the user invokes the composite service, and suspends for a reply (line 18). The reply can signal that the request cannot be satisfied, or propose an offer. In the first case, the service simply terminates (line 23). In the second case, the user then decides whether to accept or not such offer, and signals his decision before terminating (lines 24-40).

1.1.2 Experiments: composition

Here, we tackle the automated composition task, with the goal to generate the executable code of the P&S, which sells a product at destination whenever possible, and, if this is not possible, guarantees that no pending commitments with the partners are left. In achieving this goal, the P&S has to respect the protocols described by the three abstract WS-BPEL specifications of Producer, Shipper, and P&S, and must obey a constraint

```
1  ⟨process name = "User"⟩
2    ⟨partnerLinks⟩
3      ⟨partnerLink name = "client" partnerLinkType = "U_PLT" myRole = "U_PT" partnerRole = "UC_PT"/⟩
4    ⟨/partnerLinks⟩
5    ⟨variables⟩
6      ⟨variable name = "requestVar" messageType = "requestMsg"/⟩
7      ⟨variable name = "unavailVar" messageType = "unavailMsg"/⟩
8      ⟨variable name = "offerVar" messageType = "offerMsg"/⟩
9      ⟨variable name = "ackVar" messageType = "ackMsg"/⟩
10     ⟨variable name = "nackVar" messageType = "nackMsg"/⟩
11     ⟨variable name = "acceptsVar" messageType = "xsd : boolean"/⟩
12   ⟨/variables⟩
13   ⟨sequence name = "main"⟩
14     ⟨assign name = "prepareRequest"⟩
15       ⟨copy⟩⟨from opaque = "yes"/⟩⟨to variable = "requestVar" part = "art"/⟩⟨/copy⟩
16       ⟨copy⟩⟨from opaque = "yes"/⟩⟨to variable = "requestVar" part = "loc"/⟩⟨/copy⟩
17     ⟨/assign⟩
18     ⟨invoke name = "sendRequest" operation = "request" inputVariable = "requestVar" partnerLink = "client"
19         portType = "U_PT"⟩
20     ⟨pick⟩
21       ⟨onMessage name = "getUnavail" operation = "unavail" variable = "unavailVar" partnerLink = "client"
22           portType = "UserC_PT"⟩
23       ⟨/onMessage⟩
24       ⟨onMessage name = "getOffer" operation = "offer" variable = "offerVar" partnerLink = "client"
25           portType = "UserC_PT"⟩
27         ⟨assign name = "prepareAcceptOrReject"⟩
28           ⟨copy⟩⟨from opaque = "yes"/⟩⟨to variable = "acceptsVar" ⟩⟨/copy⟩
29         ⟨/assign⟩
30         ⟨switch name = "checkAcceptance"⟩
31           ⟨case condition = "bpws : getVariableData('acceptsVar') = xsd : False" ⟩
32             ⟨invoke name = "refused" operation = "nack" inputVariable = "nackVar" partnerLink = "client"
33                 portType = "U_PT"⟩
34           ⟨/case⟩
35           ⟨otherwise
36             ⟨invoke name = "accepted" operation = "ack" inputVariable = "ackVar" partnerLink = "client"
37                 portType = "U_PT"⟩
38           ⟨/otherwise⟩
39         ⟨/switch⟩
40       ⟨/onMessage⟩
41     ⟨/pick⟩
42   ⟨/sequence⟩
43 ⟨/process⟩
```

Figure 1.4: The abstract WS-BPEL for the User process.

that associates the cost paid by the User with the prices proposed by the Producer and Shipper, including a profit margin.

Figure 1.5 shows a possible implementation of the P&S as an executable WS-BPEL process.

As we see, the service feature three `partnerLink`s, one for each of the three component services it has to interact with. Its variables refer to the typed declared in the WSDL files associated to each component; for instance, the `U_RequestVar` has the `requestMsg` type declared for the User, which contains an item and a location, while `S_RequestVar` has the `requestMsg` type declared for the Shipper, and contains a size and a location.

The P&S is activated by a request from the customer — the request specifies the desired product and delivery location (line 28). The P&S asks the Producer for information about the product, namely its size (line 35). If the item can be produced, then the Producer gives the P&S information about the size, the cost, and the time required for the service (line 44). The P&S then asks the delivery service the price and time needed to transport an object of such a size to the desired locations (line 57). If the Shipper makes an offer to deliver the item, then the P&S sends an aggregated offer to the customer. This offer takes into account the overall production and delivery costs and time (line 94). If the customer sends a confirmation, the final acknowledge is sent both to the Producer and to the Shipper (lines 98-107), via a `flow` construct that specifies that the associated `invoke` activities can be executed asynchronously in parallel. As a consequence, all three component services terminate successfully.

While we have described only the nominal flow of interaction, the executable WS-BPEL of the P&S takes also into account the cases in which the service cannot succeed, i.e., the cases in which either the Producer cannot provide the product, or the Shipper cannot deliver, or the customer does not accept the offer, by canceling the pending orders (see lines 41, 72, and 112-121).

Notice that the executable WS-BPEL for the P&S is much more complex than any of the abstract components. This is related to two aspects. First, while the abstract WS-BPELs only describe the protocols exported by the partners, the executable P&S must implement all the communication with the three component services (the User, the Producer and the Shipper), as well as all the computations over the internal variables, e.g., those computing the total cost and time for the offer to the customer. Second, P&S must deal with the fact that our requirement to sell the item may be not always satisfiable by the composed service, since it depends on decisions taken by third parties that are out of its control: the Producer and the Shipper may refuse to provide the service for the requested product and location (e.g., since the product is not available or the location is out of the area of service of the shipper), and the User may refuse the offer by the P&S (e.g., since it is too expensive). If this happens, we require that the P&S should step back from both orders, since we do not want the P&S to buy something that cannot be delivered, as well as we do not want it to spend money for delivering an item that we cannot buy. In terms of abstract

WS-BPEL, this means the User should terminate execution either at line 32 or at line 21 in Figure 1.4), and the Producer should terminate either at line 45 or at line 32 in Figure 1.3.

In our experiments, we formulated the high-level requirements stated above using three different languages, which are amenable to making use of three different composition techniques. First, we modeled the goal at the *ground level*, that is mentioning explicitly the relationships amongst the values of data that the customer must exchange with P&S. This modeling corresponds to a ground representation of the involved services, where actual values are instantiated, and to a search technique that considers such a modeling. Second, we adopted an automated mechanism to convert our requirement to a *knowledge-level* goal: a goal where no explicit mention is made about actual variable values, but only the relationship between variables is expressed in terms of "knowledge about values". This technique allows us to relieve from the computational burden of considering every possible combination of actual values, and therefore completely solves the issues of scalability against different data ranges. However, the knowledge-based representation comes at the cost of a non-trivial construction of a so-called knowledge base, and forces us to adopt certain restrictive assumptions. Our third approach starts from a different requirement language altogether, expressing relationship amongst data in terms of a network called "*data-net*". This allows a clean-cut distinction between the portion of requirements that specifies the control logics of the composed service, and the one that specifies the relationship amongst data. The following subsections detail each experimentation in turn.

Ground-level composition

Taking the representation of component services in terms of STS (see [PTB05, PBB+04]), requirements like the one we expressed can be formulated in terms of reachability over sets of states of the component services. In particular, we can structure our requirements as the conjunction of requirements over the control flow, and requirements over the control.

In the control part, we specify that, in case all partners are available, they should all terminate in a "successful" state, i.e. a state where the final agreement to buy or sell has been achieved with the P&S. Otherwise, each partner must either remain inactive, or terminate in a "failure" state where the service is aware of the impossibility to agree on the buy/sell and any commitment to buy or sell has been withdrawn. In order to more compactly define this part of the requirements, we introduce `Fail` and `Succ` macros for each partner, which correspond to the set of program counter values that identify failing and successful terminations respectively; for instance,

`Producer.Succ := (Producer.pc := DONE_getAccepted)`, and `Producer.Fail := (Producer.pc IN {DONE_getRefused,DONE_Nack})`. Similarly, we introduce an `Init` macro that indicates, for each component, that the

```
1  ⟨process name = "PandS_Executable"⟩
2    ⟨partnerLinks⟩
3      partnerLink myRole = "PandS_Service" name = "client" partnerLinkType = "tns : PandS_PLT"
4              partnerRole = "PandS_Customer"⟩
5      partnerLink myRole = "Shipper_Customer" name = "PandS_Shipper" partnerLinkType = "Shipper : S_PLT"
6              partnerRole = "Shipper_Service"⟩
7      partnerLink myRole = "Producer_Customer" name = "PandS_Producer" partnerLinkType = "Producer : P_PLT"
8              partnerRole = "Producer_Service"⟩
9    ⟨/partnerLinks⟩
10   ⟨variables⟩
11     ⟨variable name = "U_RequestVar" messageType = "User : requestMsg"/⟩
12     ⟨variable name = "U_OfferVar" messageType = "User : offerMsg"/⟩
13     ⟨variable name = "U_AckVar" messageType = "User : ackMsg"/⟩
14     ⟨variable name = "U_NackVar" messageType = "User : nackMsg"/⟩
15     ⟨variable name = "P_RequestVar" messageType = "Producer : requestMsg"/⟩
16     ⟨variable name = "P_UnavailVar" messageType = "Producer : unavailMsg"/⟩
17     ⟨variable name = "P_InfoVar" messageType = "Producer : infoMsg"/⟩
18     ⟨variable name = "P_NackVar" messageType = "Producer : nackMsg"/⟩
19     ⟨variable name = "P_AckVar" messageType = "Producer : ackMsg"/⟩
20     ⟨variable name = "P_OfferVar" messageType = "Producer : offerMsg"/⟩
21     ⟨variable name = "S_AckVar" messageType = "Shipper : ackMsg"/⟩
22     ⟨variable name = "S_NackVar" messageType = "Shipper : nackMsg"/⟩
23     ⟨variable name = "S_RequestVar" messageType = "Shipper : requestMsg"/⟩
24     ⟨variable name = "S_UnavailVar" messageType = "Shipper : unavailMsg"/⟩
25     ⟨variable name = "S_OfferVar" messageType = "Shipper : offerMsg"/⟩
26   ⟨/variables⟩
27   ⟨sequence⟩
28     ⟨receive operation = "request" variable = "U_RequestVar" partnerLink = "client" portType = "PandS_PT"⟩
29     ⟨assign ⟩
30       ⟨copy⟩
31         ⟨from variable = "U_RequestVar" part = "art"⟩
32         ⟨to variable = "P_RequestVar" part = "art"/⟩
33       ⟨/copy⟩
34     ⟨/assign⟩
35     ⟨invoke operation = "request" inputVariable = "P_RequestVar" partnerLink = "PandS_Producer"
36           portType = "Producer : P_PT"⟩
37     ⟨pick⟩
38       ⟨onMessage operation = "unavail" variable = "P_UnavailVar" partnerLink = "PandS_Producer"
39                 portType = "Producer : PC_PT"⟩
40         ⟨sequence ⟩
41           ⟨invoke operation = "unavail" inputVariable = "P_UnavailVar" partnerLink = "client" portType = "PandSC_PT"⟩
42         ⟨/sequence⟩
43       ⟨/onMessage⟩
44       ⟨onMessage operation = "getP_Info" operation = "info" variable = "P_InfoVar" partnerLink = "PandS_Producer"
45                 portType = "Producer : PC_PT"⟩
46         ⟨sequence⟩
47           ⟨assign ⟩
48             ⟨copy⟩
49               ⟨from variable = "P_InfoVar" part = "size"⟩
50               ⟨to variable = "S_RequestVar" part = "size"/⟩
51             ⟨/copy⟩
52             ⟨copy⟩
53               ⟨from variable = "U_RequestVar" part = "loc"⟩
54               ⟨to variable = "S_RequestVar" part = "loc"/⟩
55             ⟨/copy⟩
56           ⟨/assign⟩
57           ⟨invoke operation = "request" inputVariable = "S_RequestVar" partnerLink = "PandS_Shipper"
58                 portType = "Shipper : S_PT"⟩
59           ⟨pick⟩
60             ⟨onMessage operation = "unavail" variable = "S_UnavailVar" partnerLink = "PandS_Shipper"
61                       portType = "Shipper : ShipperC_PT" ⟩
62               ⟨sequence⟩
63                 ⟨flow⟩
64                   ⟨sequence⟩
65                     ⟨invoke operation = "unavail" inputVariable = "U_Unavail" partnerLink = "client"
66                           portType = "PandSC_PT"⟩
67                   ⟨/sequence⟩
68                   ⟨sequence⟩
69                     ⟨invoke operation = "nack" inputVariable = "P_NackVar" partnerLink = "PandS_Producer"
70                           portType = "Producer : P_PT"⟩
71                   ⟨/sequence⟩
72                 ⟨/flow⟩
73               ⟨/sequence⟩
74             ⟨/onMessage⟩
75             ⟨onMessage operation = "offer" variable = "S_OfferVar" partnerLink = "PandS_Shipper"
76                       portType = "Shipper : ShipperC_PT"⟩
77               ⟨sequence⟩
78                 ⟨invoke operation = "ack" inputVariable = "P_AckVar" partnerLink = "PandS_Producer"
79                       portType = "Producer : P_PT"⟩
```

Figure 1.5: The executable WS-BPEL for the P&S process (pt.1).

```
79  ..........
80          ⟨receive operation = "offer" variable = "P_OfferVar" partnerLink = "PandS_Producer"
81                portType = "Producer : PC_PT"⟩
82          ⟨assign⟩
83            ⟨copy⟩
84              ⟨from expression = "bpws : getVariableData('P_OfferVar','cost')+
85                        bpws : getVariableData('S_OfferVar','cost')"⟩
86              ⟨to variable = "U_OfferVar" part = "cost"/⟩
87            ⟨/copy⟩
88            ⟨copy⟩
89              ⟨from expression = "bpws : getVariableData('P_OfferVar','delay')+
90                        bpws : getVariableData('S_OfferVar','delay')"⟩
91              ⟨to variable = "U_OfferVar" part = "delay"/⟩
92            ⟨/copy⟩
93          ⟨/assign⟩
94          ⟨invoke operation = "offer" inputVariable = "U_OfferVar" partnerLink = "client" portType = "PandSC_PT"⟩
95          ⟨pick⟩
96            ⟨onMessage operation = "ack" variable = "U_AckVar" partnerLink = "client" portType = "PandS_PT"⟩
97              ⟨sequence⟩
98                ⟨flow⟩
99                  ⟨sequence⟩
100                   ⟨invoke operation = "ack" inputVariable = "S_AckVar" partnerLink = "PandS_Shipper"
101                         portType = "Shipper : S_PT"⟩
102                 ⟨/sequence⟩
103                 ⟨sequence⟩
104                   ⟨invoke operation = "ack" inputVariable = "P_AckVar" partnerLink = "PandS_Producer"
105                         portType = "Producer : P_PT"⟩
106                 ⟨/sequence⟩
107               ⟨/flow⟩
108             ⟨/sequence⟩
109           ⟨/onMessage⟩
110           ⟨onMessage operation = "nack" variable = "U_NackVar"partnerLink = "client" portType = "PandS_PT"⟩
111             ⟨sequence
112               ⟨flow⟩
113                 ⟨sequence⟩
114                   ⟨invoke operation = "nack" inputVariable = "S_NackVar" partnerLink = "PandS_Shipper"
115                         portType = "Shipper : S_PT"⟩
116                 ⟨/sequence⟩
117                 ⟨sequence⟩
118                   ⟨invoke operation = "nack" inputVariable = "P_NackVar" partnerLink = "PandS_Producer"
119                         portType = "Producer : P_PT"⟩
120                 ⟨/sequence⟩
121               ⟨/flow⟩
122             ⟨/sequence⟩
123           ⟨/onMessage⟩
124         ⟨/pick⟩
125       ⟨/sequence⟩
126     ⟨/onMessage⟩
127   ⟨/pick⟩
128 ⟨/sequence⟩
129 ⟨/onMessage⟩
130 ⟨/pick⟩
131 ⟨/sequence⟩
132 ⟨/process⟩
```

Figure 1.6: The executable WS-BPEL for the P&S process (pt.2).

program counter for that component has its initial value. This said, the control flow part ρ_c of the requirement goes as follows:

$$((Prod.AvailVar \ \wedge \ Ship.AvailVar \ \wedge \ User.AvailVar) \rightarrow$$
$$(Prod.Succ \ \wedge \ Ship.Succ \ \wedge \ User.Succ))$$
$$\wedge$$
$$(\neg(Prod.AvailVar \ \wedge \ Ship.AvailVar \ \wedge \ User.AvailVar) \rightarrow$$
$$((Prod.Fail \ \vee \ Prod.Init) \wedge (Ship.Fail \ \vee \ Ship.Init) \wedge (User.Fail \ \vee \ User.Init)))$$

The data flow identifies the data dependencies amongst the partners. More specifically, we expect that the Producer and Shipper receive (from P&S) the request data sent by the User, that the Producer is the one to provide the size information to the Shipper, and that costs and times are coherently computed and communicated to the User by P&S. Of course there may be cases where some data are not actually produced nor circulated, e.g. if the Producer is unavailable to provide the item, he will not communicate any location or article data to the other partners. This is captured by stating that "only in case the component responsible for generating some data actually provides it, then it has to be communicated to the partner" - i.e., by a logical implication. The data section of the requirement, ρ_d, is thus as follows, where $DEFINED(d)$ is a condition that holds if d has a value different from a default 'undefinedness' value:

$$DEFINED(Producer.requestVar\_loc) \rightarrow User.requestVar\_loc = Producer.requestVar\_loc \wedge$$
$$DEFINED(Producer.requestVar\_art) \rightarrow User.requestVar\_art = Producer.requestVar\_art \wedge$$
$$DEFINED(Shipper.requestVar\_size) \rightarrow Producer.infoVar\_size = Shipper.requestVar\_size \wedge$$
$$DEFINED(Shipper.requestVar\_loc) \rightarrow User.requestVar\_loc = Shippher.requestVar\_loc \wedge$$
$$DEFINED(Shipper.requestVar\_art) \rightarrow User.requestVar\_art = Shippher.requestVar\_art \wedge$$
$$DEFINED(User.offerVar\_delay) \rightarrow$$
$$User.offerVar\_delay = Shipper.offerVar\_delay + Producer.offerVar\_delay \wedge$$
$$DEFINED(User.offerVar\_cost) \rightarrow$$
$$User.offerVar\_cost = Shipper.offerVar\_cost + Producer.offerVar\_cost \wedge$$

Of course, ground-level composition is rather sensitive to the ranges of exchanged data, due to the need of instantiating each and every possible combination of values. However, as discussed in [PTB05], ranges of size two are sufficient to produce a meaningful composed service which, submitted to an appropriate abstraction post-processing, is general enough to work even with larger data ranges. In this setting, the ground-level composition for the P&S scenario described above takes less than 10 seconds on a 2.8 Ghz, 512 Mbytes Linux machine.

Knowledge-level composition

Knowledge-level composition has been first introduced in [PATB05], and it is concerned with re-casting the goal and the service models to speak of "knowledge over (functions of) values", rather than of the actual values. This is performed by taking the goal and the components services, and automatically constructing a suitable knowledge-level abstraction, leading to a model called the "knowledge base". To illustrate and test our knowledge-level approach, we start from an alternate way to express our goal, based on a simple "try-do" preference pattern implemented using the EAGLE language ([DLPT02, PTB05]), and considering (uninterpreted) functions to relate values:

TryReach
```
user.pc = SUCC ∧ producer.pc = SUCC ∧ shipper.pc = SUCC ∧
user.offer_cost =
  addCost(producer.costOf(user.article),
  shipper.costOf(producer.sizeOf(user.article), user.location))
```

In essence, we declare that we want all the services to reach the situation where the order has been confirmed. Moreover the offered cost must be obtained by applying the function `addCost` to the costs offered by the producer and the shipper. The operator **TryReach** is one of the modal operators provided by the EAGLE goal specification language. It requires that the plan reaches the goal condition whenever possible in the domain. For further details, see [PTB05]

From this, we flatten the functions, introducing auxiliary variables until only basic propositions are left; the obtained "flat" goal provides us with the set of variables (and functions) whose knowledge we actually need, so we can re-cast the goal in terms of "knowledge atoms".

TryReach
```
user.pc = SUCC ∧ producer.pc = SUCC ∧ shipper.pc = SUCC ∧
user.offer_cost = goal.added_cost ∧
goal.added_cost = addCost(goal.prod_cost, goal.ship_cost) ∧
goal.prod_cost = producer.costOf(goal.user_art) ∧
goal.ship_cost = shipper.costOf(goal.prod_size, goal.user_art) ∧
goal.user_art = user.article ∧
goal.prod_size = producer.sizeOf(goal.user_art) ∧
goal.user_loc = user.location
```

In particular, from this flattened goal we can extract the goal variables: `goal.added_cost`, `goal.prod_cost`, `goal.ship_cost`, `goal.prod_size`, `goal.user_art`, and `goal.user_loc`; and the goal function `goal.addCost(Cost, Cost)`:`Cost`.

The goal can then be automatically translated into its corresponding knowledge level

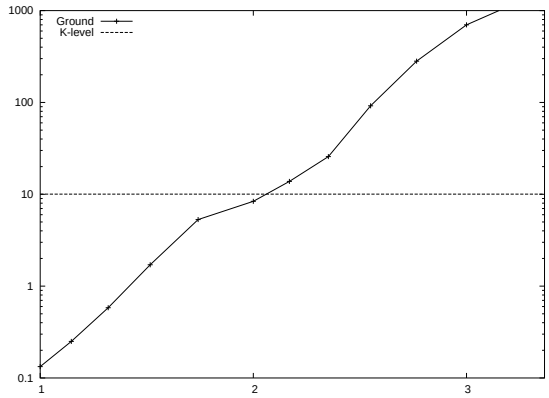


Figure 1.7: Experiments with P&S.

goal:

TryReach

K(user.pc = SUCC) ∧ K(producer.pc = SUCC) ∧ K(shipper.pc = SUCC) ∧
K(user.offer_cost = goal.added_cost)∧
K(goal.added_cost = goal.addCost(goal.prod_cost, goal.ship_cost))∧
K(goal.prod_cost = producer.costOf(goal.user_art))∧
K(goal.ship_cost = shipper.costOf(goal.prod_size, goal.user_art))∧
K(goal.user_art = user.article)∧
K(goal.prod_size = producer.sizeOf(goal.user_art))∧
K(goal.user_loc = user.location)

Figure 1.7 shows the results of our experiments. The same planning engine is run to perform knowledge-level composition and ground level compositions, where different ranges of values can be produced and exchanged by the Shipper, Producer and User. The horizontal axis reports the cardinality $\overline{n}$ of the data types (i.e. Size, Location, Cost, Delay) handled by the services. We also consider intermediate cases where we have, for instance, two possible values for the Cost and only one value for the Size), reporting the average cardinality in the figure. On the vertical axis, we report the composition time. As expected, ground level composition is only feasible for the unrealistic cases where processes may exchange only data with 2 or at most 3 values. Indeed, the time for ground composition grows exponentially with the cardinality of the data types, and even the simple case where types have cardinality 4 is unmanageable. On the contrary, knowledge-level composition takes about 10 seconds to complete, with a performance similar to that of the ground level for $\overline{n} = 2$. This is a reasonable result, since, basically, binary variables at the ground level correspond to (binary) knowledge atoms at the knowledge-level.

Data-net composition

We now step to using the data flow modeling language presented in [MPT06] is to allow the specification of complex requirements concerning data manipulation and exchange. In particular, data flow requirements specify explicitly how output messages (messages sent to component services) must be obtained from input messages (messages received from component services). This includes several important aspects: whether an input message can be used several times or just once, how several input messages must be combined to obtain an output message, whether all messages received must be processed and sent or not, and so on.

In essence, a data-net is an (acyclic) graph composed by a set of *connection nodes*, and a set of *gates*. *External* connection nodes are associated to an output (or an input) external port, characterizes an external source (target) of data used to exchange data with the outside world. *Internal* connection nodes are associated to the passage of data amongst different gates. *Gates* are the key element of the language; a gate has some input node and some output node, and a precise formal semantics that relate its outputs to its inputs. For instance, an "operation" gate computes the input result according to a function and forwards it to the output channel; a "fork" forwards its only input to each of its outputs, and so on.

A possible specification of the data net for the Purchase and Ship example is presented in Figure 1.11, which we will (partially) explain in the following example. The first data-flow requirement from the top

id(C.request.item)(P.request.item)

specifies that the item information sent to the Producer must be obtained from the request received from the Customer.
The last requirement

oper[addDelay](P.offer.delay, S.offer.delay)(C.offer.delay)

specifies that the P&S must apply its internal function addDelay on the delay received in the offers from the Producer and from the Shipper to obtain the delay to be sent in the offer to the Customer. The remaining parts of the requirement can be explained analogously.

	Knowledge Level		**Data Net**	
	model construction	composition	model construction	composition
P&S	4.391 sec.	0.672 sec.	2.207 sec.	0.027 sec.

Table 1.1: Experiments with P&S.

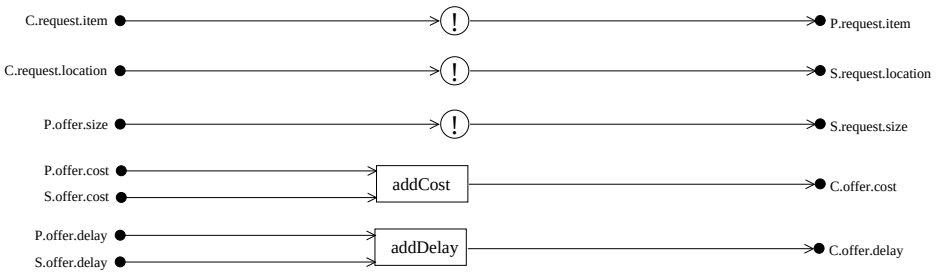
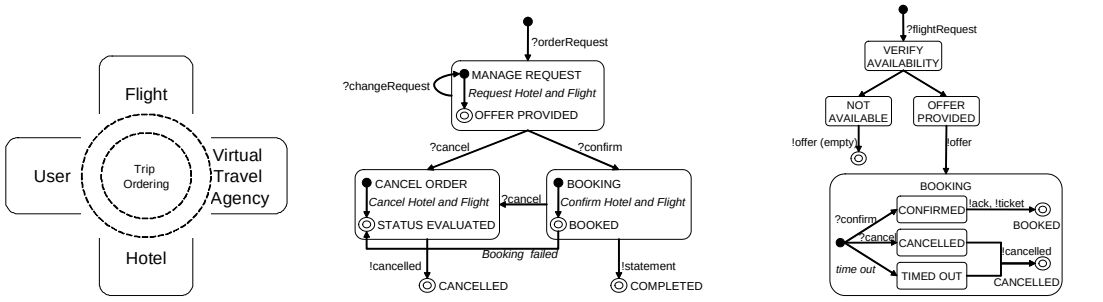


Figure 1.8: The data net for the *Purchase and Ship* composition scenario

1.2 The VTA scenario

1.2.1 Description



a) VTA participants **b) VTA service diagram** **c) Flight service diagram**

Figure 1.9: Choreography Model for VTA

We now consider a scenario named Virtual Travel Agency, rather well-known in the SOC community. In this scenario (which will be referred to as "VTA"), we assume two service to be already existing, and independently published: a Flight booking service, and a Hotel booking service. The idea is that it would be useful to integrate them to provide a combined flight and hotel booking service. Thus, the composition describes the interactions of four partners: User, Virtual Travel Agency (VTA), Hotel and Flight services (see Fig. 1.9.a). Although the User appears explicitly in the choreography, the real negotiation is performed among VTA, Hotel and Flight services that should agree on their needs and constraints.

The case study describes the behavior exposed by VTA that allows the user to book a flight to the specified place and reserve a room in the hotel at that place for a given period of time.

The choreography model may be defined in different notations, varying from the set of global requirements to the set of interfaces for each of the partners, at different level of abstractions. Different languages (see e.g. [AAF+02, ACD+03]) provide wide capabilities to define choreography model for interacting Web services. For the sake of simplicity

we model the initial choreography with the set of state diagrams representing abstract behavior of each participant.

In the case study the User is able to submit a request for a trip, change it, accept or reject the offer provided, stop the procedure by sending a cancellation message etc.

The VTA service model is supposed to provide this functionality (Fig. 1.9.b). Here VTA is supposed to receive an order request and the ordering process starts. The request is propagated to the Flight and Hotel services and the offer is obtained. The User is allowed to change its request in any moment thus forcing re-requesting services for corresponding offers. If the User confirms the order, providing all the information necessary for booking (e.g. personal info, payment data), the booking subprocess is initiated by the VTA service. If the booking was successful the user receives a statement information. The User is also allowed to cancel the whole process at any moment in which case the partners are canceled also.

The Flight service behavior is modeled as follows (Fig. 1.9.c). On the reception of the request it verifies the availability of tickets and provides either the flight offer or an empty offer if there are no flight/tickets available. Then the partner (VTA) can either confirm or cancel the offer, and respective messages are provided. Or, if there was no response from it, the booking is automatically canceled on time out. Analogous model for the Hotel service can be thought of.

Similar to the P&S scenario, we assume that the standard language has been adopted to publish both the component Flight and Hotel services, and for the VTA service. Figure 1.10 shows the abstract WS-BPEL of the Hotel Service, slightly simplified for readability purposes.

The hotel booking service becomes active upon a request for a room in a given location for a given period of time (see the `<receive>` statement named `Receive_request` at line 9; notice that this statement corresponds to a web service invocation for operation `hRequest`, and that the information on the location and period of time carried by the invocation are stored in variable `hRequestMsg`). In the case the booking is not possible (e.g., there are no available rooms), (`<otherwise>` statement at lines 50-56), this is signaled to the request applicant (`invoke` statement at line 53), and the protocol terminates with failure. If the booking is possible (`<case>` statement at lines 11-49), the applicant is notified with information about the hotel, cost of the room, etc. (`<assign>` statement at line 13 and `<invoke>` statement at line 14; notice that the assign statement is "opaque", i.e., it describes the fact that the values of the offer are decided at this point, but does not disclose the details on how these data are defined; similar opaque statements also appear in other points of the code). Then, the protocol stops waiting for either a positive (`<onMessage>` statement at lines 16) or negative acknowledgment (`<onMessage>` statement at lines 44). In the first case (lines 17-42), an agreement has been reached and the room is booked. The hotel waits a certain amount of time before emitting the reservation ticket. If a cancellation request is received before the deadline (`<onMessage>` statement at line 18), the room booking is cancelled and a cancellation acknowledge mes-

```
 7   ...
 8 <sequence>
 9  <receive name="Receive_request" operation="hRequest" variable="hRequestMsg" partnerLink="Hotel_PLT"/>
10  <switch name="Is_available">
11    <case condition="isAvailable">
12      <sequence>
13        <assign name="Prepare_offer"><copy><from opaque="yes"/><to variable="hOfferMsg" part="hotel"/></copy></assign>
14        <invoke name="Hotel_offer" operation="hOffer" inputVariable="hOfferMsg" partnerLink="Hotel_PLT"/>
15        <pick>
16          <onMessage operation="hAck" variable="hAckMsg" partnerLink="Hotel_PLT">
17            <pick>
18              <onMessage name="YesCancel" operation="hCancel" variable="hCancelMsg" partnerLink="Hotel_PLT">
19                <sequence name="on_YesCanc">
20                  <assign name="Prepare_Canc"/>
21                  <invoke name="Yes" operation="hYes" inputVariable="hYesMsg" partnerLink="Hotel_PLT"/>
22                  <empty name="CANC"/>
23                </sequence>
24              </onMessage>
25              <onAlarm>
26                <sequence name="on_Timeout">
27                  <assign name="Prepare_Ticket"/>
28                  <invoke name="Send_Ticket" operation="hSendTicket" inputVariable="hTicketMsg" partnerLink="Hotel_PLT"/>
29                  <pick name="Sent_Ticket">
30                    <onMessage name="NoCancel" operation="hCancel" variable="hCancelMsg" partnerLink="Hotel_PLT">
31                      <assign name="Prepare_no"/>
32                      <invoke inputVariable="hNoMsg" name="Send_No" operation="hNo" partnerLink="Hotel_PLT"/>
33                      <receive name="Ticket_ACK" operation="hTicketAck" variable="hTicketAckMsg" partnerLink="Hotel_PLT"/>
34                      <empty name="NO_CANC"/>
35                    </onMessage>
36                    <onMessage operation="hTicketAck"  variable="hTicketAckMsg" partnerLink="Hotel_PLT">
37                      <empty name="SUCC"/>
38                    </onMessage>
39                  </pick>
40                </sequence>
41              </onAlarm>
42            </pick>
43          </onMessage>
44          <onMessge operation="hNack" variable="hNackMsg" partnerLink="Hotel_PLT">
45            <empty name="FAIL"/>
46          </onMessage>
47        </pick>
48      </sequence>
49    </case>
50    <otherwise>
51      <sequence>
52        <assign name="Prepare_Answer"/>
53        <invoke name="Not_available" operation="hNot_avail" inputVariable="hNot_availMsg" partnerLink="Hotel_PLT"/>
54        <empty name="FAIL"/>
55      </sequence>
56    </otherwise>
57  </switch>
58 </sequence>
59 </process>
```

Figure 1.10: Hotel abstract WS-BPEL

sage is sent to the client (`<invoke>` statement at line 21). If no cancellation is received before the deadline (`onAlarm` statement at line 25), the room booking is confirmed and a ticket is sent to the client (`<invoke>` statement at line 28), the hotel waits for a final acknowledge from the client and terminates with success (line 37). Any cancellation that is received after the ticket is emitted, is refused (lines 30-32). The protocol provided by the Flight booking service is similar to that of the Hotel, and we omit it.

1.2.2 Experiments: composition

Even for a rather simple scenario such as the VTA, we need a way to express requirements that define complex conditions, both for what concerns the control flow and for the data exchanged among the component services.

First, we need to express conditions over the termination of the service: we want the VTA to reach a situation where the customer has accepted the offer and a flight and a hotel have been booked. However, it may be the case that there are no available flights (or no available hotels) satisfying the customer request, or that the customer doesn't like the flight or the hotel offer and thus cancels the booking. We cannot avoid these situations, therefore we cannot ask the composite service to guarantee this requirement. In case this requirement cannot be satisfied, we do not want the VTA to book a flight (nor a hotel) without being sure that our customer accepted the offer, as well as we do not want displeased customers that have booked holidays for which there are no available flights or hotels. Thus, our global termination requirement would be something like: do whatever is possible to "sell holiday packages" and if something goes wrong guarantee that there are "no single commitments".

This termination requirement is only a partial specification of the constraints that the composition should satisfy. Indeed, we need to specify also complex requirements on the data flow.

In order to provide consistent information, the VTA service needs to exchange data with the components and its customer in an appropriate way. For instance, when invoking the Flight service, the information about the location and date of the flight must be the same ones that the VTA received in the customer request; similarly, the information sent to the customer about the distance between the proposed hotel and the airport must be those obtained from the last interaction with the AllMaps service; and such a service must receive the information on the airport and hotel location according to the last offer proposed by the Hotel service. Moreover, the cost proposed to the customer for the holiday package must be the sum of the hotel and flight cost plus some additional fee for the travel agency service; thus the cost offered to the customer must be computed by means of a function internal to the VTA service. And so on.

This goes to show that, even for apparently simple composition problems, we need a way to express complex data flow requirements: from simple data links between incoming and outgoing message parts (e.g., forwarding the information received by the customer about the location to the Flight service), to the specification of complex data manipulation (e.g., when computing the holiday package cost), to more subtle requirements concerning data (e.g., all the time the VTA invokes the AllMaps service it must send the location information of the last hotel offer received, while the same airport location information can be used more than once).

In particular, we exploit the "data net" methodology proposed in [MPT06], where a diagram os obtained by suitably composing data-flow elements by means of connection nodes. Figure 1.11 represents the data-flow portion of the requirements for the VTA scenario, which we used to test composition by a suitable recasting within the framework presented in [PTB05].

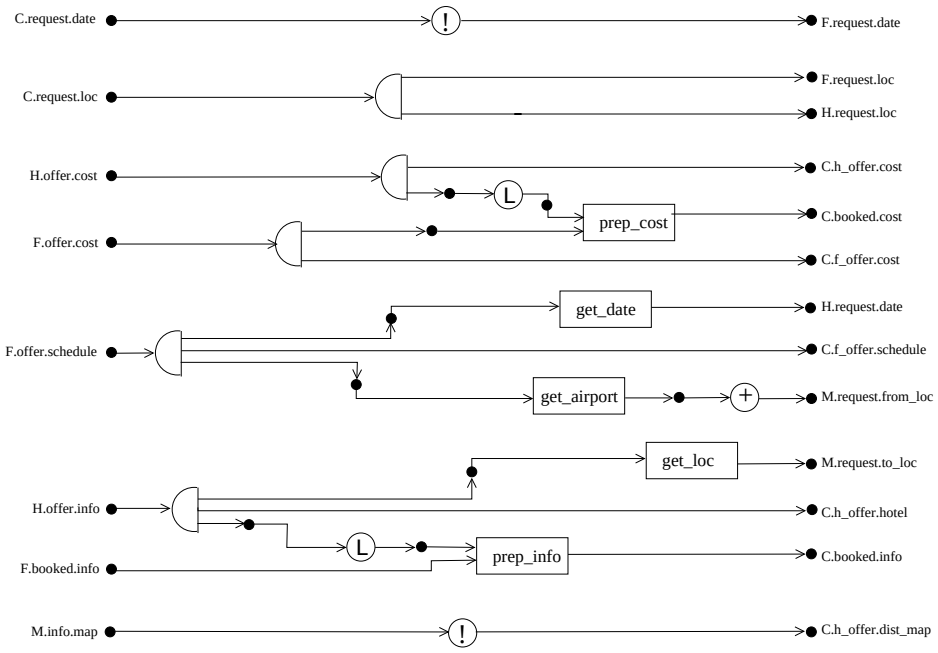


Figure 1.11: The data flow requirements for the Virtual Travel Agency

1.2.3 Experiments: verification

We used the VTA scenario to test a verification methodology and tool based on the idea of parametric communication models, presented in [KPS06], and implemented within the Astro toolkit.

We conducted series of experiments in order to evaluate the presented approach. The aim of the evaluation was to demonstrate that the less general model shown to be adequate is more efficient for the analysis, and to see the overall performance of the composition verification based on the presented approach.

In particular, we were interested in the performance and in the memory usage of the composition analysis. In these experiments we used variations of the VTA example, where the number of the participating processes grows from two up to seven processes. We remark that the VTA process also grows since it interacts with increasing number of services. The ranges of the domain types used in the messages (e.g. Flight, Time) were set to three values for each type. Although the examples described below are relatively simple, they still are considerably more complex with respect to the samples presented in other tools (e.g., [FBS04, FUMK03]).

In order to compare the verification complexity on the same scenarios under different communication models, we have used domains where the synchronous model is adequate. We used two properties in the experiments specified as Linear-time Temporal Logic (LTL) formula. The first property (P1) requires that the user process terminates successfully only if also the reservation services do. This property is expected to be valid in the domain,

Property	N	NuSMV						Spin					
		Sync		LO		UO		Sync		LO		UO	
		Time	Size	Time	Size	Time	Size	Time	Size	Time	Size	Time	Size
P1	2	0.08	5	0.09	6	0.09	6	0.87	48	0.89	56	0.89	76
	3	0.11	8	0.13	9	0.15	10	1.05	60	1.09	76	1.13	120
	4	0.19	11	0.22	13	0.24	15	1.51	72	1.60	100	1.75	172
	5	0.26	14	0.32	17	0.36	20	3.67	84	4.79	120	6.37	220
	6	0.32	17	0.42	21	0.50	24	90	100	$>$ 1Gb		$>$ 1Gb	
	7	0.44	20	0.64	24	3.75	27	$>$ 1Gb		$>$ 1Gb		$>$ 1Gb	
P2	2	0.01	5	0.02	6	0.02	6	0.87	48	0.89	56	0.92	76
	3	0.33	8	0.48	9	0.52	10	1.05	60	1.09	76	1.13	120
	4	1.14	11	2.28	13	8.35	15	1.38	72	1.46	100	1.52	172
	5	20.1	14	36.8	17	69.0	20	1.85	84	1.85	120	1.95	220
	6	114	17	221	21	469	24	2.31	100	2.33	140	2.46	272
	7	771	20	1279	24	$>$ 1 hour		2.89	112	2.98	160	3.27	320

Table 1.2: Verification results

i.e., to be respected by all the executions of the Web service composition. The second property (P2) expresses the possibility for the partners to terminate successfully. This property is expected to be satisfiable, i.e., there are some executions of the composition where the property is true. Moreover, we expect that the verification task produces a trace corresponding to one these executions, thus witnessing the validity of this possibility.

The results of the verification of these properties are summarized in Table 1.2. The verification was performed using two state of the art model checkers, namely NuSMV [CCGR00] and Spin [Hol97]. We tested the specifications of the composition under synchronous product (Sync), under locally order model (LO) and under the most general model (UO). The table contains information on the time used for the verification and counterexample generation in seconds, and on the size of the state vector in bytes. That is, if the state vector size is 14 bytes, the state space is $2^{8\times 14}$ states.

Some comments are in order, on the difference in the performance of NuSMV between the two properties. This is due to the fact that the second property requires the generation of a witness scenario, and this takes a lot of time. The time required by NuSMV to report that the second property can be satisfied without extracting the witness trace is similar to the time required to verify the first property. On the contrary, the verification using Spin model checker requires much more time for the first property. Indeed, in this case all the behaviors have to be considered to prove the correctness of the property, while a single witness trace is sufficient for the second property.

The presented results demonstrate the reduction of the verification performance when more general communication model is applied. This is explained by the fact that more general model introduces more queue variables and therefore increases the state space size. This is particularly important for the NuSMV model checker, since the techniques used there strictly depend on the number of variables, their domains and relations.

1.2.4 Experiments: monitoring

We used the VTA scenario also to experiment with a technique to automatically generate monitors expressed as Java programs that, by intercepting messages exchanged between the partners in the protocol, are able to detect whether certain properties are obeyed at runtime, and report misbehaviors to e.g. trigger recovery actions. This technique has been presented in [MMF+06].

Several interesting properties, and associated techniques, can be thought of. First, we can define a *hotel protocol monitor* checks whether the hotel behaves consistently with the agreed interaction flow defined in its exposed abstract WS-BPEL. Its task is to rise a flag when the hotel does not respect the protocol. e.g., since it is modified without any notification or warranty. This may be indeed the case, since it is and external partners that can act in autonomy. The hotel protocol monitor is a so-called *domain monitor*: it checks whether the domain behaves coherently with what has been published. We can as well define a *hotel cancellation monitor*, which is an example of monitoring an agreement with an external partner. Its aim is to check whether the hotel actually cancels the reservation if the client cancels it on time, i.e., within the cancellation deadline. We can then define a *full compensation monitor* that detects a global misbehavior that is caused by several processes (monitors of this kind are called *"choreographic"* monitors in the following). It checks whether a complete cancellation of the reservations takes place after a user cancellation request has been accepted by the agency. If only a partial cancellation occurs (i.e., the flight booking or the hotel booking are not canceled), then a warning is issued. Finally, the *selling report monitor* is an example of a monitor of situations of interest that can be useful to business analysts to decide future business strategies. It raises a flag every time the client rejects an offer. These observations can thus be aggregated and statistics can reveal, e.g., that too many offers are rejected. This indicator should be reported to the management, and some strategic action should be taken, e.g., prices should be decreased. The hotel cancellation, the full compensation, and the selling products monitors are all*assumption monitors*: corresponding assumptions can be used by the planner to generate the composition plan: the assumption that the hotel cancels reservations correctly, that a user's cancellation causes a cancellation by all the partners, and the assumption that users will accept offers of the agency.

An example of a hand-written Java code implementing the *hotel cancellation monitor* for the VTA is presented in Figure 1.12. The monitor checks that whenever an input message of type `hCancel` is sent to the hotel service, then the hotel does not emit an output of type `hSendTicket`, i.e., if a cancellation is requested, then it is accepted and no ticket is sent. The monitor can be in one of the following internal states: `USER_CANCELS` when a cancellation has been requested by the user, `HOTEL_REFUSES_CANCEL` when a ticket has been sent to the user despite the fact that the user requested a cancellation, and `OTHER` in all the other cases. The property to be monitored is valid if and only if its internal state is different from `HOTEL_REFUSES_CANCEL`.

However, all the monitors presented above can be automatically generated. Domain

```
public class HotelCancel implements IMonitor {
    private enum HotelState { USER_CANCELS, HOTEL_REFUSES_CANCEL, OTHER }
    private HotelState _state;
    public void init()
    {
        _state = HotelState.OTHER;
    }
    public boolean isValid() {
        return !(_state.equals(HOTEL_REFUSES_CANCEL));
    }
    public void terminate() {}
    public String getErrorString()
    {
        if(!isValid()){
            return "Property Cancel violated";
        } else {
            return "No error";
        }
    }
    public void evolve(BpelMsg msg)
    {
        if(msg.getOperation().equals("hCancel")){
            _state = HotelState.USER_CANCELS;
        } else if(msg.getOperation().equals("hSendTicket")){
            if(_state.equals(USER_CANCELS)){
                _state = HotelState.HOTEL_REFUSES_CANCEL;
            }
        }
    }
    public HotelCancel()
    {
        init();
    }
    public String getProcessName() { return "Hotel"; }
    public String getPropertyName() { return "Cancel"; }
    public String getPropertyDescription() { return "Hotel correct cancellation"; }
}
```

Figure 1.12: The Java code for the Hotel Cancellation Monitor

monitors can be built by an exhaustive (belief-level) visit of the STSs representing the services that need be monitored. Assumption and choreographic monitors, on top of this, require the statement of temporal properties to be monitored; these are converted in an automaton whose evolution at runtime constitutes the monitor proper. We implemented such an approach in our toolset, and presented it in [MMF+06].

1.3 The WMO scenario

1.3.1 Description

We now consider a scenario taken from an actual situation, and related to government of environmental issues according to the Italian regulations.

The goal of the application is to provide a service that manages user requests to open sites for the disposal of dangerous waste. According to the existing Italian laws, such a request involves the interaction of different actors of the public administration, namely a Citizen Service, a Waste Management Office (WMO), a Secretary Service, a Procedure Manager, a Technical Committee, and a Political Board. In this application, the whole procedure is already implemented as a composition of Web services that serve as inter-

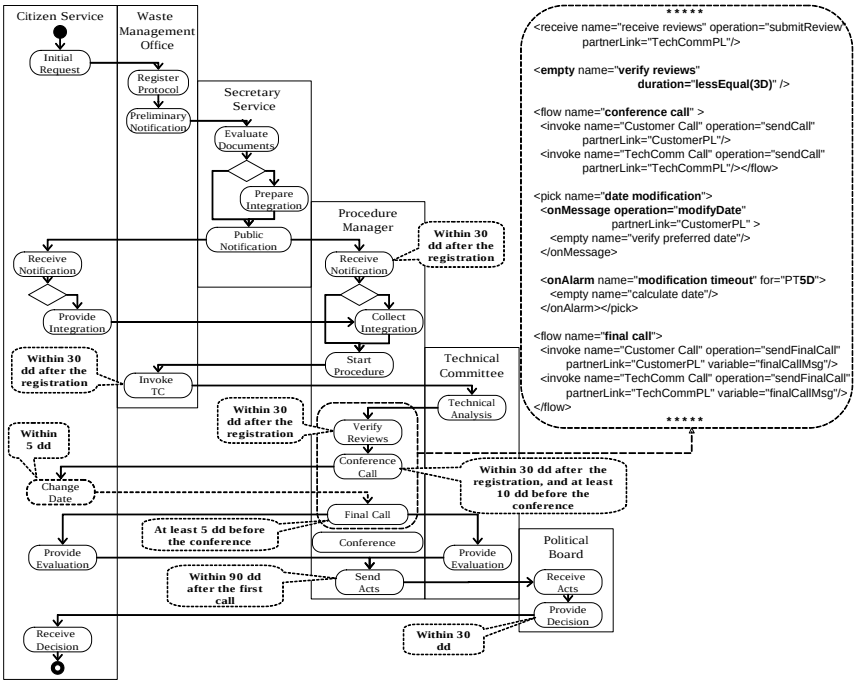


Figure 1.13: Waste management application processes

faces to the processes of the above actors. Each of the actors are represented by means of WS-BPEL interfaces, as represented in Fig. 1.13. The interaction between the WS-BPEL actors makes up a multi-phase procedure for the application management, where the request is registered, the documentation is evaluated and collected, the application is analyzed regarding the ecological impact of the site, the public conference is scheduled and organized, and final decision is provided. Given this setting, our main interest consisted in studying the WMO scenario for the purposes of statically verifying the properties of its components.

1.3.2 Experiments: verification

The WMO features interesting aspects for what concerns verification. Indeed, apart from the functional requirements, the execution of the process in the choreography should respect a set of timed requirements and constraints, dictated by Italian laws or by the agreement among the involved parties. These requirements (callouts in Fig. 1.13) specify, for example, that the period of time between the application registration and the notification of the Procedure Manager should not exceed 30 days, or that the participants can change the date within 5 days after the preliminary call. The behavior of the composition and the possibility to satisfy these requirements depend on the time needed for the execution of the activities the involved parties are responsible for. We remark that the critical parameter is the duration of internal activities of the participants, and not to the communication

time, which can be neglected.

In these settings, the Web service composition analysis may become a long, error-prone process of finding boundary time values that would ensure correctness of the composition with respect to functional and timed requirements. Consider, for instance, the problem of determining the maximum time which can be spent by the Citizen to provide integration documents and the Technical Committee to perform the analysis, such that the requirement to announce a conference within 30 days after registration is satisfied.

The analysis of time-related aspects of the compositions requires explicit representation of timeouts, operation durations, and even complex properties expressing various timed requirements. While *timeouts* can be represented in BPEL, durations and timed requirements can not, and require specific way to be modelled. In our framework, we assume that the answer times are negligible by default, and that activities that have a non-negligible duration are annotated in the BPEL specification with an extra *duration* attribute. In Fig. 1.13 an excerpt of the annotated BPEL is presented. Here a BPEL event handler "date modification" is used to model a time-bounded possibility to change the date of the conference. That is, the `onAlarm` activity is triggered if the user does not call the "modifyDate" operation within 5 days. On the contrary, the internal activity "verify reviews" is equipped with a duration annotation to express that certain time may be used for the reviews analysis.

While durations and timeouts can be easily represented within BPEL, *timed requirements* can not and require more powerful notations. Consider, for instance, the requirement that the interval from the registration to the conference call should not exceed 30 days, and it is followed by the interval of length of at least 10 days, ending with the conference. This requirement spans over many activities performed by different parties. In order to be able to handle it, it is necessary to provide a model of the behavior of the BPEL processes that allows for an explicit representation of time. Moreover, it is necessary to exploit techniques for reasoning about time to check if this requirement is satisfied by the BPEL timed model. To address these issues, in [KPP06a], we designed a Timed Transition-based model, and recurred to duration annotations and duration calculus to express complex time requirements. This boils down to an algorithm that converts timed requirements into automata, which can then be verified using the NuSMV model checker. We verified the behavior of the Waste Management Application composition against various properties of interest, and under different assumptions expressed both as the duration annotations and also using QDDC. In particular, we verified the (annotated) composition against the following properties, and experimented with the quantitative analysis:

- Deadlock freeness of the specification;
- Assertion that the procedure always terminates within a given period;
- Possibility to obtain an official conclusion within a given bound;
- Maximal duration of the procedure;

Table 1.3: Experimental results

Property	Result	Time	States
Deadlock	true	0.48 sec	1005
Assertion	false	5,56 sec	2919
Possibility	true	2,28 sec	2919
Max	∞	0.28 sec	1005
Min	10	0.32 sec	1005

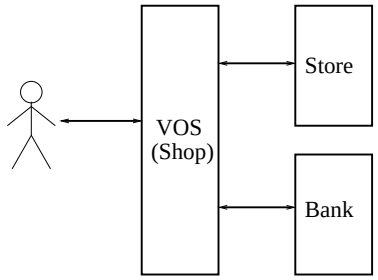


Figure 1.14: The Virtual Online Shop example: component services and monitors

- Minimal time needed to obtain an official conclusion.

The results of the verification are presented in Table 1.3. The table demonstrates the outcome of the analysis, the verification time, and the size of the reachable state space (i.e., the number of states of the GTTS of the composition specification). Interestingly enough, the computation time is considerably smaller than the verification of the corresponding property, which conforms with the results presented in [Pan04].

1.4 The VOS scenario

1.4.1 Description

The VOS is a real e-commerce scenario that bears a structure similar to that of the P&S, but where more complicated protocols enter into play. Here, the existing published services are a Store and a Bank, and a possible goal is that of obtained a composed Virtual Online Shop (VOS) service that offers a combined sell and payment service to clients, by interacting with them (see Figure 1.14). When the VOS receives a request for an item from a client, it contacts the Store, and, if the item is available, it gets back an offer including the price. Once the client has accepted the offer, the VOS sends an acknowledgment to the Store, which in turn replies by sending back its bank account data. After obtaining also the client's bank account, the VOS invokes the Bank by sending the amount to be transferred from the client's account to the Store's account. The Bank performs usual authentication procedures, and if the credential of the Store are not refused by the bank,

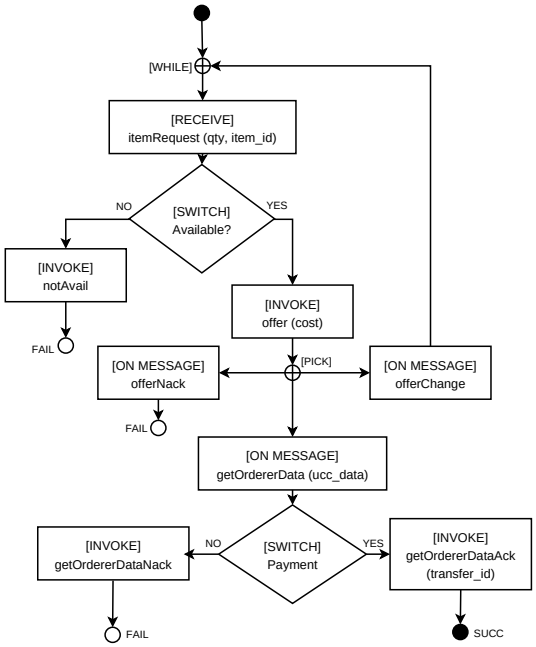


Figure 1.15: VOS abstract WS-BPEL

it sends back to the VOS a "transfer id" that identifies unequivocally the transaction and that the VOS routes to the Store. If the Store confirms the payment, the VOS confirms to the Bank, which performs the money transfer.

In the case the client does not accept an offer, he/she can either terminate the interaction with the VOS, or request an offer for a different item. We remark that the Bank can refuse to authenticate the account data of both the client and the store.

In this example, the Store and the Bank are the external component services. We assume their abstract WS-BPEL specifications are available on the Web. Their abstract WS-BPEL specifications describe the interaction protocols that the VOS is expected to respect when interacting with them. The VOS, instead, is assumed to run in the local engine, and is defined as executable WS-BPEL. Its abstract WS-BPEL, depicted in figure 1.15[1], defines the protocol interaction between the VOS and the client. The VOS becomes active upon a request for an item, which includes information about the quantity (see the `[RECEIVE] itemRequest` box). If the requested item is not available, the user is informed about this (`[INVOKE] notAvail` box) and the interaction terminates, the VOS sends an offer with a cost (`[INVOKE] offer` box), otherwise. The availability condition (`[SWITCH] Available?`) depends on an interaction between the VOS and the Store that is hidden to the client. If the item is available, the VOS stops waiting for either a positive reply from the client (`[ON MESSAGE] getOrdererData`), or a negative response (`[ON MESSAGE] offerNack`). The client is given also the possibility to ask for another

[1]For lack of space, we simply depict the interaction flow rather than reporting the actual WS-BPEL specification

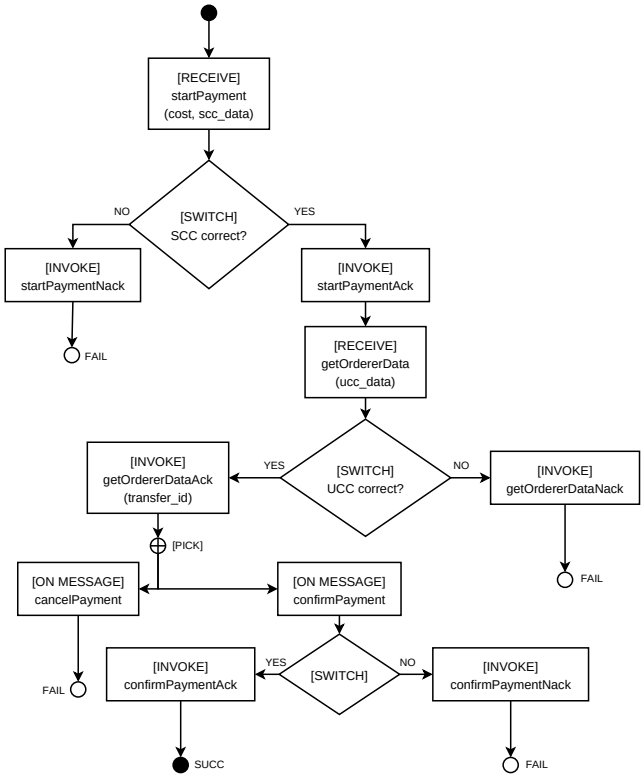


Figure 1.16: Bank abstract WS-BPEL

offer (`[ON MESSAGE] offerChange`) by iterating the item request. If the client accepts the offer, its message (`[ON MESSAGE] getOrdererData`) contains its bank account information for the payment. Finally, depending on whether the payment procedure is successful (this involves interactions of VOS with the Store and with the Bank), the client is notified with either an acknowledgment (`[INVOKE] getOrdererDataAck`) or a refusal (`[INVOKE] getOrdererDataNack`).

The two other abstract WS-BPEL specifications define the interaction protocol between the VOS and the Store and between the VOS and the Bank. In Figure 1.16 we show the interaction flow of the abstract WS-BPEL of the Bank. The interaction with the Bank service is structured in three phases. In the first phase, the client that receives the money, in our case the Store, is authenticated. The bank receives a request from for a money transfer of a given amount (corresponding to the `cost` parameter of the message `[RECEIVE] startPayment`) to the bank account identified by the parameter `scc_data`. The second phase starts after the client that receives the money has been successfully validated (`[SWITCH] SCC Correct?`), In this phase, the client that has to pay, in our case the VOS client, is authenticated (`[RECEIVE] getOrdererData`). The third phase starts when both Bank clients have been validated. The Bank sends back an acknowledgment and stops waiting for a confirmation or a cancellation by the client. On confirmation (`[ON MESSAGE] confirmPayment`), the Bank can either refuse to perform the

money transfer (this is notified to the client with `[INVOKE] confirmPaymentNack`) or actually perform the transfer. In the latter case, a final acknowledgment is sent to the service's client (`[INVOKE] confirmPaymentAck`).

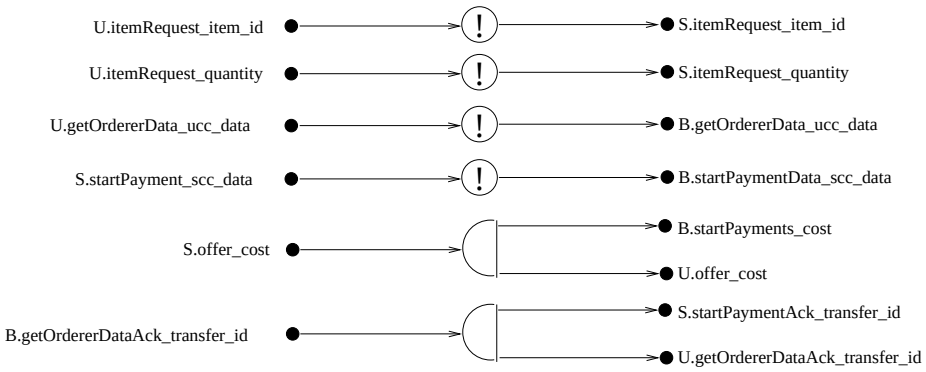


Figure 1.17: The graphical requirements language.

1.4.2 Experiments: composition

We used the VOS scenario to test the ground-level approach of [PTB05], where dataflow requirements are left implicit in the goal expression, and to compare it with the one based on the explicit data-net requirements language presented in [MPT06].

In particular, to evaluate the scalability of the two approaches when the number of (complex) component services grows, we considered a scalable version of the VOS, varying the number of stores participating to the composition. The following tables report the experimental results.

	Implicit					**WS-BPEL**
	domain			**time (sec.)**		num
	goal vars	nr. of states	max path	model construction	composition & emission	goal activities
VOS	6	1357	54	11.937	3.812	52
VO2S	9	8573	64	185.391	84.408	84
VO3S	12	75289	74	M.O.	-	109
VO4S	-	-	-	-	-	136
VO5S	-	-	-	-	-	165
VO6S	-	-	-	-	-	196

	Explicit						WS-BPEL
	domain				time (sec.)		num complex activities
	goal vars	data constr.	nr. of states	max path	model construction	composition & emission	
VOS	14	6	390	26	1.612	0.218	52
VO2S	20	9	1762	32	1.796	0.688	84
VO3S	26	12	12412	38	1.921	2.593	109
VO4S	32	15	122770	44	2.218	12.500	136
VO5S	38	18	1394740	50	2.547	26.197	165
VO6S	44	21	16501402	56	2.672	246.937	196

For each considered scenario the table shows some parameters that characterize the complexity of the composition domain, the automated composition time, and the size of the generated composite process. The complexity of the implicit approach is given in terms of the number of goal variables that encode the pieces of "knowledge" that the composite process acquires while interacting with the component services and manipulating messages (see [PATB05] for the details). For what concerns the explicit approach, we consider the number of data-flow constraints and the number of goal variables (as shown in [MPT06], the variables correspond to number of nodes of the data net obtained by combining all the data-flow constraints). To complete the characterization of the complexity of the composition domain, for both approaches we report the number of states and the number of transitions of the longest path in the composition domain that is passed as input to the automated generation techniques of [PTB05]. These measures characterize the size of the search space for the composed service.

The complexity of the composition task can also be deduced from the size of the new composite WS-BPEL process, which is reported in the last column of the table. We remark that we report the number of WS-BPEL basic activities (e.g. `invoke`, `receive`, `reply`, `assign`, `onMessage`) and do not count the WS-BPEL structured activities that are used to aggregate basic activities (e.g. `sequence`, `switch`, `flow`). Indeed, the former activities are a better measure of the complexity of the generated process, while the latter are more dependent on the coding style used in the composite WS-BPEL process. Notice that we report only one measure for the composite process. Indeed, the processes generated by the two approaches are basically identical: they implement the same strategy, handle exceptions and failures in the same way and present the same number of activities. The only difference is the way in which such activities are arranged, e.g. the order of invocation of the different shops or of the assignments when preparing different parts of a message to be sent.

The composition times have been obtained on a Pentium Centrino 1.6 GHz with 512 Mb RAM of memory running Linux. We distinguish between model construction time and composition and emission time. The former is the time required to obtain the composition domain, i.e., to translate the WS-BPEL component services into a finite state domain and to encode the composition goal. The latter is the time required to synthesize the controller according to [PTB05] and to emit the corresponding WS-BPEL process. The

experiments show that the implicit approach has worse performances both for the model construction time and for the composition time. In particular, the implicit approach is not able to synthesize the VOS scenario with three shops: a memory out is obtained in model construction time. In the case of the explicit approach, instead, the time required to generate the composition domain is very low for all the scenarios, and also the performance for the composition scales up to very complex composition scenarios: the VO6S example (6 Stores, 1 Bank and 1 Customer) can be synthesized in about 4 minutes. We remark that this example is very complex, and requires several hours of work to be manually encoded: the corresponding WS-BPEL process contains about 200 non-trivial activities! We also remark that the number of states in the implicit domain for the VO6S example is much larger than the number of states of the VO3S example in the explicit approach. The fact that the former composition has success while the latter has not shows another important advantage of the explicit approach: the domain is very modular, since each data-flow constraint is modeled as a separated "service", which allows for a very efficient exploitation of the techniques implemented by [PATB05].

For what concerns usability, the judgment is not so straightforward, since the two approaches adopt very different perspectives. From the one side, modeling data-flow composition requirements through a data net requires to explicitly link all the messages received from component services with messages sent to component services. Moreover, a second disadvantage is that component services are black boxes exporting only input and output ports: there is no way to reason about their internal data manipulation behaviors. From the other side, data nets models are easy to formulate and understand through a very intuitive graphical representation. It is rather intuitive for the designer to check the correctness of the requirements on data routings and manipulations. Also detecting missing requirements is simple, since they usually correspond to WS-BPEL messages (or part of messages) that are not linked to the data net. Finally, the practical experience with the examples of the experimental evaluations reported in this section is than the time required to specify the data net is acceptable, and much smaller that the time required to implement the composite service by hand. In the case of the VO6S scenario, for instance, just around 20 minutes are sufficient to write the requirement, while several hours are necessary to implement the composite service.

The implicit approach adopts a more abstract perspective, through the use of annotations in the process-level descriptions of component services. This makes the goal independent from the specific structure of the WS-BPEL processes implementing the component services, allowing to leave out most implementation details. It is therefore less time consuming, more concise, more re-usable than data nets. Moreover, annotations provide a way to give semantics to WS-BPEL descriptions of component services, thus opening up the way to reason on semantic annotations capturing the component service internal behavior. Finally, implicit knowledge level specifications allow for a clear separation of the components annotations from the composition goal, and thus for a clear separation of the task of the designers of the components from that of the designer of the composition, a separation that can be important in, e.g., cross-organizational application domains. How-

ever, taking full advantage of the implicit approach is not obvious, especially for people without a deep know-how of the exploited composition techniques. There are two main opposite risks for the analysts: to over-specify the requirements adding non mandatory details and thus performing the same amount of work required by the explicit approach; to forget data-flow requirements which are necessary to find the desired composite service. Moreover, our experiments have shown that, while the time required to write the implicit data-flow requirements *given* the annotated WS-BPEL processes is much less than the time to specify the data net, the time required to write the requirements *and* to annotate the WS-BPEL processes is much more (and the errors are much more frequent) than for the data net.

1.4.3 Experiments: monitoring

The VOS scenario is such that there exist several different properties that it can be relevant to monitor. Fir this purpose, it has the subject of a thorough investigation of automated monitor synthesis techniques, presented in [BTPT06]. A first class of properties are those that constrain the correct behaviors of the composition. An example is property:

- **StoreCcNotRefused**: the credentials of the Store are not refused by the Bank.

We remark that the interaction protocol with the bank allows for such a refusal; however, the VOS expects this refusal never to happen due to the contracts regulating the relations with the Bank. We expect to promptly detect and promptly reveal such possible but highly unexpected event, which corresponds to an assumption violation that can be detected only at run-time. This is an example of a boolean property that we want to check on all instances of the VOS process. Another boolean property is the following:

- **OfferBeforeBank**: the interaction with the Bank does not start before the User has accepted an offer.

The VOS may also be interested in counting how many times a given event occurs in the execution of a process instance. Examples of this properties are:

- **NotAvailCount**: count the number of times the Store reports that a requested item is unavailable.
- **RetriesOnSuccCount**: count the number of items offered to the User before the User accepts to buy.

Finally, the VOS may be interested in measuring the time spent to perform certain activities, for instance:

- **PaymentTime**: compute the time requested to finalize the payment with the bank from the payment request.

All the properties described above are instance-level properties, that is, they are evaluated on one instance on execution of the VOS service. On top of these properties, it is possible to define aggregated class-level properties, i.e., properties that consider all the instances of a business process.

- **GlobalStoreCcNotRefuse**: the credentials of the Store have never been refused by the Bank in any execution of the VOS.

This is an example of a boolean property for a class monitor. A related numerical property for a class monitor is the following:

- **CountStoreCCRefused**: count the total number of times the Bank has refused the credentials of the Store on all the executions of the VOS.

The following two properties perform statistical analysis of properties related to the number of times a given event is repeated and of properties related to the time required to perform given activities:

- **AverageUserRetriesCount**: average number of times the user gets and refuses an offer from the VOS.
- **AveragePaymentTime**:average duration of the interactions with the back for the payment procedure.

The properties mentioned above can be defined by RTML formulas (we use the abbreviation $\mathsf{Average}\,(n) = \mathsf{Sum}\,(n)/\mathsf{Count}\,(\mathsf{O}\,\mathsf{start})$ for computing the average value of numeric formula n):

- **StoreCcNotRefused**:
  $\neg \mathsf{O}\,\mathtt{msg}(\mathtt{Store.input} = \mathtt{startPaymentNack})$
- **OfferBeforeBank**:
  $\mathtt{msg}(\mathtt{Bank.input} = \mathtt{startPayment}) \Rightarrow$
  $\mathsf{O}\,\mathtt{msg}(\mathtt{Store.input} = \mathtt{offerAck}))$
- **NotAvailCount**:
  $\mathsf{count}\,(\mathtt{msg}(\mathtt{Store.output} = \mathtt{notAvail}))$
- **RetriesOnSuccCount**:
  $\mathsf{O}\,(\mathtt{cause}(\mathtt{VOS.state} = \mathtt{SUCC}))?$
  $\mathsf{count}\,(\mathtt{msg}(\mathtt{VOS.output} = \mathtt{offer}))) : 0$
- **PaymentTime**:
  $\mathsf{time}\,((\neg(\mathtt{cause}(\mathtt{Bank.state} = \mathtt{SUCC}, \mathtt{FAIL}))\mathsf{S}$
  $\mathtt{msg}(\mathtt{Bank.input} = \mathtt{startPayment})))$
- **GlobalStoreCcNotRefuse**:
  $\mathsf{And}\,(\neg\mathsf{O}\,\mathtt{msg}(\mathtt{Store.input} = \mathtt{startPaymentNack}))$
- **CountStoreCCRefused**:
  $\mathsf{Count}\,(\mathsf{O}\,\mathtt{msg}(\mathtt{Store.input} = \mathtt{startPaymentNack}))$

- **AverageUserRetriesCount**:
  Average (count (msg(VOS.output = offer)))
- **AveragePaymentTime**:
  Average (time ((¬(cause(Bank.state = SUCC, FAIL))
  S msg(Bank.input = startPayment))))

For both classes of properties, we have implemented a procedure that translates automatically an instance RTML formula into the Java code implementing the monitor. This has been realized within our toolset, and presented e.g. in [MMF+06].

1.4.4 Experiments: verification

As discussed in [MMF+06], both for the VOS and for its components, there are several properties that it may be interesting to verify. We tested the validation functionality provided by the WS-verify component of the toolset for this purpose, e.g. we checked that some previously defined properties do not hold in general. As shown in Fig. 1.18, counterexamples are produced and emitted as Message Sequence Charts.

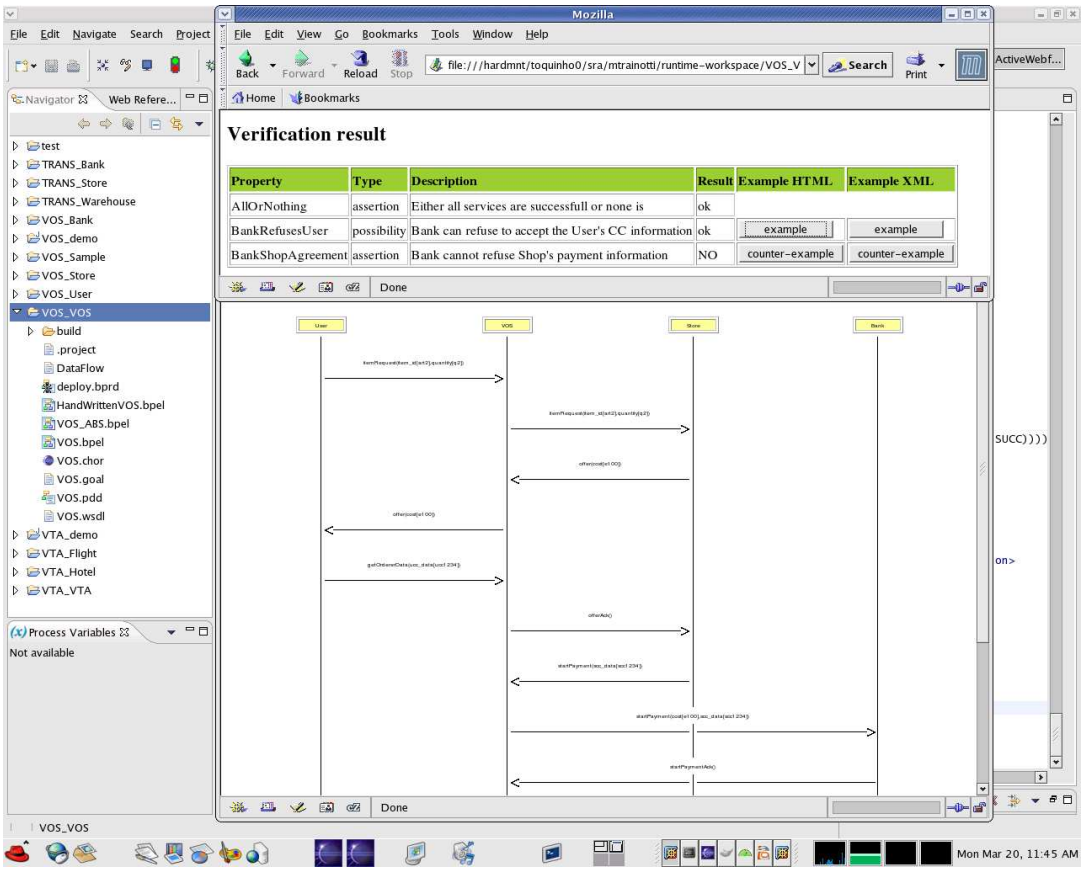


Figure 1.18: Verifying the services.

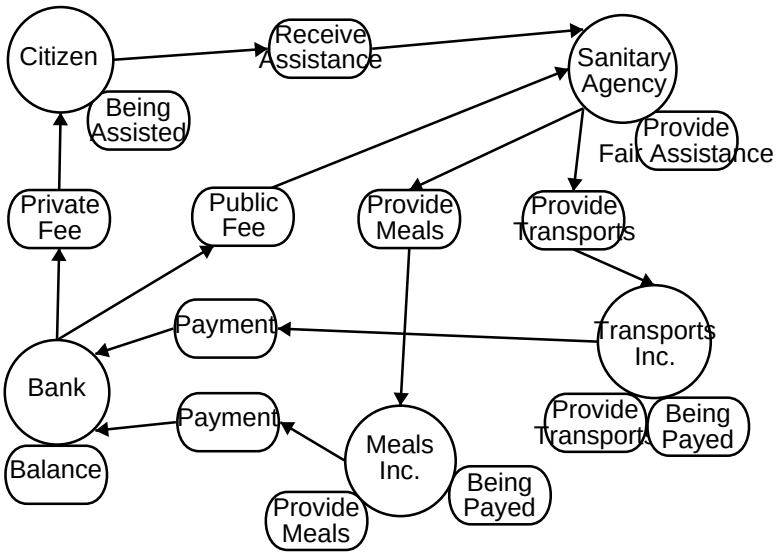


Figure 1.19: High level business requirements model.

1.5 The Health-care scenario

1.5.1 Description

We now step to consider a case-study in the field of public welfare, extracted from a larger domain analysis concerning the local government of Trentino (Italy); this will be used as a testbed for the Formal Tropos development methodology, which allows refining (informal) Tropos [GMPS05, BGG+04] diagrams into formal specifications that define the dynamic aspect of models, which can be then formally validated. Figure 1.19 is a Tropos diagram that provides a high-level description of the case-study domain. It represents the main *actors* and *goals* of the domain: the Citizen that aims at being assisted; the SanitaryAgency which aims at providing a fair assistance to the citizens; the TransportsInc which provides transportation services; the MealsInc which delivers meals at home; and the Bank which handles the government's finances. The picture also describes the dependencies and expectations that exist among these actors. For instance, the citizen depends on the sanitary agency for being assisted, and this is formulated in the model with dependency ReceiveAssistance from Citizen to SanitaryAgency.

Starting from this high-level view of the organizational or business system, the Tropos methodology proceeds with an incremental refinement process (see Figure 1.20). Goals are decomposed into sub-goals, or operationalized into tasks, taking into account the dependencies existing among the different actors. For instance, the goals BeingAssisted and ProvideFairAssistance are refined in order to reflect the "contract" that governs the way the assistance is provided by the SanitaryAgency to the Citizen. More precisely, the Citizen refines the goal BeingAssisted into the three sub-task of DoRequest, ReceiveService

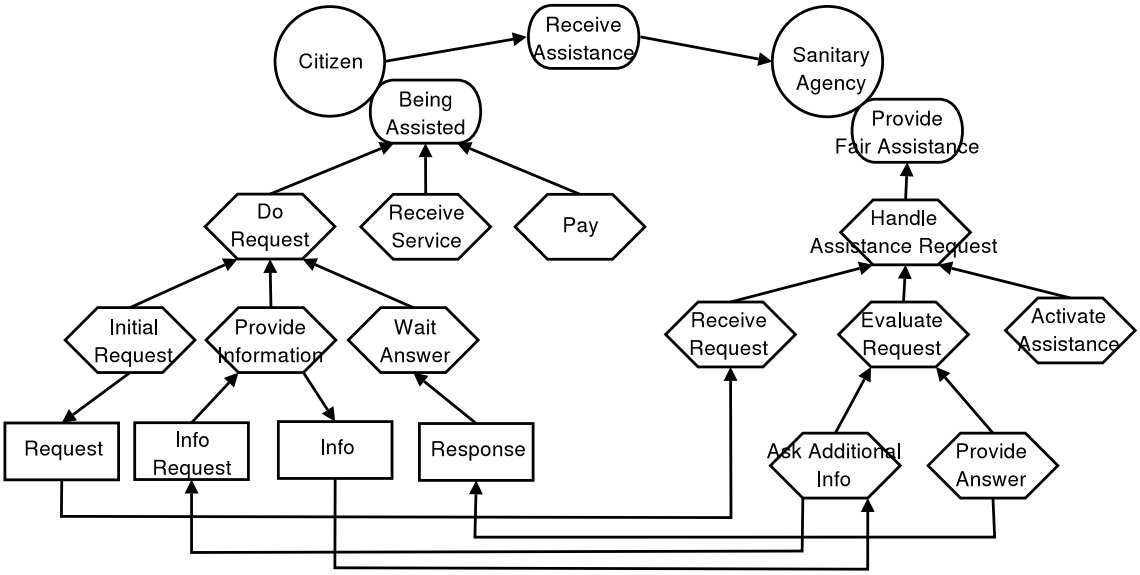


Figure 1.20: Requirements model refinement.

and Pay. DoRequest is further refined into InitialRequest, ProvideInformation, WaitAnswer. On the other side, the SanitaryAgency refines the goal ProvideFairAssistance into the task HandleAssistanceRequest, which is further refined into ReceiveRequest, EvaluateRequest and ActivateAssistance.

The refinement procedure ends once we have identified all basic tasks that define the business process. To these basic tasks we associate messages that describe the basic interactions among actors. For instance, task InitialRequest requires to send a message Request to the SanitaryAgency. This message is received and processed by the SanitaryAgency task ReceiveRequest. The task AskAdditionalInfo requires to send a message InfoRequest to the Citizen which receives and processes it with task ProvideInformation and responds with an Info message. Once sufficient information has been gathered, the SanitaryAgency sends a Response message to the Citizen. Figure 1.20 shows the refinement for the interactions between Citizen and SanitaryAgency. Similar refinements need to be done also for the other tasks and interactions in the domain.

The Tropos graphical models produced in the refinement process have a formal counterpart described in the Formal Tropos specification language. *Formal Tropos* (hereafter FT) has been designed to supplement Tropos models with a precise description of their dynamic aspects. In FT the focus is on the circumstances in which the goals and tasks arise, and on the conditions that lead to their fulfillment. In this way, the dynamic aspects of a requirements specification are introduced at the strategic level, without requiring an operationalization of the specification. A precise definition of FT and of its semantics can be found in [FLM$^+$03].

In the FT specification of Figure 1.21, the first three invariants of task DoRequest describe the expected evolution of the task and its relations with the subtasks. Namely,

```
ENTITY AssistanceNeed
ENTITY Query

ACTOR Citizen
ACTOR SanitaryAgency

GOAL DEPENDENCY ReceiveAssistance
  Mode maintain
  Depender Citizen
  Dependee SanitaryAgency
  Creation condition EXISTS ba: BeingAssisted (ba.actor = depender)
  Invariant F EXISTS pfa: ProvideFairAssistance (pfa.actor = dependee & Fulfilled(pfa))
  Fulfillment condition FORALL dr: DoRequest (
                          (dr.actor = depender & Fulfilled(dr) & dr.result) ->
                             F EXISTS rs: ReceiveService (rs.actor = depender & Fulfilled(rs)))

TASK DoRequest
  Mode achieve
  Actor Citizen
  Super BeingAssisted
  Attribute constant need: AssistanceNeed
            result: boolean
  Invariant F EXISTS ir: InitialRequest (ir.super = self)
  Invariant EXISTS ir: InitialRequest (ir.super = self & Fulfilled(ir))
               -> F EXISTS pi: ProvideInformation (pi.super = self)
  Invariant EXISTS pi: ProvideInformation (pi.super = self & Fulfilled(pi))
               -> F EXISTS wa: WaitAnswer (wa.super = self)
  Invariant Fulfilled(self) -> EXISTS wa: WaitAnswer
               (wa.super = self & Fulfilled(wa) & (result <-> wa.result))
  Fulfillment definition EXISTS wa: WaitAnswer (wa.super = self & Fulfilled(wa))

TASK InitialRequest
  Mode achieve
  Actor Citizen
  Super FareRichiesta
  Invariant F EXISTS r: Request (r.sender = actor & r.need = super.need)
  Fulfillment definition EXISTS r: Request (r.sender = actor & r.need = super.need)

MESSAGE Request
  Sender Citizen
  Receiver SanitaryAgency
  Attribute constant need: AssistanceNeed
  Creation condition Exists dr: DoRequest (dr.actor = sender & dr.need = need)
```

Figure 1.21: Formal Tropos specification.

if the task DoRequest is started, then eventually sub-task InitialRequest is entered (1st invariant). After InitialRequest has ended, sub-task ProvideInformation is eventually entered (2nd invariant). And after also this sub-task has ended, WaitAnswer is eventually started (3rd invariant). The fourth invariant constrains the value of attribute result of the task to the value of the same attribute of sub-task WaitAnswer once this sub-task has ended. Finally, the **Fulfillment definition** tells us that the sub-task WaitAnswer has to complete before we can consider the DoRequest task fulfilled (necessary condition) and that, if WaitAnswer has completed, then DoRequest will eventually be fulfilled (sufficient condition).

As discussed in [KPR04b], it is possible to refine the business requirements model described above into a business process model. The key idea is to associate WS-BPEL code to the high-level tasks of the actors of the domain (e.g., task DoRequest of actor Citizen, or task HandleAssistanceRequest of the SanitaryAgency).

The formal business requirements model already contains several pieces of information that can be exploited to generate a WS-BPEL specification. For instance, it is possible to automatically generate the definition of messages, ports, and services for the business domains — these elements define the WSDL document associated to the WS-BPEL specification. The description of the process model has to be completed by defining the body of the business process corresponding to the task. In Tropos, this is achieved by associating to the task a business process defined in the WS-BPEL language. For instance, the business process corresponding to the task of submitting a request is described by the WS-BPEL specification in Figure 1.22.

The process contains the variables need and result, which are already present in the formal requirements specification, and the additional variables waitResponse, vRequest, vInfoRequest, vInfo, and vResponse. The process behaves as follows. First, an initialization step is performed, during which the variable waitResponse is set to true, and the message Request is prepared by setting its need field. The Request message is sent in the following ⟨`invoke`⟩ command. A ⟨`while`⟩ loop is then entered, and its body is repeated until variable waitResponse becomes false. The body consists of a ⟨`pick`⟩ instruction which suspends the execution of the process until a InfoRequest or a Response message is received. If a InfoRequest message is received, a corresponding Info message is prepared and sent. The emitted Info message refers to the query contained in the received InfoRequest message. If a Response message is received, then the result variable of the process is set to reflect the result field of the received message. Moreover, the waitResponse variable is set to false, so that we can exit from the ⟨`while`⟩ loop.

Some additional attributes, which are specific of Tropos, are added to the WS-BPEL commands. These attributes are used to connect the evolution of the WS-BPEL process with the evolution of the requirements model. The event attributes describe which subtasks of DoRequest are supposed to be created or fulfilled in the requirements model when a given point is reached in the WS-BPEL code. For instance, sub-task InitialRequest is created during the initialization step and is fulfilled after the Request message has been

```
<variables>
  <variable name="need" messageType="Need"/>
  <variable name="result" type="boolean"/>
  <variable name="vRequest" messageType="Request"/>
  <variable name="vInfoRequest" messageType="InfoRequest"/>
  <variable name="vInfo" messageType="Info"/>
  <variable name="vResponse" messageType="Response"/>
  <variable name="waitResponse" type="boolean"/></variables>

<sequence name="DoRequestBody">
  <assign name="Initialization" event="Create ir: InitialRequest (ir.super = self)">
    <copy><from expression="true()"/><to variable="waitResponse"/></copy>
    <copy><from variable="need"/><to variable="vRequest" part="need"/></copy></assign>
  <invoke name="SendRequest" operation="oRequest" inputVariable="vRequest"/>
  <empty name="PhaseSwitch"
    event="Fulfill ir: InitialRequest (ir.super = self) & Create pi: ProvideInformation (pi.super = self)"/>
  <while name="Cycle" condition="getVariableData('waitResponse')">
    <pick name="WaitMessage">
      <onMessage name="InfoRequest" operation="oInfoRequest" outputVariable="vInfoRequest">
        <sequence name="AnswerToInfoRequest">
          <assign name="PrepareInfo">
            <copy><from variable="vInfoRequest" part="query"/>
                  <to variable="vInfo" part="query"/></copy></assign>
          <invoke name="Info" operation="oInfo" inputVariable="vInfo"/>
        </sequence></onMessage>
      <onMessage name="Response" operation="oResponse" outputVariable="vResponse"
         event="Fulfill pi: ProvideInformation (pi.super = self) & Create wa: WaitAnswer (wa.super = self)">
        <assign name="LeaveLoop">
          <copy><from expression="false()"/><to variable="waitResponse"/></copy>
          <copy><from variable="vResponse" part="result"/><to variable="result"/></copy></assign></onMessage>
    </pick>
  </while>
  <empty name="DoRequestFulfilled" event="Fulfill wa: WaitAnswer (wa.super = self)"
     constraint="Forall wa: WaitAnswer (wa.super = self -> G (wa.result <-> self.result))"/>
</sequence>
```

Figure 1.22: WS-BPEL process for task DoRequest of actor Citizen.

sent (the WS-BPEL command ⟨empty⟩ is used to place this fulfillment event in the right position of the process). The constraint attributes define additional constraints between the requirements layer and the process layer. They are typically used to define the values of the attributes of the sub-tasks. For instance, the constraint attribute of Figure 1.22 binds the value of attribute result of the WaitAnswer sub-task to the value of variable result of the WS-BPEL process.

1.5.2 Experiments: verification

Our experiments on the Health-care scenario focused on two formal validation techniques first presented in [KPR04b, KPR04a]. The first relies on FT's precise semantics to allow the verification of business requirements. The second consists in validating the WS-BPEL process obtained as a refinement of business requirements, considering formal requirements. Both validation techniques have been implemented within T-TOOL [FLM+03]. T-TOOL uses symbolic model checking techniques to perform the verification, and is based on the NUSMV [CCG+02] state-of-the-art symbolic model checker.

In our first round of tests, we validate a requirements specification in FT, by allowing the designer to specify properties that the requirements model is supposed to satisfy. We distinguish between **Assertion** properties, which describe conditions that should hold for all valid evolutions of the specification, and **Possibility** properties, which describe conditions that should hold for at least one valid evolution.

```
POSSIBILITY P1
  Exists dr: DoRequest (Fulfilled(dr))

ASSERTION A1
  Forall c: Citizen (
    Forall r: Response (r.receiver = c -> ! r.result) ->
      Forall rs: ReceiveService (rs.actor = c -> ! Fulfilled(rs)))

ASSERTION A2
  Forall dr: DoRequest (
   (Exists ra: ReceiveAssistance (ra.depender = dr.actor & Fulfilled(ra)
    & Forall r: Request (r.sender = dr.actor & r.need = dr.need -> r.receiver = ra.dependee)))
      -> (F Fulfilled(dr)))
```

Figure 1.23: Validation properties on the requirements model.

Figure 1.23 reports an excerpt of desired properties for the considered case-study. Possibility P1 aims at guaranteeing that the set of constraints of the formal business requirements specification allow for the fulfillment of the task of doing a request in some scenario of the model. Assertion A1 requires that it is not possible for the citizen to fulfill its goal of receiving assistance services unless a positive answer to a request from the sanitary agency has been received. Finally, assertion A2 requires that the task of doing a request is eventually fulfilled along every scenario under the condition that: there is a sanitary agency that is bounded to provide assistance to the user (citizen's dependency ReceiveAssistance); and, the citizen sends the requests to that particular sanitary agency.

To perform the validation, the T-TOOL translates an FT specification into the input language of NUSMV, which is then asked to perform the actual verification. Since model checking requires a finite state model, for translation purposes, upper bounds need to be specified to the number of instances of the different classes that appear in the formal specification. Given these bounds, a finite state automaton is built. Its states describe valid configurations of class instances, according to the class signatures and attributes that appear in the formal specification. Its transitions define valid evolutions of these configurations according to some generic constraints that capture the semantics of FT, e.g., that constant attributes should not change over time, or that, once fulfilled, a goal stays fulfilled forever. The creation, invariant, and fulfillment constraints of the various classes are collected in a set $\{C_i \mid i \in I\}$ of temporal constraints. In this way, the valid behaviors of a model are those executions of the finite-state automaton that satisfy all temporal constraints C_i. Checking if assertion A is valid corresponds to checking whether the implication $\bigwedge_{i \in I} C_i \Rightarrow A$ holds in the model, i.e., if all valid scenarios also satisfy the assertion A. Checking if possibility P holds amount to check whether $\bigwedge_{i \in I} C_i \wedge P$ is satisfiable, i.e., if there is some scenario that satisfies the constraints and the property. In both cases, the verification of a property is translated to the verification of an LTL formula. In [FLM+03] we have shown how this verification can be performed efficiently using NUSMV.

All the properties in Figure 1.23 are true on the final version of the formal requirements model of the considered case-study. However, this result has required several revision steps, where both the model and the properties have been adjusted to capture the intended behaviors of the domain. For instance, assertion A2 had a crucial role in the

process of precisely defining the mutual expectations incarnated by dependency Receive-Assistance, and captured by the fulfillment constraints specified for this dependency as it can be seen in Figure 1.21.

To verify the WS-BPEL business processes obtained by our refinement methodology, we first considered the formal queries that appear in Figure 1.23 on the more detailed model. Another possibility is checking that the refined model satisfies the requirements described by the **Creation**, **Invariant**, and **Fulfi llment** constraints enforced in the requirements model for task DoRequest and its sub-tasks.

To support these kinds of verification, we have extended the T-TOOL with a translation of WS-BPEL processes in NUSMV finite state machines, considering a subset of WS-BPEL which covers all the constructs used in Figure 1.22.

By applying this approach to the verification of the WS-BPEL process of Figure 1.22 we obtain that all verification tasks are successful, and hence this process is a correct implementation of the requirements of task DoRequest. If we modify the code of the process, e.g., by disallowing the reception of one of the two message in the ⟨`pick`⟩ command, then the verification detects problems. If we disallow the reception of the InfoRequest message, for instance, assertion A2 turns out to be false. Indeed, if the sanitary agency is requesting some information, the citizen is not able to answer to the request and a deadlock in the process is reached. The counter-example of Figure 1.24 is generated in this case. If we disallow the reception of the Response, not only assertion A2, but also possibility P1 becomes false. Indeed, if we do not receive the response, it is not possible to fulfill DoRequest.

We remark that the approach described here allows also for another kind of verification. Namely, in order to check that the process model is correct, one can show that it is equivalent to the requirements model according to a suitable behavioral equivalence.

1.6 The Loan Approval scenario

1.6.1 Description

We now discuss a Loan Approval scenario which illustrates clearly the data-related problems of composition analysis. The nominal scenario may be summarized as follows. The user requests a loan approval specifying a desired amount and the Loan Approval (LA) process is aimed to iteratively look for an amount that may be safely approved for the user. Each iteration is performed as follows. The current amount is being checked. If the amount is low, then the Assessor service is called, otherwise the Approver service is used. If the Assessor considers the amount to be risky, then it is also passed to the Approver, otherwise the amount is approved and the loop terminates. If the Approver provides a negative answer, the amount is reduced and the iteration continues. We model the composition as a set of WS-BPEL specifications that represent the behaviors of every participant.

```
-- Assertion A2 is false as demonstrated by the
-- following execution sequence
-> State 3.1 <-
    Citizen_1.exists = 1
    AssistanceNeed_1.exists = 1
    BeingAssisted_1.exists = 1
    BeingAssisted_1.actor = Citizen_1
    BeingAssisted_1.fulfilled = 0
    DoRequest_1.exists = 1
    DoRequest_1.actor = Citizen_1
    DoRequest_1.fulfilled = 0
    DoRequest_1.waitResponse = 0
    DoRequest_1.answer = 0
    DoRequest_1.pc = Initialization
    DoRequest_1.need = AssistanceNeed_1
    DoRequest_1.super = BeingAssisted_1
-> State 3.2 <-
    SanitaryAgency_1.exists = 1
    InitialRequest_1.exists = 1
    InitialRequest_1.actor = Citizen_1
    InitialRequest_1.fulfilled = 0
    InitialRequest_1.super = DoRequest_1
    Request_1.exists = 1
    Request_1.need = AssistanceNeed_1
    Request_1.initiator = InitialRequest_1
    Request_1.sender = Citizen_1
    Request_1.receiver = SanitaryAgency_1
    DoRequest_1.waitResponse = 1
    DoRequest_1.pc = SendRequest
-> State 3.3 <-
    InitialRequest_1.fulfilled = 1
    ProvideInformation_1.exists = 1
    ProvideInformation_1.actor = Citizen_1
    ProvideInformation_1.super = DoRequest_1
    ProvideInformation_1.fulfilled = 0
    DoRequest_1.pc = PhaseSwitch
    Query_1.exists = 1
    ProvideFairAssistance_1.exists = 1
    ...
    ProvideFairAssistance_1.fulfilled = 1
    ...
    HandleAssistanceRequest_1.exists = 1
    ...
    HandleAssistanceRequest_1.fulfilled = 1
    ReceiveRequest_1.exists = 1
    ...
    ReceiveRequest_1.fulfilled = 1
    EvaluateRequest_1.exists = 1
    ...
    EvaluateRequest_1.fulfilled = 0
-> State 3.4 <-
    ReceiveAssistance_1.exists = 1
    ReceiveAssistance_1.dependee = SanitaryAgency_1
    ReceiveAssistance_1.depender = Citizen_1
    ReceiveAssistance_1.fulfilled = 0
    DoRequest_1.pc = Cycle
-> Loop starts here <-
-> State 3.5 <-
    InfoRequest_1.exists = 1
    InfoRequest_1.query = Query_1
    InfoRequest_1.ref = Request_1
    InfoRequest_1.sender = SanitaryAgency_1
    InfoRequest_1.receiver = Citizen_1
    ReceiveAssistance_1.fulfilled = 1
    DoRequest_1.pc = WaitMessage
-> Loop <-
```

Figure 1.24: An example of counter-example generated by NuSMV.

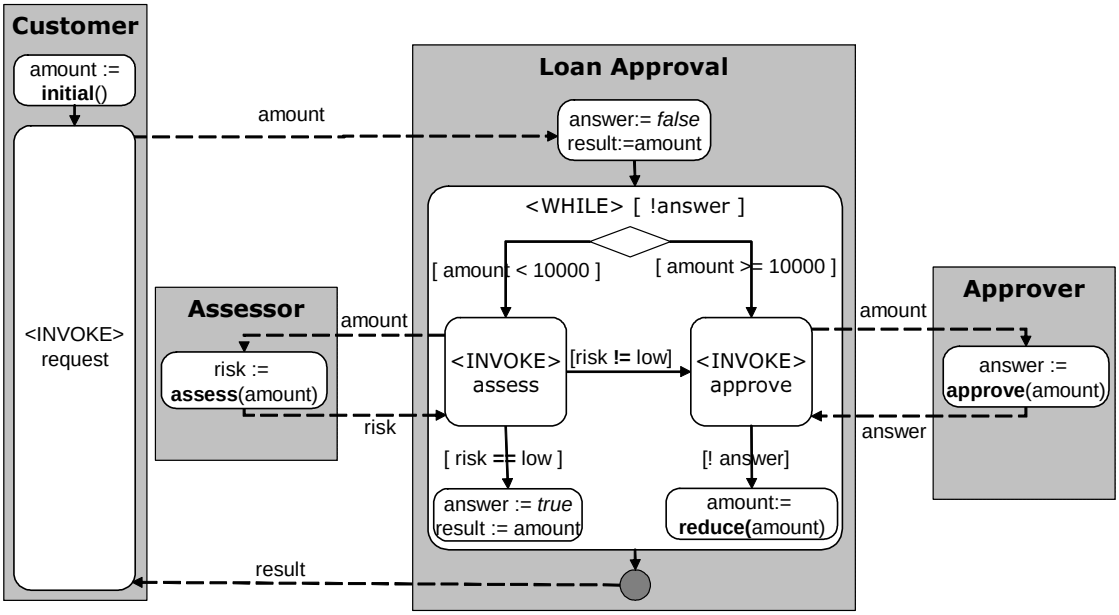
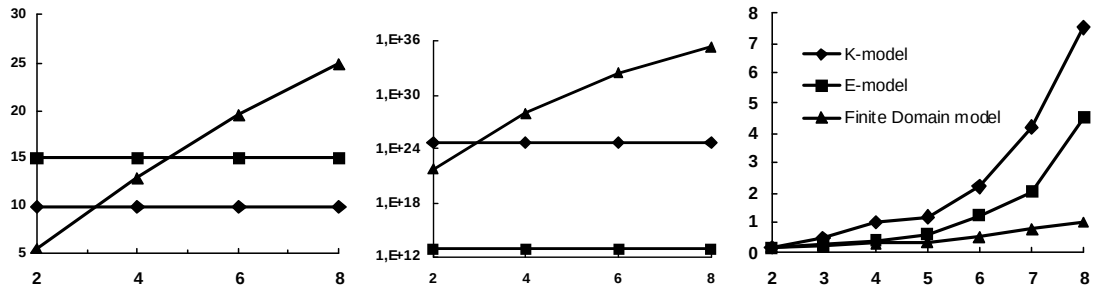


Figure 1.25: Loan Approval Example

A conceptual model of the composition scenario is represented in Fig. 1.25.



(a) (b) (c)

Figure 1.26: Experimental results

1.6.2 Experiments: verification

The interesting fact about the Loan Approval scenario is that verification techniques based on standard FSM encoding and model checking, such as those presented in [KPR04b], cannot be applied.

Indeed, the data variables, being manipulated in this scenario have infinite domains, such as strings, integers, structures, and aggregations of them. Therefore a finite representation of the system is not possible. A predicate abstraction approach [GS97] allows to avoid this problem by considering only relevant data flow properties and ignoring the others. For instance, in the above case study it is not needed to enumerate all the possible amounts that the user may request. Instead, it might be sufficient to evaluate the fact that the amount is big, which can be represented with one proposition.

The abstraction, however, is not always able to faithfully describe the real system. Indeed, the abstract model may *over-approximate* the real system, that is may contain some unrealizable behaviors. If one is interested to check that every behavior satisfies a property (*universal* property), then the violation of the property in an over-approximated abstraction does not imply the violation in a real system. Analogously, if the verification of the *existential* property (i.e. some behavior satisfies the property) is violated in *under-approximation* (i.e. the abstract system that contains less behaviors then the real one), it is not necessarily violated in the real system.

The way we tackled these issues in [KP06] is through an interplay of over- and under-approximations during the verification process, whose formal details are spelled out in Deliverable D3.3, and which we then implemented within the Astro toolkit (available at `http://astroproject.org`). We exploited the logic of *equalities and uninterpreted functions* (UIF, [BD94]) for the computation of abstract models. This computation is performed by a parametric reasoner that, given a composition specification, a set of abstract predicated, and possibly a set of function constraints, generates two abstract models

of the composition, namely B-model and K-model. These models define a control flow of Web service composition in terms of states and transitions, and a data flow in terms of boolean variables corresponding to equalities between terms (e.g. variables, functions, and constants). In case of B-model the resulting specification contains a set of axioms, specifying rules for equalities consistency, functions rigidness, etc. In case of K-model these axioms are used to compute a predicate valuation that defines a transition execution effect.

We evaluated this approach on series of experiments. First, we applied it to the analysis of (extended) Loan Approval case study. We used this example to compare the performance of the verification in the presented framework versus the verification under finite data domains (like, for instance, in [FBS04]). We used 35 abstract propositions (and their negations) for 19 process variables and constants of 3 data types. The comparison of the verification times (in seconds) and of the state spaces are represented in Fig. 1.26(a) and Fig. 1.26(b) respectively. The horizontal axis of the graphs shows the number of domain values per type. In the finite domains approach the verification time (and space) grows when the data domains increase. On the contrary, the verification performance in our abstraction-based approach does not change.

Chapter 2

Synthetic scenarios

2.1 Approach

On top of analyzing scenarios coming from real case studies and from existing literature on SOC, we considered a variety of synthetic scenarios; our main aim, in particular, was that of identifying which structural and dimensional features of web services impact on the complexity of their composition, and in which way.

In particular, our analysis will be organized as follows:

1. We first consider "simple" services, which encode question-and-answer protocols that simply receive a message, produce an output and terminate. We will study both deterministic services, whose answer is a function of the input, and nondeterministic ones, which may also return a failure message based on an internal choice.

2. Then, we consider more complex services, whose protocols are more lengthy, and which feature sources of internal nondeterminism. This means that, in order to satisfy a requirement, a composed service will be forced to take complex choices, depending on the behavior of the components.

   In particular, in order to study the impact of structural factors over the composition, we will consider in turn:

   (a) "binary unbalanced" protocols, where, at each step, a conclusive failure may be signaled by the protocol; in fact, these protocols correspond to "concatenations" of simple nondeterministic services;

   (b) "binary balanced protocols", where two possible answers are possible at each interaction, none of which signaling a conclusive failure;

   (c) "n-ary balanced protocols", which (at a high level) generalize binary balanced protocols in terms of number of possible answers.

For both sets, varied the size of the sets of services that need to be composed, analyzing the performance of the various phases in the composition task: construction of the belief-level domain, internalization and search. In all cases, the modeling of services has been carried out in terms of WS-BPEL services, built automatically (together with the necessary WSDL files associated to them) via specific test generation scripts.

2.2 Simple services

2.2.1 Description

We start by considering the simpler services that may be thought of, which model a (fully deterministic, predictable) function computation: each component W_i receives an input D, computes a function $f_i(D)$ and returns it as an output, terminating.

We then step to the smaller possible web services that encode a possibly failing protocol; i.e., each web service will accept a request, and either compute a function, or fail. In the former case, the service will wait for a ack/nack signal, depending on which it finally fails or terminates successfully. Thus such services will feature two branching points; the first one in the protocol encodes a form of "internal" nondeterminism, while the second represents and "external" nondeterminism which is necessary in order to be capable to drive the service to fail or to succeed.

2.2.2 Experiments: composition

Given a set of "atomic" components, which simply compute a function, the composition requirement consists of computing the nested function $f_{i_0}(f_{i_1}(\ldots(f_{i_n}(D))))$ (and have each component terminate successfully). This goal is easily generated in an automated way, and its solution requires a composed protocol that suitably interleaves the invocations to the components, passing at each step the current result.

We applied the ground-level composition technique presented in [PTB05], and the results are shown in Fig. 2.1. We are able to combine quite up to 20 services within a very reasonable time. We observe that belief-level construction has a significant relative cost; however, as expected, the partitioned representation helps out, and the time spent in this phase grows only linearly with the number of component services.

Once the belief-level system has been represented, it has to be internalized; constructing its internal representation, which implies computing the parallel product of each module, has a marginal cost, which however grows polynomially in the number of components.

For large sets of components, the most relevant cost is due to the search phase, which seems to grow up polynomially, but with some notable oscillations.

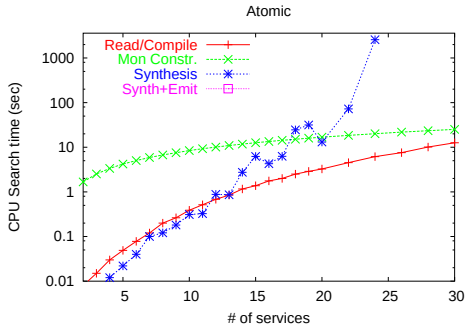


Figure 2.1: Combining N atomic services

To explain this, we remark that, in the vast majority of cases, exponentially large sets of states can be represented by polynomially large BDD structures, and thus manipulated within polynomial time. Since we adopt a breadth-first backward search style, given the domain under exam, composing N services will require computing a number of layers proportional to N; the number of nodes in each layer can be thought, at a high level of approximation, as growing polynomially at each backward iteration. All this would then result in an overall search time polynomial in N. However, breadth-first search is very memory-demanding, and when N is high, the garbage collection mechanism of the BDD package enters into play to rearrange data structures, and remove unused ones. This is what influences the performance of the search for the highest values of N.

When we step to simple services which may branch to failure, a reasonable composition requirement consists in asking *preferably* that the set of services compute a nested function $f_{i_0}(f_{i_1}(\ldots(f_{i_n}(D))))$ of a datum D, and have each component terminate successfully. In suborder, we require to have every started component terminate with failure. We remark that allowing the externally commanded refusal is necessary so to have services "controllable" enough to be able to compose them according to the requirement. Again, such a goal is easily generated via a script.

The composed protocol must implement the following behavior, which combines transferring data "in the right order" between services, and detecting and handling failure situations by appropriately driving each suspended service to fail.

- invoke the service W_{i_n} computing f_{i_n}, passing it D
- if the service answers with failure, terminate
- receive D_n from W_{i_n}, and invoke the service $W_{i_{n-1}}$ computing $f_{i_{n-1}}$, passing it D_n
- if $W_{i_{n-1}}$ answers with failure, send a $nack$ to W_{i_n} and terminate
- ...
- receive D_1 from W_{i_1}, and invoke the service W_{i_0} computing f_{i_0}, passing it D_1

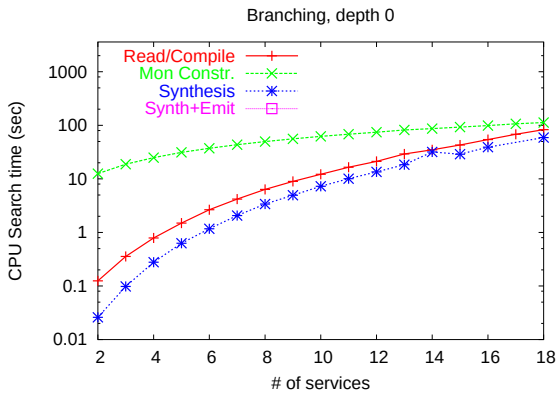


Figure 2.2: Combining N branching services

- if W_{i_0} answers with failure, send a $nack$ to every service $W_{i_1}, \ldots, W_{i_n}$ and terminate
- send ack to every service $W_{i_0}, \ldots, W_{i_n}$ and terminate.

The results of applying ground-level composition for this test are reported in fig 2.2.

We observe that it is possible to achieve, in a reasonable time, the composition of a rather high number of such services - approximately 20. In this case, the harder phase in the composition consists in the construction of the belief-level system $\Sigma_{\|}$. Again, thanks to the modular modeling allowed by the SMV language, the weight of constructing and emitting $\Sigma_{\|}$ grows only linearly with the number of component services - while its interpretation to build the internal representation of $\Sigma_{\|}$ is polynomial. Both costs are about an order of magnitude higher than in the case of deterministic components.

The complexity of searching for a controller grows according to a polynomial curve, not unlikely the one for the predictable services. Indeed, the cost of the composition is only marginally higher than the one for predictable services.

This result is somehow to be expected: the cost of the search basically depends on the number of states to be represented in the number N of layers generated by the backward search, and N amounts to the length of the longer interaction needed to compose the services. But, in this test, given a set of n components, N is the same regardless of whether the components are predictable or not, and the number of visited states do not differ significantly.

This indicates that our approach is particularly suited to deal with nondeterministic services: not only our representation is general enough to capture nondeterminism, but also the performance of composition seems to be affected only marginally by the need of handling different contingencies. In fact, for the specific case of fully predictable atomic services, it is possible to adopt simpler formalisms (such as STRIPS [FN71]) and more focused search techniques (e.g. forms of classical heuristic planning, see [HN01]) to

obtain a better scalability. We remark, however, that we intend to provide an approach that is general and encompasses situations like the ones found in practice, where services obey to an internal logic and expose a nondeterministic behavior.

2.3 Binary unbalanced protocols

2.3.1 Description

We start by analyzing services which generalize the kind of nondeterministic service used in the "simple" test, by encoding a "sequential" pattern of nondeterministic interaction, . Each "cell" of the pattern represents one interaction analogous to the one explained in the previous subsection: a request is received, and either a refusal message is given back, or a function f_i is computed, and the service suspends for either an ack or a nack message. As such, each cell may fail (because it refuses computing f_i, or because it receives a $nack$) or succeed. Failure of the cell corresponds to failure of the service; success of the cell, instead, activates the next "cell", and the success of the service corresponds to the success of the last cell. We call such components "binary unbalanced"; the number of their branches grows linearly with the number of cells; Fig. 2.3 schematically show a component service with 3 cells.

2.3.2 Experiments: composition

We first consider the combination of a number of services containing 1,2,3,4 cells each, where the goal is, once more, to compute a nested function, or to have every component fail. The results are shown in Fig. 2.4.

In general, we observe that, similarly to the case of simple nondeterministic services, (a) the performance of belief-level construction degrades quasi-linearly with the number of the involved services (b) its internalization and the search degrade polynomially, and (c) the search degrades polynomially (with a higher polynomial factor).

On top of this, we can observe the following:

- Regarding belief-level construction and interpretation, as expected, their cost grow up when the components are more complex: approximately, of one order of magnitude between for each additional cell. Similarly to the previous tests, belief-level construction grows up linearly with the number of components, and belief-level internalization grows up polynomially.

- the performance of the search phase of our composition chain depends mainly on the total number of "cells" in the set of components, and quite marginally on whether the cells are glued together in few, large component services, or distributed

over several, small ones. For instance, we observe that to compose 12 cells, it takes about 10 seconds, regardless of whether we use 12 components with 1 cell each, 6 with two cells, or 4 with three cells. For large numbers of cells, fewer complex services are somehow preferable; this is due to the fact that in this case, the search is more constrained, since a smaller variety of input/output actions can be performed at each interaction step, and the cell ordering internal to a component W_i also constrains the ordering of the input/outputs concerning the functions computed by W_i.

- when composing the more complex amongst these services, memory consumption becomes a major issue, mainly due to the breadth-first style adopted by our search algorithm. Indeed, instances where services have depth $d \geq 3$, as well as the composition of large sets of services with depth $d = 2$, are terminated not due to timeouts, but due to exceeding the 4GByte memory bound.

To study in more detail the relationships sketched above, we also analyze the combination of a fixed set of such services, where we vary the average number of cells in the component services. Fig. 2.5 reports the results for the composition of 2 to 5 services. These graphics show clearly that both the belief-level construction and internalization, and the search times, grow up exponentially with the number of cells in the components. Concerning belief-level construction and internalization, this is due to the fact that the size of the belief-level system of a component is (potentially) exponential in the number of states, and thus in the number of cells. Concerning search, this is due to the fact that the length of the composed service is, in this case, also proportional to the number of cells. Also, we can observe a sort of "step" discontinuity in the curves that refer to belief-level construction and internalization times. This comes from the fact that the time required for the larger component in the set dominates the overall performance in such phases.

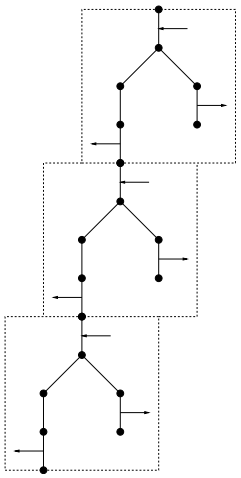


Figure 2.3: A 3-cell binary unbalanced component.

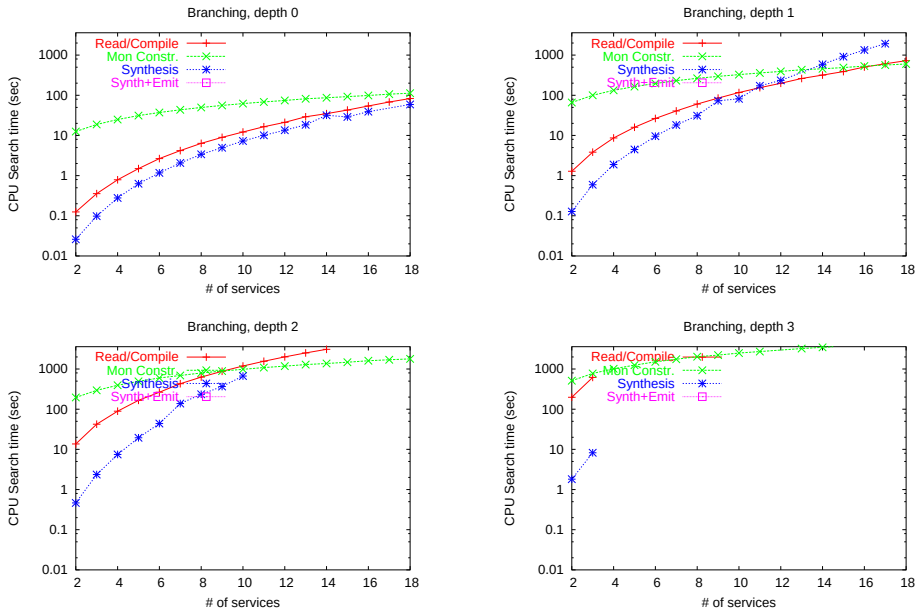


Figure 2.4: Combining N binary unbalanced services of depth 1,2,3,4

2.4 Binary balanced protocols

2.4.1 Description

The pattern above is asymmetric, and such that there exists only one successful path for each service. As a consequence, similar to what happens for simple services, the combination of the services fails as soon as one of the involved services fails.

To model a more general situation, where different success paths exist for the combination of services, and to analyze the combination of services with a non-trivial branching factor, we generalize the pattern above into a "balanced binary tree" of nondeterministic interaction cells. In this case, an interaction cell computes one of two different functions, sending out two different messages, and in each case awaiting for an ack/nack message to either terminate with success or fail. A balanced tree of such cells is such that the each success state of a cell coincides with the start cell of another one; the success of such a tree is the success of a leaf cell of the tree. Fig. 2.6 shows a schema of such a balanced component of depth 2.

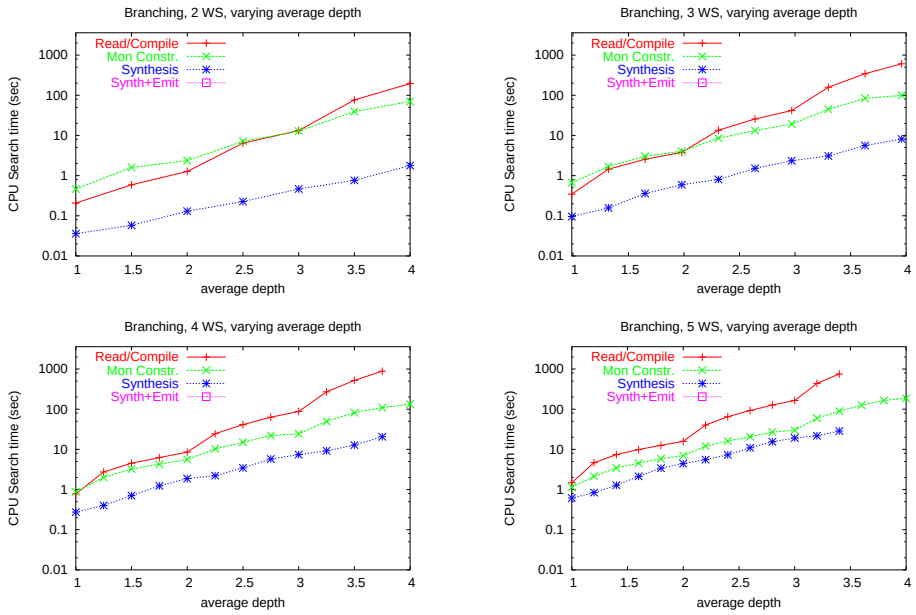


Figure 2.5: Combining 2,3,4,5 branching services of increasing depth

2.4.2 Experiments: composition

A goal for a set of such web services consists in computing one of several possible combination of functions

$$\bigvee_j f_{i_{0_j}}(f_{i_{1_j}}(\ldots(f_{i_{n_j}}(D))))$$

and having every component service succeed, or in suborder to have every service fail. The composed service then must decide "on the fly" in which order to invoke the component services, since the answer of one service may not only make it impossible to achieve success, but also determine which of the possible functions combinations can still be pursued.

We first consider combining increasingly large sets of balanced components of a fixed depth. The results are shown in Fig. 2.7.

Analogously to what we have seen for the composition of unbalanced nondeterministic components, the computational cost of performing the belief-level construction grows up quasi-linearly with the number of the component services, and similarly, the cost of the search seems to grow up only polynomially. The cost of belief-level construction is largely predominant over that of the search unless many services are combined, or services are small - but even for the smaller balanced components possible, the cost of the search becomes significant only with $N > 7$.

Similarly to what we did for unbalanced components, we also combine sets of fixed size of balanced branching components, and we vary the their average depth (i.e. their average number of "cells"). The results are presented in Fig. 2.8. Again, we witness an

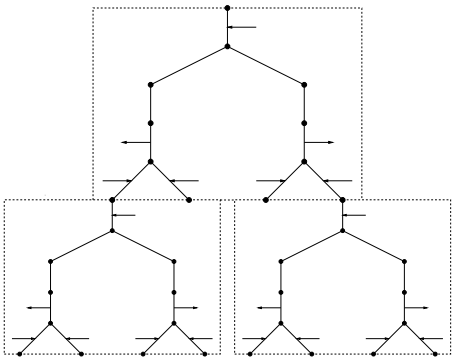


Figure 2.6: A 2-depth “balanced branch” component.

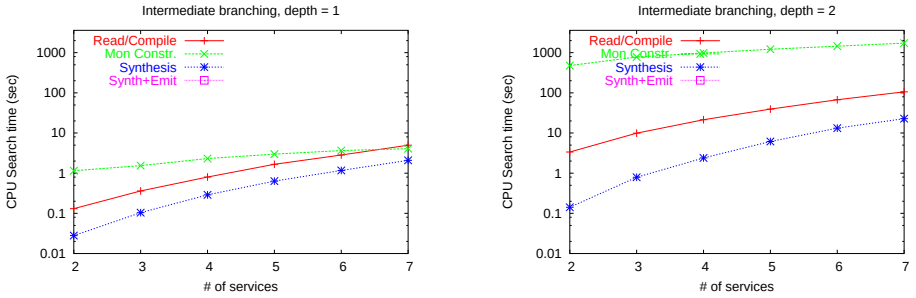


Figure 2.7: Combining N binary balanced services of fixed depth (1,2)

exponential dependency of the performance of both belief-level composition and search upon the depth of the component services. The fact the the time for belief-level construction grows by steps, being dominated by the one of the larger component, is even more evident in this case, where the size of a component is exponential (rather than linear) in its depth. Indeed, it is just because of the “step” introduced by components of depth 3 that tests fail for larger instances of the problems.

2.5 N-ary balanced protocols

2.5.1 Description

Finally, we remark that, while the services considered above generalize the structure of simple ones by combining sets of “interaction cells”, they all share the fact that only binary choices are involved: either two input messages can be received, or two different branches can be nondeterministically taken by the service.

We consider a variation in this aspect which generalizes in the structural dimension of width. In particular, we combine two services which implement a short nondeterministic protocol admitting a variety of choices, either because of internal nondeterminism, or

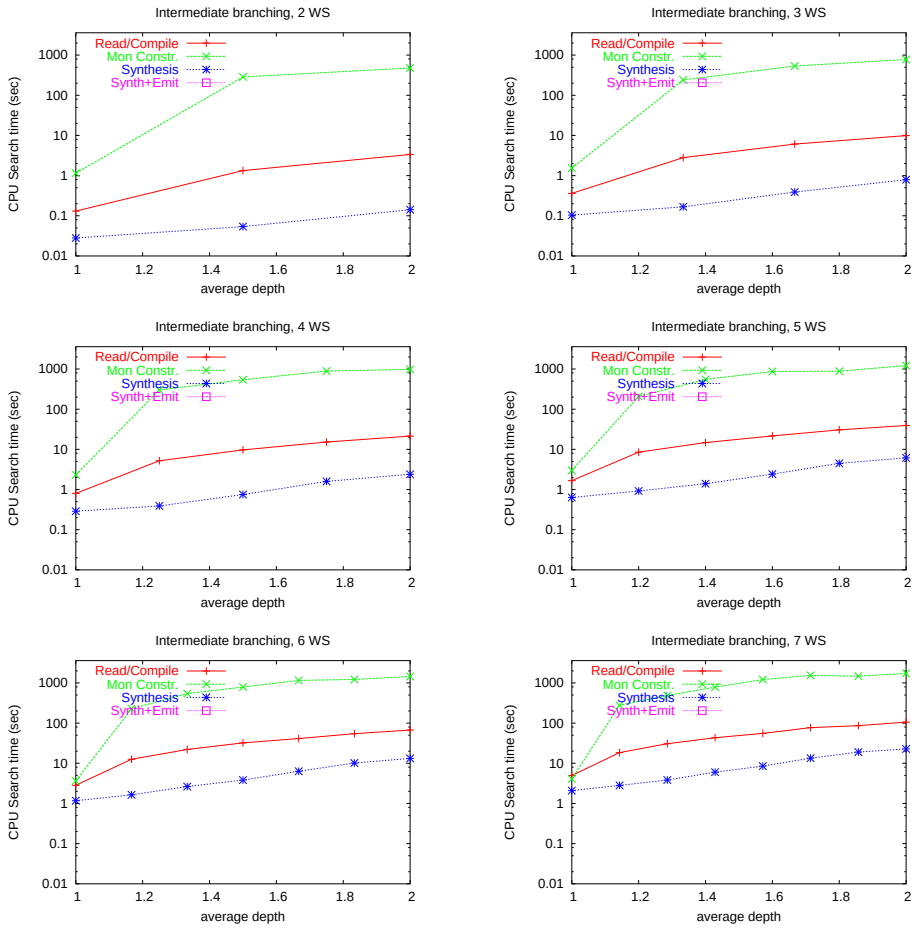


Figure 2.8: Combining sets of 2 to 7 balanced branching services of varying average depth

because different external controls are possible.

The first service S_1, upon receiving an input datum D, performs an internal choice and decides which function $f_i(D)$, out of a set of n, it will compute. After computing it, it returns the results and suspends for an ack/nack signal that leads it to success or failure states respectively. The second service S_2 may receive one of m different input signals, together with a datum D; depending on which signal did it receive, it computes one of m functions $g_i(D)$, and returns the result. After this, it suspends for an ack/nack signal that leads it to success or failure states respectively.

2.5.2 Experiments: composition

These two services can be composed by requiring that one of the possible function combinations

$$\bigvee g_i(f_j(D)))$$

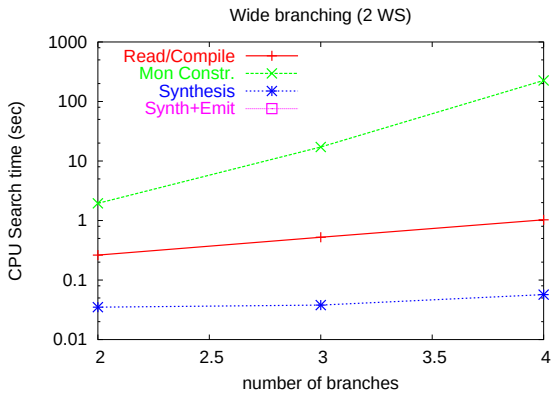


Figure 2.9: Combining two branching services of increasing branching factor

is computed; similarly to the case of balanced branching tree, this requires an "on-the-fly" decision, on the part of the composite service, that depends on the internal determinism of S_1.

The results are shown in Fig. 2.9, and highlight that belief-level construction is the crucial bottleneck for such a kind of composition, growing exponentially with the branching factor and dominating the search.

2.5.3 Correlation between results

Given the range of complexity and structures adopted in the tests, we want to analyze comparatively their results to study the impact of the various features over the performance of the composition task, decomposed in the belief-level construction and internalization phases, and in the search phase.

All tests confirm that the performance of the first two phases depends crucially on the size of the components into play, the larger component dominating the others. In fact, the time spent in belief-level construction seem to grow up exponentially with the number of states. The partitioned representation, however, allows building the belief-level system for several components, as the overall spent for constructing and internalizing it grows only polynomially with the number of components. As a result, large sets of reasonably simple components are effectively dealt with in these phases.

Concerning the search, its performance results from the complexity of two factors: the length of the composed protocol (i.e. the number of interactions with the components which are necessary to achieve a situation where the requirement is obeyed) and the variety of the input-output actions possible at each step. The breadth-first search style adopted during the search, in spite of the symbolic representation, implies a behavior which is essentially exponential in the size of the components and in the variety of exchanged messages, while it is polynomial in the number of components into play.

Nonetheless, we remark that our composition tool is capable of effectively tackling significantly complex composition problems, whose ad-hoc solution would have been far from trivial, time-demanding and error prone.

Chapter 3

History of the Deliverable

In this chapter, we summarize the way in which the activities described in this deliverable have evolved along the 4 years of the project.

3.1 1st Year

Year one of the project was mostly devoted to surveying the existing wealth of approaches that support web-service enactment, and to clearly draw connections with advanced AI techniques developed by the project partners. During this activity, a number of very prototypical examples have been tought of, and in many cases solved by manually recasting them as standard planning or model-checking problems, under strict assumptions.

As first result of this activity, we have been able to assess the feasibility of tackling such problems by adapting symbolic search techniques.

This triggered the adaptation activity, which started by the design of a unifying framework of the problems, based on the interpretation of WS-BPEL services in terms of finite state machines.

3.2 2nd Year

In the second year, having implemented the first version of the framework where automated translation of services is supported, we have started experimenting automated composition and verification of web services.

In particular, the P&S scenario has been the main testbed for experimenting "ground-level" techniques for composition, based on explicit instantiation of data values (see e.g. [PBB+04]).

The Health-care scenario has been instead the reference scenario to discuss our For-

mal Tropos-based methodology to develop and verify services starting from high-level requirements (see [KPR04b]).

As a result, we have been able to identify a number of directions to improve our verification and composition approaches, leading us to pursue their optimization, re-design and implementation.

Moreover, we started tackling the web service monitoring problem, by handcrafted tests first. As a result, we incorporated monitoring into out architecture, and proceeded to design a fully automated approach to monitor construction.

3.3 3rd Year

During the third year, we optimized our ground-level composition techniques, and developed a novel composition technique that improves scalability by a adopting a suitable "knowledge-level" abstraction. Our main testbed for these techniques have been the P&S and VTA scenarios, see [PTB05, PATB05, MMG+05]. We also started an activity of systematic analysis of composition scalability over synthetic scenarios.

At the same time, we achieved the integration of automated monitoring in our toolset, and tested it, together with automated verification, over the VTA, see [MMG+05]. We identified the need to be improve our monitoring property language, and to monitor service classes, other than service instances. This triggered a redesign activity in that respect.

3.4 4th Year

In the 4th year, we applied the idea of reasoning at a more abstract level, both to composition and verification. This has led to the notion of "data net"-based composition [MPT06], and approximation-based verification - which also encompasses the validation of time properties.

We have been testing these approaches over more complex scenarios, in particular the VOS and the Loan Approval. Results can be found in [MPT06, KP06, KPP06a, KPP06b].

We finalized the approach to automatically generate class monitors, and also tested it on the VOS scenario, see [BTPT06].

Bibliography

[AAF+02] A. Arkin, S. Askary, S. Fordin, W. Jekeli, K. Kawaguchi, D. Orchard, S. Pogliani, K. Riemer, S. Struble, P. Takacsi-Nagy, I. Trickovic, and S. Zimek. *Web Service Choreography Interface (WSCI) 1.0*, August 2002.

[ACD+03] T. Andrews, F. Curbera, H. Dolakia, J. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weeravarana. Business Process Execution Language for Web Services (version 1.1), 2003.

[Act] ActiveBPEL. The Open Source BPEL Engine - http://www.activebpel.org.

[BD94] J.R. Burch and D.L. Dill. Automated verification of pipelined microprocessor control. In *Proc. CAV'94*, 1994.

[BGG+04] Paolo Bresciani, Paolo Giorgini, Fausto Giunchiglia, John Mylopoulos, and Anna Perini. Tropos: An agent-oriented software development methodology. *Journal of Autonomous Agents and Multi-Agent Systems*, May 2004.

[BTPT06] Fabio Barbon, Paolo Traverso, Marco Pistore, and Michele Trainotti. Run-time monitoring of instances and classes of web service compositions. In *Proceedings of ICWS'06*, 2006.

[CCG+02] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NUSMV 2: An OpenSource Tool for Symbolic Model Checking. In *Proceedings of Computer Aided Verification Conference*, Copenhagen (DK), July 2002. Springer.

[CCGR00] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. NUSMV: a new symbolic model checker. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(4), 2000.

[CCMW] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web Service Definition Language (WSDL 1.1) - http://www.w3.org/TR/wsdl.

[DLPT02] U. Dal Lago, M. Pistore, and P. Traverso. Planning with a Language for Extended Goals. In *Proc. AAAI'02*, 2002.

[FBS04] X. Fu, T. Bultan, and J. Su. Analysis of Interacting BPEL Web Services. In *Proc. WWW'04*, 2004.

[FLM+03] A. Fuxman, L. Liu, J. Mylopoulos, M. Pistore, M. Roveri, and P. Traverso. Specifying and analyzing early requirements in Tropos. *Requirements Engineering*, 2003.

[FN71] R. E. Fikes and N. J. Nilsson. STRIPS: A new approach to theorem proving in problem solving. *Journal of Artificial Intelligence*, 2:189–208, 1971.

[FUMK03] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Model-based verification of Web Service Compositions. In *Proc. ASE'03*, 2003.

[GMPS05] Paolo Giorgini, John Mylopoulos, Anna Perini, and Angelo Susi. The tropos metamodel and its use. *Informatical journal*, 2005.

[GS97] S. Graf and H. Saidi. Construction of abstract state graph with PVS. In *Proc. CAV'97*, 1997.

[HN01] Jörg Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.

[Hol97] Gerard J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[KP06] Raman Kazhamiakin and Marco Pistore. Static Verification of Control and Data in Web Service Compositions. In *Proc. ICWS'06*, 2006.

[KPP06a] Raman Kazhamiakin, Paritosh K. Pandya, and Marco Pistore. Representation, Verification, and Computation of Timed Properties in Web Service Compositions. In *Proc. ICWS'06*, 2006.

[KPP06b] Raman Kazhamiakin, Paritosh K. Pandya, and Marco Pistore. Timed Modelling and Analysis in Web Service Compositions. In *Proc. ARES'06*, 2006.

[KPR04a] Raman Kazhamiakin, Marco Pistore, and Marco Roveri. Formal Verification of Requirements using SPIN: A Case Study on Web Services. In *Proc. SEFM'04*, pages 406–415, 2004.

[KPR04b] Raman Kazhamiakin, Marco Pistore, and Marco Roveri. A Framework for Integrating Business Processes and Business Requirements. In *Proc. EDOC'04*, pages 9–20, 2004.

[KPS06] Raman Kazhamiakin, Marco Pistore, and Luca Santuari. Analysis of Communication Models in Web Service Compositions. In *Proc. WWW'06*, 2006.

[MMF+06] M.Trainotti, M.Pistore, F.Barbon, P.Bertoli, A. Marconi, P.Traverso, and G. Zacco. ASTRO: Supporting Web Service Development by Automated Composition, Monitoring and Verification. In *Proceedings of Demos, ICAPS'06*, 2006.

[MMG+05] M.Trainotti, M.Pistore, G.Calabrese, G.Zacco, G.Lucchese, F.Barbon, P.Bertoli, and P.Traverso. ASTRO: Supporting Composition and Execution of Web Services. In *Proceedings of Demos, ICAPS'05*, 2005.

[MPT06] A. Marconi, M. Pistore, and P. Traverso. Specifying Data-Flow Requirements for the Automated Composition of Web Services. In *Proc. SEFM'06*, 2006.

[Ora] Oracle. Oracle BPEL Process Manager - http://www.oracle.com/products/ias/bpel/.

[Pan04] P.K. Pandya. Finding extremal models of discrete duration calculus formulae using symbolic search. In *Proc. AVOCS'2004*, 2004.

[PATB05] M. Pistore, A.Marconi, P. Traverso, and P. Bertoli. Automated Composition of Web Services by Planning at the Knowledge Level. In *Proc. IJCAI'05*, 2005.

[PBB+04] M. Pistore, F. Barbon, P. Bertoli, D. Shaparau, and P. Traverso. Planning and monitoring web service composition. In *Proc. 11th Int. Conf. on Artificial Intelligence: Methodology, Systems, Architectures*, 2004.

[PTB05] M. Pistore, P. Traverso, and P. Bertoli. Automated Composition of Web Services by Planning in Asynchronous Domains. In *Proc. ICAPS'05*, 2005.