Specification of Web Service Composition Requirements

ITC-irst
Università di Trento

Abstract. Automated Composition of Web Services aims at generating a new executable process that satisfies a given composition goal interacting with existing services published on the Web. Several approaches have been proposed to tackle this problem, but little effort has been devoted to the problem of modeling the composition requirements. The goal of this research work is to define a new modeling language which allows to specify composition requirements for realistic composition problems in an effective, user-friendly and easy refining way.

Document Identifier	Deliverable 4.4
Project	MIUR-FIRB project RBNE0195K5 "Knowledge Level Automated Software Engineering"
Version	v1.0
Date	Oct 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

The main idea underneath Web Service composition is to combine existing services published on the Web in order to achieve a given business goal. The manual development of the new composite service is a difficult and error-prone task, because human domain experts have to take care about all possible situations during the service execution process. The ability to automatically compose web services is an essential step to substantially decrease time and costs in the development, integration, and maintenance of complex services.

Given a set of available services and a composition requirements, the problem of automated composition is concerned with synthesizing a new composite service that satisfy the composition goal, by suitably coordinating the available services. It is widely recognized that solving this problem in practice is far from trivial: the component services are stateful processes whose interactions are intrinsically asynchronous and each of them exhibits a complex, nondeterministic and partial observable behavior. Several methods have been proposed to cope with the automated composition problem. In particular, most researches conducted fall in the realm of AI planning (see e.g. [MS02, MF02, NM02, PTB05, HBCS03, BCG$^+$03]): component services are used to obtain the planning domain, composition requirements are formalized as planning goals, and planning algorithms are used to synthesize plans that compose the published services.

An important aspect that hasn't been investigated so far is how to derive and model the composition requirements. In order to cope with a wide range of real composition problems we need a way to express requirements of different strength, preferences among different goals, complex conditions on the exchanged data and on the execution states of the component services. Moreover, to make the automated composition an effective and practical task, the requirements specification should be as user-friendly and easy refining as possible.

The aim of this document is to describe in details the approaches we are proposing to the specification of both control-flow and data-flow composition requirements (see also [MPT06b, PMTB05]) and show how they can be integrated within a state-of-the-art automated composition framework proposed in [PTB05, PTBM05]. Moreover, we evaluate the proposed approaches on realistic composition scenarios from a technical point of view, from the point of view of the performance, and for what concerns the usability.

Contents

1 Introduction **1**

2 A Web Service Composition Domain **4**

3 Specifying Composition Requirements **8**

3.1 Control Flow Requirements 8

3.1.1 CF Requirements as Reachability Goal 8

3.1.2 CF Requirements as EAGLE Goal 10

3.1.3 CF Requirements as Preference Goal 11

3.2 Data Flow Requirements 13

3.2.1 Implicit Data Flow Requirements: Knowledge Base 13

3.2.2 Explicit Data Flow Requirements: Data Net 14

4 Integration within an Automated Composition Framework **21**

4.1 An Automated Composition Framework 21

4.2 Integrating Implicit Data Flow Requirements 23

4.3 Integrating Explicit Data Flow Requirements 25

4.3.1 Data Requirements as STSs 26

4.3.2 Generating the Composite Process 28

5 Evaluation of the Proposed Approaches **31**

5.1 Experimental Evaluation 31

5.2 Usability 34

6 Conclusions and Related Works **35**

7 History of the Deliverable **37**

7.1 2nd year 37

7.2 3rd year 37

7.3 4th year 38

Chapter 1

Introduction

Web Services are platform-independent applications that export a description of their functionalities and make it available using standard network technologies. Services are able to perform a wide spectrum of activities, from simple receive-response protocols to complicated business workflows. Therefore, Web Services provide the basis for the development and execution of business processes that are distributed over the network and available via standard interfaces and protocols.

Service composition is one of the fundamental ideas underlying service-oriented applications: composed services perform new functionalities by interacting with component services that are available on the Web. In most real-world applications, service compositions must be at the "process-level", i.e., they must take into account the behaviors of component services that are not atomic and cannot be executed in a single request-response step. Component services are instead stateful processes, and they require to follow an interaction protocol which may involve different sequential, conditional, and iterative steps [HBCS03]. The automated synthesis of composed services is one of the key tasks that supports the design and development of service oriented applications: given a set of available component services and a composition goal, the task corresponds to the synthesis of a composition, e.g., a new service, or a set of constraints on the behaviors of existing services, that satisfies the requirements expressed by the composition goal.

Recent works address the problem of the automated synthesis of composed services at the process level, see, e.g., [HBCS03, BCG$^+$03, BCGM05, BP05, TPC$^+$05, PTBM05, PTB05, PMTB05]. However, most of them do not take into account a key aspect of the composition problem: how to derive and model the composition requirements. This is a significant and challenging problem, since, in real life scenarios, business analysts and developers need a way to express complex requirements both on the exchanged data (*data-flow requirements*) and on the execution states of the component services (*control-flow requirements*). Moreover, to make the automated composition an effective and practical task, the requirements specification should be easy to write and to understand for the analyst. Surprisingly, very little effort has been devoted in the literature to address this problem.

We consider component services that are described in WS-BPEL [ACD+03], the well known standard language for describing services at the process-level. We present and discuss an approach to the specification of control-flow requirements and two different approaches to the specification of data-flow requirements for the automated synthesis of composed services. *Implicit data-flow requirements*, see [PMTB05], exploit the functions that define the tasks carried out by the component services, and that annotate their WS-BPEL descriptions. Composition goals contain references to such functions that implicitly define constraints on data flows. *Explicit data-flow requirements*, instead, define composition goals that contain constraints that explicitly define the valid routings and manipulations on the messages of the component services. These constraints can be described in a graphical way, e.g., as data-nets [MPT06b].

Consider, for instance, a virtual travel agency that composes two component services, a flight and a hotel reservation service. The main goal of the virtual travel agency is to sell holiday packages (hotel and flight) to potential customers. However, it can be the case that there are no available flights (or hotels) satisfying the customer request, or that the customer refuses the offer or cancels the booking. In all these cases, we want to be sure that there are no 'pending' (flight, hotel or customer) bookings. Therefore, when modeling control-flow composition requirements we need a way of specifying preferences among sub-goals and recovery conditions.

For what concerns data-flow requirements (i.e. requirements on data that are exchanged among component services), an implicit data-flow requirement can state that the price of the offer to the customer is a specific function of the costs of the hotel and of the flight. Two cost functions annotate the abstract WS-BPEL of the flight and hotel components. This requirement implicitly specifies a data flow from the (flight and hotel) messages of the components to the composed service, the agency service. The composition task should determine from this specification which messages and which data should be sent from a service to another. A corresponding explicit data flow requirement directly specifies instead that a specific hotel service output message containing the data about the cost of the hotel, as well as the flight output message with the data for the cost of the flight should be the input message for the composed virtual travel agency, which should be manipulated and sent to the customer.

This deliverable provides an overview of the approaches, shows how they can be used to specify composition requirements within a state-of-the-art automated composition framework [PTB05, PTBM05, PMTB05], and finally compares them w.r.t. their efficiency and usability (see also [MPT06a]). We experiment with the proposed approaches on some composition problems taken from a realistic domain in the field of e-commerce involving electronic purchases and bank payments. From the one side, explicit data-flow requirements allow for a better performance and a much higher scalability, due to the very modular encoding of requirements. Each explicit requirement can be locally transformed into a (typically) small automaton that constraints the search for a composition.

Another advantage of explicit requirements is that they are easy to formulate and understand, e.g., through a very intuitive graphical representation. From the other side, the advantages of implicit requirements are a consequence of the use of annotations in the process-level descriptions of component services. Annotations provide a way to give semantics to WS-BPEL descriptions of component services, and they are intrinsically more abstract and general than explicit data-flows, thus allowing for a higher degree of re-use. Finally, they allow for a clear separation of the components annotations from the composition goal, and thus for a a clear separation of the task of the designers of the components from that of the designer of the composition, a separation that can be important in, e.g., cross-organizational application domains.

The rest of the deliverable is organized as follows. In Chapter 2 we provide some examples of composition requirements in an explanatory example that we will use all along the deliverable. In Chapter 3 we describe in details the approaches we are proposing to the specification of both control-flow and data-flow composition requirements and in Chapter 4 we show how they can be integrated within a state-of-the-art automated composition framework proposed in [PTB05, PTBM05, PMTB05]. In Chapter 5 we evaluate the proposed approaches from a technical point of view, from the point of view of the performance, and for what concerns the usability. Finally, in Chapter 6 we discuss related work and final remarks and in Chapter 7 we recap the history of this deliverable.

Chapter 2

A Web Service Composition Domain

By automated composition of Web services we mean the generation of a new composite service that interacts with a set of existing component services in order to satisfy given composition requirements.

In this section we illustrate on a case study the problem of the automated composition of Web services. We will focus in particular on the problem of specifying the requirements of such a composition.

Example 1 (Virtual Travel Agency)**.** *Our reference example consists in providing a virtual travel agency service, say the VTA service, which offers holiday packages to potential customers, by combining three separate existing services: a flight booking service Flight, a hotel booking service Hotel, and a service that provides maps AllMaps. The idea is that of combining these three services so that the customer may directly interact with the composed service VTA to organize and possibly book his holiday package.*

When addressing a Web service composition problem, the (manual or automatic) development of the new composite Web service must be driven by the analysis of published process specifications of component services (i.e. the Hotel, Flight and AllMaps services in the VTA example) and by requirements and constraints the composite service has to satisfy.

We assume that component services are described as BPEL4WS processes.BPEL4WS (Business Process Execution Language for Web Services) [ACD+03] is an industrial language for the specification and execution of business processes made available thorough Web services. BPEL4WS provides an operational description of the (stateful) behavior of web services on top of the service interfaces defined in their WSDL specifications. A BPEL4WS description identifies the partners of a service, its internal variables (together with their type specification), and the operations that are triggered upon the invocation of the service by some of the partners. Operations include assigning variables (possibly using internal functions), invoking other services and receiving responses, forking parallel threads of execution, and nondeterministically picking one amongst different courses of actions. Standard imperative constructs such as if-then-else, case choices, and loops, are also supported.

Example 2 (VTA Component Protocols)**.** *In the following, we describe informally the three available services, whose interaction protocols are depicted in Figure 2.1*[1]*. Hotel accepts requests for providing information on available hotels for a given date and a given location. If there are hotels available, it chooses a particular hotel and return an offer with a cost and other hotel information. This offer can be accepted or refused by the external service that has invoked the Hotel. In case of acceptance, the Hotel proceeds with the booking and sends a confirmation message to the client. The Flight service receives requests for booking flights for a given date and location. If there are available flights, it sends an offer with a cost and a flight schedule. The client can either accept or refuse the offer. If he decides to accept, the Flight will book the flight and provide additional information such as an electronic ticket. The AllMaps service receives requests with two locations and provides a digital map depicting distance information.*

Intuitively, the VTA service should try to satisfy a given customer request by providing information on available flights and hotels (e.g., holiday cost, flight schedule, hotel description and a map showing distance from the airport) and book the holiday according to customer final decision. Figure 2.2 describes a possible protocol that the VTA could expose to the customer. According to it, the customer sends a request for an holiday, then, if there is an available flight, it receives a flight offer. If the customer agrees on the flight schedule and cost and there is an available hotel, he receives an hotel offer consisting of the hotel cost, the distance of the hotel from the airport and other information about the hotel. The customer can either decide to accept the offer or to terminate the interaction with the VTA. If he decides to accept, he receives the booking confirmation with the overall holiday cost and other information about the chosen hotel and flight.

Given the description of the component services and of the customer interface, the next step towards the definition of the automated composition domain is the formal specification of the composition requirements. As we will see from the examples presented in the rest of this chapter, even for simple case studies we need a way to express requirements that define complex conditions, both for what concerns the control flow and for the data exchanged among the component services.

Example 3 (Control-flow requirements)**.** *The VTA service main goal is to "sell holiday packages". This means we want the VTA to reach a situation where the customer has accepted the offer and a flight and a hotel have been booked. However, it may be the case that there are no available flights (or no available hotels) satisfying the customer request, or that the customer doesn't like the flight or the hotel offer and thus cancels the booking. We cannot avoid these situations, therefore we cannot ask the composite service to guarantee this requirement. In case this requirement cannot be satisfied, we do not want*

[1]In the figure, labels of input transitions start with a "?", labels of output transitions start with a "!", other transitions correspond to internal operations performed by the services.

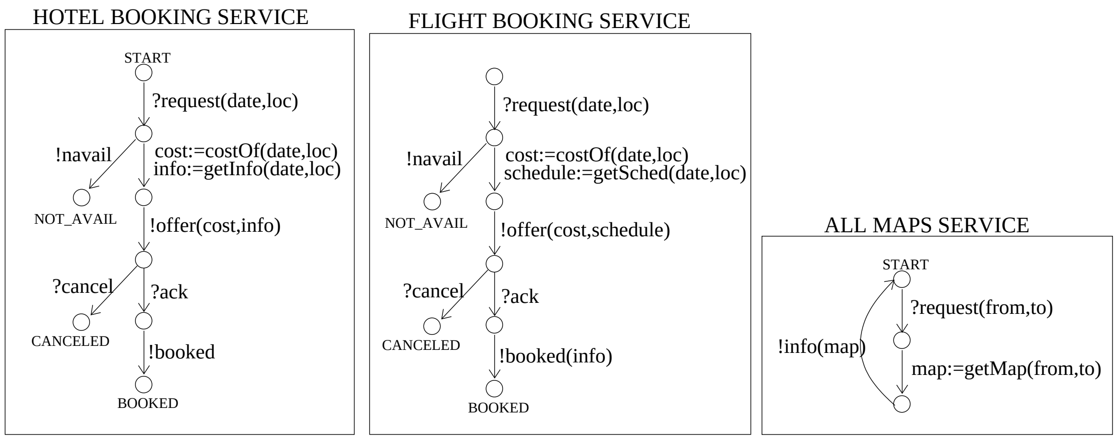


Figure 2.1: The Virtual Travel Agency Component Services

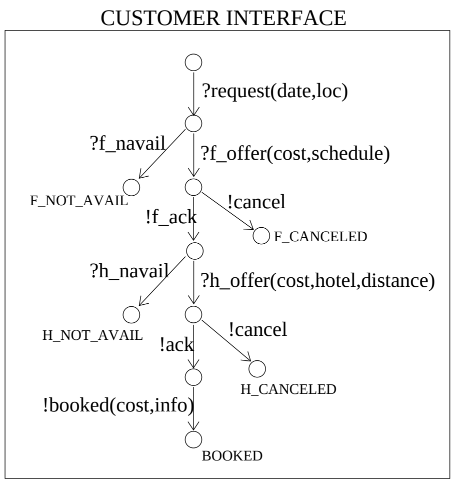


Figure 2.2: The VTA Customer Interface

the VTA to book a flight (nor a hotel) without being sure that our customer accepted the offer, as well as we do not want displeased customers that have booked holidays for which there are no available flights or hotels. Our control flow requirements would therefore be something like:

> *try to "sell holiday packages";*
> *upon failure,*
> *do "never a single commitment".*

Notice that the secondary requirement ("never a single commit") has a different strength w.r.t. the primary one ("sell holiday packages"). We write "do" satisfy, rather than "try" to satisfy. Indeed, in the case the primary requirement is not satisfied, we want the secondary requirement to be guaranteed.

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

Example 4 (Data-flow requirements)**.** *In order to provide consistent information, the VTA service needs to exchange data with the components and its customer in an appropriate way. For instance, when invoking the Flight service, the information about the location and date of the flight must be the same ones that the VTA received in the customer request; similarly, the information sent to the customer about the distance between the proposed hotel and the airport must be those obtained from the interaction with the AllMaps service; and such a service must receive the information on the airport and hotel location according to the offer proposed by the Flight and Hotel service. In particular, the VTA must obtain the airport location from the flight schedule offered by the Flight service and must obtain the hotel location from the information received in the Hotel offer. Moreover, the cost proposed to the customer for the holiday package must be the sum of the hotel and flight cost plus some additional fee for the travel agency service; thus the cost offered to the customer must be computed by means of a function internal to the VTA service. And so on.*

The example shows that, even for apparently simple composition problems, we need a way to express complex data flow requirements: from simple data links between incoming and outgoing message parts (e.g., forwarding the information received by the customer about the location to the Flight service) to the specification of complex data manipulation (e.g., when computing the holiday package cost or obtaining the airport and hotel locations).

The research challenge we are addressing is the definition of formal modeling languages that allow to specify composition requirements for realistic composition problems in an effective, user friendly and easy refining way. To make this languages usable in practice we integrated them within the automated composition framework described in [PTB05, PTBM05] that, given the descriptions of the component processes and the composition requirements, automatically generates a new BPEL4WS process implementing the required composition.

Chapter 3

Specifying Composition Requirements

3.1 Control Flow Requirements

As shown in Example 3, in the control flow requirements we need to express conditions on the termination of the component services. In particular, we must specify that, in case all services are available and the final user accepts, they should all terminate in a 'successful' state, i.e. a state where the final agreement to book or sell has been achieved with the VTA. Otherwise, each service must either remain inactive, or terminate in a 'failure' state where the service is aware of the impossibility to agree on the book/sell and any commitment to buy or sell has been withdrawn.

This kind of requirements can be expressed in several ways. In this Section we present a well-known approach, which consists in expressing control flow requirements as reachability goal, and two approaches where control flow requirements are encoded with languages that allow the specification of recovery conditions and preferences among subgoals.

All these approaches exploits the semantic annotations in the description of the service protocols that specify whether a state is a successful one (e.g. C.BOOKED, H.BOOKED in Figure 2.1) or a failing one (e.g. H.NOT_AVAIL, H.CANCELED, C.F_CANCELED in Figure 2.1).

3.1.1 CF Requirements as Reachability Goal

Encoding the control flow requirements of Example 3 as a reachability goal, means being able to formalize that:

"If all the services are available and the client accepts"
then all the services must be in a successful state,
otherwise none of them must be in a successful state."

To express this kind of requirements we exploit the semantic annotations in the description of the component services that specify whether a termination state is failing or successful and we need additional semantic annotations specifying whether in that state the service is available (the client has accepted) or not. Figure 3.1 presents the abstract protocols of the component services enriched with this kind of information[1].

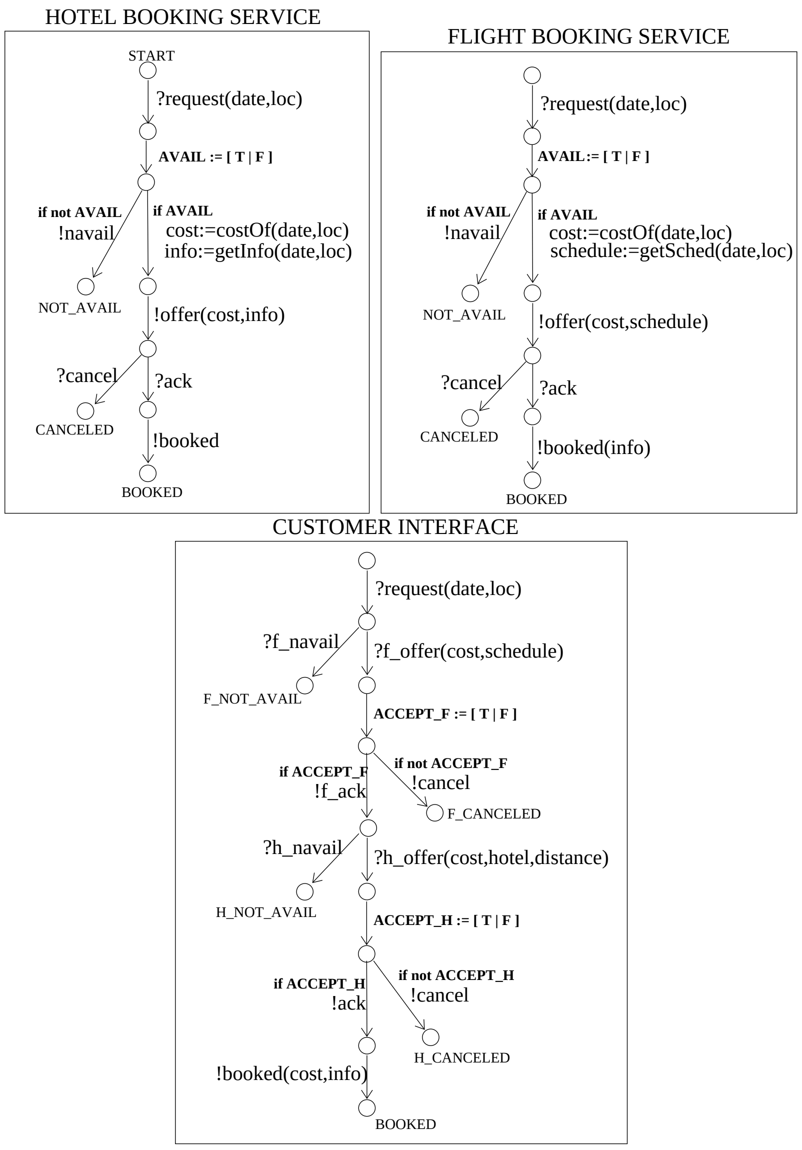


Figure 3.1: Availability/Acceptance semantic annotations in the Component Services

[1]We do not need this kind of semantic annotations for the AllMaps service since its protocol doesn't present any possibility of failure.

Example 5. *The formalization of the requirement in Example 3 as a reachability goal is the following.*

$$((\mathit{H.AVAIL} \wedge \mathit{F.AVAIL} \wedge \mathit{C.ACCEPT\_F} \wedge \mathit{C.ACCEPT\_H}) \rightarrow$$
$$(\mathit{C.BOOKED} \wedge \mathit{F.BOOKED} \wedge \mathit{H.BOOKED} \wedge \mathit{M.START}))$$
$$\wedge$$
$$(\neg(\mathit{H.AVAIL} \wedge \mathit{F.AVAIL} \wedge \mathit{C.ACCEPT\_F} \wedge \mathit{C.ACCEPT\_H}) \rightarrow$$
$$((\mathit{C.F\_NOT\_AVAIL} \vee \mathit{C.F\_CANCELED} \vee \mathit{C.H\_NOT\_AVAIL} \vee \mathit{C.H\_CANCELED})$$
$$\wedge(\mathit{H.NOT\_AVAIL} \vee \mathit{H.CANCELED} \vee \mathit{H.START})$$
$$\wedge(\mathit{F.NOT\_AVAIL} \vee \mathit{F.CANCELED})$$
$$\wedge \mathit{M.START}))$$

3.1.2 CF Requirements as EAGLE Goal

When considering composition control flow requirements, as those of Example 3, it arise the necessity of expressing conditions of different strengths (like "try" and "do"), and preferences among different (e.g., primary and secondary) requirements. Thus, we decide to propose an approach where these requirements are encoded using a formal language designed with this specific purpose.

EAGLE is a requirement language whose operators are similar to CTL operators, but their semantics, formally defined in [DLPT02], take into account the notion of preference and the handling of failure when subgoals cannot be achieved.

Let propositional formulas $p \in \mathcal{P}rop$ define conditions on the states of a given service. In EAGLE, a *composition requirement* ρ over $\mathcal{P}rop$ is defined as follows:

$$\rho := p \mid \rho \textbf{ And } \rho \mid \rho \textbf{ Then } \rho \mid \rho \textbf{ Fail } \rho \mid \textbf{Repeat } \rho \mid$$
$$\textbf{DoReach } p \mid \textbf{TryReach } p \mid \textbf{DoMaint } p \mid \textbf{TryMaint } p.$$

DoReach p specifies that condition p has to be eventually reached in a mandatory way, for all possible evolutions of the service. Similarly, **DoMaint** q specifies that property q must mandatorily be maintained true. **TryReach** p and **TryMaint** q are weaker versions of the previous requirements, where the composite service is required to do "everything that is possible" to achieve condition p or maintain condition q, but failure is accepted if unavoidable. Construct ρ_1 **Fail** ρ_2 is used to model preferences among requirements and recovery from failure. More precisely, requirement ρ_1 is considered first. Only if the achievement or maintenance of this requirement fails, then ρ_2 is used as a recovery or second-choice requirement. Consider for instance the requirement "**TryReach** c **Fail DoReach** d". **TryReach** c requires a service that tries to reach condition c. During the execution of the service, a state may be reached from which it is not possible to reach c. When such a state is reached, the requirement **TryReach** c fails and the recovery condition **DoReach** d is considered. Constructs **Then** , **Repeat** and **And** are used to model sequencing, infinite iteration, and conjunction of goals respectively.

A detailed definition and a formal semantics for the EAGLE language can be found in [DLPT02]. Here we just explain how EAGLE can express the control flow composition requirements of the VTA composition scenario presented in Example 3.

Example 6. *The* EAGLE *formalization of the requirement in Example 3 is the following.*

$$
\begin{array}{l}
\textbf{TryReach} \\
\quad \mathit{C.BOOKED} \wedge \mathit{F.BOOKED} \wedge \mathit{H.BOOKED} \wedge \mathit{M.START} \\
\textbf{Fail DoReach} \\
\quad (\mathit{C.F\_NOT\_AVAIL} \vee \mathit{C.F\_CANCELED} \vee \\
\qquad \mathit{C.H\_NOT\_AVAIL} \vee \mathit{C.H\_CANCELED}) \wedge \\
\quad (\mathit{H.NOT\_AVAIL} \vee \mathit{H.CANCELED} \vee \mathit{H.START}) \wedge \\
\quad (\mathit{F.NOT\_AVAIL} \vee \mathit{F.CANCELED}) \wedge \\
\quad (\mathit{M.START})
\end{array}
$$

*The goal is of the form "**TryReach** c **Fail DoReach** d". **TryReach** c requires a service that tries to reach condition c. In our case the condition c is to* "sell holiday packages", *since it requires both the Customer, the Hotel and the Flight service to be in a state where the booking has been confirmed (see Figure 2.1 and Figure 2.2).*

*During the execution of the service, a state may be reached from which it is not possible to reach c, e.g., since the product is not available. When such a state is reached, the requirement **TryReach** c fails and the recovery condition **DoReach** d, is considered. In our case the condition d requires all the component services to be in a state where the booking has (for some reason) failed, and thus to be in a state where* "no single commitments" *have been done.*

If we compare the reachability goal of Example 5 with the EAGLE formalization of Example 6 it is clear that the latter is preferable for several reasons: first of all it is much more compact, moreover it requires to reason only on the successful/failing nature of the final states, and last but not least the formal encoding perfectly reflects the high level specification of the control flow requirements (see Example 3).

3.1.3 CF Requirements as Preference Goal

In this Section we propose a simple core goal language that allows to express qualitative preferences among goals.

In particular, we allow for an explicit, conditional, and qualitative model of goal preferences. The model is *explicit*, since it is possible to specify explicitly that one goal is better than another one. It is *conditional* since it is possible to express that if some condition holds on the domain, then we prefer some goal, we prefer a different goal, otherwise. It is *qualitative*, since preferences are interpreted as a partial order over goals.

We propose to encode the goal as a hierarchical structure where any goal can have an unbounded set of nested goals. The hierarchy is organized by the combination of two operators **OneOf** and **And**. The operator **And** is used to conjunct subgoals when all of them have to be satisfied. **OneOf** operator is used for the goal composition when only one of them has to be satisfied. The motivation of **OneOf** construction is the fact that in many application domains the strong plans do not exist for the most preferable goals. The goal designer need a possibility to specify a secondary (less preferable, compensation or recovery) goals to manage the situation when the focused goal became unreachable as result of the non-deterministic action outcome. In order to satisfy such kind of goal a plan has to guarantee that at least one of listed subgoals will be achieved in spite of non-determinism. Moreover, the plan has to do its best to reach as more preferable goal as possible. **OneOf** assigns to each its subgoal a natural number which expresses the user preferences to the goal in respect to others. The usage of natural number for user preferences description enable to set up a total ordering relation not only between goals from the same **OneOf** operator, but also between combination of goals from different hierarchy levels.

We propose the formal definition of the goal language that extends classical reachability goals by adding preferences and compositional operators as follows.

Definition 1 (Goal Language with Preferences). *Let $\mathcal{P}$ be the set of basic propositions. The propositional formulas p and the* preference goals *$g \in \mathcal{G}$ over $\mathcal{P}$ are defined as follows:*

1. $p \ := \ \top \mid \bot \mid b \mid \neg p \mid p \wedge p \mid p \vee p \mid$
2. $g \ := \ p \mid$ **OneOf**$(g_1, c_1; ...; g_n, c_n) \mid$ **And**$(g_1, ..., g_n)$

where $c \in \mathbb{N}$ is a natural number that describes a priority and $b \in \mathcal{P}$ is a basic proposition.

Example 7. *The goal with preferences for the control flow requirements of Example 3 is the following:*

$$\textbf{\textit{OneOf}}(g_1, 1; g_2, 0)$$

where:

$$\begin{aligned} g_1 = &\quad \textit{C.BOOKED} \wedge \textit{F.BOOKED} \wedge \textit{H.BOOKED} \wedge \textit{M.START} \\ g_2 = &\quad (\textit{C.F\_NOT\_AVAIL} \vee \textit{C.F\_CANCELED} \vee \textit{C.H\_NOT\_AVAIL} \vee \textit{C.H\_CANCELED}) \wedge \\ &\quad (\textit{H.NOT\_AVAIL} \vee \textit{H.CANCELED} \vee \textit{H.START}) \wedge \\ &\quad (\textit{F.NOT\_AVAIL} \vee \textit{F.CANCELED}) \wedge \\ &\quad \textit{M.START} \end{aligned}$$

The intuition of this goal is that the VTA has to reach a situation where either all the component services are in a 'successful' state, goal g_1, or they are all in a 'failing' state, goal g_2, but g_1 is a more preferable goal and the goal g_2 is considered only if the goal g_1 becomes unsatisfiable.

In [SPT06] we describe a correct and complete planning algorithm, for planning with goal preferences, that is guaranteed to find optimal solutions and show the practical potentialities of this approach through a preliminary experimental evaluation.

3.2 Data Flow Requirements

In this section we show how to express data-flow requirements, so that they can be integrated in a general framework for the automated composition of Web services. In particular, we present two different approaches, implicit and explicit, to the specification of data-flow requirements. The former exploits the knowledge level model for web services presented in [PMTB05], while the latter uses the data-flow requirements modeling language proposed in [MPT06b]. Both approaches have been integrated in the formal framework for the automated composition of web services described in [PTB05, PTBM05].

3.2.1 Implicit Data Flow Requirements: Knowledge Base

The specification of the data-flow requirements within the knowledge level composition approach [PMTB05] exploits the semantic annotations in the abstract process descriptions of the component services. Such annotations specify how incoming messages are manipulated within the component service (by means of internal functions and assignments) to obtain outgoing messages.

Example 8. *In the Hotel service protocol of Figure 2.1, the semantic annotation cost := costOf(date, loc) models the fact that the Hotel obtains the cost offered to the client by means of its internal function costOf applied on the date and loc information received in the client request. Similarly, the semantic annotation map := getMap(from_loc, to_loc) in the AllMaps service protocol specifies that the map provided by the AllMaps service is obtained by means of its internal function getMap on the location information received from the client.*

In the following example we show how the specification of the implicit data-flow requirements exploits the functions of the component services and uses additional functions specifying how the data computed by the components need to be combined in the composite service.

Example 9 (Implicit data flow requirements)**.** *A possible specification of the data-flow composition requirements in Example 4 is the following:*

> C.f_offer.cost =
>   F.costOf(C.request.date,C.request.loc) ∧
> C.f_offer.schedule =
>   F.getSchedule(C.request.date,C.request.loc) ∧
> C.h_offer.cost =
>   H.costOf(getDate(C.f_offer.schedule),C.request.loc) ∧
> C.h_offer.info =
>   H.getInfo(getDate(C.f_offer.schedule),C.request.loc) ∧
> C.h_offer.distance =
>   A.getMap(getAirport(C.f_offer.schedule),getLocation(C.h_offer.info)) ∧
> C.booked.cost =
>   prepareCost(C.h_offer.cost, C.f_offer.cost) ∧
> C.booked.info =
>   prepareInfo(C.h_offer.info, F.booked.info)

The first requirement declares that the flight cost offered to the Customer must be obtained by means of the Flight function costOf on the location and date information in the Customer request (and similarly for the flight schedule in the second requirement). The third requirement specifies that the VTA must obtain the hotel cost to be offered to its Customer using the Hotel internal function costOf on the location information received in the Customer request and on the date information that the VTA can obtain (by means of its internal function getDate) from the flight schedule. And so on.

Within this approach, the data-flow requirements specify how the information sent to the customer must be obtained by composing functions of the component services (e.g. F.costOf, F.getSchedule, H.costOf, H.getInfo, A.getMap) and of the new composite service (i.e. getDate, getAirport, getLocation, prepareCost, prepareInfo). These requirements implicitly constrain the message exchanges that must occur with the component services and the internal data manipulation that the composite service must perform. Notice that the internal functions of the new composite service can be used both to aggregate information received from the components (e.g. when applying the prepareCost function to compute the overall cost of the holiday package) and to extract information from the received messages (e.g. when obtaining the airport location from the flight schedule by means of the getAirport function, or when extracting the hotel location from the hotel information by means of the getLoc function).

3.2.2 Explicit Data Flow Requirements: Data Net

The aim of the data flow modeling language presented in [MPT06b] is to allow for the specification of complex requirements concerning data manipulation and exchange. In

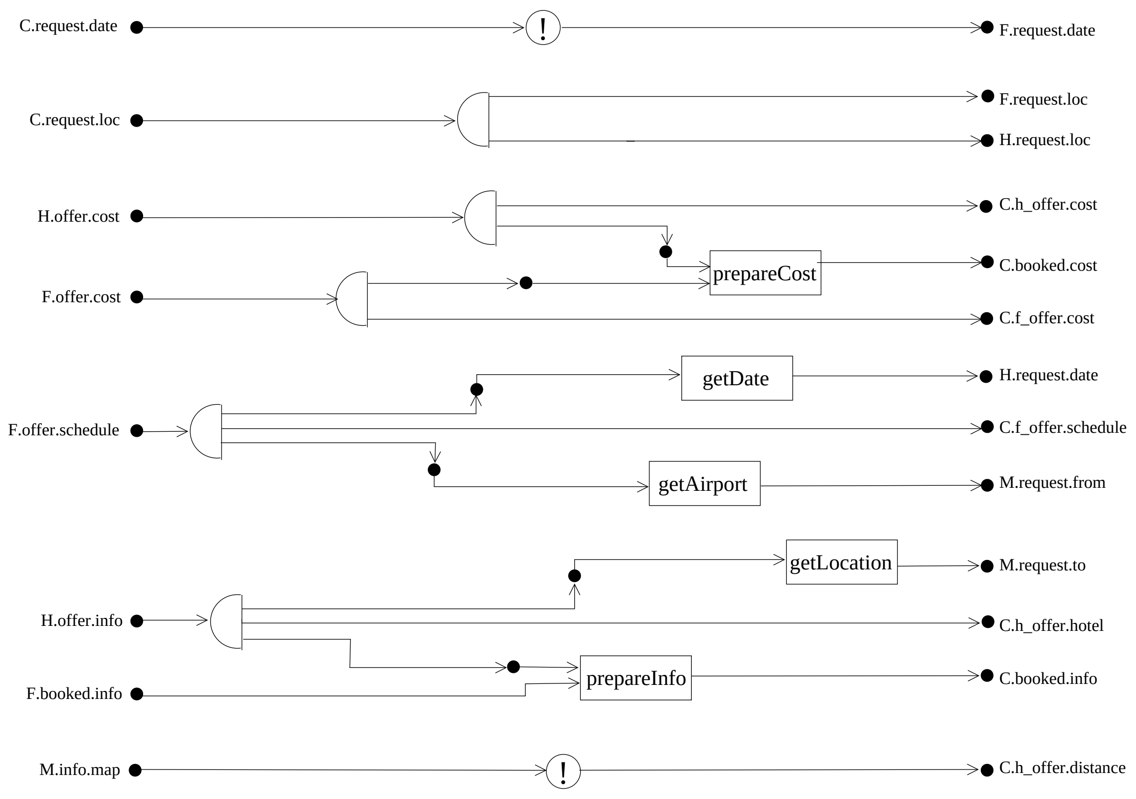


Figure 3.2: The data flow requirements for the Virtual Travel Agency

particular, data flow requirements specify explicitly how output messages (messages sent to component services) are obtained from input messages (messages received from component services). This includes several important aspects: whether an input message can be used several times or just once, how several input messages must be combined to obtain an output message, whether all messages received must be processed and sent or not, etc..

Syntax

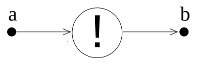
In the following we describe the basic elements of the language, show how they can be composed to obtain complex expressions and provide an intuitive semantics.

- *Connection Node*

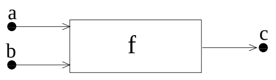
  ●

  A *connection node* can be external or internal. An external connection node is associated to an output (or an input) external port. Intuitively, an external input (output) node characterizes an external source (target) of data and it is used to exchange data with the outside world.

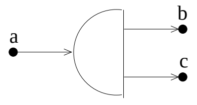
- *Identity*
  It is connected to one connection node in input and one node in output. The requirement states that data received from the input node should be forwarded to the output node. The graphical notation for the data-flow identity element `id(a)(b)`, with input node `a` and output node `b`, is the following:



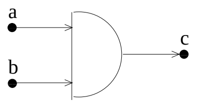
- *Operation*
  It is related to a function definition; it is connected to as many input nodes as the number of function parameters and only to one output node corresponding to the function result. The requirement states that, when data is received from all the input nodes, the result of the operation should be forwarded to the output node. The graphical notation for the data-flow operation element `oper[f](a,b)(c)` characterizing function `f`, with input nodes `a` and `b` and output node `c`, is the following:



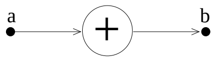
- *Fork*
  It is connected to a node in input and to as many nodes as necessary in output. It forwards data received on the input node to all the output nodes. The graphical notation for the data-flow fork element `fork(a)(b,c)`, with input node `a` and output nodes `b` and `c`, is the following:



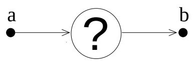
- *Merge*
  It is connected to one node in output and as many nodes as necessary in input. It forwards data received on some input node to the output node. It preserves the temporal order of data arriving on input nodes (if it receives data on two or more input nodes at the same time, the order is nondeterministic). We represent the data-flow merge element `merge(a,b)(c)`, with input nodes `a` and `b` and output node `c` as:



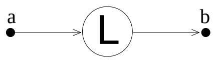
- *Cloner*
  It is connected to one node in input and one node in output. It forwards, one or more times, data received from the input node to the output node. The data-flow cloner element `clone(a)(b)`, with input node `a` and output node `b` is represented as:



- *Filter*
  It is connected to one node in input and one node in output. When it receives data on the input node, it either forwards it to the output node or discards it. We represent the data-flow filter element `filt(a)(b)`, having input node `a` and output node `b` as:



- *Last*
  It is connected to one node in input and one node in output. It requires that at most one data is forwarded to the output node: the last data received on the input node. All other data that are received should be discarded. The graphical notation for the data-flow last element `last(a)(b)`, with input node `a` and output node `b`, is the following:



The diagram obtained by suitably composing data-flow elements by means of connection nodes is called *data net* (see Figure 3.2 for an example). A data net is characterized by a set of external connection nodes associated to input ports N^I_{ext}, a set of external connection nodes associated to output ports N^O_{ext}, a set of internal connection nodes N_{int}, a set of data-flow elements D (corresponding to the basic elements described in this section) and a set of data values V. Given a data-flow element d, we denote with $in\_nodes(d)$ the set of input connection nodes of d and with $out\_nodes(d)$ the set of output connection nodes of d.

Definition 2 (Data Net)**.**
A data net $\mathcal{D}$ *is a tuple* $\langle N^I_{ext}, N^O_{ext}, N_{int}, D, V\rangle$ *where:*

- *for each* $n \in N^I_{ext}$ *there exists a unique data-flow element* $d \in D$ *s.t.* $n \in in\_nodes(d)$*;*

- *for each* $n \in N^O_{ext}$ *there exists a unique data-flow element* $d \in D$ *s.t.* $n \in out\_nodes(d)$*;*

- *for each* $n \in N_{int}$ *there exists a unique data-flow element* $d_1 \in D$ *s.t.* $n \in in\_nodes(d_1)$ *and there exists a unique data-flow element* $d_2 \in D$ *s.t.* $n \in out\_nodes(d_2)$*;*

- *for each* $d \in D$*,* $in\_nodes(d) \subseteq N^I_{ext} \cup N_{int}$ *and* $out\_nodes(d) \subseteq N^O_{ext} \cup N_{int}$*.*

Notice that it is possible to associate a type to each connection node in the network. Indeed, external nodes inherit the types from the corresponding BPEL4WS ports, and the types of internal nodes can be deduced from the structure of the data net. We do not consider this aspect to make the formalization more understandable; completing the model to handle typed connection nodes is straightforward.

A possible specification of the data net for the Virtual Travel Agency example is presented in Figure 3.2, which we will (partially) explain in the following example.

Example 10. *When the VTA receives a request from the Customer, it must forward the date information to the Flight and the loc information both to the Flight and to the Hotel.*

To obtain the cost to be sent in the booked message to the Customer, the VTA must apply its internal function prepareCost on the cost received in the offer from the Flight and on the cost information in the offer received from the Hotel.

The VTA must obtain the date information that it sends in the request to the Hotel by computing its internal function getDate on the schedule received in the offer of the Flight. The schedule received in the offer of the Flight is also forwarded to the client, as part of the f_offer message.

Moreover, the VTA exploits the internal function getAirport on the flight schedule to obtain the from location information to be sent to the AllMaps; similarly, it obtains the to location information by means of its internal function getLocation on the info received in the offer from the Hotel.

And so on.

Semantics

We now formalize the semantics of the data flow modeling language. Given a data net $\mathcal{D} = \langle N^I_{ext}, N^O_{ext}, N_{int}, D, V\rangle$, we denote with N_{ext} the sets of all external connection nodes, formally $N_{ext} = N^I_{ext} \cup N^O_{ext}$. An *event* e of $\mathcal{D}$ is a couple $\langle n, v\rangle$, where $n \in N_{ext} \cup N_{int}$, and $v \in V$, which models the fact that the data value v passes through the connection node n. An *execution* ρ of $\mathcal{D}$ is a finite sequence of events $e_0, ..., e_n$. Given an execution ρ we define its projection on a set of connection nodes $N \subseteq N_{ext} \cup N_{int}$, and denote it with $\Pi_N(\rho)$, the ordered sequence $e'_0, ..e'_m$ representing the events in ρ which correspond to nodes in N.

We formally define the semantics of our language in terms of *accepted executions* of a data net $\mathcal{D}$. In the following definition, we exploit regular expressions to define the accepted execution. We use notation $\Sigma_{v\in V}$ to express alternatives that range over all the possible values $v \in V$ that can flow thorough the net.

Definition 3 (data net accepting execution)**.**
An execution ρ is accepted by a data net $\mathcal{D} = \langle N^I_{ext}, N^O_{ext}, N_{int}, D, V\rangle$ if it satisfies all the following properties:

- *for each identity element* `id(a)(b)` *in $\mathcal{D}$:*

$$\Pi_{\{a,b\}}(\rho) = \left(\sum_{v\in V}\langle a, v\rangle \cdot \langle b, v\rangle\right)^*$$

- *for each operation element* `oper[f](a,b)(c)` *in $\mathcal{D}$:*

$$\Pi_{\{a,b,c\}}(\rho) = \left(\sum_{v,w\in V}(\langle a, v\rangle \cdot \langle b, w\rangle + \langle b, w\rangle \cdot \langle a, v\rangle) \cdot \langle c, f(v, w)\rangle\right)^*$$

- *for each fork element* `fork(a)(b,c)` *in $\mathcal{D}$:*

$$\Pi_{\{a,b,c\}}(\rho) = \left(\sum_{v\in V}\langle a, v\rangle \cdot (\langle b, v\rangle \cdot \langle c, v\rangle + \langle c, v\rangle \cdot \langle b, v\rangle)\right)^*$$

- *for each merge element* `merge(a,b)(c)` *in $\mathcal{D}$:*

$$\Pi_{\{a,b,c\}}(\rho) = \left(\sum_{v\in V}(\langle a, v\rangle \cdot \langle c, v\rangle + \langle b, v\rangle \cdot \langle c, v\rangle)\right)^*$$

- *for each cloner element* `clone(a)(b)` *in $\mathcal{D}$:*

$$\Pi_{\{a,b\}}(\rho) = \left(\sum_{v\in V}\langle a, v\rangle \cdot \langle b, v\rangle \cdot \langle b, v\rangle^*\right)^*$$

- *for each filter element* `filt(a)(b)` *in $\mathcal{D}$:*

$$\Pi_{\{a,b\}}(\rho) = \left(\sum_{v\in V}\langle a, v\rangle \cdot (\langle b, v\rangle + \epsilon)\right)^*$$

- *for each last element* `last(a)(b)` *in $\mathcal{D}$:*

$$\Pi_{\{a,b\}}(\rho) = \left(\sum_{v \in V} \langle a, v \rangle\right)^* \cdot \left(\sum_{v \in V} \langle a, v \rangle \cdot \langle b, v \rangle\right) + \epsilon$$

Notice that this definition considers data net elements having at most two input/output nodes, however it can easily be extended to handle elements of the data net having more input/output nodes.

Data Net Satisfiability

A data net can be used to specify the desired behavior of a service or everything that concerns the exchange of data with its communication partners. In particular, as shown in the data net of Figure 3.2, external connection nodes are associated to input (or output) ports which model BPEL4WS messages, or message parts, which are used to store data received (or sent) by the process while interacting with other services.

Since the behavioral aspect we are interested in concerns the data flow among the process and its partners, we characterize an execution of a BPEL4WS process W, denoted with $exec(W)$, as the set of all possible ordered sequence of input/output messages (or message parts) received and sent by the process from its activation to its termination. Notice that each message carries both the information about the external port on which it has been sent/received and about its content (value).

Definition 4 (data net satisfiability)**.**
Let W be a BPEL4WS *process and $\mathcal{D} = \langle N^I_{ext}, N^O_{ext}, N_{int}, D, V\rangle$ a data net. We say that W* satisfies *$\mathcal{D}$ if for each process execution $\rho_W \in exec(W)$ there exists an accepting execution ρ of $\mathcal{D}$ such that $\Pi_{N_{ext}}(\rho) = \rho_W$.*

Chapter 4

Integration within an Automated Composition Framework

In this chapter we show how the proposed approaches for the specification of composition requirements can be integrated in the framework for the automated composition of Web services presented in [PTB05, PTBM05].

4.1 An Automated Composition Framework

The work in [PTB05] (see also [PTBM05, PMTB05]) presents a formal framework for the automated composition of Web services which is based on planning techniques: component services define the planning domain, composition requirements are formalized as a planning goal, and planning algorithms are used to generate the composite service. The framework of [PTB05] differs from other planning frameworks since it assumes an asynchronous, message-based interaction between the domain (encoding the component services) and the plan (encoding the composite service). We now recall the most relevant features of the framework defined in [PTB05]. An overview of the framework is given in Figure 4.1, please refer to D4.3 for a detailed description.

The composition domain is modeled as a *state transition system* (STS from now on) which describes a dynamic system that can be in one of its possible *states* (some of which are marked as *initial states* and/or as *final 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 *internal actions*, which represent internal evolutions that are not visible to external services, i.e., data computation that the system performs without interacting with external services. A *transition relation* describes how the state can evolve on the basis of inputs, outputs, or internal actions.

Definition 5 (state transition system (STS))**.**

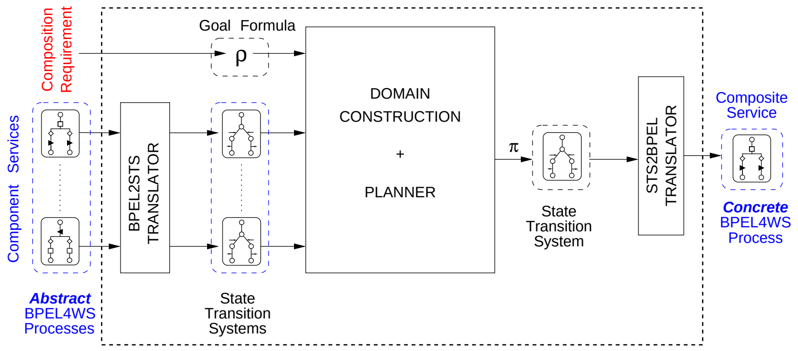


Figure 4.1: The general automated composition framework.

A state transition system Σ *is a tuple* $\langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{A}, \mathcal{R}, \mathcal{F} \rangle$ *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;*
- $\mathcal{A}$ *is the finite set of internal actions;*
- $\mathcal{R} \subseteq \mathcal{S} \times (\mathcal{I} \cup \mathcal{O} \cup \mathcal{A}) \times \mathcal{S}$ *is the transition relation;*
- $\mathcal{F} \subseteq \mathcal{S}$ *is the set of final states.*

We have defined a translation that associates a state transition system to each component service, starting from its BPEL4WS specification. We omit the formal definition of the translation, which can be found at `http://astroproject.org`. Intuitively, input actions of the STS represent messages received from the component services, output actions are messages sent to the component services, internal actions model assignments and other operations which do not involve communications, and the transition relation models the evolution of the service. Examples of the STS representation of BPEL4WS component services can be found in Figure 2.1.

The automated synthesis problem consists in generating a state transition system Σ_c that, once connected to Σ, satisfies the composition requirements. We now recall the definition of the state transition system describing the behavior of Σ when connected to Σ_c.

Definition 6 (Controlled system)**.**
Let $\Sigma = \langle \mathcal{S}, \mathcal{S}^0, \mathcal{I}, \mathcal{O}, \mathcal{A}, \mathcal{R}, \mathcal{F} \rangle$ *and* $\Sigma_c = \langle \mathcal{S}_c, \mathcal{S}_c^0, \mathcal{I}_c, \mathcal{O}_c, \mathcal{A}, \mathcal{R}_c, \mathcal{F}_c \rangle$ *be two state transition systems such that* $\mathcal{I} = \mathcal{O}_c$ *and* $\mathcal{O} = \mathcal{I}_c$*. The state transition system* $\Sigma_c \triangleright \Sigma$*, describing the behaviors of system* Σ *when controlled by* Σ_c*, is defined as follows:*

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

where:

$$\langle (s_c, s), a, (s'_c, s') \rangle \in (\mathcal{R}_c \triangleright \mathcal{R}), \textit{ if}$$
$$\langle s_c, a, s'_c \rangle \in \mathcal{R}_c \textit{ and } \langle s, a, s' \rangle \in \mathcal{R}$$

In an automated synthesis problem, the composition requirements are formalized as a specification ρ, and the composition task consists in generating a Σ_c that guarantees that the controlled system $\Sigma_c \triangleright \Sigma$ satisfies the requirement ρ, written $\Sigma_c \triangleright \Sigma \models \rho$. The definition of whether ρ is satisfied is given in terms of the executions that $\Sigma_c \triangleright \Sigma$ can perform.

Given this, we can characterize formally an automated synthesis problem.

Definition 7 (Automated Synthesis)**.**
Let Σ be a state transition system, and let ρ be an EAGLE *formula defining a composition requirement. The automated synthesis problem for Σ and ρ is the problem of finding a state transition system Σ_c such that*

$$\Sigma_c \triangleright \Sigma \models \rho.$$

The work in [PTB05, PTBM05] shows how to adapt to this task the "Planning as Model Checking" approach, which is able to deal with large nondeterministic domains and with requirements expressed in EAGLE. It exploit powerful BDD-based techniques developed for Symbolic Model Checking to efficiently explore domain Σ during the construction of Σ_c.

4.2 Integrating Implicit Data Flow Requirements

In this section we show how we integrated the implicit approach (recalled in Section 3.2.1) within the automated composition framework briefly presented in Section 4.1.

The implicit approach of [PMTB05] requires to re-write the data-flow requirement in a simpler form, more suitable for automated composition, as described in the next example.

Example 11. *Consider a subset of the implicit data flow requirements of Example 9:*

C.f_offer.cost =
  F.costOf(C.request.date,C.request.loc) $\wedge$
C.f_offer.schedule =
  F.getSchedule(C.request.date,C.request.loc) $\wedge$
C.h_offer.cost =
  H.costOf(getDate(C.f_offer.schedule),C.request.loc) $\wedge$
C.booked.cost =
  prepareCost(C.h_offer.cost, C.f_offer.cost) $\wedge$

We flatten the functions introducing auxiliary variables until only basic propositions are left:

$$\begin{aligned}
&\mathit{goal.flight\_cost = F.costOf(goal.cust\_date, goal.cust\_loc)} \wedge\\
&\mathit{goal.flight\_cost = C.f\_offer.cost} \wedge\\
&\mathit{goal.cust\_date = C.request.date} \wedge\\
&\mathit{goal.cust\_loc = C.request.loc} \wedge\\
&\mathit{goal.flight\_sched = F.getSchedule(goal.cust\_date, goal.cust\_loc)} \wedge\\
&\mathit{goal.flight\_sched = C.f\_offer.schedule} \wedge\\
&\mathit{goal.flight\_date = getDate(goal.flight\_sched)} \wedge\\
&\mathit{goal.hotel\_cost = H.costOf(goal.flight\_date, goal.cust\_loc)} \wedge\\
&\mathit{goal.hotel\_cost = C.h\_offer.cost} \wedge\\
&\mathit{goal.added\_cost = prepareCost(goal.hotel\_cost, goal.flight\_cost)} \wedge\\
&\mathit{goal.added\_cost = C.booked.cost}
\end{aligned}$$

From this flattened goal we can extract the goal variables (goal.added_cost, goal.added_delay, goal.prod_cost, goal.prod_delay, goal.ship_cost, goal.ship_delay, goal.prod_size, goal.user_art, and goal.user_loc) and a set of elementary constraints on these variables.

The flattened representation of data requirements in terms of goal variables and constraints among them allows for an efficient encoding of these requirements in the composition domain where the actual data values are never considered. Indeed, the satisfaction of the requirements is achieved by reasoning at the *knowledge-level*, on the relations that we *know* to be true among the goal and the process variables during the executions.

Example 12. *Consider the requirement goal.flight_cost = F.costOf(goal.cust_date, goal.cust_loc). If we call the operation F.request with values goal.cust_date, goal.cust_loc, then we* know *that F.date = goal.cust_date and F.loc = goal.cust_loc will hold. Due to the semantic annotation in the Flight process, we* know *that if we receive an F.offer, then F.cost = F.costOf(F.date, F.loc) holds. If we assign the received cost to goal.flight_cost, then we know that goal.flight_cost = F.cost.*

From the knowledge-level propositions K(F.date = goal.cust_date), K(F.loc = goal.cust_loc), K(F.cost = F.costOf(F.date, F.loc)), K(goal.flight_cost = F.cost), applying simple inference rules, we can deduce that the requirement goal.flight_cost = F.costOf(goal.cust_date, goal.cust_loc) holds.

In [PMTB05] we formally defined the knowledge level model described in the previous example in terms of a suitable knowledge base, i.e., of a set of knowledge-level propositions that define the constraints among goal and process variables that are known to hold at a given point in the evolution of the system. From the representation of the data-flow composition requirements given in Example 11 we can straightly obtain a set of knowledge-level propositions. Such propositions define therefore constraints on the knowledge base that the new composite service must have at the end of its execution.

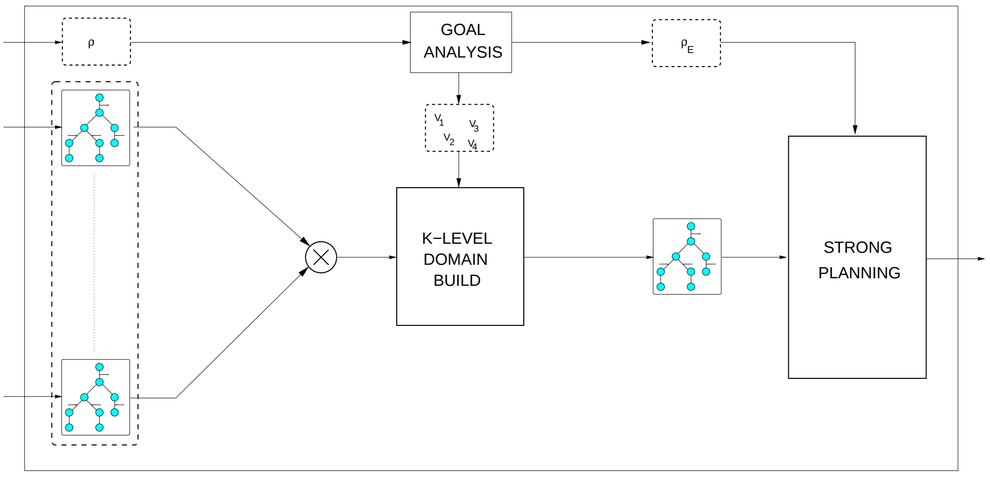


Figure 4.2: 'Domain Construction + Planner' Module for the Implicit Approach.

Figure 4.2 shows how the knowledge level approach can be integrated within the automated composition framework presented in Section 4.1, for details please refer to D4.3. The knowledge-level composition domain Σ_K 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. Each state of Σ_K models both the evolution of the services (process states) and the evolution of the data knowledge (knowledge base). The execution of a transition affects not only the states of the processes but also the knowledge base according to the data transformation performed (similarly to what we have shown in Example 12). The planning goal ρ is the EAGLE formalization of the composition control-flow requirements. We enrich ρ by adding to the TryReach part the flattened data-flow requirements (see Example 11). Given the domain Σ_K described above, we can apply the approach presented in [PTB05] and obtain a Σ_c that controls Σ_K by satisfying the composition requirements ρ.

Despite of the fact that the synthesized controller Σ_c is modelled 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 Σ_c.

4.3 Integrating Explicit Data Flow Requirements

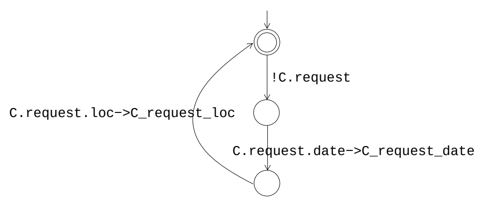
In this section we show how the explicit approach presented in Section 3.2.2 can be integrated within the automated composition framework briefly recalled in Section 4.1.

4.3.1 Data Requirements as STSs

As we have seen in Section 3.2.2, a data net $\mathcal{D}$ of a particular composition problem specifies how messages received from the component services can be used by the new composite process to generate outgoing messages. Therefore, it is possible to represent $\mathcal{D}$ as a STS $\Sigma_{\mathcal{D}}$, which models the allowed data flow actions. In particular, input actions in $\Sigma_{\mathcal{D}}$ represent messages received by the component services, output actions represent messages sent by the component services and internal actions represent assignments that the composite process performs on its internal variables.

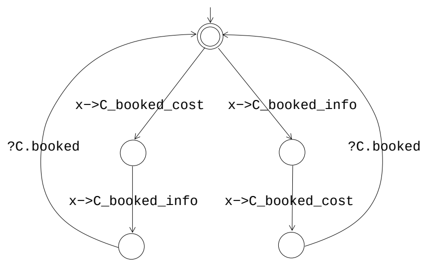
We assume that, in the BPEL4WS specification of the composite service, a variable will exist for each connection node in $\mathcal{D}$; variables associated to external connection nodes are those used by the new composite process to store received messages and to prepare the messages to be sent, while variables associated to internal connection nodes are those used to manipulate messages by means of internal functions and assignments. Then $\Sigma_{\mathcal{D}}$ defines constraints on the possible operations that the composite process can perform on these variables. A nice feature of our approach is that this can be done compositionally, i.e., a "small" automaton can be associated to each element of the data net, and STS $\Sigma_{\mathcal{D}}$ is obtained as the product of all these small automata.

More precisely, for each output operation of a component service, which is associated to some external input port in the data net, we define a STS which represents the sending of the message (as an output action) and the storing of all message parts (as internal actions). As an example, considering the VTA composition problem, for the output operation C.request with message parts date and loc we define the following STS:[1]



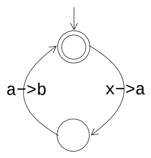
Similarly, for each input operation of a component service, which is associated to some external output port in the data net, we define a STS which represents the storing of all message parts (as internal actions) and the reception of the message (as an input action). As an example, for the input operation C.booked with message parts info and cost we define the following STS:

[1]In the STS that we use to model data net requirements, we represent input operations by a `?` followed by the operation name, output actions by a `!` followed by the operation name, while internal actions, denoted with `a->b`, model an internal operation that copies the value of `a` in variable `b`. We use `x` as a place-holder for arbitrary nodes/expression: so for instance `x->a` denotes all internal actions copying any variable/expression to variable `a`, and similarly for `a->x`. Final states are marked with an internal circle.

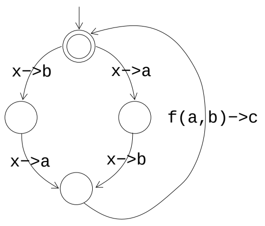


Finally, we define a STS for each data-flow element of the data net. These STSs have no input/ouput actions since they model manipulation of variables through assignments. In particular:

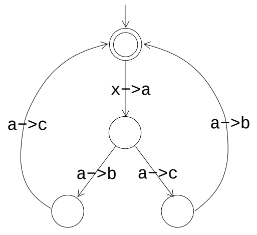
- for each identity element `id(a)(b)` in the data net we define the following STS:



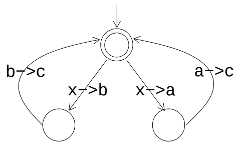
- for each operation element `oper[f](a,b)(c)` in the data net we define the following STS:



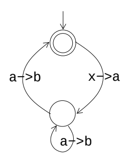
- for each fork element `fork(a)(b,c)` in the data net we define the following STS:



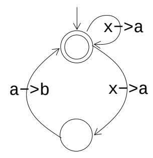
- for each merge element `merge(a,b)(c)` in the data net we define the following STS:



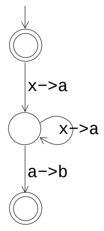
- for each cloner element `clone(a)(b)` in the data net we define the following STS:



- for each filter element `filt(a)(b)` in the data net we define the following STS:



- for each last element `last(a)(b)` in the data net we define the following STS:



Example 13. The small automata corresponding to the first two requirements in Figure 3.2 are represented in Figure 4.3.

The STS $\Sigma_{\mathcal{D}}$ modeling the data net $\mathcal{D}$ is the synchronized product of all the STSs corresponding to external connection nodes and to data-flow elements of $\mathcal{D}$. The synchronized product $\Sigma_1||\Sigma_2$ models the fact that the systems Σ_1 and Σ_2 evolve simultaneously on common actions and independently on actions belonging to a single system.

4.3.2 Generating the Composite Process

We are ready to show how we can integrate the proposed composition approach within the automated composition framework presented in Section 4.1 (see Figure 4.4 for an overview of the Domain Construction and Planner module).

Given n component services $W_1, ..., W_n$ and a data net $\mathcal{D}$ modeling the data-flow composition requirements, we encode each component service W_i as a STS Σ_{W_i} and the data net $\mathcal{D}$ as a STS $\Sigma_{\mathcal{D}}$. The composition domain Σ for the automated composition problem is the synchronized product of all these STSs. Formally, $\Sigma = \Sigma_{\mathcal{D}} \parallel \Sigma_{W_1} \parallel .. \parallel \Sigma_{W_n}$. The planning goal ρ is the EAGLE formalization of the composition termination requirements, enriched with the requirements that $\Sigma_{\mathcal{D}}$ need to terminate in a final state.

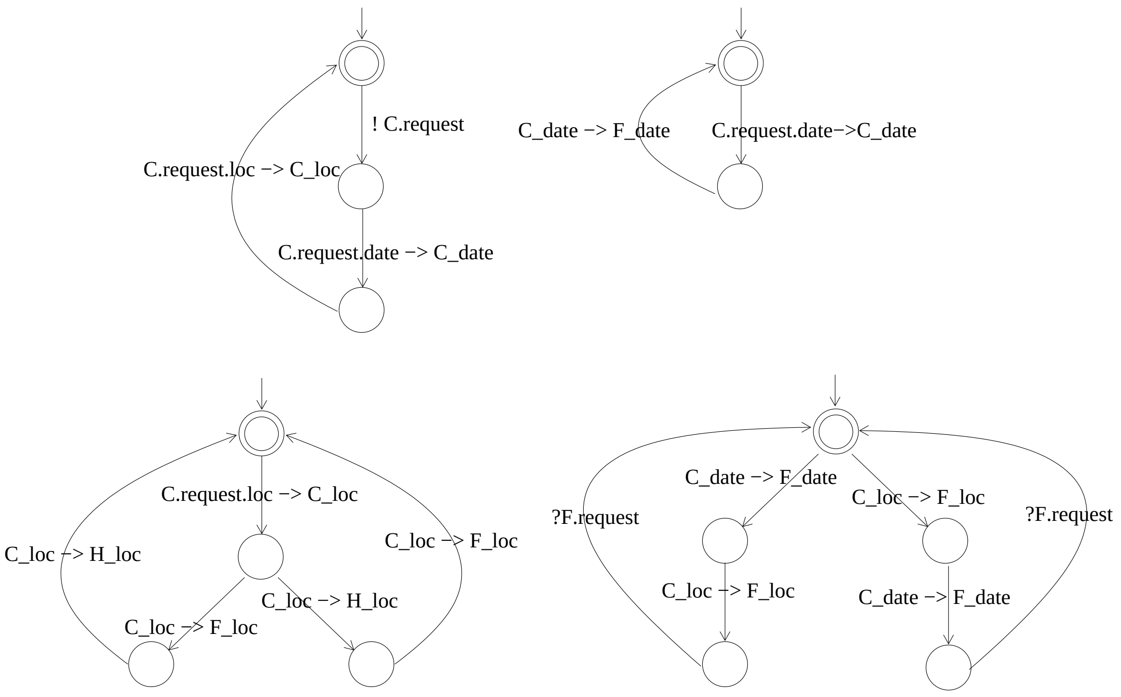


Figure 4.3: Examples of automata generated for data net requirements.
.

Given the domain Σ and the planning goal ρ we can apply the approach presented in [PTB05] to generate a controller Σ_c, which is such that $\Sigma_c \triangleright \Sigma \models \rho$. Once the state transition system Σ_c has been generated, it is translated into BPEL4WS to obtain the new process which implements the required composition. The translation is conceptually simple; intuitively, input actions in Σ_c model the receiving of a message from a component service, output actions in Σ_c model the sending of a message to a component service, internal actions model manipulation of data by means of expressions and assignments.

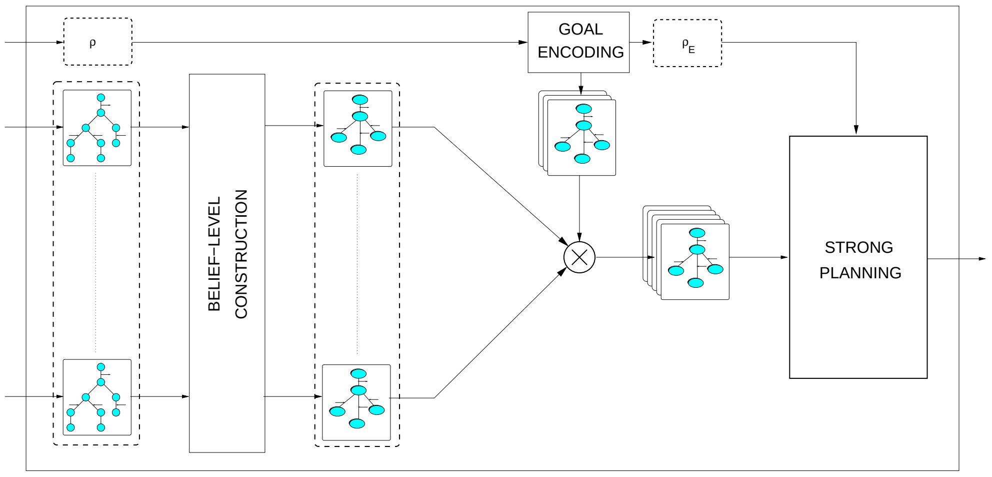


Figure 4.4: 'Domain Construction + Planner' Module for the Explicit Approach.

Chapter 5

Evaluation of the Proposed Approaches

In this chapter we compare the (implicit and explicit) proposed approaches from a technical point of view, from the point of view of the performance, and for what concerns the usability.

The two approaches present some similarities. First of all both of them extend the automated composition framework proposed in [PTB05, PTBM05]. Moreover, they adopt the same strategy to encode data manipulation and exchange at an abstract level in the composition domain: introducing goal variables (which model variables of the new composite BPEL4WS process) and encoding data-flow requirements as constraints on the operations that can be performed on goal variables. A first difference can be seen in the way they extract these variables and constraints: the knowledge-level approach obtains them by flattening the data-flow goal, while the other approach directly obtains them from the data net, where they are explicitly defined. However, the main difference lies in the way the two approaches reason on goal variables: the implicit approach encodes the knowledge on the equalities between these variables (knowledge base) in the states of the composition domain and uses this information to check whether the goal is satisfied; while the data net based approach doesn't explicitly encode data within the composition domain (states simply model the evolution of the processes) and introduces new STSs modeling the synchronization between operations that manipulate goal variables. This difference is reflected in the complexity of the obtained composition domain, which in the knowledge level approach is much heavier.

5.1 Experimental Evaluation

In order to test the performance of the proposed approaches, we have conducted some experiments on different domains (see also [MPT06a]). One of these is the VTA, which we used as a reference example throughout this deliverable. This is a simple example: the interactions among the components and the composite service are quite straight and the

	Knowledge Level						**Data Net**						**BPEL**
	domain				**time (sec.)**		**domain**				**time (sec.)**		num
	kl props	goal vars	nr. of states	max path	model construction	composition & emission	dn vars	dn STSs	nr. of states	max path	model construction	composition & emission	complex activities
VOS	27	6	1357	54	11.937	3.812	14	14	390	26	1.612	0.218	52
VO2S	35	9	8573	64	185.391	84.408	20	20	1762	32	1.796	0.688	84
VO3S	46	12	75289	74	M.O.	-	26	26	12412	38	1.921	2.593	109
VO4S	-	-	-	-	-	-	32	32	122770	44	2.218	12.500	136
VO5S	-	-	-	-	-	-	38	38	1394740	50	2.547	26.197	165
VO6S	-	-	-	-	-	-	44	44	16501402	56	2.672	246.937	196

Table 5.1: Experiments with realistic composition case studies.

data-flow requirements are pretty simple. In [PMTB05] we have shown that the knowledge level composition approach scales up quite well when considering this simple composition domain; while in [MPT06b] we have shown that, within the data net approach, obtaining the composed process takes less then 1 sec.

Since we want to compare the two approaches on realistic domains, we consider here a more complex scenario, the Virtual Online Shop (VOS from now on). The VOS consists in providing an electronic purchase and payment service by combining a set of independent existing services: a given number of e-commerce services Shops and a credit-card payment service Bank. This way, the Customer, also described as a service, may directly ask the composite service VOS to purchase some given goods, which are offered by different Shops, and pay them via credit-card in a single payment transaction with the Bank. Notice that the Bank models a real on-line payment procedure offered by an Italian bank. Such a process handles several possible failures: it checks both the validity of the target bank account (the Store's one in our case) and the validity of the credit card, it checks whether the source bank account has enough money and whether the owner is authorized for such a money transfer. The Shop models a hypothetical e-commerce service, providing a complex offer negotiation and supporting a transactional payment procedure. With respect to the VTA, this composition problem requires a high degree of interleaving between components (to achieve the goal, it is necessary to carry out interactions with all component services in an interleaved way) and both the knowledge base of the goal and the data net are much more complex (due to the number of functions and to the need of manipulating data in a complex way).

Table 5.1 reports the experimental results. For each considered scenario it shows some parameters that characterize the complexity of the composition domain, the automated composition time, and the size of the generated composite process. The complexity of the knowledge level composition domain is given in terms of the number of knowledge level propositions and the number of goal variables. As we have seen in Section 4.2, the knowledge level propositions are used to model the knowledge on data that the composite process acquires while interacting with the component services and manipulating messages. The number of KL propositions strongly affects the complexity of the domain: it determine the size of the knowledge base that characterize each state of the STS Σ_K. For what concerns the approach using Data Nets (DN), we consider the number of goal variables (i.e., the number of nodes of the data net) and the number of STS used to encode data-flow requirements.[1] We recall that these STSs are directly obtained from the data net: each of them models a particular node and constraints either the receiving (sending) of a message, or an internal message manipulation. To complete the characterization of the complexity of the composition domain, we report for both approaches the number of states in the domain Σ and the number of transitions of the longest path in Σ.

The complexity of the composition task can also be deduced from the size of the new composite BPEL4WS process, which is reported in the last column of Table 5.1. We remark that we report the number of BPEL4WS basic activities (e.g. `invoke`, `receive`, `reply`, `assign`, `onMessage`) and do not count the BPEL4WS structured activities that are used to aggregate basic activities (e.g. `sequence`, `switch`, `flow`). Indeed, the former activities are a better measure of the complexity of the generated process, while the latter are more dependent on the coding style used in the composite BPEL4WS process. Notice that we report only one measure for the composite process. Indeed, the processes generated by the two approaches are basically identical: they implement the same strategy, handle exceptions and failures in the same way and present the same number of activities. The only difference is the way in which of such activities are arranged, e.g. the order of invocation of the different shops or of the assignments when preparing different parts of a message to be sent.

The composition times reported in Table 5.1 have been obtained on a Pentium Centrino 1.6 GHz with 512 Mb RAM of memory running Linux. We distinguish between model construction time and composition and emission time. The former is the time required to obtain the composition domain, i.e., to translate the BPEL4WS component services into STS and to encode the composition goal. The latter is the time required to synthesize the controller and to emit the corresponding BPEL4WS process. The experiments show that the knowledge level approach has worse performances both for the model construction time and for the composition time. In particular, the knowledge level approach is not able to synthesize the Virtual Online Shop scenario with three shops: a memory out is obtained in model construction time, due to the very high number of knowledge level propositions that need to be taken into account in the construction of the domain. In the case of the data net approach, instead, the time required to generate the composition domain is very small for all the scenarios, and also the performance for the composition scales up to very complex composition scenarios: the VO6S example (6 Stores, 1 Bank and 1 Customer) can be synthesized in about 4 minutes. We remark that this example is very complex, and requires several hours of work to be manually encoded: the corresponding BPEL4WS process contains about 200 non-trivial activities! We also remark that the number of states in the DN domain for the VO6S example is much larger than the number of states of the VO3S example at the knowledge level. The fact that the former composition has success while the latter has not show another important advantage on the data-net approach: the domain is very modular, since it consists of the composition of several very small STSs, which allows for a very efficient exploitation of the BDD-based techniques

[1]In the case of the different VOS scenarios, the number of goal variables is the same as the number of STS. This is a coincidence which does not hold in the case of other scenarios.

we adopt for the composition.

5.2 Usability

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

The implicit approach adopts a more abstract perspective, through the use of annotations in the process-level descriptions of component services. This makes the goal independent from the specific structure of the WS-BPEL processes implementing the component services, allowing to leave out most implementation details. It is therefore less time consuming, more concise, more re-usable than data nets. Moreover, annotations provide a way to give semantics to WS-BPEL descriptions of component services, along the lines described in [PST05], thus opening up the way to reason on semantic annotations capturing the component service internal behavior. Finally, implicit knowledge level specifications allow for a clear separation of the components annotations from the composition goal, and thus for a a clear separation of the task of the designers of the components from that of the designer of the composition, a separation that can be important in, e.g., cross-organizational application domains. However, taking full advantage of the implicit approach is not obvious, especially for people without a deep know-how of the knowledge level composition framework. There are two main opposite risks for the analysts: to over-specify the requirements adding non mandatory details and thus performing the same amount of work required by the explicit approach; to forget data-flow requirements which are necessary to find the desired composite service. Moreover, our experiments have shown that, while the time required to write the KL goal *given* the annotated WS-BPEL processes is much less than the time to specify the data net, the time required to write the KL goal *and* to annotate the WS-BPEL processes is much more (and the errors are much more frequent) than for the data net.

Chapter 6

Conclusions and Related Works

We have presented our approach for the specification of Web service composition requirements and show how the proposed approach can be integrated within an existing automated composition framework.

In particular, we have proposed to model control-flow requirements using EAGLE, a language for extended goals, which allows to express recovery conditions and preferences among sub-goals. For what concerns the specification of data-flow composition requirements, we have described and compared two different approaches: implicit requirements [PMTB05] compose functions that annotate WS-BPEL descriptions of component services; explicit requirements [MPT06b] directly specify the routing and manipulations of data exchanged in the composition. Both approaches have their pros and cons. Explicit models allows for much better performances, due to the very modular encoding of requirements. Moreover, they are rather easy to write and to understand for the analyst. Implicit requirements allow for a higher degree of re-use and abstraction, as well as for a clear separation of the components annotations from the composition goal.

Several methodologies have been proposed to model different aspects of requirements for service oriented applications, from goal-oriented approaches, see, e.g., [LM04, CMS05], to UML-based object-oriented methodologies, see, e.g., [SGS04]. The methodology proposed in [LM04] is founded on Tropos [PBG$^+$04], an agent-oriented software development technique that supports early and late requirements analysis, as well as architectural and detailed design. The key idea of this approach is that Web Services are designed by starting from stake-holders goals, that are then analyzed and refined into subgoals and tasks. However, the starting point for Web Service compositions are usually abstract workflows, which are derived from business scenarios, rather then high level business goals. In [SGS04] the authors propose a UML-based model-driven method for Web Service composition. They show how composite Web Service models can be transformed into executable models by means of UML model transformations. The weakness of UML-based approaches is that modeling complex composition scenarios is a demanding and difficult task since they require to use several different UML diagrams (e.g. activity, class, collaboration diagrams); moreover, the relationships between these diagrams do not have a clear and formal semantic. In all these works [SGS04, LM04, CMS05], high-level requirements are used to guide and construct by hand the composition. The problem of the automated synthesis of compositions is not addressed. These methodologies can be integrated with our approach and used in a pre-analysis step to guide the analyst to the definition of the detailed implicit or explicit data-flow requirements that we use to generate automatically the composition.

Most of the works that address the problem of the automated synthesis of process-level compositions do not take into account data flow specifications. This is the case of the work on synthesis based on automata theory that is proposed in [HBCS03, BCG$^+$03, BCGM05], and of work within the semantic web community, see, e.g., [MS02]. For what concerns the formal specification of the requirements that the new composite service should satisfy, most of the existing approaches specify them as reachability conditions (e.g. [NM02])

Some other approaches, see, e.g., [PF02], are limited to simple composition problems, where component services are either atomic and/or deterministic. In this approach component services are defined in terms of their inputs and outputs; given the inputs and outputs of the service, a rule is then defined which indicates which outputs can be obtained by the service given which inputs. When a developer wishes to create and deploy a new composite service, he specifies the inputs and outputs of the composite service and submits it to SWORD which determines if the composite service can be realized using the existing services. This rule based approach can be adopted exclusively for modeling simple composition problems, where component services are atomic and deterministic.

The work closest to ours is the one described in [BP05], which proposes an approach to service aggregation that takes into account data flow requirements. The main difference is that data flow requirements in [BP05] are much simpler and at a lower level than in our framework, since they express direct identity routings of data among processes, and do not allow for manipulations of data. The examples reported in this paper clearly show the need for expressing manipulations in data-flow requirements and higher level requirements. Finally, the work in [BP05] does not present any experimental evaluation.

Chapter 7

History of the Deliverable

The presented research works have been carried out in the last three years of the KLASE project. In this section, the history of the deliverable is described along these three years.

7.1 2nd year

During the second year of the KLASE project, a preliminary activity was conducted to determine what kind of requirements need to be specified when considering Web Service composition problems.

In this initial phase we concentrate on the control flow part. The result of this survey is that there is the need to express termination conditions on the executions of the component services and in particular to specify recovery conditions and preferences among different subgoals. For this reason we decided to formalize the control flow requirements using EAGLE, a requirement specification language designed with this specific purpose.

7.2 3rd year

The third year of the KLASE project was devoted to address the problem of handling realistic composition problems, overcoming the main limitation of our existing automated composition approach: the possibility to handle only scenarios where data can have only finite (and few) values.

Solving this problem required both to deal with the specification of data-flow requirements and to encode the reasoning on data within the planning domain in an efficient way. We proposed to apply knowledge level planning techniques and we developed an approach that adapts these techniques to the peculiarities of the Web Service composition problem. Finally, we integrated the knowledge level approach within our existing automated composition framework.

7.3 4th year

During the last year of the KLASE project we addressed two problems :

- making the specification of data flow composition requirements a more effective and affordable task
- encoding the reasoning on data within the planning domain in a more efficient and light way (wrt the knowledge level approach)

We addressed these problem through the explicit DF requirements specification approach. On the one hand we propose a data flow modeling language which, thanks also to a user-friendly graphical notation, allows to specify requirements in an easy and intuitive way. On the other hand, this approach doesn't explicitly encode data within the composition domain and thus the resulting encoding is much lighter.

Finally, we integrated the data net approach within our automated composition framework and compared the (implicit and explicit) proposed approaches from a technical point of view, from the point of view of the performance, and for what concerns the usability.

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.

[BCG$^+$03] D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Mecella. Automatic composition of E-Services that export their behaviour. In *Proc. ICSOC'03*, 2003.

[BCGM05] D. Berardi, D. Calvanese, G. De Giacomo, and M. Mecella. Composition of Services with Nondeterministic Observable Behaviour. In *Proc. ICSOC'05*, 2005.

[BP05] A. Brogi and R. Popescu. Towards Semi-automated Workflow-Based Aggregation of Web Services. In *Proc. ICSOC'05*, 2005.

[CMS05] E. Colombo, J. Mylopolous, and P. Spoletini. Modeling and Analyzing Context-Aware Compositions of Services. In *Proc. ICSOC'05*, 2005.

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

[HBCS03] R. Hull, M. Benedikt, V. Christophides, and J. Su. E-Services: A Look Behind the Curtain. In *Proc. PODS'03*, 2003.

[LM04] D. Lau and J. Mylopoulos. Designing Web Services with Tropos. In *Proc. ICWS'04*, 2004.

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

[MPT06a] A. Marconi, M. Pistore, and P. Traverso. Implicit vs. Explicit Data-Flow Requirements in Web Service Composition Goals. In *Workshop Proc. AISC'06*, 2006.

[MPT06b] A. Marconi, M. Pistore, and P. Traverso. Specifying Data-Flow Requirements for the Automated Composition of Web Services. In *Proc. SEFM'06*, 2006.

[MS02] S. McIlraith and S. Son. Adapting Golog for Composition of Semantic Web Services. In *Proc. KR'02*, 2002.

[NM02] S. Narayanan and S. McIlraith. Simulation, Verification and Automated Composition of Web Services. In *Proc. WWW'02*, 2002.

[PBG$^+$04] A. Perini, P. Bresciani, F. Giunchiglia, P. Giorgini, and J. Mylopoulos. A Knowledge Level Software Enineering Methodology for Agent Oriented Programming. In *Proc. of the Fifth International Conference on Autonomous Agents*, 2004.

[PF02] S. Ponnekanti and A. Fox. SWORD: A Developer Toolkit for Web Service Composition. In *Proc. WWW'02*, 2002.

[PMTB05] M. Pistore, A. Marconi, P. Traverso, and P. Bertoli. Automated Composition of Web Services by Planning at the Knowledge Level. In *Proc. IJCAI'05*, 2005.

[PST05] M. Pistore, L. Spalazzi, and P. Traverso. A Minimalist Approach to Semantic Annotations of Web Processes. In *Proc. of ESWC'05*, 2005.

[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 Composite BPEL4WS Web Services. In *Proc. ICWS'05*, 2005.

[SGS04] D. Skogan, R. Gronmo, and I. Solheim. Web Service Composition in UML. In *Proc. EDOC'04*, 2004.

[SPT06] D. Shaparau, M. Pistore, and P. Traverso. Contingent Planning with Goal Preferences. In *Proc. AAAI'06*, 2006.

[TPC$^+$05] M. Trainotti, M. Pistore, G. Calabrese, G. Zacco, G. Lucchese, F. Barbon, P. Bertoli, and P. Traverso. ASTRO: supporting the Composition and Execution of Web Services. In *Demo Paper: ICSOC'05*, 2005.