Automated Composition of Web Services via Planning

ITC-irst
Università di Trento

Abstract. In this deliverable, we describe the way we tackle web service composition by recasting it as a specific, complex form of planning problem. We introduce a general framework for this purpose, and we analyze in detail a range of different planning techniques and encodings we have used to solve the problem of composing web services. Our practical achievements are reported and discussed at length.

Document Identifier	Deliverable D4.3
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

Web services are a fast emerging paradigm for distributed, network-enabled publication and provision of functionalities by independent providers. The principled reuse of published services is key to the success of this approach, allowing the design of complex functionalities on the shoulders of existing ones and offering the possibility to link different businesses into integrated functionalities needed by customers. At the same time, understanding existing services and programming new ones that, by interacting with them, implement complex functionalities is a very demanding, error-prone task if performed by a human. The success of the web service approach will strongly rely on the existence of support tools that address the following three problems:

1. automated web service composition, i.e. synthesizing new services that provide a given functionality, by interacting with existing ones;
2. automated web service verification, i.e. formally validating whether (the implementation of) existing services obey some desired property;
3. web service monitoring, i.e. checking that the actual behavior of services at run-time follow the desired pattern, and evaluating statistical data from the executions.

In this deliverable, we focus on the first problem, which, amounting to a problem of program synthesis, is theoretically (and often in practice) the harder of the three. In particular, we show that the problem can be recast as a problem of search, within a certain search space, for a "plan of action" - i.e. the executable service to be produced. That is, the problem can be seen as a problem of planning, and solved by automated planning techniques. More specifically, the features of web services, represented by standard languages such as WS-BPEL, impose some strict requirements on the kind of planning techniques that need to be used for this purpose: web services can be seen as nondeterministic, asynchronous, partially observable automates, and as such, we will need to plan against a partially observable, asynchronous, nondeterministic environment.

Within such a framework, we developed a variety of planning technologies, starting from the adaptation of a large amount of existing work on planning as model checking, and then departing into more specific and effective techniques.

In this deliverable, we summarize our work on the issue of web service composition, describing in detail the variety of techniques we developed, spelling out practical results we obtained, and discussing the boundaries of their applicability, and next steps to improve them.

Contents

1 Introduction 1

2 The problem 3

3 Web services as STSs 14

4 Framework and approaches 19

5 The ground level approach 21

5.1 Problem statement 21

5.1.1 Domain and Controller 21

5.1.2 Composition Requirements 25

5.2 Search by belief-level pre-compilation 33

5.2.1 Implementation and results 34

5.3 On-the-fly belief-level search 48

5.3.1 Implementation and results 52

6 The knowledge-level approach 56

6.1 Problem statement 57

6.2 Implementation and results 62

7 History of the Deliverable 65

7.1 1st Year 65

7.2 2nd Year 65

7.3 3rd year 66

7.4 4th Year 66

Chapter 1

Introduction

Planning is one of the most promising techniques for the automated composition of web services. Several recent works in planning have addressed different aspects of this problem, see, e.g., [BDG03, WPS^{+}03, Der98, SdF03, MS02, MF02]. In these works, automated composition is described as a planning problem: services that are available and published on the Web, the *component services*, are used to construct the planning domain, *composition requirements* can be formalized as planning goals, and planning algorithms can be used to generate *composed services*, i.e., plans that compose the component services.

A challenge for planning is the *automated composition of Web service at the process level*, i.e., the composition of component services that are described as "business processes" that publish a specification of the flow that is required to interact with them. Indeed, in most real cases, services cannot be considered simply as atomic components, which, given some inputs, return some outputs in a single request-response step. In several application domains, they need to be represented as stateful processes that require to follow an interaction protocol that may involve different, sequential, conditional, and iterative steps. For instance, we cannot in general interact with an "flight booking" service in an atomic step. The service may require a sequence of different operations including an authentication, a submission of a specific request for a flight, the possibility to submit iteratively different requests, acceptance (or refusal) of the offer, and finally, a payment procedure. In these cases,the process, the published interaction flow, is a key aspect for the (automated) composition of services.

The planning problem corresponding to the automated composition of services that are published as processes is far from trivial. First, component services cannot be simply represented as atomic actions of the planning domain. As a consequence, it is not obvious, like in the case of atomic component services, which is the planning domain that corresponds to the composition problem. Second, in realistic cases, component services publish nondeterministic and partially observable behaviors, since, in general, the outputs of a service cannot be predicted *a priori* and its internal status is not fully available to external services. For instance, whether a payment transaction will succeed cannot be known a priori of its execution, and whether there are still seats available on a flight cannot be known until a specific request is submitted to the service. Third, the plan composing component services cannot be simply a sequence of actions that call atomic components, but needs to interleave the (partial execution of) component services with the typical programming language constructs, such as conditionals, loops, etc. Finally, most often Web service interactions are intrinsically asynchronous: each process evolves independently and with unpredictable speed, synchronizing with the other processes only through asynchronous message exchanges. Message queues are used in practical implementations to guarantee that processes don't loose messages that they are not ready to receive.

As a consequence of all these characteristics of Web services, it is far from obvious how their automated composition can be adequately represented as a planning problem. Moreover, their nondeterministic, partially observable, and asynchronous behavior poses strong requirements and introduce novel problems for the planning techniques that can be used.

In this deliverable, we formalize the composition problem, provide a general framework for its solution by planning techniques, based on the idea that component services are seen as STSs and are the domain to be controlled, and we describe a variety of solutions to the problem, experimenting them and discussing their properties.

The remaining part of the document is organized as follows. In Chapter 2, we provide an informal description of the web service composition problem, by making use of an explanatory, yet realistic running example. In Chapter 3, we discuss the way in which web services, expressed in the standard WS-BPEL language, can be rewritten, and later manipulated, as state transition systems. In Chapter 4 we describe a general framework for solving the composition problem. This will be first instantiated in Chapter 5, where "ground-level techniques", based on concretely estimating the behavior of services for each possible combination of exchanged data, will be discussed at length. In Chapter 6, a different technique, based on abstracting away from concrete data and working at a "knowledge-level", is presented and tested. We wrap up with a recap on the history of this deliverable.

Chapter 2

The problem

The composition problem amounts to construct a new service (the *composed service*) that performs some desired functionality by interacting with available services (the *component services*). In our approach, we consider the case of component services that require complex interaction protocols to perform their task. This is indeed the case in many realistic composition scenarios, such as the one described in the following reference example.

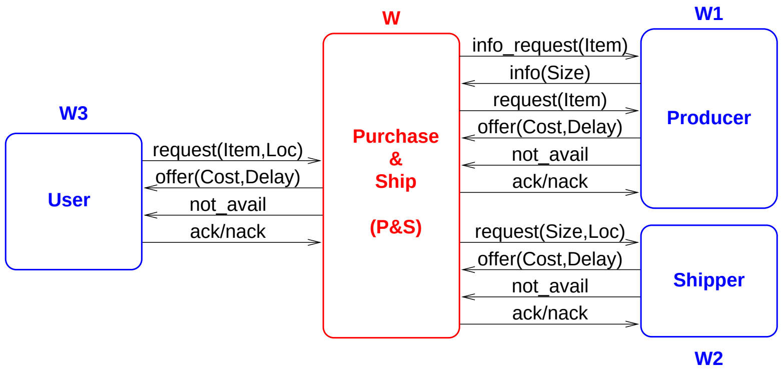


Figure 2.1: The P&S scenario.

Example 1 *Our 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. 2.1. The component 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.*

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 in 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.

In the rest of this section, we illustrate the WS-BPEL specifications of the component and composed services in the P&S scenario.

Example 2 *Fig.2.2 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.*[1] *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 2.3 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.*

[1]In our example, we only exploit one-way operations, i.e., operations that consist of a communication from the sender to the receiver. WSDL and WS-BPEL also support invoke-and-response operations, which define an input message and an output message (plus a failure message to manage non-nominal outcomes). Since invoke-and-response operations can always be realized as copies of unidirectional operations, in the

```
⟨definitions name = "Producer"⟩
  ⟨types⟩
    ⟨schema targetNamespace = "http://www.astroproject.org/Producer"⟩
      ⟨element name = "Item"⟩
        .......
      ⟨/element⟩
      ⟨element name = "Size"⟩
        .......
      ⟨/element⟩
      ⟨element name = "Location"⟩
        .......
      ⟨/element⟩
      ⟨element name = "Cost"⟩
        .......
      ⟨/element⟩
      ⟨element name = "Delay"⟩
        .......
      ⟨/element⟩
    ⟨/schema⟩
  ⟨/types⟩
  ⟨message name = "requestMsg"⟩
    ⟨part name = "art" type = "Item"/⟩
  ⟨/message⟩
  ⟨message name = "infoMsg"⟩
    ⟨part name = "size" type = "Size"/⟩
  ⟨/message⟩
  ⟨message name = "offerMsg"⟩
    ⟨part name = "cost" type = "Cost"/⟩
    ⟨part name = "delay" type = "Delay"/⟩
  ⟨/message⟩
  ⟨message name = "ackMsg"⟩
  ⟨message name = "nackMsg"⟩
  ⟨message name = "unavailMsg"⟩
  ⟨portType name = "P_PT"⟩
    ⟨operation name = "request"⟩
      ⟨input message = "requestMsg"/⟩
    ⟨/operation⟩
    ⟨operation name = "ack"⟩
      ⟨input message = "ackMsg"/⟩
    ⟨/operation⟩
    ⟨operation name = "nack"⟩
      ⟨input message = "nackMsg"/⟩
    ⟨/operation⟩
  ⟨/portType⟩
  ⟨portType name = "PC_PT"⟩
    ⟨operation name = "info"⟩
      ⟨input message = "infoMsg"/⟩
    ⟨/operation⟩
    ⟨operation name = "unavail"⟩
      ⟨input message = "unavailMsg"/⟩
    ⟨/operation⟩
    ⟨operation name = "offer"⟩
      ⟨input message = "offerMsg"/⟩
    ⟨/operation⟩
  ⟨/portType⟩
  ⟨plnk:partnerLinkType name = "P_PLT"⟩
    ⟨plnk:role name = "P_Service"⟩
      ⟨plnk:portType name = "P_PT"/⟩
    ⟨plnk:role⟩
    ⟨plnk:role name = "P_Customer"⟩
      ⟨plnk:portType name = "PC_PT"/⟩
    ⟨plnk:role⟩
  ⟨plnk:partnerLinkType⟩
⟨/definitions⟩
```

Figure 2.2: The WSDL for the Producer process.

```
⟨process name = ”Producer”⟩
  ⟨partnerLinks⟩
    ⟨partnerLink name = ”client” partnerLinkType = ”P_PLT” myRole = ”P_PT” partnerRole = ”PC_PT”/⟩
  ⟨/partnerLinks⟩
  ⟨variables⟩
    ⟨variable name = ”requestVar” messageType = ”requestMsg”/⟩
    ⟨variable name = ”unavailVar” messageType = ”unavailMsg”/⟩
    ⟨variable name = ”infoVar” messageType = ”infoMsg”/⟩
    ⟨variable name = ”offerVar” messageType = ”offerMsg”/⟩
    ⟨variable name = ”ackVar” messageType = ”ackMsg”/⟩
    ⟨variable name = ”nackVar” messageType = ”nackMsg”/⟩
    ⟨variable name = ”availVar” messageType = ”xsd : boolean”/⟩
  ⟨/variables⟩
  ⟨sequence name = ”main”⟩
    ⟨receive name = ”getRequest” operation = ”request” variable = ”requestVar” partnerLink = ”client” portType = ”P_PT”/⟩
    ⟨assign name = ”setAvail”⟩
      ⟨copy⟩⟨from opaque = ”yes”/⟩⟨to variable = ”availVar”/⟩⟨/copy⟩
    ⟨/assign⟩
    ⟨switch name = ”checkAvail”⟩
      ⟨case condition = ”bpws : getVariableData(′avail′) = xsd : False”⟩
        ⟨invoke name = ”sendUnavail” operation = ”unavail” inputVariable = ”unavailVar” partnerLink = ”client”
              portType = ”PC_PT”/⟩
      ⟨/case⟩
      ⟨otherwise
        ⟨assign name = ”prepareInfo”⟩
          ⟨copy⟩⟨from opaque = ”yes”/⟩⟨to variable = ”infoVar” part = ”size”/⟩⟨/copy⟩
        ⟨/assign⟩
        ⟨invoke name = ”sendInfo” operation = ”info” inputVariable = ”infoVar” partnerLink = ”client”
              portType = ”PC_PT”/⟩
        ⟨pick⟩
          ⟨onMessage name = ”getNack” operation = ”nack” variable = ”nackVar” partnerLink = ”client”
                  portType = ”P_PT”/⟩
          ⟨/onMessage⟩
          ⟨onMessage name = ”getAck” operation = ”ack” variable = ”ackVar” partnerLink = ”client” portType = ”P_PT”⟩
            ⟨assign name = ”prepareOffer”⟩
              ⟨copy⟩⟨from opaque = ”yes”/⟩⟨to variable = ”offerVar” part = ”delay”/⟩⟨/copy⟩
              ⟨copy⟩⟨from opaque = ”yes”/⟩⟨to variable = ”offerVar” part = ”cost”/⟩⟨/copy⟩
            ⟨/assign⟩
            ⟨invoke operation = ”sendOffer” inputVariable = ”offerVar” partnerLink = ”client” portType = ”PC_PT”⟩
            ⟨pick⟩
              ⟨onMessage name = ”getAccepted” operation = ”ack” variable = ”ackVar” partnerLink = ”client”
                      portType = ”P_PT”/⟩
              ⟨/onMessage⟩
              ⟨onMessage name = ”getRefused” operation = ”nack” variable = ”nackVar” partnerLink = ”client”
                      portType = ”P_PT”/⟩
              ⟨/onMessage⟩
            ⟨/pick⟩
          ⟨/onMessage⟩
        ⟨/pick⟩
      ⟨/otherwise⟩
    ⟨/switch⟩
  ⟨/sequence⟩
⟨/process⟩
```

Figure 2.3: The abstract WS-BPEL for the Producer process.

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 become 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.

Example 3 *Fig. 2.4 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

paper we just consider the latter ones, even if the approach works also for the former ones.

```
⟨process name = "User"⟩
  ⟨partnerLinks⟩
    ⟨partnerLink name = "client" partnerLinkType = "U_PLT" myRole = "U_PT" partnerRole = "UC_PT"/⟩
  ⟨/partnerLinks⟩
  ⟨variables⟩
    ⟨variable name = "requestVar" messageType = "requestMsg"/⟩
    ⟨variable name = "unavailVar" messageType = "unavailMsg"/⟩
    ⟨variable name = "offerVar" messageType = "offerMsg"/⟩
    ⟨variable name = "ackVar" messageType = "ackMsg"/⟩
    ⟨variable name = "nackVar" messageType = "nackMsg"/⟩
    ⟨variable name = "acceptsVar" messageType = "xsd : boolean"/⟩
  ⟨/variables⟩
  ⟨sequence name = "main"⟩
    ⟨assign name = "prepareRequest"⟩
      ⟨copy⟩⟨from opaque = "yes"/⟩⟨to variable = "requestVar" part = "art"/⟩⟨/copy⟩
      ⟨copy⟩⟨from opaque = "yes"/⟩⟨to variable = "requestVar" part = "loc"/⟩⟨/copy⟩
    ⟨/assign⟩
    ⟨invoke name = "sendRequest" operation = "request" inputVariable = "requestVar" partnerLink = "client"
         portType = "U_PT"⟩
    ⟨pick⟩
      ⟨onMessage name = "getUnavail" operation = "unavail" variable = "unavailVar" partnerLink = "client"
                portType = "UserC_PT"⟩
      ⟨/onMessage⟩
      ⟨onMessage name = "getOffer" operation = "offer" variable = "offerVar" partnerLink = "client"
                portType = "UserC_PT"⟩
        ⟨assign name = "prepareAcceptOrReject"⟩
          ⟨copy⟩⟨from opaque = "yes"/⟩⟨to variable = "acceptsVar" ⟩⟨/copy⟩
        ⟨/assign⟩
        ⟨switch name = "checkAcceptance"⟩
          ⟨case condition = "bpws : getVariableData('acceptsVar') = xsd : False" ⟩
            ⟨invoke name = "refused" operation = "nack" inputVariable = "nackVar" partnerLink = "client"
                  portType = "U_PT"⟩
          ⟨/case⟩
          ⟨otherwise
            ⟨invoke name = "accepted" operation = "ack" inputVariable = "ackVar" partnerLink = "client"
                  portType = "U_PT"⟩
          ⟨/otherwise⟩
        ⟨/switch⟩
      ⟨/onMessage⟩
    ⟨/pick⟩
  ⟨/sequence⟩
⟨/process⟩
```

Figure 2.4: The abstract WS-BPEL for the User process.

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).

The goal of the composition task is 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 that associates the cost paid by the User with the prices proposed by the Producer and Shipper, including a profit margin.

Example 4 *Figure 2.5 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. Indeed, 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.

```
⟨process name = "PandS_Executable"⟩
  ⟨partnerLinks⟩
    partnerLink myRole = "PandS_Service" name = "client" partnerLinkType = "tns : PandS_PLT"
          partnerRole = "PandS_Customer"⟩
    partnerLink myRole = "Shipper_Customer" name = "PandS_Shipper" partnerLinkType = "Shipper : S_PLT"
          partnerRole = "Shipper_Service"⟩
    partnerLink myRole = "Producer_Customer" name = "PandS_Producer" partnerLinkType = "Producer : P_PLT"
          partnerRole = "Producer_Service"⟩
  ⟨/partnerLinks⟩
  ⟨variables⟩
    ⟨variable name = "U_RequestVar" messageType = "User : requestMsg"/⟩
    ⟨variable name = "U_OfferVar" messageType = "User : offerMsg"/⟩
    ⟨variable name = "U_AckVar" messageType = "User : ackMsg"/⟩
    ⟨variable name = "U_NackVar" messageType = "User : nackMsg"/⟩
    ⟨variable name = "P_RequestVar" messageType = "Producer : requestMsg"/⟩
    ⟨variable name = "P_UnavailVar" messageType = "Producer : unavailMsg"/⟩
    ⟨variable name = "P_InfoVar" messageType = "Producer : infoMsg"/⟩
    ⟨variable name = "P_NackVar" messageType = "Producer : nackMsg"/⟩
    ⟨variable name = "P_AckVar" messageType = "Producer : ackMsg"/⟩
    ⟨variable name = "P_OfferVar" messageType = "Producer : offerMsg"/⟩
    ⟨variable name = "S_AckVar" messageType = "Shipper : ackMsg"/⟩
    ⟨variable name = "S_NackVar" messageType = "Shipper : nackMsg"/⟩
    ⟨variable name = "S_RequestVar" messageType = "Shipper : requestMsg"/⟩
    ⟨variable name = "S_UnavailVar" messageType = "Shipper : unavailMsg"/⟩
    ⟨variable name = "S_OfferVar" messageType = "Shipper : offerMsg"/⟩
  ⟨/variables⟩
  ⟨sequence⟩
    ⟨receive operation = "request" variable = "U_RequestVar" partnerLink = "client" portType = "PandS_PT"⟩
    ⟨assign ⟩
      ⟨copy⟩
        ⟨from variable = "U_RequestVar" part = "art"⟩
        ⟨to variable = "P_RequestVar" part = "art"/⟩
      ⟨/copy⟩
    ⟨/assign⟩
    ⟨invoke operation = "request" inputVariable = "P_RequestVar" partnerLink = "PandS_Producer"
         portType = "Producer : P_PT"⟩
    ⟨pick⟩
      ⟨onMessage operation = "unavail" variable = "P_UnavailVar" partnerLink = "PandS_Producer"
             portType = "Producer : PC_PT"⟩
        ⟨sequence ⟩
          ⟨invoke operation = "unavail" inputVariable = "P_UnavailVar" partnerLink = "client" portType = "PandSC_PT"⟩
        ⟨/sequence⟩
      ⟨/onMessage⟩
      ⟨onMessage operation = "getP_Info" operation = "info" variable = "P_InfoVar" partnerLink = "PandS_Producer"
             portType = "Producer : PC_PT"⟩
        ⟨sequence⟩
          ⟨assign ⟩
            ⟨copy⟩
              ⟨from variable = "P_InfoVar" part = "size"⟩
              ⟨to variable = "S_RequestVar" part = "size"/⟩
            ⟨/copy⟩
            ⟨copy⟩
              ⟨from variable = "U_RequestVar" part = "loc"⟩
              ⟨to variable = "S_RequestVar" part = "loc"/⟩
            ⟨/copy⟩
          ⟨/assign⟩
          ⟨invoke operation = "request" inputVariable = "S_RequestVar" partnerLink = "PandS_Shipper"
               portType = "Shipper : S_PT"⟩
          ⟨pick⟩
            ⟨onMessage operation = "unavail" variable = "S_UnavailVar" partnerLink = "PandS_Shipper"
                   portType = "Shipper : ShipperC_PT" ⟩
              ⟨sequence⟩
                ⟨flow⟩
                  ⟨sequence⟩
                    ⟨invoke operation = "unavail" inputVariable = "U_Unavail" partnerLink = "client"
                         portType = "PandSC_PT"⟩
                  ⟨/sequence⟩
                  ⟨sequence⟩
                    ⟨invoke operation = "nack" inputVariable = "P_NackVar" partnerLink = "PandS_Producer"
                         portType = "Producer : P_PT"⟩
                  ⟨/sequence⟩
                ⟨/flow⟩
              ⟨/sequence⟩
            ⟨/onMessage⟩
            ⟨onMessage operation = "offer" variable = "S_OfferVar" partnerLink = "PandS_Shipper"
                   portType = "Shipper : ShipperC_PT"⟩
              ⟨sequence⟩
                ⟨invoke operation = "ack" inputVariable = "P_AckVar" partnerLink = "PandS_Producer"
                     portType = "Producer : P_PT"⟩
```

Figure 2.5: The executable WS-BPEL for the P&S process (pt.1).

```
..........
          ⟨receive operation = ”offer” variable = ”P_OfferVar” partnerLink = ”PandS_Producer”
                portType = ”Producer : PC_PT”⟩
          ⟨assign⟩
            ⟨copy⟩
              ⟨from expression = ”bpws : getVariableData(′P_OfferVar′,′ cost′)+
                        bpws : getVariableData(′S_OfferVar′,′ cost′)”⟩
              ⟨to variable = ”U_OfferVar” part = ”cost”/⟩
            ⟨/copy⟩
            ⟨copy⟩
              ⟨from expression = ”bpws : getVariableData(′P_OfferVar′,′ delay′)+
                         bpws : getVariableData(′S_OfferVar′,′ delay′)”⟩
              ⟨to variable = ”U_OfferVar” part = ”delay”/⟩
            ⟨/copy⟩
          ⟨/assign⟩
          ⟨invoke operation = ”offer” inputVariable = ”U_OfferVar” partnerLink = ”client” portType = ”PandSC_PT”⟩
          ⟨pick⟩
            ⟨onMessage operation = ”ack” variable = ”U_AckVar” partnerLink = ”client” portType = ”PandS_PT”⟩
              ⟨sequence⟩
                ⟨flow⟩
                  ⟨sequence⟩
                    ⟨invoke operation = ”ack” inputVariable = ”S_AckVar” partnerLink = ”PandS_Shipper”
                          portType = ”Shipper : S_PT”⟩
                  ⟨/sequence⟩
                  ⟨sequence⟩
                    ⟨invoke operation = ”ack” inputVariable = ”P_AckVar” partnerLink = ”PandS_Producer”
                          portType = ”Producer : P_PT”⟩
                  ⟨/sequence⟩
                ⟨/flow⟩
              ⟨/sequence⟩
            ⟨/onMessage⟩
            ⟨onMessage operation = ”nack” variable = ”U_NackVar”partnerLink = ”client” portType = ”PandS_PT”⟩
              ⟨sequence
                ⟨flow⟩
                  ⟨sequence⟩
                    ⟨invoke operation = ”nack” inputVariable = ”S_NackVar” partnerLink = ”PandS_Shipper”
                          portType = ”Shipper : S_PT”⟩
                  ⟨/sequence⟩
                  ⟨sequence⟩
                    ⟨invoke operation = ”nack” inputVariable = ”P_NackVar” partnerLink = ”PandS_Producer”
                          portType = ”Producer : P_PT”⟩
                  ⟨/sequence⟩
                ⟨/flow⟩
              ⟨/sequence⟩
            ⟨/onMessage⟩
          ⟨/pick⟩
         ⟨/sequence⟩
        ⟨/onMessage⟩
       ⟨/pick⟩
      ⟨/sequence⟩
     ⟨/onMessage⟩
    ⟨/pick⟩
   ⟨/sequence⟩
⟨/process⟩
```

Figure 2.6: The executable WS-BPEL for the P&S process (pt.2).

Construct	Meaning
⟨types⟩	The types element encloses data type definitions that are relevant for the exchanged messages. The XSD type system can be used to define the types.
message	Messages consist of one or more logical parts. Each part is associated with a type from some type system. WSDL defines two different message-typing attributes: element,which refers to an XSD element using a QName, or type, which refers to an XSD simpleType or complexType using a QName.
portType	A PortType is a named set of operations.
operation	An operation is a named entity that specifies, through the output and input elements, the abstract message format for the solicited request and response, respectively. WSDL defines four types of operation: • One-way, the operation can receive a message but will not return a response • Request-response, the operation can receive a request and will return a response • Solicit-response, the operation can send a request and will wait for a response • Notification, the operation can send a message but will not wait for a response
ports and bindings	While PortTypes define abstract functionality by using abstract messages, Ports provide actual access information, including communication endpoints and (by using extension elements) other deployment-related information such as public keys for encryption. Bindings provide the glue between the two.
service	A service groups a set of related ports together

Figure 2.7: Synopsis table for WSDL.

Construct	Meaning
PartnerLink	In BPEL a Web Service that is involved in the process is always modeled as a partnerLink. Every partnerLink is characterized by a partner-LinkType which is defined in the WSDL definition. The role of the process in the communication is specified by the attribute myRole and the role of the partner is specified by the attribute partnerRole.
Variable	Variables offer the possibility to store data. The messages that get stored are most of the time either coming from partners or going to partners. Variables also offer the possibility to store data that is only state based and never send to partners. There are three types of variables: WSDL message type, XML Schema simple type and XML Schema element.
Assign	It allows to copy the data from one source (a variable, an XPath expression, ..) to a variable. In abstract processes assigns can be opaque. Finally, this activity can also be used to copy endpoint references to and from partner links (dynamic binding).
CorrelationSet	Each correlation set in BPEL4WS is a named group of properties that, taken together, serve to define a way of identifying an application-level conversation within a business protocol instance. After a correlation set is initiated, the values of the properties for a correlation set must be identical for all the messages in all the operations that carry the correlation set and occur within the corresponding scope until its completion.
Receive	A receive activity specifies the partner link it expects to receive from, and the port type and operation that it expects the partner to invoke, and a variable that is to be used to store the message data received. In addition, receive activities play a role in the life cycle of a business process. The only way to instantiate a business process in BPEL4WS is to annotate a receive activity with the createInstance attribute set to ”yes”.
Reply	A reply activity is used to send a response to a request previously accepted through a receive activity. Such responses are only meaningful for synchronous interactions. An asynchronous response is always sent by invoking the corresponding one-way operation on the partner link.
Invoke	Is used to invoke a service operation. Such an operation can be a synchronous request/response or an asynchronous one-way operation. BPEL4WS uses the same basic syntax for both with some additional options for the synchronous case. An asynchronous invocation requires only the input variable of the operation because it does not expect a response as part of the operation. A synchronous invocation requires both an input variable and an output variable
Sequence	A sequence activity contains one or more activities that are performed sequentially, that is, in the order in which they are listed within the ¡sequence¿ element.
Switch	The activity consists of an ordered list of one or more conditional branches defined by case elements, followed optionally by an otherwise branch. The case branches of the switch are considered in the order in which they appear. The first branch whose condition holds true is taken and provides the activity performed for the switch. If no branch with a condition is taken, then the otherwise branch is taken.
While	The while activity supports repeated performance of a specified iterative activity. The iterative activity is performed until the given Boolean while condition no longer holds true.
Pick	The pick activity awaits the occurrence of one of a set of events and then performs the activity associated with the event that occurred. Note that after the pick activity has accepted an event for handling, the other events are no longer accepted by that pick. The possible events are the arrival of some message in the form of the invocation of an operation, or an ”alarm” based on a timer.
Flow	The most fundamental semantic effect of grouping a set of activities in a flow is to enable concurrency. A flow completes when all of the activities in the flow have completed. It further enables expression of synchronization dependencies between activities that are nested directly or indirectly within it. The link construct is used to express these synchronization dependencies.

Figure 2.8: Synopsis table for WS-BPEL.

Chapter 3

Web services as STSs

We encode WS-BPEL processes as *state transition systems* which describe dynamic systems that can be in one of their possible *states* (some of which are marked as *initial states*) and can evolve to new states as a result of performing some *actions*. Actions are distinguished in *input actions*, which represent the reception of messages, *output actions*, which represent messages sent to external services, and a special action τ, called *internal action*. The action τ is used to represent internal evolutions that are not visible to external services, i.e., the fact that the state of the system can evolve without producing any output, and independently from the reception of inputs. A *transition relation* describes how the state can evolve on the basis of inputs, outputs, or of the internal action τ. Finally, a *labeling function* associates to each state the set of properties in $\mathcal{P}rop$ that hold in the state. These properties will be used to define the composition requirements.

Defi nition 5 (State transition system (STS)) *A* state transition system Σ *is a tuple* $\langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{R}, \mathcal{L} \rangle$ *where:*

- $\mathcal{S}$ *is the finite set of states;*
- $\mathcal{S}^0 \subseteq \mathcal{S}$ *is the set of initial states;*
- $\mathcal{I}$ *is the finite set of input actions;*
- $\mathcal{O}$ *is the finite set of output actions;*
- $\mathcal{R} \subseteq \mathcal{S} \times (\mathcal{I} \cup \mathcal{O} \cup \{\tau\}) \times \mathcal{S}$ *is the transition relation;*
- $\mathcal{L} : \mathcal{S} \rightarrow 2^{\mathcal{P}rop}$ *is the labeling function.*

We require that $\mathcal{I} \cap \mathcal{O} = \emptyset$ *and that* $\tau \notin \mathcal{I} \cup \mathcal{O}$.

A state $s \in \mathcal{S}$ is *divergent* if there is an infinite sequence of τ transition starting from s, i.e, if there is an infinite sequence of states $s = s_1, s_2, s_3, \ldots$ such that $(s_i, \tau, s_{i+1}) \in \mathcal{R}$ for all $i \geq 1$. Divergent states and infinite τ-sequences correspond to undesirable behaviors of the service, which never interacts with the environment, and cannot be controlled in any way. For this reason, we will implicitly assume in the rest of the paper that the STS will not have divergent states.

Some standard definitions on STS are now in order. We say that an action $a \in (\mathcal{I} \cup \mathcal{O} \cup \{\tau\})$ is *applicable* on a state $s \in \mathcal{S}$, denoted with Appl(a, s), if there exists a state $s' \in \mathcal{S}$ s.t. $(s, a, s') \in \mathcal{R}$. A state of an STS is *final* if no action $a \in (\mathcal{I} \cup \mathcal{O} \cup \{\tau\})$ is applicable in s, i.e., if there is no transition leaving s. The execution of an STS is represented by its set of possible *runs*, i.e., of sequences of states $s_0, s_1, \ldots$ such that $s_0 \in \mathcal{S}^0$ and $(s_i, a, s_{i+1}) \in \mathcal{R}$ for some $a \in \mathcal{I} \cup \mathcal{O} \cup \{\tau\}$. In general, such executions may be finite or infinite. A run is said to be *completed* if it is finite, and if its last state is final. A state $s \in \mathcal{S}$ will be said *reachable* if and only if there exists a state $s_0 \in \mathcal{S}^0$ and a sequence $s_0, s_1, \ldots, s_n$ such that $s_n = s$, and for all $i = 0, \ldots, n-1$ there is some $a \in (\mathcal{I} \cup \mathcal{O} \cup \{\tau\})$ such that $\langle s_i, a, s_i \rangle \in \mathcal{R}$. We will denote with $Reachable(\Sigma) \subseteq \mathcal{S}$ the set of reachable states of Σ.

We have implemented a translation that associates a STS to each component service, starting from its abstract WS-BPEL specification. For the moment, the translation is restricted to a significant subset of the WS-BPEL languages. More precisely, we support all WS-BPEL *basic* and *structured activities*, like invoke, receive, sequence, switch, while, pick, and flow (without links). Moreover we support restricted forms of assignments (specifically, we restrict the expressions that can appear in the `from` part of the copy statements) and of correlations[1]. We omit the formal definition of the translation, which requires some technicality but which is conceptually simple. We provide instead the underlying intuitions by illustrating the translation through the example of the Producer abstract WS-BPEL introduced in the previous section.

Example 6 *Figure 3.1 shows the STS for the abstract* WS-BPEL *process of the Producer, represented in the internal language that is used by the* WS-BPEL *to STS translator. The specification starts with a list of* TYPE*s (*`Article`*,* `Size`*,* `Cost`*,* `Delay`*) exploited in the STS, which are derived from the WSDL specification of the Producer. The* STATE*s are defined by a set of variables. A special variable* `pc` *that implements a "program counter", that holds the current execution step of the service, e.g.,* `pc` *has value* `getRequest` *when the process is waiting to receive a request for an item, and value* `checkAvail` *when it is ready to check whether the item is available. These values of the program counter correspond to the* `<receive>` *and* `<switch>` *statements at lines 15 and 20 of the* WS-BPEL *source, respectively. Whenever the* name *of a* WS-BPEL *activity is specified, our translation uses that name as the value attained by the program counter at the start of that activity; for instance, line 47 is associated to the activity named* `pickProducer1` *in the code. Otherwise, a unique PC value is created by the translator*

[1]The considered subset does not deal at the moment with WS-BPEL constructs like scopes, fault, event and compensation handlers. However, we found the considered subset expressive enough for describing services as business processes in several applications in different domains.

```
PROCESSProducer;
TYPE
Article;
  Size;
  Cost;
  Delay;
STATE
pc : {main, getRequest, setAvail, checkAvail, sendUnavail, seq1, prepareInfo, sendInfo,
  pick1, getNack, getAck, seq2, prepareOffer, pick2, getAccepted,
    getRefused, DONE_getNack, DONE_getAccepted, DONE_getRefused};
  infoVar_size : Size;
  offerVar_delay : Delay;
  offerVar_cost : Cost;
  requestVar_article : Article;
  availVar : Boolean;
INIT
  pc := main;
  infoVar_size := UNDEF;
  offerVar_delay := UNDEF;
  offerVar_cost := UNDEF;
  requestVar_article := UNDEF;
  availVar := UNDEF;
INPUT
request(Article);
  nack();
  ack();
OUTPUT
unavail();
  offer(Cost, Delay);
  info(Size);
TRANS
  pc = main − [TAU]− > pc := getRequest;
  pc = getRequest − [INPUT request(requestVar_article)]− > pc := setAvail;
  pc = setAvail − [TAU]− > pc = checkAvail, availVar ! = UNDEF;
  pc = checkAvail , availVar = ⊥ − [TAU]− > pc := sendUnavail;
  pc = sendUnavail − [OUTPUT unavail()]− > pc := DONE_sendUnavail;
  pc = checkAvail , availVar = ⊤ − [TAU]− > pc := seq1;
  pc = seq1 − [TAU]− > pc := prepareInfo;
  pc = prepareInfo − [TAU]− > pc := sendInfo, infoVar_size ! = UNDEF;
  pc = sendInfo − [OUTPUT info(infoVar_size)]− > pc := pick1;
  pc = pick1 − [INPUT nack()]− > pc := DONE_getNack;
  pc = pick1 − [INPUT ack()]− > pc := seq2;
  pc = seq2 − [TAU]− > pc := prepareOffer;
  pc = prepareOffer − [TAU]− > pc := sendOffer, offerVar_cost ! = UNDEF, offerVar_delay ! = UNDEF;
  pc = sendOffer − [OUTPUT offer(offerVar_cost, offerVar_delay)]− > pc := pick2;
  pc = pick2 − [INPUT nack()]− > pc := DONE_getAccepted;
  pc = pick2 − [INPUT ack()]− > pc := DONE_getRefused;
```

Figure 3.1: The STS for the Producer process.

(e.g. in the case of the `<pick>` activity at line 31. If an activity takes place immediately prior to termination, its name (if defined) is also used to describe the final value of the PC: if the name has value `nameAct`, it corresponds to `pc = DONE_nameAct` (see for instance line 51 and line 52).

The other variables (e.g., `offerVar_delay`, `offerVar_cost`*) correspond to the parts of the* WS-BPEL *message variables. In the* `INIT`*ial states all the variables are undefined but* `pc` *that is set to the initial* WS-BPEL *activity* `main`.

The `INPUT` *declarations in the file model all the incoming requests to the process and the information they bring (i.e.,* `request` *is used for the receiving of the shipping request, and has a single parameter of type* `Article`*). Similarly, the* `OUTPUT` *declarations model the outgoing messages (e.g.,* `unavail` *is used when the shipping is not supported by the process, and* `offer` *is used to bid the transportation of an item at a particular price).*

The evolution of the process is modeled through a set of possible `TRANS`*itions. Each transition defines its applicability conditions on the source state, its firing action, and the destination state. For instance, "*`pc = main -[TAU]-> pc = getRequest`*" states that an action τ can be executed in state* `main` *leading to state* `getRequest`*.. Transitions such as the one above are directly associated to* WS-BPEL *activities, e.g. in this case the starting* `main` *activity; other transitions are inserted by the translation mechanism, e.g. to link two activities that must be performed in sequence (see for instance "*`pc = _seq2 -[TAU]-> pc = _prepareOffer`*").* WS-BPEL *assignments are mapped into transitions whose effects also affect variables different from the program counter; for instance, the assignment at line 44 corresponds to the one in 27 in the* WS-BPEL *code.*

Figure 3.1 gives a schematic representation of the STS. In order to obtain a concrete STS according to Definition 5, types `Article`, `Size`, `Cost`, *and* `Delay` *need to be instantiated with finite ranges (we will come back to this point in the experimental evaluation). States $\mathcal{S}$ then correspond to assignments to the variables, inputs $\mathcal{I}$ and outputs $\mathcal{O}$ to the instantiations of the messages defined in the* `INPUT` *and* `OUTPUT` *sections.*

Each `TRANS` *clause of Figure 3.1 corresponds to the different elements in the transition relation $\mathcal{R}$: e.g. "*`pc = checkAvail -[TAU]-> pc = _sendUnavail`*" generates different elements of $\mathcal{R}$, depending on the values of variables* `requestVar_article`,`offerVar_cost` *and so on. The properties associated to the STS are expressions of the form* `<variable> = <value>`, *where* `<variable>` *is either a message part variable or* `pc`. *The labeling function is the obvious one.*

A similar mapping from WS-BPEL process to STS holds also for the Shipper and the User processes.

We remark that the STSs feature nondeterminism of two different natures, often referred to as '"external" and "internal" nondeterminism. The first, "external" nondeterminism, is due to the fact that several input actions are possible on a given state. This models the possibility for an STS to handle different requests, responding adequately to several possible behaviors of the partners that interact with it. This kind of nondeterminism occurs for instance in state `pc = pickProducer1`, where messages `ack` and `nack` can be received. The second, "internal" nondeterminism, refers to the fact that an STS may evolve in different ways according to internal decisions that are not explicit in the STS. For instance, the fact that a single state may originate different τ models internal choices whose conditions are not described in the STS. This is the case, for instance, of the decision about the availability of the product taken in state `pc = checkAvail`. Similar considerations hold for states where different output transitions can be performed.

Example 6 shows that both forms of nondeterminism are necessary to model the component services. In the case of the composed web services, we can restrict, without loss of generality, to STS than feature only "external" nondeterminism, which is needed to respond correctly to various messages coming from the component services. "Internal" nondeterminism is instead not needed, since there is no reason to model "abstract" choices in an executable process: executable WS-BPEL cannot feature internal nondeterminism.

According with the following definition, we will call an STS *deterministic* if it does not contain "internal" nondeterminism (while it may contain "external" nondeterminism). The STS2BPEL translator that is in charge of converting into an executable WS-BPEL the STS obtained by the synthesis procedure, which represents the composed service, handles such form of STSs.

Defi nition 7 (Deterministic State Transition System) *An STS $\Sigma = \langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{R}, \mathcal{L} \rangle$ is deterministic if and only if:*

1. $|\mathcal{S}^0| = 1$*;*
2. *for each state $s \in \mathcal{S}$, if $(s, \tau, s') \in \mathcal{R}$, then no other transition from s belongs to $\mathcal{R}$;*
3. *for each state $s \in \mathcal{S}$, if $(s, a, s') \in \mathcal{R}$, with $a \in \mathcal{O}$, then no other transition from s belongs to $\mathcal{R}$;*

Basically, a deterministic STS has a single initial state and admits, for each state, either a unique τ-transition, or a unique output transition, or a nonempty set of input transitions.

Chapter 4

Framework and approaches

Given our ability to convert web services (originally expressed in WS-BPEL) as STSs, all of our approaches will revolve around the general framework represented in Fig. 4.1. The idea is that component web services are first converted as STSs, and then treated as a domain that needs to be controlled by sending messages and handling answers. Notice that, as a part of the domain, also the description of the intended protocol that the user needs to establish with the composed web service enters into play - and that, similarly, it consists of an STS translation of a WS-BPEL interface.

Once the domain is available, the expected properties that need to be obeyed by the composed service are seen as the goal property that needs to be satisfied by the controlled domain; as such, we can use planning techniques to look for a way to control the domain with that goal in mind.

The resulting 'plan of action' is in fact a (deterministic) automaton that, on the basis of responses from the domain, i.e. from the component services and from the user, act by processing the received data and sending messages to the components (and to the user). That is, such plan is our component web service, and we can convert back such an STS into an (executable) web service that we will be able to deploy and run.

Within such a general framework, a variety of techniques can be used to synthesize the desired web service. There are two main points of intervention that originate such a variety of techniques.

First, the way in which components services are encoded as STS, and vice versa. A first family of techniques, called "ground-level" techniques, rely on a translation where every possible instantiation of data exchanged by the service are considered and mapped into the STS. A different approach is envisaged by so-called "knowledge-level" techniques, where the STS maps a more abstract view of the behavior of the service, encoding only the state of knowledge over the values of variables, but not their concrete value. Ground-level and knowledge level techniques differ in a quite radical way, since the underlying search space is quite different.

The second choice that originates a variety of techniques stands in the way the search space is represented and searched upon. Here, the main difference stands between “off-line” and “on-the-fly” techniques. Off-line techniques construct and represent the whole search space beforehand, before searching it for a solution. The evident drawback is that the whole search space being often very large, its construction can be rather expensive: on the other side, once the search space is available, it is possible to make use of effective representation techniques to visit it, considering large portions of it at each progression step of the visit. Vice versa, on-the-fly techniques build the search space as they visit it; this can be convenient if the search is smart enough (and the problem simple enough) that just a small portion of the search space needs to be visited.

In the following chapters, we will first describe and test ground-level techniques, and then step on the knowledge-level ones.

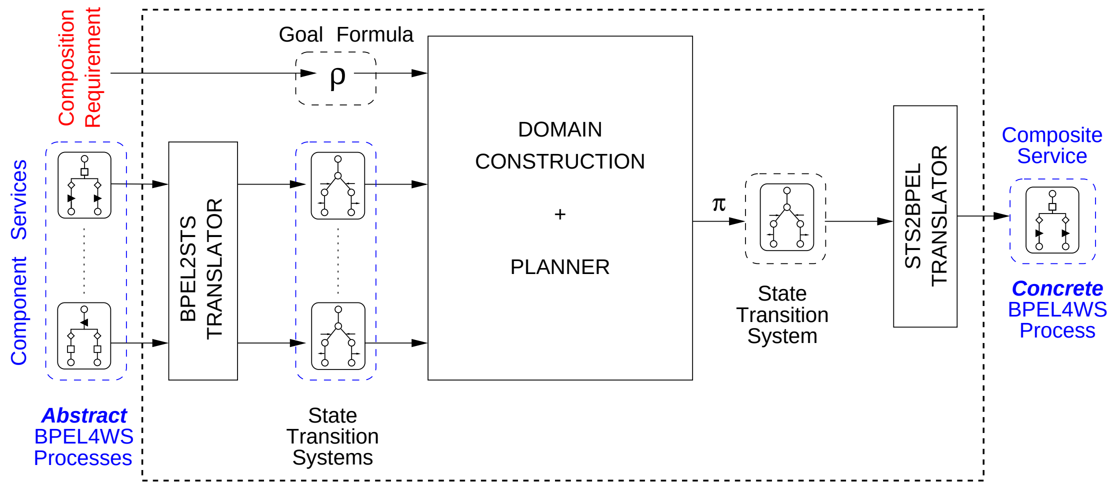


Figure 4.1: The general framework.

Chapter 5

The ground level approach

5.1 Problem statement

To formally state the composition problem, we start (Sec. 5.1.1) by modeling the environment to be controlled as a *parallel product* $\Sigma_{\|}$ of the independently evolving components $\Sigma_1, \ldots, \Sigma_n$. We then define what it means to control $\Sigma_{\|}$ by an STS Σ_c; the behaviors of the controlled system are again represented as an STS $\Sigma_c \triangleright \Sigma_{\|}$.

In Sec. 5.1.2, we formalize composition requirements, and the conditions to satisfy them. We then state the composition problem as that of finding a controller Σ_c such that $\Sigma_c \triangleright \Sigma_{\|}$ obeys a given composition requirement.

Finally, in order to be able to tackle service composition by means of effective search techniques, we rephrase the problem by considering a *belief-level* counterpart of $\Sigma_{\|}$, where observationally equivalent states are grouped together into so-called *beliefs*. We show that controllers that solve the problem can be uniquely associated to portions of the belief-level system, allowing us to exploit the belief-level system as our search space.

5.1.1 Domain and Controller

The automated composition problem has two inputs (see Figure 4.1): the formal composition requirement ρ and the set of component services $\Sigma_{W_1}, \ldots, \Sigma_{W_n}$. The components $\Sigma_{W_1}, \ldots, \Sigma_{W_n}$ evolve independently, and in fact represent, under a planning perspective, the domain to be controlled. Such domain $\Sigma_{\|}$ is obtained as the first step of the composition, by combining $\Sigma_{W_1}, \ldots, \Sigma_{W_n}$ by means of a *parallel product* operation.

Definition 8 (Parallel product) *Let* $\Sigma_1 = \langle \mathcal{S}_1, \mathcal{S}_1^0, \mathcal{I}_1, \mathcal{O}_1, \mathcal{R}_1, \mathcal{L}_1 \rangle$ *and* $\Sigma_2 = \langle \mathcal{S}_2, \mathcal{S}_2^0, \mathcal{I}_2, \mathcal{O}_2, \mathcal{R}_2, \mathcal{L}_2 \rangle$ *be two STSs with* $(\mathcal{I}_1 \cup \mathcal{O}_1) \cap (\mathcal{I}_2 \cup \mathcal{O}_2) = \emptyset$*. The parallel product* $\Sigma_1 \| \Sigma_2$ *of* Σ_1 *and* Σ_2 *is defined as:*

$$\Sigma_1 \| \Sigma_2 = \langle \mathcal{S}_1 \times \mathcal{S}_2, \mathcal{S}_1^0 \times \mathcal{S}_2^0, \mathcal{I}_1 \cup \mathcal{I}_2, \mathcal{O}_1 \cup \mathcal{O}_2, \mathcal{R}_1 \| \mathcal{R}_2, \mathcal{L}_1 \| \mathcal{L}_2 \rangle$$

where:

- $\langle(s_1, s_2), a, (s'_1, s_2)\rangle \in (\mathcal{R}_1 \| \mathcal{R}_2)$ *if* $\langle s_1, a, s'_1\rangle \in \mathcal{R}_1$*;*
- $\langle(s_1, s_2), a, (s_1, s'_2)\rangle \in (\mathcal{R}_1 \| \mathcal{R}_2)$ *if* $\langle s_2, a, s'_2\rangle \in \mathcal{R}_2$*;*

and $(\mathcal{L}_1 \| \mathcal{L}_2)(s_1, s_2) = \mathcal{L}_1(s_1) \cup \mathcal{L}_2(s_2)$*.*

The system representing (the parallel evolutions of) the component services $W_1, \ldots, W_n$ of Figure 4.1 is formally defined as $\Sigma_{\|} = \Sigma_{W_1} \| \ldots \| \Sigma_{W_n}$.

We remark that our definition of parallel product requires that the inputs/outputs of Σ_1 and those of Σ_2 are disjoint. This means that the component services may not communicate directly with each other. While an extension to the case where such component services may directly communicate is possible (and rather simple), we will not consider it, since in our scenario, we intend to compose independent existing services, which we assume not to be aware of each other.

The automated composition problem consists in generating a STS Σ_c that controls $\Sigma_{\|}$ so that its executions satisfy the requirement ρ (according to a formal notion of requirement satisfaction). Without loss of generality, we can restrict to consider internally deterministic controllers, see Def. 7. Moreover, we require that the inputs of a controller coincide with the outputs of its corresponding environment, and vice versa: situations where some I/O only appears on either the controller or the domain are useless, and may cause deadlock situations.

Defi nition 9 (controller) *A controller for* $\Sigma_{\|} = \langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{R}, \mathcal{L}\rangle$ *is a deterministic STS* $\Sigma_c = \langle \mathcal{S}_c, \mathcal{S}_c^0, \mathcal{O}, \mathcal{I}, \mathcal{R}_c, \mathcal{L}_\emptyset\rangle$*, where* $\forall s \in \mathcal{S}_c : \mathcal{L}(s) = \emptyset$*.*

The behaviors of a STS Σ when controlled by a controller Σ_c can be represented as an STS, which we call the *controlled system*:

Defi nition 10 (controlled system) *Let* $\Sigma = \langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{R}, \mathcal{L}\rangle$ *be a state transition system, and let* $\Sigma_c = \langle \mathcal{S}_c, \mathcal{S}_c^0, \mathcal{O}, \mathcal{I}, \mathcal{R}_c, \mathcal{L}_\emptyset\rangle$ *be a controller for* Σ*. The STS* $\Sigma_c \triangleright \Sigma$*, describing the behaviors of system* Σ *when controlled by* Σ_c*, is defined as:*

$$\Sigma_c \triangleright \Sigma = \langle \mathcal{S}_c \times \mathcal{S}, \mathcal{S}_c^0 \times \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{R}_c \triangleright \mathcal{R}, \mathcal{L}\rangle$$

where:

- $\langle(s_c, s), \tau, (s'_c, s')\rangle \in (\mathcal{R}_c \triangleright \mathcal{R})$ *if* $\langle s_c, \tau, s'_c\rangle \in \mathcal{R}_c$*;*
- $\langle(s_c, s), \tau, (s_c, s')\rangle \in (\mathcal{R}_c \triangleright \mathcal{R})$ *if* $\langle s, \tau, s'\rangle \in \mathcal{R}$*;*
- $\langle(s_c, s), a, (s'_c, s')\rangle \in (\mathcal{R}_c \triangleright \mathcal{R})$*, with* $a \neq \tau$*, if* $\langle s_c, a, s'_c\rangle \in \mathcal{R}_c$ *and* $\langle s, a, s'\rangle \in \mathcal{R}$*.*

Notice also that, although the systems are connected so that the output of one is associated to the input of the other, the resulting transitions in $\mathcal{R}_c \triangleright \mathcal{R}$ are labeled by input/output actions. This allows us to distinguish the transitions that correspond to τ actions of Σ_c or Σ from those deriving from communications between Σ_c and Σ.

In general, an STS Σ_c may not be adequate to control a system Σ. In particular, deadlocks may occur in case Σ_c performs an output at a point in time where Σ is not able to accept it, or vice versa. Of course, we want to only consider controllers that prevent these situations - guaranteeing that, whenever Σ_c performs an output transition, then Σ is able to accept it, and vice versa. Moreover, for sake of generality, we need to do so in a way which is independent from low-level engine-dependent implementation details, and in particular from the way I/O queuing mechanisms are realized. This requires a careful, conservative approach: we need to rule out any cases where the presence of a queuing mechanism is essential to the avoidance of deadlock situations.

We define a deadlock-freedom condition that relies on the assumption that messages are associated to a buffer, but does not rely on the existence of any message queuing/buffering mechanism. This corresponds to the minimal requirement that allows for asynchronous execution of WS-BPEL services, and as such is guaranteed to be supported by any web-service execution engine. In particular, we assume that s can accept a message a if there is some successor $s' \in \mathcal{S}$ of s, reachable from s through a chain of τ transitions, such that s' can perform an input transition labeled with a. Vice versa, if state s has no such successor s', and message a is sent to Σ, then a deadlock situation is reached.[1]

In the following definition, and in the rest of the paper, we denote by τ-closure(s) the set of the states reachable from s through a sequence of τ transitions, and by τ-closure(S) with $S \subseteq \mathcal{S}$ the union of the τ-closures on all the states of the set S:

Definition 11 (τ-closure) *Let $\Sigma = \langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{R}, \mathcal{L} \rangle$ be a STS, and $s \in \mathcal{S}$. Then τ-closure$(s) = \{t : \exists s^0, s^1, \ldots, s^n : s = s^0, t = s^n, \langle s^i, \tau, s^{i+1} \rangle \in \mathcal{R}\}$. Moreover if $S \subseteq \mathcal{S}$, τ-closure$(S) = \bigcup_{s \in S} \tau$-closure$(s)$.*

Definition 12 (deadlock-free controller) *Let $\Sigma = \langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{R}, \mathcal{L} \rangle$ be a STS and $\Sigma_c = \langle \mathcal{S}_c, \mathcal{S}_c^0, \mathcal{O}, \mathcal{I}, \mathcal{R}_c, \mathcal{L}_\emptyset \rangle$ be a controller for Σ. Σ_c is* deadlock free *for Σ if all states $(s_c, s) \in Reachable(\Sigma_c \triangleright \Sigma)$ satisfy the following conditions:*

1. *if $\langle s, a, s' \rangle \in \mathcal{R}$ with $a \in \mathcal{O}$ then there is some $s'_c \in \tau$-closure(s_c) such that $\langle s'_c, a, s''_c \rangle \in \mathcal{R}$ for some $s''_c \in \mathcal{S}_c$; and*
2. *if $\langle s_c, a, s'_c \rangle \in \mathcal{R}_c$ with $a \in \mathcal{I}$ then there is some $s' \in \tau$-closure(s) such that $\langle s', a, s'' \rangle \in \mathcal{R}_c$ for some $s'' \in \mathcal{S}$.*

[1]We remark that, if there is such a successor s' of s, a deadlock can still occur. This can happen if a different chain of τ transitions is executed from s that leads to a state s'' for which a cannot be executed anymore. In this case, the deadlock is recognized in s''.

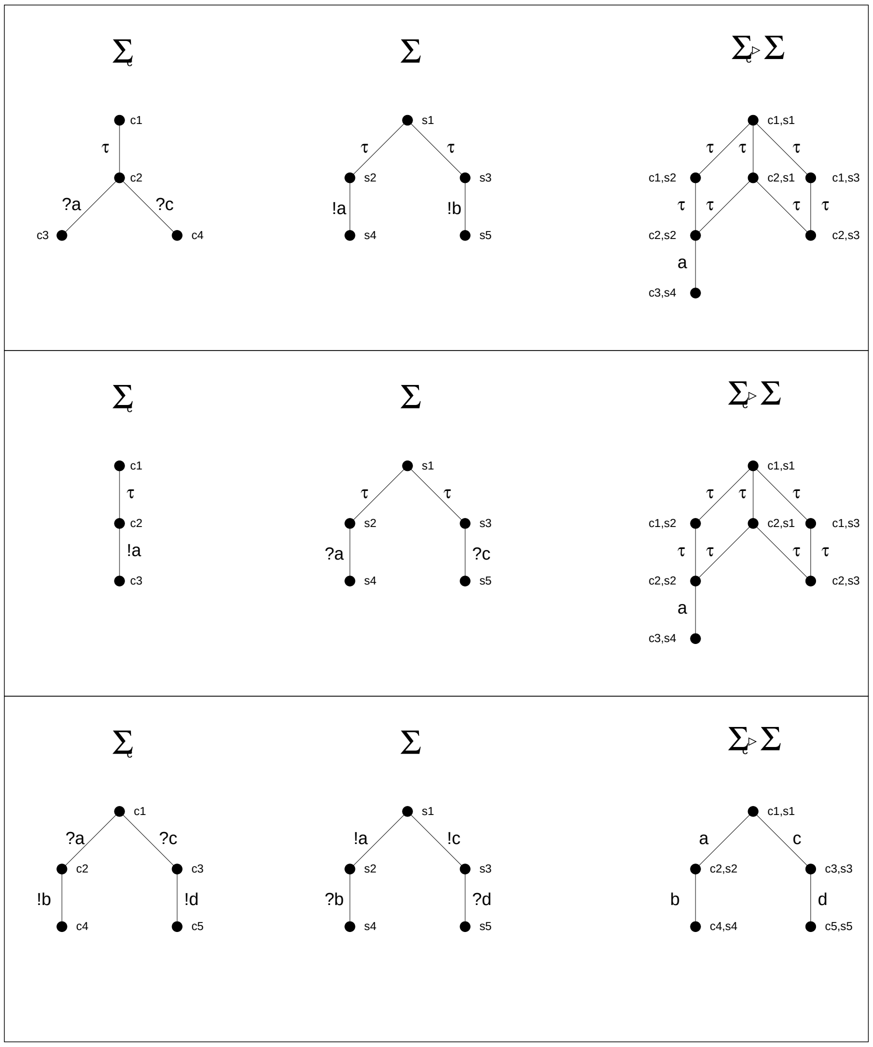


Figure 5.1: Some examples of deadlock-prone and deadlock-free controllers

Example 13 *Fig. 5.1 shows some examples where the controller is prone to deadlocks, and an example of deadlock-free controller.*

In the top section of the figure, Σ_c is not deadlock-free for Σ due to the fact that it may not be able to handle one of the outputs produced by Σ, violating requirement (1) of Def. 12 (more specifically, output b does not match input c in state $(c2, s3)$ of the controlled system).

In the middle section of the figure, Σ_c is not deadlock-free for Σ due to the fact that it may produce an output which is not handled by Σ, violating requirement (2) of Def. 12 (more specifically, output a does not match input c in state $(c2, s3)$ of the controlled system).

In the bottom section of the figure, Σ_c is deadlock-free for Σ, as it is easy to verify by analyzing the controlled system.

Notice that our definitions of controller and deadlock-freedom rule out the cases where, in a controlled system, both input and output actions can be executed on the same state, or on states connected by just a sequence of τ-transitions. This is because, at each step of execution, the controller, which we recall is a *deterministic* STS, may uniquely perform, as its next action, either an input or an output. Indeed, avoiding situations where both input and outputs can be executed on the same state, or on states connected just by a sequence of τ-transitions, is desirable: in cases where the environment $\Sigma_{\|}$ can both receive inputs commands or independently evolve by emitting an output, and in the absence of a queuing mechanism, the relative speeds of the I/Os would decide whether the controller gets deadlocked or not. This is shown by the following example.

Example 14 *Consider Fig. 5.2; we intend to control Σ in such a way that some dark shaded states are reached (namely, $s8$ or $s9$).*

Driving Σ to always get to $s8$ or $s9$ is not possible without incurring into the possibility of a deadlock. Indeed, the STS Σ_c in the upper part of the figure is designed with this idea in mind, but it is non-deterministic, and as a consequence, critical runs may occur in states $(c2, s5)$ and $(c2, s6)$ of the corresponding controlled system, breaking the deadlock-freedom requirements in Def. 12.

Let us now relax our requirement, and also admit reaching the bright shaded state $s7$. The deterministic controller in the bottom part of the figure satisfies such a requirement. The associated controlled system admits no chance of deadlock, and as one can see, at each step of its execution, it is fully determined whether (apart from internal evolutions) the next step will be an input or an output operation.

5.1.2 Composition Requirements

Consider the following example.

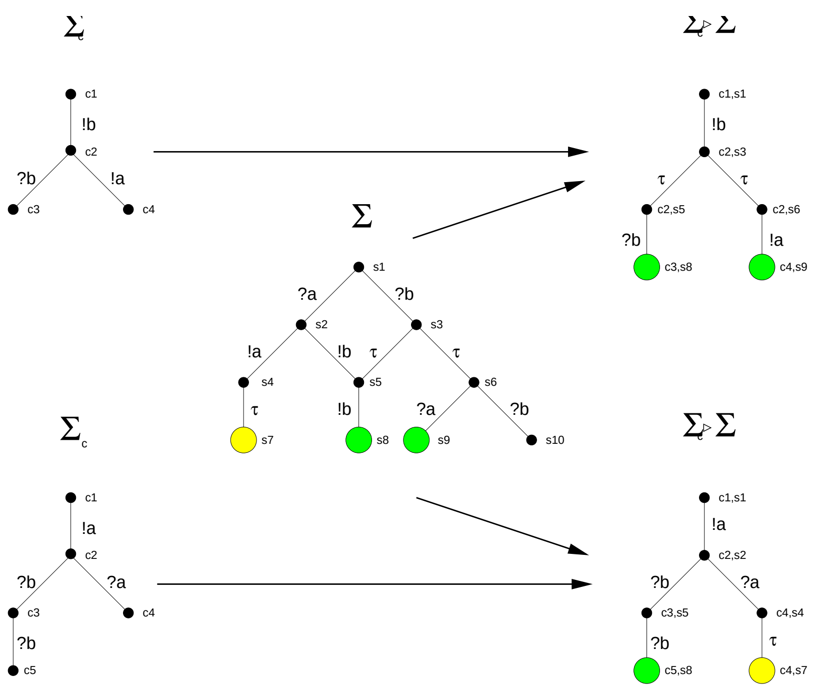


Figure 5.2: Controllers and input/output.

Example 15 *A reasonable requirement for the P&S is to sell the product at destination as requested by the customer. This means to conclude a transaction started by a customer request successfully. In terms of the interaction flows specified by the three available abstract* WS-BPEL, *this means to reach the situation where the User, the Producer, and the Shipper receive a final confirmation (the User should get to execute line 36 in Figure 2.4, while the Producer should execute line 42 in Figure 2.3).*

However, this requirement 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 2.4), and the Producer should terminate either at line 45 or at line 32 in Figure 2.3.*

Requirements like the one above 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 flow.

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 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:

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

$$DEFINED(User.offerVar\_delay) \rightarrow$$
$$User.offerVar\_delay = Shipper.offerVar\_delay + Producer.offerVar\_delay \wedge$$
$$DEFINED(User.offerVar\_cost) \rightarrow$$
$$User.offerVar\_cost = Shipper.offerVar\_cost + Producer.offerVar\_cost \wedge$$

Intuitively, a controller is a *solution* for the requirement $\rho = \rho_c \wedge \rho_d$ if and only if it guarantees that ρ is achieved. We can formally express this by requiring that every execution of the controlled system $\Sigma_c \triangleright \Sigma_{||}$ ends up in a state where ρ holds.

Defi nition 16 (Satisfi ability) *Let $p, \not p$ be propositional formulas, Σ an STS, $s, s', \ldots$ states of Σ.*

We say that Σ satisfies a proposition p, denoted with $\Sigma \models_f p$, if and only if

- *there exists no infinite run of Σ*
- *every final state s_f of Σ satisfies p according to the standard notion of satisfaction of a property on a state.*

Defi nition 17 (Solution controller) *A controller Σ_c is a solution for goal ρ if and only if $\Sigma_c \triangleright \Sigma_{||} \models \rho$.*

We can now characterize formally a (web service) composition problem.

Defi nition 18 (composition problem) *Let $\Sigma_1, \ldots, \Sigma_n$ be a set of state transition systems, and let ρ be a composition requirement. The composition problem for $\Sigma_1, \ldots, \Sigma_n$ and ρ is the problem of finding a deadlock-free controller Σ_c such that $\Sigma_c \triangleright (\Sigma_1 \parallel \ldots \parallel \Sigma_n) \models \rho$*

The definition of composition problem above does not hint at a practical way to identify a deadlock-free controller Σ_c that solves the problem, since the search space used to look for Σ_c is not explicitly represented. In this section, we rephrase the problem so that the search space used to look for Σ_c is explicitly represented; this way, it will be easy to devise a search algorithm that solves the problem.

We start by observing that $\Sigma_{||} = \Sigma_1 \parallel \ldots \parallel \Sigma_n$ is a (nondeterministic) STS that is only partially observable by Σ_c. That is, at execution time, the composite service Σ_c cannot in general get to know exactly what is the current state of the component services modeled by $\Sigma_{W_1}, \ldots, \Sigma_{W_n}$. This uncertainty has two different sources. The first is the presence of non-deterministic transitions in a situation where direct observation of the domain state is impossible; this is something also considered by several approaches to planning. The second source of uncertainty is due to the fact that we are modeling an asynchronous framework, where entities may evolve internally through τ-transitions. (So far this has never been considered in planning, where systems are assumed to behave synchronously).

Such incomplete knowledge on the runtime state of the environment under control must be taken into account by grouping together its observationally indistinguishable executions. In particular, at each step of the execution, we need to consider that a set of different states may be equally plausible given the partial knowledge that we have of the system. Such a set of states is called a *belief state*, or simply *belief*.

The initial belief for the execution is the τ-closure of the initial states $\mathcal{S}^0$ of $\Sigma_{\|}$. This belief is updated whenever $\Sigma_{\|}$ performs an observable (input or output) transition. More precisely, if $B \subseteq \mathcal{S}$ is the current belief and an action $a \in \mathcal{I} \cup \mathcal{O}$ is applied, then the new belief $B' = \textit{Evolve}(B, a)$ is defined as the set of states reachable by first applying a, and then considering every (possibly empty) sequence of internal transitions on the resulting states. We remark that a has to be applicable over the belief, and that such notion of applicability is not a direct generalization of the one over the states $s \in B$. First of all, a state in B may evolve within B due to τ-transitions, and during such internal evolution it is perfectly admissible that transient states are traversed where the action a is not handled. Moreover, since output actions are executed just on some states, depending on the choice performed by the STS under exam, we shall not require that they are executable on every state of B, but just on some state of B.

Defi nition 19 (belief applicability and evolution)
Let $B \subseteq \mathcal{S}$ be a belief of some state transition system Σ. We say that an action $a \in \mathcal{I} \cup \mathcal{O}$ is applicable on a belief B, denoted Appl(B, a), iff:

- *$a \in \mathcal{O}$, and $\exists s \in B$: Appl(s, a), or*
- *$a \in \mathcal{I}$, and $\forall s \in B : \exists s' \in \tau$-closure$(s)$: Appl(s', a).*

The evolution of B under action a is the belief $B' = \textit{Evolve}(B, a)$, where

- *if Appl(B, a), then Evolve$(B, a) = \tau$-closure$(\{s'.\exists s \in B, a \in \mathcal{I} \cup \mathcal{O} : \langle s, a, s'\rangle \in \mathcal{R}\})$;*
- *otherwise, Evolve$(B, a) = \emptyset$*

By starting from the τ-closure of $\mathcal{S}^0$, and iteratively evolving the belief via every applicable action, we obtain the set of beliefs $\textit{Reachable}_{\mathcal{B}}(\Sigma)$ reachable from $\mathcal{S}^0$:

- τ-closure$(\mathcal{S}^0) \in \textit{Reachable}_{\mathcal{B}}(\Sigma)$
- if $B \in \textit{Reachable}_{\mathcal{B}}(\Sigma)$ and $\exists a \in \mathcal{I} \cup \mathcal{O} : \emptyset \neq B' = \textit{Evolve}(B, a)$, then $B' \in \textit{Reachable}_{\mathcal{B}}(\Sigma)$

To evaluate whether a system satisfies a given reachability goal, we must not consider "transient" states of beliefs, that are exited on the basis of internal, unobservable transitions - since we would never be able to establish whether the system is in one of those states or not, and we have no control on the system to prevent (or force) internal transitions. Indeed, we must focus on "stable" states of beliefs, i.e. those that are reached after asynchronously completing every internal evolution, before a new I/O is performed (if possible). This is what we call the *τ-frontier* of B:

Definition 20 (τ-frontier) *Let B be a belief; we call the τ-frontier of B the set τ-frontier$(B) = \{s : \exists s' \in \mathcal{S}, a \in \mathcal{I} \cup \mathcal{O} : (s, a, s') \in \mathcal{R} \vee \forall s' \in \mathcal{S} : (s, \tau, s') \notin \mathcal{R}\}$.*

Since we deal with reachability goals, we can state that a belief satisfies a propositional property p by just considering its τ-frontier: it satisfies p if and only if every states in its τ-frontier of does.

Definition 21 (belief satisfying a property)
Let $\Sigma = \langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{R}, \mathcal{L} \rangle$ be a STS, $p \in \mathcal{P}rop$ be a property for Σ, and $B \subseteq \mathcal{S}$ be a belief. We say that B satisfies p, written $B \models_\Sigma p$, if and only if for any state $s \in \tau$-frontier(B), $p \in \mathcal{L}(s)$.

We can now define a "belief-level" STS for $\Sigma_\|$, whose states are the beliefs that may be traversed by the executions of $\Sigma_\|$, and whose transitions describe belief evolutions:

Definition 22 (belief-level system)
Let $\Sigma = \langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{R}, \mathcal{L} \rangle$ be a STS. The corresponding belief-level STS is $\Sigma_\mathcal{B} = \langle \mathcal{S}_\mathcal{B}, \mathcal{S}^0_\mathcal{B}, \mathcal{I}, \mathcal{O}, \mathcal{R}_\mathcal{B}, \mathcal{L}_\mathcal{B} \rangle$, where:

- *$\mathcal{S}_\mathcal{B} = Reachable_\mathcal{B}(\Sigma)$;*
- *$\mathcal{S}^0_\mathcal{B} = \tau$-closure$(\mathcal{S}^0)$;*
- *transitions $\mathcal{R}_\mathcal{B}$ are defined as follows: if Evolve$(B, a) = B' \neq \emptyset$ for some $a \in \mathcal{I} \cup \mathcal{O}$, then $\langle B, a, B' \rangle \in \mathcal{R}_\mathcal{B}$;*
- *$\mathcal{L}_\mathcal{B}(B) = \{p \in \mathcal{P}rop : B \models_\Sigma p\}$.*

We remark that a belief-level STS is a particular case of a deterministic STS, where there are no τ transitions.

We can now recast the notion of composition problem by imposing conditions not on the controller, but on the observationally equivalent executions it induces on the controlled domain, which are represented by the belief-level system $\text{BEL}(\Sigma_\|)$. In this way, the problem can be solved by searching a satisfactory subset of the belief-level system, that is, intuitively, a subgraph such that, starting from the initial belief, every possible execution finally ends up in some belief that satisfies the property associated to the goal. More specifically, since we consider reachability goals, we will restrict our attention to controllers whose execution is loop-free; this corresponds to searching just for DAG-structured subsets of the belief level system. Moreover, we will consider two additional constraints that come from the deterministic nature of controllers, and from the deadlock-freedom requirement. The determinism of controllers imposes that, in our subtree, at each point, we will need to consider at most one input transition. Deadlock-freedom implies that such controllers never discard outputs of the controlled system; that is, if our subset contains a state B that originates some output transitions in $\textsc{Bel}(\Sigma_{\|})$, then the subset must contain *every* output transitions of $\textsc{Bel}(\Sigma_{\|})$ from B. Thus, we need only to consider portions of the belief-level system structured as follows:

Definition 23 (Execution subtree)
Let $\Sigma_{\mathcal{B}} = \langle \mathcal{S}_{\mathcal{B}}, \mathcal{S}^0_{\mathcal{B}}, \mathcal{I}, \mathcal{O}, \mathcal{R}_{\mathcal{B}}, \mathcal{L}_{\mathcal{B}} \rangle$ be the belief-level system for a domain $\Sigma = \langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{R}, \mathcal{L} \rangle$. We say that $\Sigma'_{\mathcal{B}} = \langle \mathcal{S}'_{\mathcal{B}}, \mathcal{S}^0_{\mathcal{B}}, \mathcal{I}, \mathcal{O}, \mathcal{R}'_{\mathcal{B}}, \mathcal{L}'_{\mathcal{B}} \rangle$ is a controlled execution subtree *of $\Sigma_{\mathcal{B}}$, denoted $\Sigma'_{\mathcal{B}} \subseteq \Sigma_{\mathcal{B}}$, if and only if:*

1. *$\mathcal{S}'_{\mathcal{B}} \subseteq \mathcal{S}_{\mathcal{B}}$ and $\mathcal{R}'_{\mathcal{B}} \subseteq \mathcal{R}_{\mathcal{B}}$*

2. *$\forall \langle B, o, B' \rangle \in \mathcal{R}_{\mathcal{B}} : ((o \in \mathcal{O} \wedge B \in \mathcal{S}'_{\mathcal{B}}) \rightarrow (B' \in \mathcal{S}'_{\mathcal{B}} \wedge \langle B, o, B' \rangle \in \mathcal{R}'_{\mathcal{B}}))$*

3. *If some input transition starts from a belief B, only one such transition may start from B in the subtree. (Formally, this can be stated as $\forall B \in \mathcal{S}'_{\mathcal{B}} : (\exists B' \in \mathcal{S}_{\mathcal{B}}, i' \in \mathcal{I} : (B, a, B') \in \mathcal{R}_{\mathcal{B}} \wedge \exists B'' \in \mathcal{S}'_{\mathcal{B}}, a \in \mathcal{I} \cup \mathcal{O} : (B, a', B'') \in \mathcal{R}'_{\mathcal{B}}) \rightarrow (\exists! B'' \in \mathcal{S}'_{\mathcal{B}}, a \in \mathcal{I} \cup \mathcal{O} : (B, a', B'') \in \mathcal{R}'_{\mathcal{B}} \wedge \forall (B, a', B'') \in \mathcal{R}'_{\mathcal{B}} : a' \in \mathcal{I})$*

4. *no run of $\Sigma'_{\mathcal{B}}$ can traverse the same belief more than once.*

5. *$\mathcal{L}'_{\mathcal{B}}$ is the restriction of $\mathcal{L}_{\mathcal{B}}$ to the transitions in $\mathcal{R}'_{\mathcal{B}}$.*

It can be seen that:

- conditions (1) and (4), and the fact that the initial belief of the subtree is the same of the originating belief-level system, insure that the subtree is a DAG-structured portion of the belief-level system;

- condition (2) requires that no output generated from the environment is disregarded;

- condition (3) states that, in the subtree, we only need to consider a uniquely determined way of driving the environment, since the controller is deterministic.

- since the controller is deterministic, it is never the case that both an input and an output transitions start from the same state B.

Given its acyclic structure, a subtree can be associated to a notion of *subtree depth*: the maximal length of the sequences $B_0, B_1, ..., B_n$ such that $B_i, a, B_{i+1} \in \mathcal{R}_{\mathcal{B}}$. Intuitively, the depth of a subtree identifies the number of steps needed, in the worst case, to terminate the execution of the controlled system.

A subtree π of $\textsc{Bel}(\Sigma_{\|})$ can be associated to a controller that, once connected to $\Sigma_{\|}$, drives it in such a way that only the beliefs of π can be traversed during the execution of the controlled system. Amongst the controllers that satisfy this requirement, we can focus on a synchronous controller for which we can give an immediate constructive definition: the structure of the controller mirrors the one of the subtree, featuring one state for each state of the subtree and contra-posing one-to-one its I/O operations with those of the subtree.

Defi nition 24 (Controller associated to a subtree) *Let $\Sigma'_{\mathcal{B}} = \langle \mathcal{S}'_{\mathcal{B}}, \mathcal{S}^0_{\mathcal{B}}, \mathcal{I}, \mathcal{O}, \mathcal{R}'_{\mathcal{B}}, \mathcal{L}'_{\mathcal{B}} \rangle \subseteq \Sigma_{\mathcal{B}}$ be an execution subtree for (the belief-level system associated with) an STS Σ. We say that the STS $Contr(\Sigma'_{\mathcal{B}}) = \langle \mathcal{S}'_{\mathcal{B}}, \mathcal{S}^0_{\mathcal{B}}, \mathcal{O}, \mathcal{I}, \mathcal{R}'_{\mathcal{B}}, \mathcal{L}_{\emptyset} \rangle$ is the controller associated to π, where $\mathcal{L}_{\emptyset}$ denotes the empty labeling function.*

Lemma 25 *Let Σ be an STS, and Σ' be a subtree for Σ. The STS $Contr(\Sigma')$ is a controller for Σ.*

Thus $Contr(\Sigma_{\mathcal{B}})$ can be used to control Σ; however, to prove that such a controller is also deadlock-free for Σ, we need to first show that the execution of $Contr(\Sigma'_{\mathcal{B}}) \triangleright \Sigma_{\|}$ traverses states (B, s) such that $s \in B$ (remember that the states of $Contr(\Sigma'_{\mathcal{B}})$ are beliefs of Σ).

Lemma 26 *Let Σ be an STS. Let $\Sigma'_{\mathcal{B}}$ be a subtree of (the belief-level system for) Σ. Every state $(B, s) \in Reachable(Contr(\Sigma'_{\mathcal{B}}) \triangleright \Sigma_{\|})$ is such that $s \in B$.*

Theorem 27 *Let $\Sigma_{\|}$ be an STS. Let $\Sigma'_{\mathcal{B}} \subseteq \textsc{Bel}(\Sigma)$. Then $Contr(\Sigma'_{\mathcal{B}})$ is deadlock-free for $\Sigma_{\|}$.*

The notion of solution controller can be directly recasted to speak instead of the corresponding controlled subtree: it is enough to require that every leaf of the controlled subtree is a belief satisfying ρ.

Defi nition 28 (Solution subtree) *Let $\Sigma_{\mathcal{B}} = \langle \mathcal{S}_{\mathcal{B}}, \mathcal{S}^0_{\mathcal{B}}, \mathcal{I}, \mathcal{O}, \mathcal{R}_{\mathcal{B}}, \mathcal{L}_{\mathcal{B}} \rangle$ be the belief-level system for a domain $\Sigma = \langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{R}, \mathcal{L} \rangle$. The execution subtree $\Sigma'_{\mathcal{B}} \subseteq \Sigma_{\mathcal{B}}$ is a solution for the requirement ρ if every leaf B of $\Sigma'_{\mathcal{B}}$ is such that $B \models_{\Sigma} \rho$.*

A solution subtree π can be associated to a solution controller $Contr(\pi)$, and vice versa.

Lemma 29 *Let π be a solution subtree of* $\text{BEL}(\Sigma_{\|})$ *for the problem of service composition over the requirement ρ. Then $Contr(\pi)$ is a solution controller for the same problem.*

To associate a subtree of $\text{BEL}(\Sigma_{\|})$ with a controller Σ_c, we observe that, following Def. 10, Σ_c constrains the behaviors of $\Sigma_{\|}$: only certain states and transitions of $\Sigma_{\|}$can ever be traversed during the execution of $\Sigma_c \triangleright \Sigma_{\|}$. That is, Σ_c *induces* a constrained subset of $\Sigma_{\|}$, which we denote with $\Sigma_{\|}(\Sigma_c)$. We now show that $\Sigma_{\|}(\Sigma_c)$ is a solution subtree if Σ_c is a solution controller:

Definition 30 (Induced STS) *Let $\Sigma_{\|} = \langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{R}, \mathcal{L} \rangle$ be an STS describing a set of component services, and let $\Sigma_c = \langle \mathcal{S}_c, \mathcal{S}_c^0, \mathcal{O}, \mathcal{I}, \mathcal{R}_c, \mathcal{L}_\emptyset \rangle$ be a controller for $\Sigma_{\|}$.*

The STS $\Sigma_{\|}(\Sigma_c) = \langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{R}', \mathcal{L} \rangle \subseteq \Sigma_{\|}$, where $\mathcal{R}' = \{(s, a, s') : \exists s_c, s'_c : ((s_c, s) \in Reachable(\Sigma_c \triangleright \Sigma_{\|}) \wedge ((s_c, s), a, (s'_c, s')) \in \mathcal{R}_c \triangleright \mathcal{R})\}$, is called the STS induced by Σ_c on $\Sigma_{\|}$.

Lemma 31 *If $\Sigma_c \triangleright \Sigma_{\|} \models \rho$, then* $\text{BEL}(\Sigma_{\|}(\Sigma_c)) \subseteq \text{BEL}(\Sigma_{\|})$ *is a solution subtree for the composition problem.*

The above lemmas can be combined to associate in a one-to-one way the existence of a solution with that of a solution subtree:

Theorem 32 (composition problem at the belief-level)
Let $\Sigma_1, \ldots, \Sigma_n$ be a set of state transition systems, and let ρ be a composition requirement. A solution controller for the problem exists if and only if a solution subtree of $\text{BEL}(\Sigma_1 \| \ldots \| \Sigma_n)$ *exists.*

Moreover, we can conclude that if a solution subtree π has been found, a solution controller $Contr(\pi)$ can be obtained: a controller Σ_c that induces a solution subtree of $\text{BEL}(\Sigma_1 \parallel \ldots \parallel \Sigma_n)$ is a solution to the composition problem.

Thus we can recast the composition problem as that of identifying a solution subtree $\pi \subseteq \text{BEL}(\Sigma_1 \parallel \ldots \parallel \Sigma_n)$; the loop-free controller $Contr(\pi)$ will be taken as the solution of the problem.

5.2 Search by belief-level pre-compilation

The recasting of the composition problem as search of a satisfactory subtree suggests that one way to solve the problem may consist in (a) building the whole belief-level system, and then (b) searching for a solution subtree inside it. The practicality of this approach is

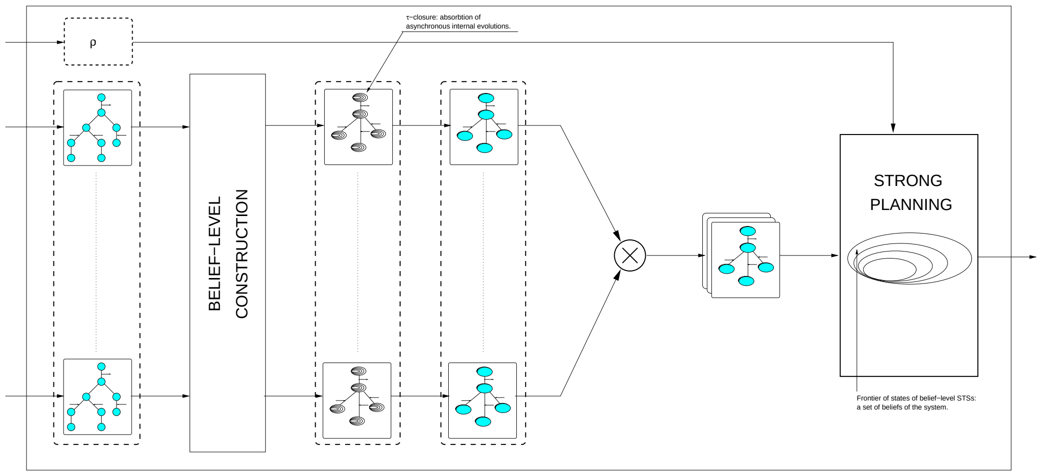

ρ
τ−closure: absorbtion of asynchronous internal evolutions.
BELIEF−LEVEL CONSTRUCTION
STRONG PLANNING
Frontier of states of belief−level STSs: a set of beliefs of the system.

Figure 5.3: Domain construction and search for belief-level pre-compilation.

helped by the fact that, under our proviso that components never communicate directly to each other, the belief-level system for their parallel product is the parallel product of their belief-level systems. This commutativity helps reducing the impact of constructing the belief-level system, allowing for a modular approach where the potentially exponential blow-up of enumerating beliefs is eliminated for good part.

Based on these considerations, our approach will be structured as follows:

1. Construct, for each component service, an STS Σ_i;
2. Construct the parallel product $\Sigma_\|$, and then its belief-level counterpart $Bel(\Sigma_\|)$;
3. Search the execution structure in $Bel(\Sigma_\|)$ to identify one execution tree associated to a solution controller;
4. Extract, from the execution tree, the controller Σ_c, and emit it as a WS-BPEL program.

In the following subsections, we detail the core of this approach, i.e. the algorithm used for constructing $Bel(\Sigma_\|)$, and the one for searching a solution subtree inside it.

5.2.1 Implementation and results

Following the ideas stated in the previous section, we solve the composition problem by first constructing the belief-level machine BEL$(\Sigma_\|)$, and then searching for a solution subtree π inside it. Once such a solution is found, the (loop-free) controller $Contr(\pi)$ is extracted and constitutes a solution to the composition problem.

To construct the belief-level machine BEL$(\Sigma_\|)$, we rely on an algorithm that implements Def. 22 by a simple fix-point algorithm that performs a forward search of the space of beliefs, starting from the initial belief. At each iteration of the fix-point, a τ-closure operator is used that realizes Def. 11, so that indistinguishable states of the STS are grouped together.

While it is possible to apply the algorithm directly on $\Sigma_{\|}$, we can do better. In particular, we exploit the fact that $\textsc{Bel}(\Sigma_1 \parallel \Sigma_2) = \textsc{Bel}(\Sigma_1) \parallel \textsc{Bel}(\Sigma_2)$. This allows to separately compute the belief-level system for each component service, and to represent the overall belief-level system as a modular composition of the various $\textsc{Bel}(\Sigma_i)$, where modules evolve on the basis of a parallel product semantics.

The impact of this approach on the effectiveness of the belief-level construction is clear: since the algorithm, given an STS Σ, explicitly enumerates the reachable beliefs of $\textsc{Bel}(\Sigma)$, and since they can be up to $2^{|\mathcal{S}|}$, the monolithic approach may have to enumerate $2^{|\mathcal{S}_1|+\ldots+|\mathcal{S}_n|}$ beliefs, while the partitioned approach only $2^{|\mathcal{S}_1|}+\ldots+2^{|\mathcal{S}_n|}$.

Once the belief-level system is built, we exploit a variant of the algorithm for strong planning presented in [CPRT03] to extract a solution subtree from it. Such algorithm receives as input the STS representing $\textsc{Bel}(\Sigma_{\|})$, produced by the belief-level construction algorithm, and performs a symbolic regression search, where frontiers of states of $\textsc{Bel}(\Sigma_{\|})$ are manipulated at once.

We remark that a state of $\textsc{Bel}(\Sigma_{\|})$ corresponds to a belief, i.e. a set of states of the component services; then, symbolically manipulating sets of states of $\textsc{Bel}(\Sigma_{\|})$ means symbolically manipulating sets of sets of states of the component services. This is an extremely powerful technique, that avoids enumerating large sets of sets of service states.

To perform such symbolic search, the algorithm relies on a strong pre-image primitive $\textsc{StrongPreImage}$, and on a pruning primitive $\textsc{PruneStates}$. Given a set S of states of $\textsc{Bel}(\Sigma_{\|})$, the pre-image primitive returns a set of state-action pairs $\{\langle s, a\rangle\}$ that corresponds to the predecessors of the states in S via some input/output action, such that the execution of the input/output action a in s is guaranteed to lead to states inside S, regardless of nondeterminism. The pre-image primitive is defined as follows:

$$\begin{aligned}\textsc{StrongPreImage}(S) \quad \doteq \quad & \{(s,i) : i \in \mathcal{I} \wedge \forall s' : (s,i,s') \in \mathcal{R} \rightarrow s' \in S\} \cup \\ & \{(s,o) : o \in \mathcal{O} \wedge \exists s' : (s,o,s') \in \mathcal{R} \rightarrow s' \in S \wedge \\ & \quad \forall s' \in \mathcal{S}, o' \in \mathcal{O} : (s,o',s') \in \mathcal{R} \rightarrow s' \in S\}.\end{aligned}$$

Notice that the deadlock-freedom requirement implies that input and output actions are treated differently when looking for a strong pre-image. In particular, since outputs correspond to uncontrollable choices taken by the component services, once we consider an output from a service, we have to consider *every* possible outputs from it, and guarantee that every outcome is in the target set.

This requirement has no correspondence in the formulation of strong planning of [CPRT03], where only the controller is responsible for taking actions. Indeed, [CPRT03] relies on a different STRONGPREIMAGE primitive, where actions are handled uniformly (and analogously to inputs here).

The pruning primitive PRUNESTATES is used, after executing a pre-image, to remove, from a state-action table π, all the pairs $\langle s,\ a\rangle$ such that a solution is already known for s and stored into S. It is defined as:

$$\text{PRUNESTATES}(\pi, S) \doteq \{\langle s,\ a\rangle \in \pi : s \notin S\}.$$

This pruning is important to guarantee that only the shortest solution from any state appears in the state-action table.

The search algorithm is depicted in Fig. 5.4, and basically consists of a fix-point iteration that incrementally constructs a so-called *state-action table*, i.e. a set of pairs that associate to a state of the domain an action to be executed. The state-action table SA is initialized as empty, and enriched, at each iteration, with states from which it is possible to control the system to achieve states in SA. The algorithm terminates either when no more states can be added to the state-action table, or when the current state-action table already includes the initial states, in which case a solution has been found.

The arguments for proving that the algorithm always terminates, and it is correct and complete, are similar to those presented in [CPRT03].

Termination is easily proved by observing that the number of states in the visited domain is finite, and that the number of states in the state-action table grows up monotonically.

Theorem 33 (Termination) *Let $\Sigma_{\mathcal{B}} = \langle \mathcal{S}_{\mathcal{B}}, \mathcal{S}^0_{\mathcal{B}}, \mathcal{I}, \mathcal{O}, \mathcal{R}_{\mathcal{B}}, \mathcal{L}_{\mathcal{B}}\rangle$ be a (belief-level) STS; let $I \in \mathcal{S}_{\mathcal{B}}$ and $G \subseteq \mathcal{S}_{\mathcal{B}}$. The execution of $plan(I, G)$ on $\Sigma_{\mathcal{B}}$ terminates.*

To prove correctness and completeness, we rely on proving that invariantly, at each iteration of the main loop at lines 4-9, the state-action table contains states for which a solution tree exists, and such tree is indeed represented by (a portion of) the state-action table. Indeed, starting from a set of initial states, a state-action table induces an STS:

Defi nition 34 (STS corresponding induced by state-action table) *Let $SA = \{\langle s, a\rangle : s \in \mathcal{S}, a \in \mathcal{I} \cup \mathcal{O}\}$ be a state-action table, let $I \subseteq \{s : \exists a \in \mathcal{I} \cup \mathcal{O}. \langle s, a\rangle \in SA\}$ be a set of initial states, and let $G \subseteq \mathcal{S}$ be a set of goal states.*

The STS induced by I on SA, denoted $STSof(SA, I, G) = \langle \mathcal{S}_{SA}, I, \mathcal{I}, \mathcal{O}, \mathcal{R}_{SA}, \mathcal{L}_{SA}\rangle$, is defined as follows:

- *$I \subseteq \mathcal{S}_{SA}$*
- *if $s \in \mathcal{S}_{SA}$, and there exists a sequence $\langle s_0, a_0\rangle, ..., \langle s_n, a_n\rangle$ such that $s_0 = s$, $\langle s_i, a_i\rangle \in SA$, $\forall i \in [0, n-1] : \langle s_i, a_i, s_{i+1}\rangle \in \mathcal{R}$, and $\langle s_n, a_n, s_f\rangle \in \mathcal{R}$, then $s_f \in \mathcal{S}_{SA}$.*

- *if $s \in \mathcal{S}_{SA}$ and $\langle s, a, s' \rangle \in \mathcal{R}$, then $\langle s, a, s' \rangle \in \mathcal{R}_{SA}$.*
- *if $\mathcal{L}_{SA}$ is the restriction of $\mathcal{L}$ to the states in $\mathcal{S}_{SA}$*

Lemma 35 (Invariant property of the state-action table) *At the i-th iteration of the main loop at lines 4-9, $G \cup \textsc{StatesOf}(SA)$ contains all the states for which a solution subtree of depth up to i exists. In particular, if $s \in G \cup \textsc{StatesOf}(SA)$, then $STSof(SA, I, G)$ is a solution subtree rooted at s.*

Theorem 36 (Correctness and completeness) *If the algorithm returns a state-action table SA, then SA contains a solution subtree to the composition problem. If the algorithm returns $Fail$, then no solution subtree to the composition problem exists.*

The implementation of the algorithms for belief-level domain construction, and for searching the solution subtree, rely on a symbolic machinery to effectively represent and manipulate sets of states, where Binary Decision Diagrams (BDDs [Bry86a]) are used as a canonical representation for formulas over state variables. This symbolic representation technology has been largely investigated in the areas of Model Checking, Diagnosis and Planning, and is at the core for the improved performances of state-of-the-art tools in these areas.

Notice that, within the two phases (belief-level construction and search), the states of the symbolic representation assume very different meanings.

In both phases, we will adopt the SMV language to describe STSs: in belief construction, both for the domain given in input and the outputted belief-level STSs; in search, as the input domain. SMV is the native language used in model-checking systems such as NuSMV [CCGR00]. It is a rich language, allowing a variety of constructs and mechanisms to compactly represent STSs. In particular, one of the most relevant features for our purpose is the possibility to define independent “modules”; semantically, independent modules are equivalent to their parallel product, while internally, a disjunctively partitioned transition relation is stored so to save significant space and computation time.

This is particularly useful in belief-level construction: given the disjointness of input/outputs amongst component services, it holds that $\textsc{Bel}(\Sigma_1 \parallel \ldots \parallel \Sigma_n) = \textsc{Bel}(\Sigma_1) \parallel \ldots \textsc{Bel}(\Sigma_n)$, and as such, we can build the belief-level independently for each component, and implicitly combine them by representing them as modules.

Thus, in belief-level construction, the states of the input domain correspond to states of the STS associated to a component web service; in this case, a BDD represents a set of indistinguishable states of one service, i.e. a belief. Thus belief-level construction explicitly enumerates the beliefs for a single component service, but symbolically evolves the sets of states that make them up by internal or input/output actions. The resulting belief-level domain for a component service is emitted as a MODULE in the SMV language, so that the domain in input to the search phase consists of a set of SMV modules whose states represent beliefs.

Finally, the search phase takes place over a domain whose states correspond to beliefs of $\textsc{Bel}(\Sigma_1 \parallel \ldots \parallel \Sigma_n)$, i.e. to tuples $\langle B_1, \ldots, B_n \rangle$ where $B_i \in \textsc{Bel}(\Sigma_i)$. This means that symbolically evolving a set of states, as performed by the search algorithm, corresponds to evolving a set of state sets of the component services; this would be extremely demanding, and practically unfeasible, by an explicit enumeration of the states, making our symbolic factored representation technique virtually necessary.

```
function $plan(I, G)$
$OldSA := Fail$
$SA := \emptyset$
while $(OldSA \neq SA \wedge I \notin (G \cup \textsc{StatesOf}(SA)))$
  $Pr := \textsc{StrongPreImage}(G \cup \textsc{StatesOf}(SA))$
  $NewSA := \textsc{PruneStates}(Pr, G \cup \textsc{StatesOf}(SA))$
  $OldSA := SA$
  $SA := SA \cup NewSA$
  done
if $(I \in (G \cup \textsc{StatesOf}(SA)))$
  return $SA$
else
      return $Fail$
endif
```

Figure 5.4: The search algorithm.

In order to test the performance of the proposed technique, we have conducted some experiments using WS-GEN. All experiments have been executed over a 1.2GHz Pentium machine, equipped with 4 GByte memory, running Linux. All of our tests have been run setting a timeout at 3600 seconds of CPU usage.

Our analysis aims at identifying which structural and dimensional features of web services impact on the complexity of their composition, and in which way. This allows us to discuss the scalability and limits of out approach.

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, we will vary 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.

We start by considering the simplest 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.

Given a set of such components, 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). In turn, this requires a composed protocol that suitably interleaves the invocations to the components, passing at each step the current result.

The results are shown in Fig. 5.5. 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.

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.

We then step to the smallest 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.

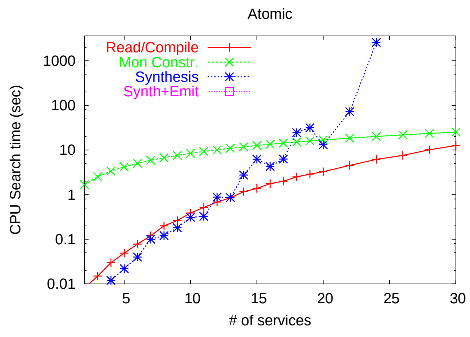


Figure 5.5: Combining N atomic services

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. The requirement for combining a number of such services consists in asking for the set of services to 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.

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
- 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 for this test are reported in fig 5.6.

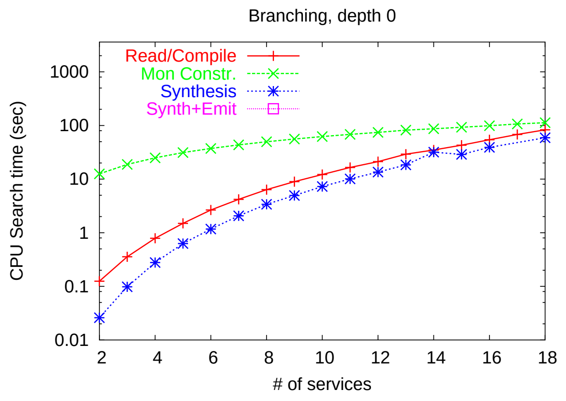


Figure 5.6: Combining N branching services

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.

We now consider services that encode multiple interactions, where each interaction may possibly lead to a failure of the service, and such that a large variety of functions may be computed.

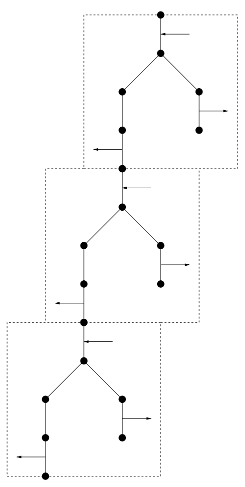


Figure 5.7: A 3-cell binary unbalanced component.

Binary unbalanced protocols 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. 5.7 schematically show a component service with 3 cells.

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. 5.8.

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

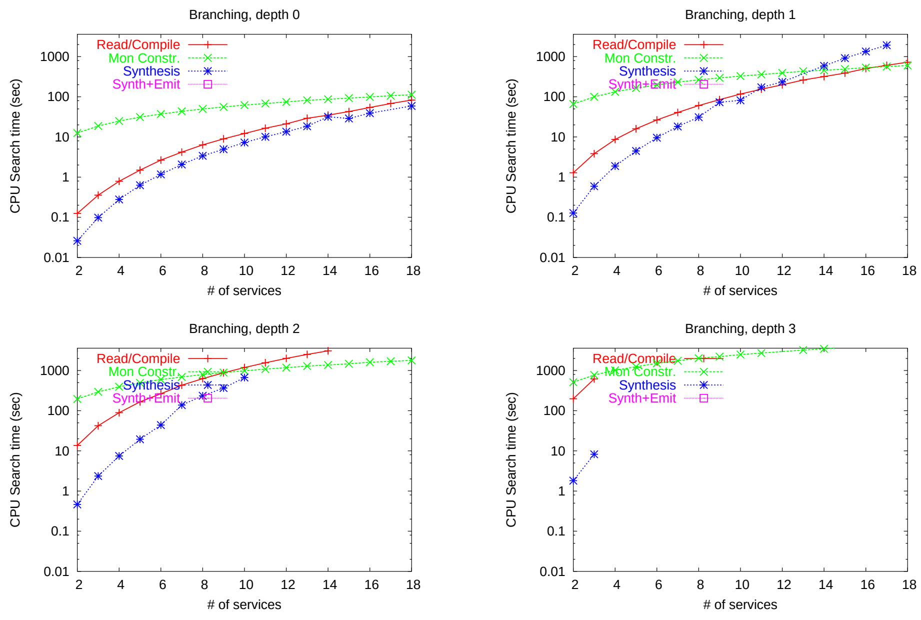


Figure 5.8: Combining N binary unbalanced services of depth 1,2,3,4

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. 5.9 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

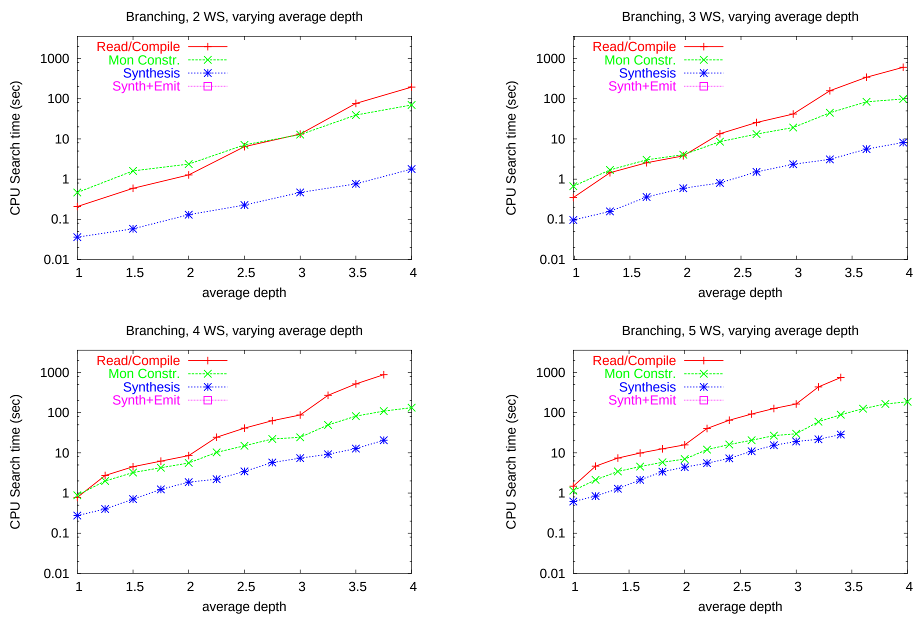


Figure 5.9: Combining 2,3,4,5 branching services of increasing depth

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.

Binary balanced protocols 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. 5.10 shows a schema of such a balanced component of depth 2.

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. 5.11.

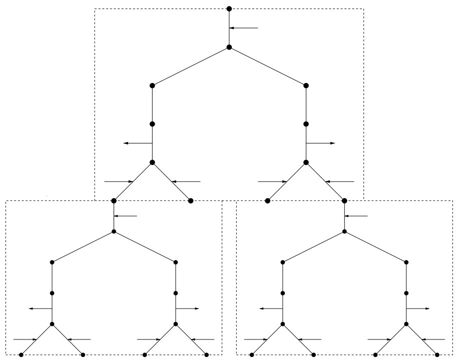


Figure 5.10: A 2-depth "balanced branch" component.

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. 5.12. Again, we witness an 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.

N-ary balanced protocols 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

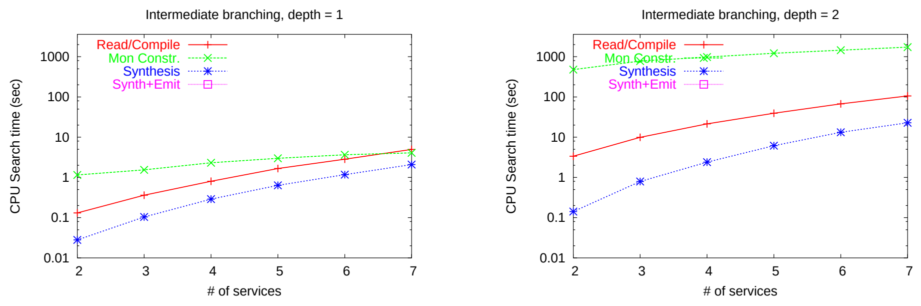


Figure 5.11: Combining N binary balanced services of fixed depth (1,2)

width. In particular, we combine two services which implement a short nondeterministic protocol admitting a variety of choices, either because of internal nondeterminism, or 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. These two services can be composed by requiring that one of the possible function combinations

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

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. 5.13, 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.

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

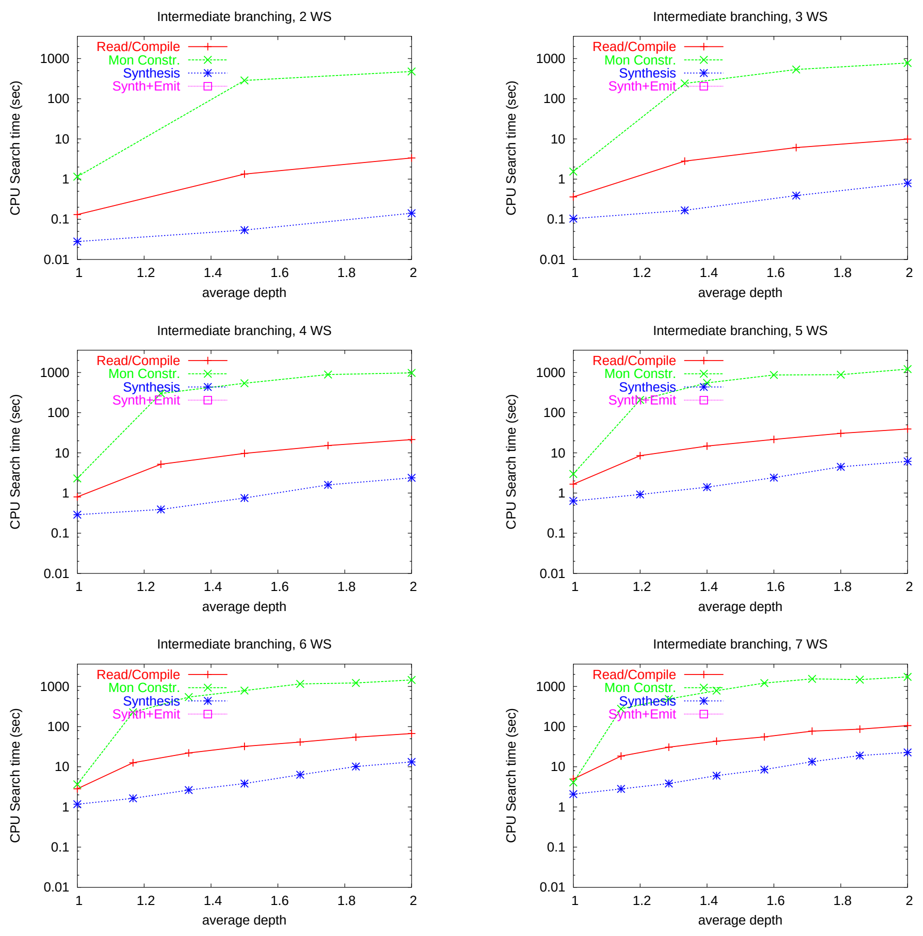


Figure 5.12: Combining sets of 2 to 7 balanced branching services of varying average depth

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.

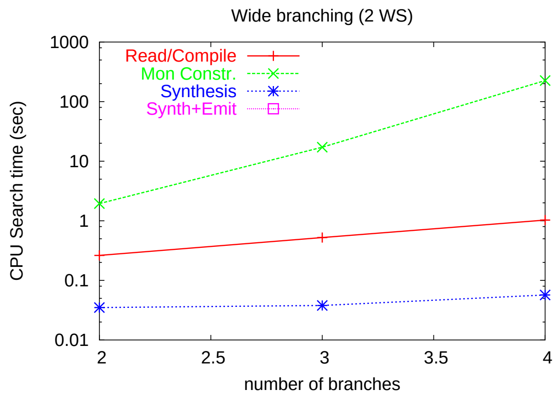


Figure 5.13: Combining two branching services of increasing branching factor

5.3 On-the-fly belief-level search

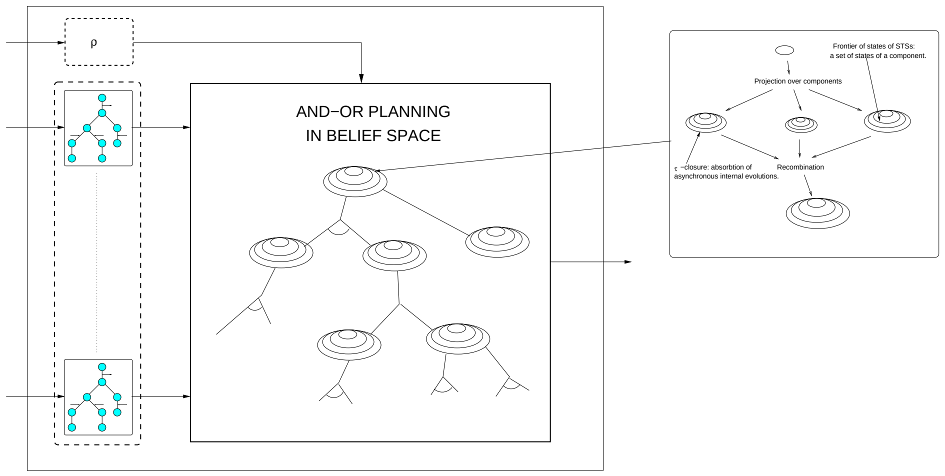

ρ
AND−OR PLANNING
IN BELIEF SPACE
Frontier of states of STSs:
a set of states of a component.
Projection over components
τ −closure: absorbtion of
asynchronous internal evolutions.
Recombination

Figure 5.14: Domain construction and search for on-the-fly belief-level search.

While the commutativity of belief-level construction and parallel product helps a lot the performance of the approach depicted so far, the possible blow-up of belief space remains a relevant issue when large components services take part to the composition - a typical scenario e.g. in e-government. In those cases, disrespectful of the complexity of the composition, the belief-level construction phase may become a hard bottleneck, and a different technique is needed.

We will take again inspiration from the belief-level recasting of the composition problem, and we observe that, considering reachability goals, the problem is rather similar to the problem of finding a conditional, tree-structure plan for a partially observable domain - indeed, a problem which can be solved by and-or search over a synchronous STS.

We will now present a search algorithm to solve our problem, inspired from the belief

```
procedure StrongPO(I,G)
  $ST$ := MkInitialTree(TauClose (I));
  if (TauClose($I$) $\subseteq G$) then
    TagNode(Root($ST$), $Success$);
  fi
  while ($\neg$isSuccess(Root($ST$)) $\wedge$ $\neg$isFailure(Root($ST$))) do
    node := ExtractNodeFromFrontier($ST$);
    ExtendTree(node, $ST$);
    if (SonsYieldSuccess(node, $ST$)) then
      TagNode(node, $Success$);
      PropagateSuccessOnTree(node, $ST$);
    else if (SonsYieldFailure(node, $ST$)) then
      TagNode(node, $Failure$);
      PropagateFailureOnTree(node, $ST$);
    fi
  done
  if isSuccess(Root($ST$)) then
    return BuildPlan(Root($ST$), $ST$);
  else
    return $Failure$;
  fi
end
```

Figure 5.15: The planning algorithm.

and-or search approach presented in [BCRT01, BCR01], to which we add the treatment of asynchronism - a notable conceptual departure from the synchronous model adopted by existing planning approaches.

The algorithm is shown in Fig. 5.15. In Fig. 5.16 we show its main subroutine, which is concerned with progressing a belief, and in Fig. 5.17 we show the subroutine that implements the τ-closure operation, and maintains the τ-frontier states. Given in input an STS and a requirement ρ, the algorithm returns either an STS representation of a tree structure satisfying ρ, or a failure message indicating that no such tree exists - i.e. that no controller may implement the required composition. In the former case, the tree structure is converted as an STS, which is amenable to being translated back into BPEL4WS via the STS2BPEL translator.

This planning algorithm performs forward-chaining by iteratively expanding an and-or graph whose nodes are beliefs, and whose edges represent input/outputs actions of $\Sigma_{\|}$. Conceptually, the and-or graph represents a prefix of the search space at the belief level, and can be interpreted as a collection of subtrees of Bel($\Sigma_{\|}$); that is, the algorithm progresses and maintains such a set of subtrees, until either it finds a satisfactory one, or no further expansion is possible.

```
1  **procedure** EXTENDTREE$(n, ST)$
2    **forall** $a \in \mathcal{I}$
3      **if** $applicable(\text{BEL}(n), a)$ **then**
4        $n' := \langle \text{TAUCLOSE}(\mathit{Exec}(\text{BEL}(n), a)), \text{PATH}(n) \circ a \rangle$
5        $Nodes(ST) := Nodes(ST) \cup \{n'\}$
6        $OrArcs(ST) := OrArcs(ST) \cup \langle n, a, n' \rangle$
7        TAGNEWNODE$(n', ST)$
8      **fi**
9    **endfor**
10   **forall** $o \in \mathcal{O}$
11     **if** $(\mathit{Exec}(\text{BEL}(n), o) \neq \emptyset)$
12       $n' := \langle \text{TAUCLOSE}(\mathit{Exec}(\text{BEL}(n), o)), \text{PATH}(n) \circ o \rangle$
13       $Nodes(ST) := Nodes(ST) \cup \{n, n'\}$
14       $AndArcs(ST) := AndArcs(ST) \cup \langle n, o, n' \rangle$
15       TAGNEWNODE$(n', ST)$
16     **endfor**
17   **end**
```

Figure 5.16: The node expansion primitive.

Consistently with the interpretation of an STS in terms of a planning domain presented in [PTBM05, PTB05], we recast input actions as planning domain actions, and output actions as the observations in the planning domain. As such, input actions contribute the "or" component of the search, and are associated to or-edges, while output actions contribute the "and" component of the search, and are associated to and-edges. Each node n in the graph is associated, other than with a belief state BEL(n), with the path of actions and observations PATH(n) used to reach it, and with a three-valued 'tag': $Success$ indicates that it is possible to control $\Sigma_{\|}$ to reach the goal starting from n, $Failure$ indicates that this is impossible, and $Undetermined$ indicates that it has not yet been possible to establish whether this is possible or not. The corresponding predicates are ISSUCCESS, ISFAILURE, and ISUNDET. The TAGNODE primitive sets the value of the tag for a node.

The algorithm first initializes the graph with the start node (line 2), and checks if an empty subtree is a solution to the problem (lines 3-4). Then, the main loop is entered (lines 6-16), where the graph is iteratively expanded until the root is tagged either as success or as failure. When the loop is exited (lines 17-20), a controller correspondent to the satisfactory subtree is constructed and returned in case of success; otherwise, $Failure$ is returned.

The body of the main loop proceeds by selecting and extracting a node from the frontier, and expanding it. With the first step in the loop, at line 7, a node is selected for expansion, and is extracted from the frontier. The EXTRACTNODEFROMFRONTIER primitive embodies the selection criterion, and is responsible for the style of the search (e.g. depth-first versus breadth-first). The selection criterion does not affect the properties of the

```
procedure TAUCLOSE(B)
  $B1 := B$
  $B0 := \emptyset$
  while $(B1 \neq B0)$ do
    $B0 := B1$
    $B1 := B0 \cup \mathit{Exec}(B0, \tau)$
  done
  $B1 = B1 \cap (\{s \in \mathcal{S} : \forall s' \in \mathcal{S} : \langle s, \tau, s' \rangle \notin \mathcal{R}\} \cup$
                $\{s \in \mathcal{S} : \exists a \in \mathcal{I} \cup \mathcal{O}, s' \in \mathcal{S} : \langle s, a, s' \rangle \in \mathcal{R}\})$
  return $B1$;
end
```

Figure 5.17: The routine for τ-closure and pruning.

algorithm, namely completeness, correctness and termination.

At line 8, *node* is expanded, considering every possible action and observation. While the extension of input and output actions is similar, the interpretation of the newly added edges in the graph is rather different: while inputs are a controller's choice, outputs are chosen by the controlled system. That is, inputs are associated to or-edges, and outputs to and-edges.

This expansion step also decides the status of each newly generated node n, updating the frontier consistently: namely, n is tagged $Success$ if $\text{BEL}(n) \subseteq \rho$; it is tagged $Failure$ if and only if there exists an ancestor node associated to the same belief, i.e. a loop has been generated; it is tagged $Undetermined$ otherwise, and added to the frontier.

If it is possible to state the success of *node* based on the status of the newly introduced sons (primitive SONSYIELDSUCCESS at line 9), i.e. if an or-son is successful node, or a pair of and-sons are successful, then *node* is tagged as success (line 10). In this case (line 11), the recursive PROPAGATESUCCESSONTREE primitive propagates success bottom-up on the search tree, to the ancestors of *node*, towards the initial node.

If the node is not detected to be successful, the SONSYIELDFAILURE primitive tries to state if a *node* is a failure. This happens when every or-son causes a loop, and for every and-edge, at least one of the and-sons causes a loop. In this case, the node is tagged as failure, and the failure is propagated bottom-up, in order to cut the search in branches which are bound to fail because some leaf has failed.

Most of the primitives in the algorithm have an immediate semantics and can be easily implemented; we only discuss here the EXTENDTREE primitive, which is responsible for expanding a frontier node by every applicable input/output action. The $applicable$ primitive checks that an action can be executed in every state of a belief, and $Exec$ collects the resulting outcomes from applying it. If the action is applicable, a node and an edge are respectively added to the nodes and to the edges of the current search tree. For every new node n generated, the TAGNEWNODE subroutine evaluates whether it is a success (i.e. $\textsc{Bel}(n) \subseteq \rho$), a failure (i.e. some node in the set of ancestors of n is associated to the same belief), or neither success nor failure; on the basis of this evaluation, n is associated to the corresponding tag.

Both when extending the search structure, and when initializing it, the TAUCLOSE routine in Fig.5.17 is invoked. This routine performs two main tasks: it computes the τ-closure, and it prunes away "transient" states, maintaining only those in the τ-frontier, according to the definition of belief evolution. The two tasks are performed by a symbolic fix-point computation, followed by a filtering operation.

The algorithm is sound and complete:

Theorem 37 *The algorithm always terminates. It returns an execution tree if and only if an execution tree that satisfies the input requirement exists; in that case, the returned execution tree satisfies the input requirement.*

The performance of this algorithm strongly relies on the efficiency of the basic operations on beliefs, and on the existence of smart heuristics for selecting the order of expansion of the nodes. Basic operations are effectively performed by symbolically representing the domain by means of binary decision diagrams [Bry86b]. Among the several possible search heuristics, we devised one implementing the following rules, implementing a high-level "greedy" approach towards maximizing the number of models satisfactory for the goal:

- if, in the DNF of the goal requirement, two facts belong exclusively to distinct disjuncts, and initially they are not both known, we disfavor actions that make them both true;
- if, in the DNF of the goal requirement, two facts belong to the same disjunct, we favor actions that make them both true;
- if the goal requirement contains implications, we disfavor actions that make the right side true, unless the left side is already true.

5.3.1 Implementation and results

In this section, we evaluate the performance of our on-the-fly belief-level search algorithm over various composition scenarios, uniformly running our experiments on a Linux equipped, 700 Mhz machine with 1 GByte of RAM. In our tests, we set a 512 MBytes limit for memory usage, and a cutoff at 3600 seconds of CPU time.

We are particularly interested in comparing with the state-of-the-art approach of [PTBM05, PTB05], on two structurally different categories of composition scenarios.

We first consider the composition of large sets of simple services; this can be seen as a representation of situations likely to be found in, e.g., grid services, where each service performs an atomic operation, and we need to compose several of them to achieve a complex goal. In particular, we consider the scenario described in [PTB05], where a set of services is available, each of them accepting a single request, and either signaling failure, or returning (upon receiving a further acknowledge message) a function of the input. A "not-acknowledge" message drives a component to a final failure state. The composition goal consists in computing a nested function, if possible, and in any case in leaving no service suspended.

The results for the tests are shown in Fig.5.18. We first report the model construction times only, and then report the overall performance. It is evident that the price of computing offline the belief-level system grows only linearly with the number of components, and paves the way to a fast symbolic search. On the contrary, the search time in our "on-line" approach grows fast, together with the number of branches of the solution subtree. Considering the overall performance, the on-line approach is convenient, thanks to the absence of belief-level construction, up to 6 web services, after which the off-line approach of [PTB05] is winning.

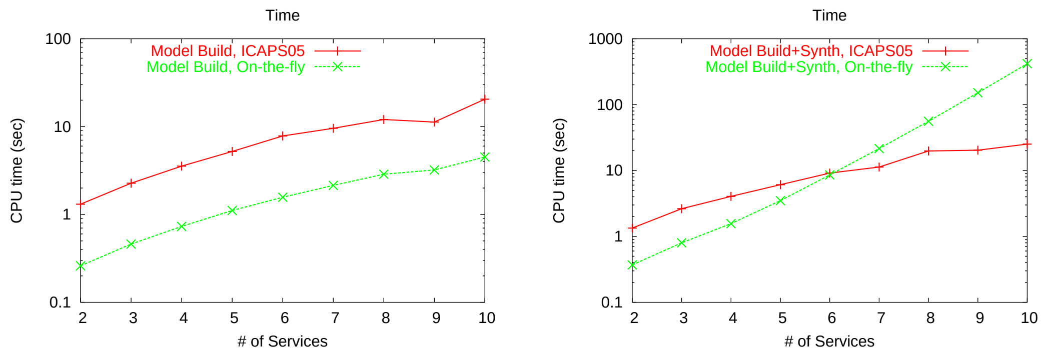


Figure 5.18: Tests for scalable services

We then consider the composition of a small set of rather large web services, each one implementing a complex protocol. This situation is typical of e.g. e-government scenarios, where the number of partners is often very small (e.g. 3), but each partner realizes a vast set of complex procedures. As an example of this kind of scenario, we start from the "producer and shipper" example of [PTB05], where each of the three involved entities describe a rather complex protocol, also reflecting in the complexity of the controller resulting from the composition. We then gradually introduce some variations to the protocols, so to consider a variety of different configurations, featuring different complexities in terms of states of the component web services (and thus of the associated belief-level system).

Fig. 5.19 shows the results, using the off-line approach of [PTB05], and our on-the-fly approach - again, we first present the model build time in isolation, and then show the overall performance. The size of the overall belief-level system is taken as a complexity measure for the tests.

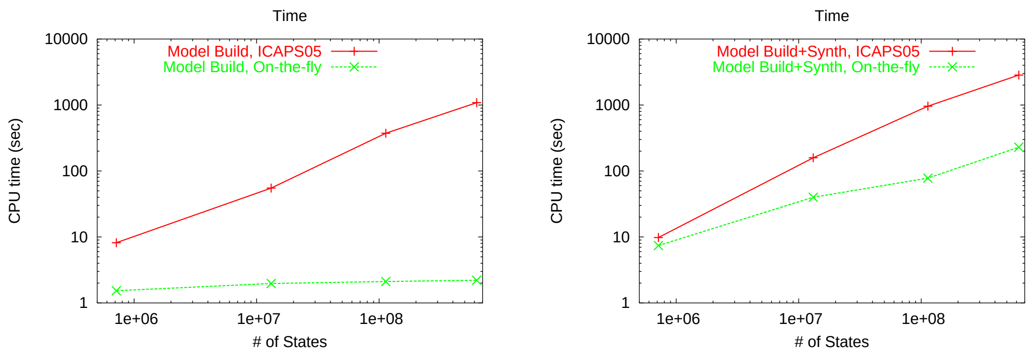


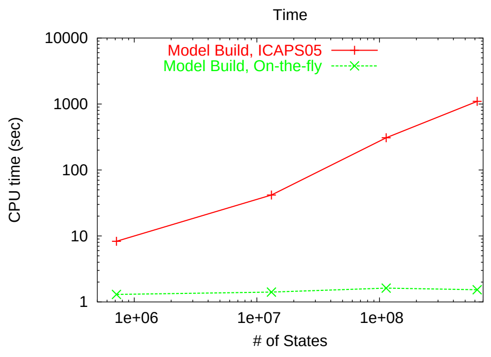
Figure 5.19: Tests for P&S, complex goal

While both approaches degrade on more complex configurations, they do so in a radically different way. In particular, the impact of constructing the ground-level planning domain is minor in our approach, while the size of the belief-level domain becomes huge and its explicit construction becomes soon unbearable for the off-line approach. The actual plan synthesis time of [PTB05] is also negatively influenced by the size of the belief domain, which - in spite of using symbolic techniques - makes it very hard to evolve large state frontiers; on the opposite, the performance of the on-the-fly approach basically depends on the size of the produced controller.

We then observe that the [PTB05] scenario, and the variants we considered so far, require that every state of each component service may have to be traversed while executing the composed service - i.e. every portion of every component is relevant to the requirement goal. In general, however, a requirement goal may involve only a partial usage of (some of) the component services. E.g. in an e-government scenario, a service may realize several different functionalities, of which only one is relevant to the goal. We represent this situation by considering a simpler goal requirement, where we intend just to establish the possibility to conduct an acquirement transaction, without then carrying it out.

The results for this test are given in Fig. 5.20. The differences are even more dramatic; the off-line approach pays, just as before, an unaffordable price to build a large belief-level domain, only to discover later that the produced plan is actually rather simple. Notice indeed that the overall performance coincides with the model building. The on-the-fly approach, instead, scales up rather nicely.

The data from our tests confirm our intuition: in the "offline" approach of [PTB05] the bottleneck is associated to the size of the component services, and the complexity of the goal only contributes in a limited way. On the the other side, the "on-the-fly" approach we envisage here only pays in proportion to the complexity of the produced controller. When component services are very complex, and/or the solution for the problem is not extremely complex, the "on-the-fly" technique exhibits a better performance. The "off-line" technique, thanks to its modular approach to belief-level construction, naturally lends to



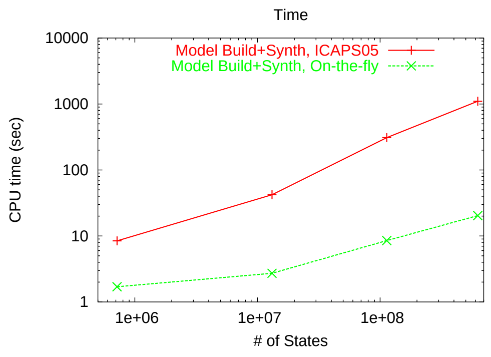


Figure 5.20: Tests for P&S, simple goal

scenarios involving a large number of simple services.

Chapter 6

The knowledge-level approach

The ground-level approaches presented so far suffer from a crucial issue, that is data-dependency.

In terms of performance, having to represent all possible behaviors of web services, considering every possible data value they will exchange, turns out to be a major bottleneck whenever such ranges are significantly large: the consequent combinatorial explosion cannot be tackled, in practice, even though symbolic representation techniques are adopted.

In terms of flexibility, data-dependency implies that, whenever a component service is modified by allowing it to manipulate/produce a new data value (or vice versa, by disallowing some data value), the web service composition, at least in principle, needs to be performed again. This is far from desirable, in those many cases where changing the data ranges does not affect the logics that need to be followed by the composed service, and as such, re-composition is actually unnecessary.

Both problems have been tackled, to some extent, by the ground-level approaches presented so far. In particular, we took the view that it is sufficient to consider, for any component service, just a restricted range of data to identify a "general pattern of behavior" that is then used for the composition; later on, we perform a suitable abstraction step that partitions out data commonalities in the composed web service and emits it, to the extent where this is possible, without mentioning data values.

While sensible, these solutions are only partially satisfactory, since for instance, the minimal data ranges sufficient for our abstraction should be identified formally rather than by an ad-hoc abstraction, and in some cases, emitted services are still data-dependent.

We now discuss a more principled solution to the issues mentioned above, consisting in devising a general abstraction mechanism to describe component services at a knowledge-level, abstracting away from actual data values. As a consequence, no data dependency will appear, in the search space and in the composed web service resulting from the search. Crucial to the applicability of the approach, the fact that such knowledge-level abstraction can be performed in a fully automated way - different from some existing, similar approaches taken in planning. In the following, we discuss the nature of our knowledge-level abstraction, and the way we realize a web composition tool based on it.

6.1 Problem statement

The key aspect for lifting the ground-level approach of [PTB05] to the knowledge level is the definition of an appropriate model for providing a knowledge level description of the component services. We now formally define such a model in terms of a suitable knowledge base (from now on KB). Then we show how to construct the STS corresponding to the planning domain by composing the knowledge bases of the component services.

Defi nition 38 (Knowledge Base)
A knowledge base KB is a set of propositions of the following form:

- $K_V(x)$ *where x is a variable with an abstract type;*
- $K(x = v)$ *where x is an enumerative variable and v is one of its possible values;*
- $K(x = y)$ *where x and y are two variables with the same type;*
- $K(x = f(y_1, \ldots, y_n))$ *where $x, y_1, \ldots, y_n$ are variables with an abstract type and f is a function compatible with the types of $x, y_1, \ldots, y_n$.*

With $K(p)$ we mean that we know that proposition p is true and with $K_V(x)$ that we know the value of the variable x. This definition of knowledge base is very simple; still, our experiments show that it is powerful enough to model web service composition problems.

We say that a knowledge base KB is *consistent* if it does not contain contradictory knowledge propositions such as $K(x = v)$ and $K(x = v')$, with $v \neq v'$. We say that KB is *closed under deduction* if it contains all the propositions that can be deduced from the propositions in KB; for instance a KB containing both $K_V(x)$ and $K(x = y)$ should contain also $K_V(y)$.

The knowledge base KB of a component service is obtained from the variables, functions and types of the service.

Example 39 *An example of* knowledge base *for the Shipper process is:*

```
KB={K(pc = waitAnswer),K_V(customer_size),K_V(customer_loc),
  K(offer_cost = costOf(customer_size, customer_loc)),
  K(offer_delay = delayOf(customer_size, customer_loc)),
  K_V(offer_cost), K_V(offer_delay)}.
```

In the following we describe when a transition can be executed in a knowledge base KB and how its execution affects KB.

We model a transition t as a triple (C, a, E) where $C = (c_1 \wedge \ldots \wedge c_n)$ are its conditions, a is its firing action and $E = (e_1;\ \ldots;\ e_n)$ are its effects. We start by defining the auxiliary *restriction* and *update* operations.

The *restriction* of a knowledge base KB with a condition C, denoted with $restrict(KB, C)$, is performed adding to KB the knowledge obtained from C and closing under deduction; for instance:

```
restrict({K(x = y)}, y = z) =
  {K(x = y), K(y = z), K(x = z)}.
```

The *update* of a knowledge base KB with an effect E, denoted with $update(KB, E)$, consists in performing the following steps: for each assignment in E, remove from KB the knowledge we had on the modified variable, add the knowledge derived from the assignment and close the KB under deduction. For instance:

```
update({K(x = y)}, z := x; x := w) =
  update({K(x = y), K(z = x), K(z = y)}, x := w) =
  {K(z = y), K(x = w)}.
```

We now give the definitions of *applicability* and *execution* which depend on a service transition t and on an action a_c performed by a peer interacting with the service; the firing action a in t and the peer action a_c correspond to the same action except that the former is instantiated on service variables, while the latter on variables of the peer. Consider for instance the following example where action `request(s, l)` performed by the peer corresponds to the Shipper `request(customer_size, customer_loc)`.

Example 40 *Let's consider the input transition* `t` *of the Shipper*

```
pc = getRequest
 -[INPUT request(customer_size, customer_loc)]->
pc := checkAvailable
```

and suppose that our current knowledge base is:

KB = { K(pc = getRequest), K_V(s), K_V(l) } .

where `s:Size` *and* `l:Location` *are additional variables. Action* `request(s, l)`*, corresponding to invoking the Shipper action using* `s` *and* `l` *as parameters, is applicable in* KB *since we know the values of* `s` *and* `l` *and the condition of transition* `t` *is obviously consistent with* KB.

The knowledge base obtained by executing on KB *transition* `t` *and action* `request(s, l)`*, is defined as follows:*

- *we restrict* KB *with the knowledge* `K(pc = getRequest)` *associated to the transition condition; in this specific case* KB *remains unchanged;*

- *then we update* `KB` *with the knowledge carried by the input action on the variables used as action parameters; this consists in removing from* `KB` *all the knowledge we had on* `customer_size` *and* `customer_loc`*, adding the new knowledge* `K(customer_size = s), K(customer_loc = l)` *and closing under deduction; we obtain:*

```
KB' = { K(pc = getRequest), K_V(s),
  K_V(l), K_V(customer_size), K_V(customer_loc),
  K(customer_size = s), K(customer_loc = l) } ;
```

- *finally we update the knowledge base with the effects, removing from* `KB'` *all the knowledge we had on the variables modified by the assignments, adding the new knowledge (in this case* `K(pc := checkAvailable)`*) and closing under deduction; we obtain:*

```
KB'' = { K(pc = checkAvailable), K_V(s),
  K_V(l), K_V(customer_size), K_V(customer_loc),
  K(customer_size = s), K(customer_loc = l) } .
```

Definition 41 (KL Applicability and Execution)
A transition $t{=}(C, a, E)$ and a corresponding action a_c are applicable *in KB, written AppltKB, a_c if:*

- $a_c{=}i(x_1, \ldots, x_n)$ *and* $a{=}i(y_1, \ldots, y_n)$*, where i is an input,* $K_V(x_1), \ldots, K_V(x_n) \in KB$ *and* $restrict(KB, C)$ *is consistent; or*

- $a_c{=}o(x_1, \ldots, x_n)$ *and* $a{=}o(y_1, \ldots, y_n)$*, where o is an output, and* $restrict(KB, C)$ *is consistent; or*

- $a_c{=}a{=}\tau$ *and* $restrict(KB, C)$ *is consistent.*

If AppltKB, a_c, then we denote with KB'=Exec(t, KB, a_c) the execution *on KB of t and a_c, defined as follows:*

- *if* $a_c{=}\tau$ *and* $t{=}(C, \tau, E)$ *then* $KB'{=}update(KB'', E)$*, where* $KB''{=}restrict(KB, C)$*;*

- *if* $a_c{=}i(x_1, \ldots, x_n)$*, where i is an input, and* $t{=}(C, i(y_1, \ldots, y_n), E)$ *then* $KB'{=}update(KB'', y_1{:=}x_1;\ \ldots;\ y_n{:=}x_n;\ E)$*, where* $KB''{=}restrict(KB, C)$*;*

- *if* $a_c{=}o(x_1, \ldots, x_n)$*, where o is an output, and* $t{=}(C, o(y_1, \ldots, y_n), E)$ *then* $KB'{=}update(KB'', x_1{:=}y_1;\ \ldots;\ x_n{:=}y_n;\ E)$*, where* $KB''{=}restrict(KB, C) \cup \{K_V(y_1), \ldots, K_V(y_n)\}$*.*

Notice that the execution of a transition increases the knowledge in KB, not only with the information in the effects, but also with those in the condition. Indeed, if the transition is executed, this means that the condition is known to hold.

The planning domain at the knowledge level is constructed from the knowledge-level models of each component service and from a knowledge-level representation of the composition goal. The latter defines which are the variables and the functions of the composite service, like, for instance, the cost of the offer to the user, or a special function that adds a mark up to the sum of the costs of the shipper and producer. We call these variables and functions, *goal variables* and *goal functions*.

Given a composition goal, we automatically generate its knowledge level representation that declares what the composite service must know and how goal variables and functions must be related with the variables and functions of the component services.

Example 42 *A possible (very simple) composition goal for our reference example is:*

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))
```

The goal declares that we want all the services to reach the situation where the order has been confirmed. Moreover it states that 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]*

To obtain the knowledge level goal, we flatten the functions introducing auxiliary variables until only basic propositions are left:

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
```

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 *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)
```

The knowledge-level representation of the composition goal defines therefore a further knowledge base, that we call the *knowledge base of the goal*.

The knowledge-level planning domain $\mathcal{D}$ is obtained by combining the knowledge bases of the component services and the knowledge base of the goal, by instantiating the input and output actions of the component services on goal variables, and by adding the private actions obtained by applying goal functions to goal variables.

Definition 43 (Knowledge-level Planning Domain)
The planning domain $\mathcal{D}$ for a composition problem is an STS $\langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{A}, \mathcal{R}, \mathcal{L} \rangle$ defined as follows:

- *the set of states $\mathcal{S}$ are all the possible KB defined on the set of typed variables $X = \bigcup_{i=0 \ldots n} X_i$ and on the set of typed functions $F = \bigcup_{i=0 \ldots n} F_i$, where $X_1, \ldots, X_n$ and $F_1, \ldots, F_n$ are the variables and functions of the component services, while X_0 and F_0 are those of the composition goal;*
- *$\mathcal{S}^0 \subseteq \mathcal{S}$ is the set of initial states corresponding to the initial knowledge bases $KB_0^1, \ldots, KB_0^n$, obtained from the initial assignments of the component services;*
- *$\mathcal{I}$ is the set of input actions $i(x_1, \ldots, x_n)$ such that $i(y_1, \ldots, y_n)$ is an input action in a transition of a component service and $x_1, \ldots, x_n$ are goal variables with the same type of service variables $y_1, \ldots, y_n$;*
- *$\mathcal{O}$ is the set of output actions $o(x_1, \ldots, x_n)$ such that $o(y_1, \ldots, y_n)$ is an output action in a transition of a component service and $x_1, \ldots, x_n$ are goal variables with the same type of service variables $y_1, \ldots, y_n$;*
- *$\mathcal{A}$ is the set of private actions $x_0 := f(x_1, \ldots, x_n)$ where f is a goal function and $x_0, x_1, \ldots, x_n$ are goal variables compatible with the type of f;*
- *$\mathcal{R}$ is the set of transitions $r = \langle s, a_c, s' \rangle$, with $s, s' \in \mathcal{S}$, such that:*
  - *if a_c is an input, output or τ action, then there exists a $t = (C, a, E)$ in the sets of transitions of the component services such that Applts, a_c and $s' =$Exec(t, s, a_c);*

   – *if a_c is a private action of the form $x_0{:=}f(x_1,\ldots,x_n)$, then $K_V(x_1),\ldots,K_V(x_n) \in s$ and $s'{=}update(s, x_0{:=}f(x_1,\ldots,x_n))$;*

- *$\mathcal{L}$ is the trivial function associating to each state the set of propositions that hold in that state.*

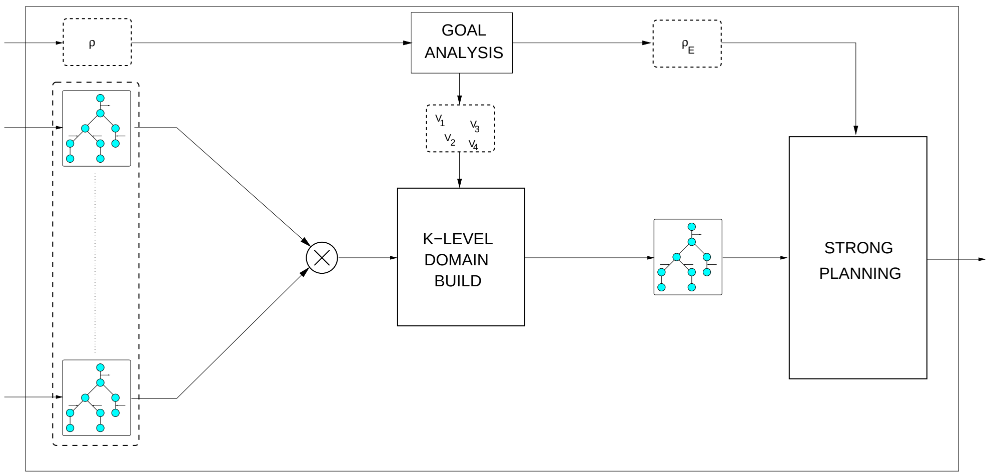


Figure 6.1: Domain construction and search for knowledge-level approach.

To sum up, in the knowledge-based approach, the general framework, and in particular the domain construction/search phase, is instantiated as shown in Fig. 6.1: the goal is analyzed to extract a set of variables which is then used to build a suitable knowledge-level database, which will be then subjected to the search.

Regarding the search, given the domain $\mathcal{D}$ described above, we can apply the approach presented in [PTB05] and obtain a deadlock free $\mathcal{D}_c$ that controls $\mathcal{D}$ by satisfying the composition requirements ρ. Despite of the fact that the synthesized controller $\mathcal{D}_c$ is modeled at the knowledge level, its elementary actions model communication with the component services (sending and receiving of messages) and manipulation of goal variables; given this, it is straightforward to obtain the executable BPEL4WS composite service from $\mathcal{D}_c$.

6.2 Implementation and results

We have implemented the proposed approach, and used the planning as symbolic model checking techniques presented in [PTB05] to perform an experimental evaluation. All the tests have run on a 3 GHz Xeon PC, limiting memory usage to 512MBytes, and with a CPU timeout of 1000 seconds.

We first considered the P&S example explained in the previous sections, which, in spite of the reduced number of components, requires a rather intricate protocol to be established for achieving the goal. Figure 6.2 shows the results of our experiments. The

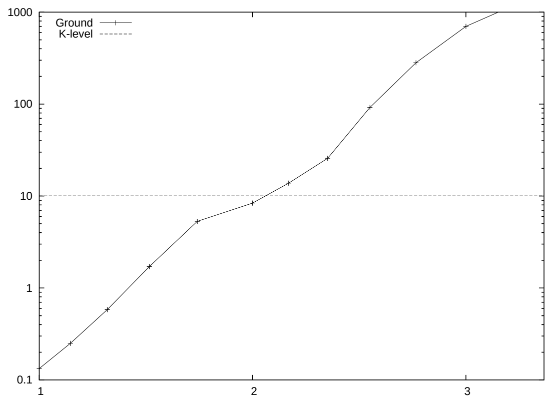


Figure 6.2: Experiments with P&S.

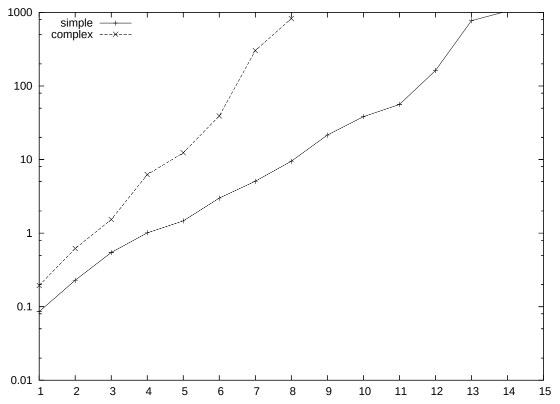


Figure 6.3: Experiments with parametrized domains.

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.

To evaluate the scalability of the knowledge-level approach when the number of component services grows, we perform two sets of experiments, considering a generalization of the example domain that involves a set of component services. In our first set of experiments, each component is represented by a very simple abstract BPEL4WS process that is requested to provide a service and can respond either by performing the service, or by refusing. The composition requirement is that either all services end successfully, or a failure is reported to the invoker of the composed service. Figure 6.3 reports the knowledge-level composition times, for increasing number of services to be composed (indicated on the horizontal axis). The composition achieves results comparable to those reported in [PTB05], where ground composition is performed only considering the case of types with range of cardinality 2. We manage to compose 14 services in 20 minutes.

In the second set of experiments, also reported in Figure 6.3, we make the protocol more complex, by requiring a higher degree of interleaving between components. Here, the interactions with each component are more complex than a single invoke-response step, and, to achieve the goal, it is necessary to carry out interactions with all components in an interleaved way. Such interleaving is common in the P&S example where, e.g., the P&S cannot confirm the order to the producer if shipping is not available or if the user does not accept the offer. The increased complexity reflects on the complexity of the composition; however, compositions of reasonable complexity (up to 8 services) can be achieved within 20 minutes. Even in this case, automated composition takes a rather low amount of time, surely faster than manual development of BPEL4WS code.

Chapter 7

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.

7.1 1st Year

In year one of the project, we developed the general idea of looking at the web service composition problem as a problem of planning, thinking of components services as (nondeterministic, asynchronous, partially observable) STSs. Our survey of existing techniques led to the design of a general planning framework and to some preliminary, hand-crafted experiments in service composition via planning. This paved the way to start the design of a first automated approach to the problem.

7.2 2nd Year

In year two of the project, we completed the design and implementation of our "ground-level" approach to web service composition, which we presented at AIMSA [PBB$^+$04]. Our experimental analysis highlighted certain scalability issues which led to start an activity on optimizing the encoding of the problem and the search techniques.

At the same time, it became clear enough that ground-level techniques based on an encoding where reachable beliefs are pre-computed first would have their scalability, and applicability, limited when significant ranges of data are involved in the composition.

7.3 3rd year

In third year, our activity on ground-level composition led to a more mature solution technique of the problem, largely improving its performance, as witnessed in [PTB05].

At the same time, to solve the inherent issues in ground-level composition, we proceeded by encoding the problem at a higher level of abstraction, speaking of the "knowledge over the values of variables", rather than of the concrete values themselves. In this way, the complexity of the composition, and the applicability of the obtained solution, becomes independent from the way data ranges are instantiated. The result of this effort is the knowledge-level approach presented at IJCAI([PMBT05]).

7.4 4th Year

The experimental analysis we performed in year three, over a variety of scenarios, revealed that, for some interesting cases, none of the techniques proposed so far was fully satisfactory, mostly due to an excessive overhead in the encoding phases.

For this reason, early in year four, we designed and implemented an alternated ground-level approach where pre-compilation of reachable beliefs is avoided in favor of an on-the-fly search (at the belief level). The approach was presented at ICAPS [PMP06] and shown to be favorable for certain configurations of the services to be composed.

Indeed, all of the solutions proposed above work extremely well under certain provisos, and suffer from scalability issues when such provisos are not met. In particular, the first ground-level approaches we designed ([PBB$^+$04, PTB05]), thanks to the modular encoding of services, scale up well with the number of components to be composed, but are extremely sensible to their size, and to the data ranges exchanged by them. The problem is solved only in part by the ground-level approach of [PMP06], which proves effective when composing restricted sets of large-sized components. However, sensitivity to the data ranges remains and, due to the different nature of the search space, performance degrades when large sets of (even simple) components are considered. The knowledge-level approach of [PMBT05] solves completely, and in a principled way, the problem of sensitivity to data ranges. However, the modular nature of encoding is lost, and therefore, the first, encoding stage of the solution, can be a bottleneck, especially if large sets of services are considered.

While the results presented here are extremely encouraging, and provide an array of techniques covering a wide range of composition problems, they also push us to pursue a more general, unifying technique where the specific scalability issues discussed above are all sorted out. This has been the main objective of our work during year four of the project, and has led to desing and realize a different way of expressing requirements, coupled with an effective, modular, and data-independent way of recasting the composition problem as one of planning. The results of such research are the focus of Deliverable D4.4.

Bibliography

[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.

[BCR01] P. Bertoli, A. Cimatti, and M. Roveri. Conditional Planning under Partial Observability as Heuristic-Symbolic Search in Belief Space. In *Proceedings of the Sixth European Conference on Planning (ECP'01)*, pages 379–384, 2001.

[BCRT01] P. Bertoli, A. Cimatti, M. Roveri, and P. Traverso. Planning in Nondeterministic Domains under Partial Observability via Symbolic Model Checking. In *Proc. IJCAI'01*, 2001.

[BDG03] J. Blythe, E. Deelman, and Y. Gil. Planning for workflow construction and maintenance on the grid. In *Proc. of ICAPS'03 Workshop on Planning for Web Services*, 2003.

[Bry86a] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[Bry86b] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[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.

[CPRT03] A. Cimatti, M. Pistore, M. Roveri, and P. Traverso. Weak, Strong, and Strong Cyclic Planning via Symbolic Model Checking. *Artificial Intelligence*, 147(1-2):35–84, 2003.

[Der98] D. Mc Dermott. The Planning Domain Definition Language Manual. Technical Report 1165, Yale Computer Science University, 1998. CVC Report 98-003.

[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.

[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.

[MF02] S. McIlraith and R. Fadel. Planning with Complex Actions. In *Proc. NMR'02*, 2002.

[MS02] S. McIlraith and S. Son. Adapting Golog for composition of semantic web Services. In *Proc. KR'02*, 2002.

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

[PBB$^+$04] M. Pistore, P. Bertoli, F. Barbon, D. Shaparau, and P. Traverso. Planning and Monitoring Web Service Composition. In *Proc. AIMSA'04*, 2004.

[PMBT05] M. Pistore, A. Marconi, P. Bertoli, and P. Traverso. Automated Composition of Web Services by Planning at the Knowledge Level. In *Proc. of IJCAI'05*, 2005.

[PMP06] P.Bertoli, M.Pistore, and P.Traverso. Automated Web Service Composition by On-The-Fly Belief Level Search. In *Proceedings of ICAPS'06*, 2006.

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

[PTBM05] M. Pistore, P. Traverso, P. Bertoli, and A. Marconi. Automated Synthesis of Executable Web Service Compositions from BPEL4WS Processes. Poster at 14th International World Wide Web Conference (WWW05), 2005.

[SdF03] M. Sheshagiri, M. desJardins, and T. Finin. A Planner for Composing Services Described in DAML-S. In *Proc. AAMAS'03*, 2003.

[WPS$^+$03] D. Wu, B. Parsia, E. Sirin, J. Hendler, and D. Nau. Automating DAML-S Web Services Composition using SHOP2. In *Proc. ISWC'03*, 2003.