Self-Coordinated and Self-Traced Composite Services with Dynamic Provider Selection

Boualem Benatallah‡, Marlon Dumas†, Marie-Christine Fauvet‡* and Hye-Young Paik‡

† Cooperative Information Systems Research Centre
Queensland University of Technology
GPO Box 2434, Brisbane QLD 4001, Australia
m.dumas@qut.edu.au

‡ School of Computer Science & Engineering
The University of New South Wales
Sydney NSW 2052, Australia
{boualem,mcfauvet,hpaik}@cse.unsw.edu.au

UNSW-CSE-TR-0108
May 2001

THE UNIVERSITY OF
NEW SOUTH WALES

School of Computer Science and Engineering
The University of New South Wales
Sydney 2052, Australia

* On leave from LSR-IMAG, University of Grenoble, France.

Abstract

The growth of Internet technologies has unleashed a wave of innovations that are having tremendous impact on the way organisations interact with their partners and customers. It has undoubtedly opened new ways of automating Business-to-Business (B2B) collaboration. Unfortunately, as electronic commerce applications are most likely autonomous and heterogeneous, connecting and coordinating them in order to build inter-organisational services is a difficult task. To date, the development of integrated B2B services is largely ad-hoc, time-consuming and requires an enormous effort of low-level programming. This approach is not only tedious, but also hardly scalable because of the volatility of the Internet, and the dynamic nature of business alliances. In this paper, we consider the efficient composition and execution of B2B services. Specifically, we present a framework through which services can be declaratively composed, and the resulting composite services can be executed in a *decentralised* way within a *dynamic* environment. The underlying execution model supports the incremental collection of the execution trace of each composite service instance. These traces are particularly useful for customer feedback, and for detecting malfunctionings in the constitution of a composite service.

1 Introduction

In order to survive the massive competition created by the new online economy, traditional businesses are under the pressure to take advantage of the information revolution the Internet and the Web have brought about. Organisations of all sizes are moving their main operations to the Web for more automation, efficient business processes, and global visibility. It is estimated that more than 60% of all businesses will move their main operations to the Web in the year 2002 [10].

With the advent of the Internet and the Web, the first generation of Web-based e-commerce was born, namely Business-to-Customer (B2C) e-commerce. In the B2C model, a business transaction involves direct purchase/sale of goods over the Web as in retailing (e.g., purchasing a book from an online bookstore). More recently, organisations started using the Web as a means to automate relationships between their business processes, i.e, creating B2B (Business-to-Business) systems. For example, GM (General Motor) and Ford are using a single B2B system (a combination of the GM's TradeXchange and the Ford's Oracle-based AutoExchange) to interact with each other and exchange automotive parts with their costumers over the Web [4].

The ability to efficiently and effectively share services on the Web is a critical step towards the development of the new on-line economy driven by the B2B e-commerce. Existing enterprises would form alliances and join their services to share costs, skills and resources in offering value-added services (virtual enterprises). By *service*, we mean any class of immaterial products whose provision involves the execution of a set of human and/or computational activities within an organisation, or across several organisations [8]. An e-service is a service which is accessible through electronic means (e.g., a web interface).

In a nutshell, a service provides a semantically well defined functionality that allows users to access and perform tasks offered, e.g., by business applications. Typical examples of services include booking an airline and a hotel, charging a credit card with a given amount, customer relationship management (CRM), or a procurement system. An example of an integrated B2B service is an accounting management system that uses payroll, tax preparation, and cash management as components. The component services might all be outsourced to business partners.

The evolution into the global information infrastructure and the concomitant increase in the available information, is offering a powerful distribution vehicle for organisations that need to coordinate the use of multiple e-services. For instance, this infrastructure will enable providers of back-office transaction processing systems such as those supporting manufacturing, supply chain, accounting, and human resources, to deliver Web-based customer access. However, the technology to organise, abstract, search, compose, evolve, analyse, monitor, and access e-services has not kept pace with the rapid growth of available information.

Indeed, the development of integrated B2B services has been largely ad-hoc, time-consuming and requiring a considerable effort of low-level programming. This approach is clearly tedious and hardly scalable because of the volatility and size of the Web. Worse, as service integration is done in an ad hoc manner, it is likely to rely on proprietary solutions, thereby rendering B2B inter-service coordination a difficult task. On the other hand, the fast, dynamic, and adaptable composition of services is an essential requirement for organisations to adapt their business practices to the dynamic nature of the Web. Hence, providing a framework for the efficient composition and management of e-services requires a high-level declarative service composition language supporting the development and deployment of new services from existing ones.

Existing point-to-point composition techniques such as EDI, component-based mediators, and cross-organisational workflows are usually appropriate to integrate small numbers of services with static relationships. However, B2B relationships, particularly on-the-fly partnerships, require more flexible composition techniques. In addition, the execution of a composite service in existing techniques is usually centralised, whereas the constituents of a composite service are distributed and autonomous. For B2B E-commerce to really take off, there is a need for effective and efficient collaborative execution of services.

In this paper, we present Self-Serv: a framework through which e-services can be declaratively composed, and the resulting composite services can be executed in a *decentralised* way within a *dynamic* environment. By dynamic environment, we mean that a composite service is not necessarily bound to a particular set of service providers. Instead, an organisation participating in the provisioning of a composite service, is free to interrupt its participation in this provisioning process, without blocking the availability of the composite service. Similarly, new organisations may decide to provide a particular constituent of a composite service in future instantiations of it. Achieving this flexibility requires in one way or another that the choice of a provider for a constituent service is delayed until the last possible moment.

Following a common requirement in the areas of business processes and services management, we also expect the composite services to be traceable, meaning that the system should in one way or another keep track of ongoing and past executions. Since in Self-Serv these executions are carried out in a decentralised way, it would be inconsistent to collect their traces through direct communication between the providers of each constituent service and a centralised entity: an approach which creates a potential bottleneck. Instead, in Self-Serv the collection of traces is carried out through peer-to-peer exchange of partial traces between the providers of constituent services, and it is not until the end of the execution that these traces are sent to a centralised repository.

More precisely, we make the following contributions:

- A declarative language for composing services. We adopt statecharts [9] as the basis for the declarative specification of the business logic of composite e-services (i.e. coordination, temporal constraints, etc.). Statecharts are widely used in the area of reactive systems, and is emerging as a standard for process-modeling as it has been adopted by the UML (Unified Modeling Language) as the semantical backbone for intra-object coordination constructs.

- A composition framework that caters for the creation of *dynamic*, as well as *static* relationships among service providers. Dynamic relationships are supported through *virtual communities*: temporary alliances that bring together a variety of providers around a common interest (e.g., selling computers). Actual providers can register with any community of interest [3].

- A collaborative service execution model, whereby the responsibility of coordinating the execution of a composite service, is distributed across several peer software components.

- A collaborative service tracing model, whereby the execution trace of a composite service is collected through peer-to-peer interaction between the providers of the constituent services, before being stored by the provider of the composite service.

The remainder of this paper is organised as follows. First we give an overview of the system's functionalities and architecture in section 2. Next, in section 3, we detail our approach to service modeling and composition using statecharts. Section 4 discusses the collaborative execution of services, while section 5 discusses the tracing of services' executions. Finally, section 6 gives a brief overview of related work and section 7 provides some concluding remarks.

2 Design overview

This section provides an overview of the main concepts and salient features of the Self-Serv system, and describes a partial view of its architecture. A complete description of this architecture and its enabling technologies are beyond the scope of this paper.

2.1 Elementary and composite services

We define a *service*, as a class of immaterial products whose provision involves the execution of a set of human and/or computational activities within an organisation, or across several organisations. Examples of services are: translating a document, booking an airline ticket and a hotel, transporting an item from one point to another, and charging a credit-card with a given amount.

Each service provides an interface which enables its instantiation (leading to a *service instance*) and the subsequent execution of the resulting instance. In other words, the interface of an elementary service provides operators such as *instantiate*, *start*, *freeze*, *unfreeze*, *cancel*, etc. The interface also describes the protocol used for invoking an operation, passing its input parameters, and collecting its outputs. This protocol can be based on remote method invocation (e.g. Java RMI [22]) or message exchange (e.g. SOAP [20]).

The interface of a service is implemented by a *wrapper*: a software component hosted by the service's provider, that acts as the service's entry point. The wrapper of a service implements the functions *instantiate*, *start*, *freeze*, *unfreeze*, *cancel*, etc., and handles conversions between the data model of the service's interface (based on e.g. SOAP [20]), and that of its implementation (based on e.g. a proprietary C++ API).

Self-Serv distinguishes *elementary services* from *composite services*. Elementary services are pre-existing services, whose instances' execution is entirely under the responsibility of some entity called the *service provider*. The provisioning of an elementary service may involve a complex business process, but in any case, its internals are hidden to the user.

A composite service on the other hand, is recursively defined as an aggregation of elementary and composite services, which are referred to as *constituent services*. The semantics of this aggregation can be described from at least four perspectives[1]:

- The *control-flow perspective* establishes the order in which the services should be invoked, the timing constraints, the signals that may interrupt or cancel their execution, etc.
- The *provider perspective* provides an organisational anchor to the composite service by establishing which entity is responsible for performing which service.
- The *data exchange perspective* (or *data-flow perspective*) is built on top of the control-flow one. It captures both the flow of data between services, and the conversion of these data between the potentially heterogeneous data models used by the services participating in the composition.
- The *transactional perspective* relates to providing execution guarantees, that is, ensuring that the execution of a composite service or part of it is atomic, that two parallel branches are executed in isolation, etc.

[1]The first three of these perspectives are similar to those discussed in [15].

In the sequel, we focus on the control-flow and the provider perspectives. The data exchange and the transactional perspectives deserve a separate work, since they involve advanced data conversion and transaction monitoring techniques.

In Self-Serv the control-flow perspective is modeled through an adapted subset of the statecharts notation defined later in the paper. On the other hand, the provider perspective is modeled by associating an organisational entity to each service offer. In other words, the concept of service offer in Self-Serv encompasses both: what has to be done? and who has to do it? In some situations however the answer to the question "who" can be a community instead of a single provider as discussed below.

As a working example, we consider the service "Travel Solutions" which is specified by the statecharts depicted in Figure 1.

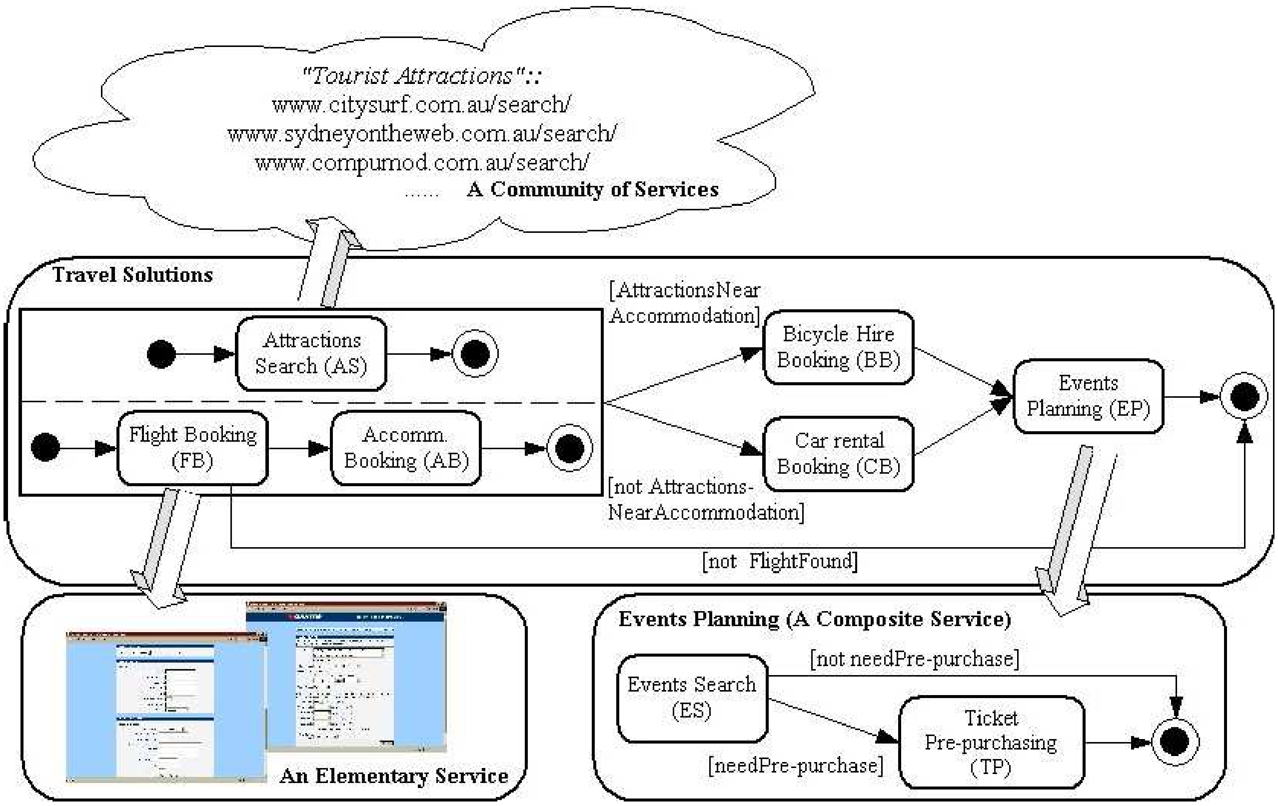


Figure 1: "Travel Solutions" modeled as a composite service.

The service is composed of several independent services like flight booking, car rental, event attendance planner, etc. It starts with an invocation to a flight booking service (FB) followed by an invocation to an accommodation booking service (AB). A service that searches for tourist attractions (AS) is executed concurrently with the former two. After all these services (AS, FB and AB) are completed, and based on how far the selected accommodation is from the major attractions, either a car rental service (CB), or a bicycle hire service (BB) is executed. Upon completion of either of these two services, a service which searches for special events occurring during the stay of the user is invoked. This "Event Planner" service is itself a composite service aggregating a service that searches for special events, and another that prepurchases tickets for these events.

Note that a constituent of a composite service can be assigned either to an individual service provider, or to a community of providers. For example, in Figure 1, the flight booking is assigned to an on-line booking web site, that does not need to delegate its task to any other entity. On the other hand, the service that searches for attractions is assigned to a community of service providers. This community federates several entities such as public tourism offices, and private tourist information sites. When a service execution request is addressed to the community, the representative of the community will forward this request to one of its members.

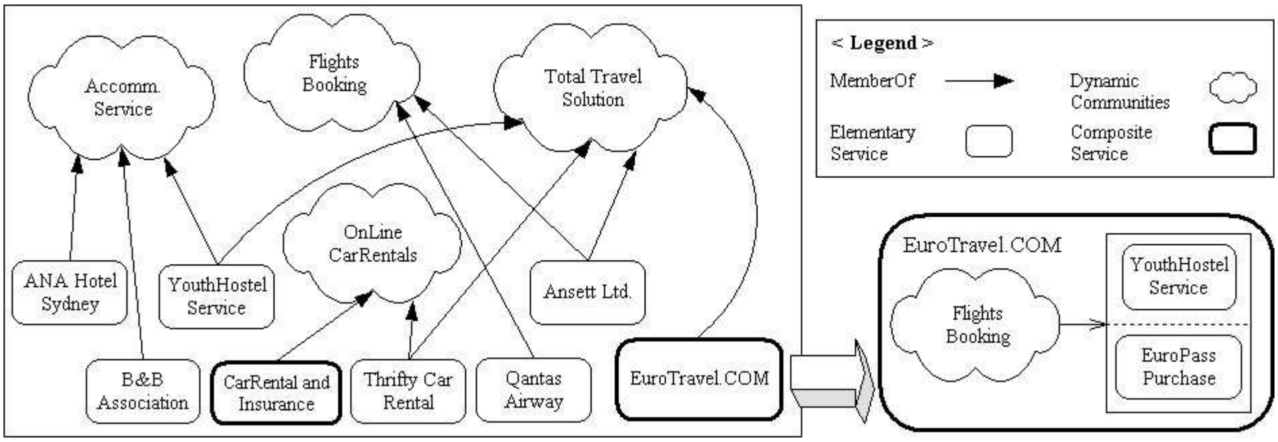


Figure 2: An example of dynamic communities and a composite service

Figure 2 shows an example of four dynamic communities which federate several elementary and composite services. A service can be a member of several communities (e.g. the YouthHostel Service is a member of "AccommodationService" as well as of "TotalTravelSolution"). The "TotalTravelSolution" community has a composite service ("EuroTravel.COM") as one of its constituents. "EuroTravel.COM" aggregates the services federated by the dynamic community "FlightBooking", as well as two elementary services: "YouthHostel" and "EuroPassPurchase", which are executed concurrently after the flight booking.

The means by which a community chooses a member to execute a request is specified via a *selection policy*. A selection policy can be based on existing auction protocols (e.g., a First-Price Sealed-Bid or an English auction). In this case, the community essentially works as an auction house in which the members bid for executing instance of the. Community-specific policies (e.g., selection based upon customer profiles, provider reliabilities, time constraints) are also supported.

2.2 Composite service execution model and architecture

An important feature of Self-Serv's execution model is *self-coordination*, whereby the responsibility of coordinating the execution a composite service, is distributed across several *state coordinators* (or *coordinators* in short). In other words, the execution of a composite service is not dependent on a central scheduler as in previous proposals (e.g. EFlow [5] and CMI [19]), but rather on software components hosted by each of the providers participating in a service composition. This software components interact in a peer-to-peer fashion in order to ensure that each instance of a composite service is executed in accordance with its control-flow and its data-flow specifications

The execution of an instance of a composite service is initiated when its *wrapper* sends an activation message to the coordinators of the initial states of the composite service. Subsequently, the state coordinators of the composite service interact among them in a peer-to-peer way to carry out the execution. Eventually, the coordinators of the final states of the composite service send their notifications of completion back to the wrapper, thereby signalling the completion of the overall execution.

State coordinators also collaborate among them in order to collect the execution trace of a composite service instance, a feature that we call *self-traceability*. Roughly speaking, as the coordinators exchange notifications between them in order to execute an instance of a composite service, they also exchange partial traces of this execution. When an execution of a composite service is completed, these partial traces are sent to the service's wrapper, that builds a full trace of the execution and stores it in a repository.

The wrappers and the state coordinators of a composite service are generated and deployed by a *service description module* provided by the Self-Serv system. This module acts as a compiler of the statechart describing a composite service. Specifically, the composite service designer (or service composer) assembles service offers advertised in a repository of services, through the service description module. This module generates and deploys the required state coordinators and wrapper. Once the service is deployed and assigned to a provider, users can invoke it through its wrapper. The wrapper maintains a repository of traces, that is made accessible to the service administrator.

A partial view of the architecture of the Self-Serv system for service composition and execution is depicted in Figure 3.

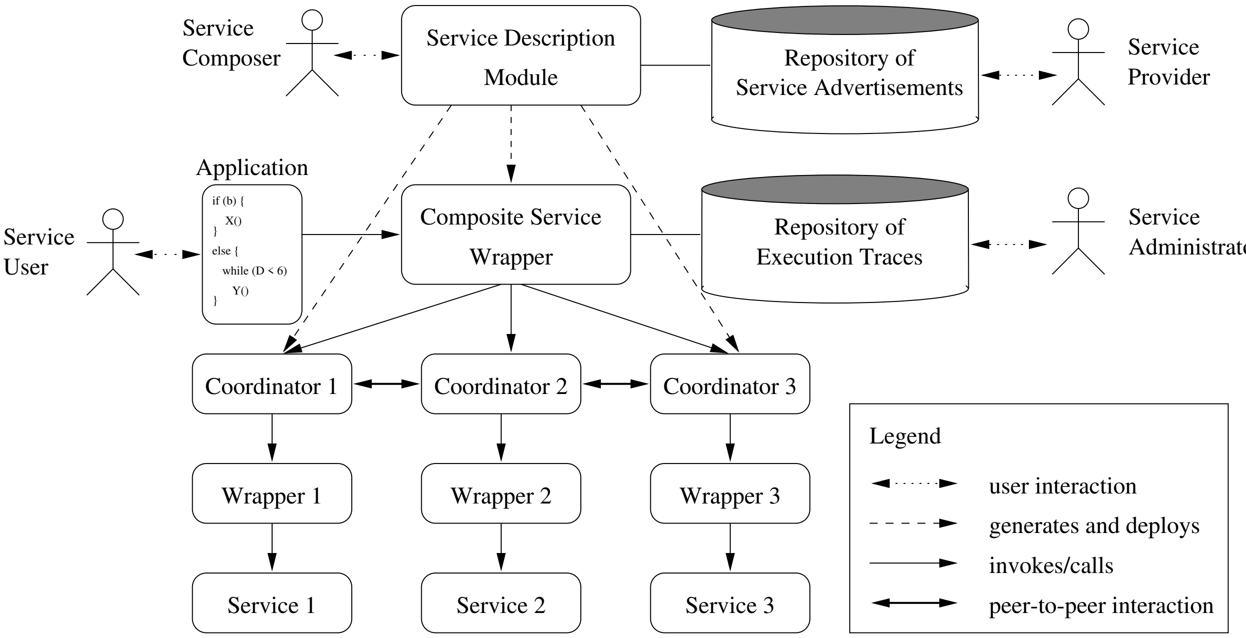


Figure 3: Architecture for composite service management

In the next three section we sequentially detail Self-Serv's model for service specification, service execution and service tracing.

3 Composite service description

This section provides a rationale for using statecharts as the foundation of a composite service specification language, and delineates the subset of statecharts that is used for this purpose in Self-Serv.

3.1 Choice of a language for composite service specification

A natural way for describing the control-flow of composite services, is to adapt to this purpose an existing process-modeling language, and especially one of those which have proven to be suitable for workflow specification. There are numerous workflow specification languages based upon different paradigms. In fact, each commercial Workflow Management System (WfMS) implements its own specification language, with little effort being done to provide some degree of uniformity between products. In this respect, the Workflow Management Coalition (WfMC) [7] has defined a set of glossaries and notations that encompass many of the concepts and constructs provided by existing workflow specification languages. Unfortunately, these efforts have had a very limited impact. To add to the lack of uniformity, most of the existing workflow specification languages, including the one defined by the WfMC, lack a formal semantics, making it difficult to compare their capabilities and expressiveness in order to make an objective choice between them [2].

The use of formal notations for workflow specification has been considered in e.g. [1] and [17]. [1] discusses several advantages of using Petri-nets for describing the control-flow perspective of workflows, among which their expressive power. However, many designers find the Petri-net formalism difficult to grasp. Moreover, Petri-nets do not provide any means for structuring a specification into blocks. As a tradeoff between expressiveness on the one hand, and ease of use and modularity on the other, [17] advocates the use of statecharts instead. The main argument is that statecharts are based upon finite automata and Event-Condition-Action (ECA) rules, two paradigms which are easy to comprehend. In addition, statecharts can be coupled with activity charts [9] so as to describe data-flow in addition to control-flow. Finally, the statechart formalism has been integrated into the Unified Modeling Language (UML) [18], as the foundation of many intra and inter-object process modeling constructs.

Based on the above arguments, we choose to express the control-flow perspective of composite services using a subset of statecharts. Still, our results can be adapted to other process specification languages, as long as they can be mapped into this subset of statecharts.

3.2 Statecharts for composite service specification

A statechart is made up of states connected by transitions. Transitions are labelled by ECA rules. The occurrence of an event fires a transition if (i) the machine is in the source state of the transition, (ii) the type of the event occurrence matches the event description attached to the transition, and (iii) the condition of the transition holds. When a transition fires, its action part is executed and its target state is entered.

An event occurrence can be the reception of a signal, or a change in the system's clock (i.e. timeouts). The event, condition, and action parts of a transition are all optional. A transition without an event is said to be *triggerless*.

The set of outgoing transitions of a state must be exclusive, that is, for a given event, only one of them may fire. This entails that if there are more than one transition with the same source state and event specification, their conditions should be disjoint.

States can be simple or compound. In our approach, a simple state corresponds to the execution of a service, whether elementary or composite. Accordingly, each simple state is labelled by a description of a service offer. A service offer (i.e. an instance of the class "Service") specifies the set of parameters that are needed to invoke the service, and the provider which is responsible for executing it. In order to support dynamic provider selection, the provider can be either an individual entity or a community of entities. When a service offer is attached to a community of entities, the choice of the actual entity that will execute the service is delayed until run-time. This discussion is summarised in Figure 10, appendix B.

When a basic state is entered, the service that labels it is invoked. The state is normally exited through one of its triggerless transitions, when the execution of the service is completed. If the state has outgoing transitions labelled with events, an occurrence of one of these events provokes the state to be exited, even if the corresponding service execution is ongoing. In this case, the execution of the service is cancelled.

Compound states on the other hand, are not directly labelled by a service invocation. Instead, they contain one or several entire statecharts within them, thereby providing a decomposition mechanism. There are two kinds of compound states: OR-states and AND-states. An OR-state contains a single statechart, while an AND-state contains several statecharts which are intended to be executed concurrently. Each of these statecharts is called a *concurrent region*. The above definitions apply recursively, that is, the sub-states of a compound state may themselves be decomposed either as OR-states or as AND-states.

When a composite state is entered, its initial state(s) become(s) active. The execution of a composite state is considered to be completed when it reaches (all) its final state(s). The initial and the final states of a compound state are pseudo-states: they are not labelled by any service invocation.

Composition is denoted by physically embedding a statechart within a state of another statechart. In the case of AND-states, concurrent regions are separated by a dashed line. Initial states are denoted by filled circles, whereas final states are denoted by two concentric circles: one filled and the other unfilled, as in figure 1, page 6.

The use of events in a composite service specification is illustrated in 4. In this example, the execution starts with a compound state with two concurrent regions being entered. At this point, service S3, is executed in parallel with service S1 and S2, which are executed one after the other. If during these executions an occurrence of event E is received, the following actions are undertaken:

- The compound state encompassing S1, S2 and S3 is exited.
- The ongoing executions of services S1, S2 and S3 are interrupted.
- Action A is performed (this may be a notification sent to the user of the composite service).
- The final state is entered.

If on the other hand the execution of the above compound state completes normally, another compound state with a single region is entered. Depending on the value of condition C, either service S4 or S5 is executed, and once this execution is completed, the compound state is exited and the final state is entered.

Statecharts provide many other constructs than those discussed above [9]. However, we focus on the above subset since it is not our intention to fully support the statechart notation, but rather to cover the basic control-flow constructs found in most workflow specification languages, such as branching, loops, forks, joins, and exceptions.

4 Self-coordination

As discussed in section 2, the coordination between the constituents of a composite service, is ensured through peer-to-peer collaboration between several state coordinators. This approach clearly provides greater scalability and availability than a centralised one where the execution of a service depends on a central scheduler.

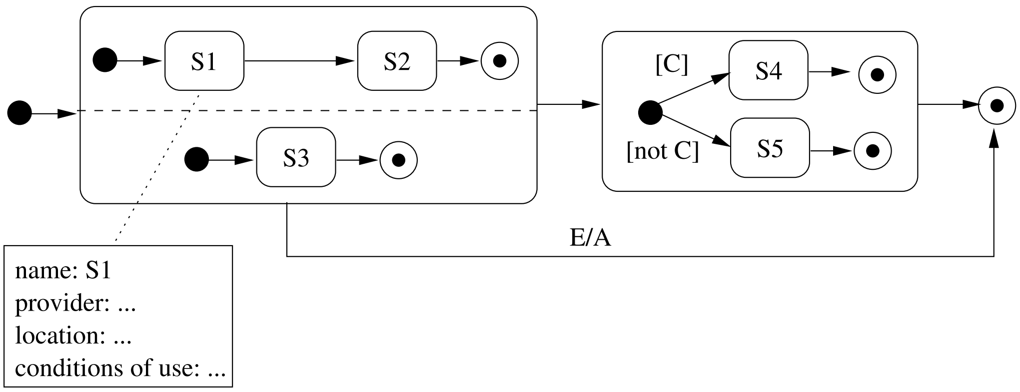


Figure 4: Sketch of a composite service description

In this section, we discuss what is the role of the state coordinators, and how they can be generated from the specification of a composite service as a statechart.

4.1 State coordinators

The provider[2] of a service labelling a state ST in a composite service specification, should host a coordinator which is responsible for:

- Initiating the execution of the service labelling ST whenever all the preconditions are met.
- Notifying the completion of this execution to the coordinators of the states which potentially need to be entered next.
- While state ST is active, receive notifications of external events, determine if ST should be exited because of these event occurrences, and if so, interrupt the service execution if it is ongoing, and notify the interruption to the coordinators of the states which potentially need to be entered next.

In other words, the coordinator of a state is a lightweight scheduler which determines when should a state belonging to a statechart be entered, and what should be done when it is exited or when a signal is received while the state is active.

To illustrate how the state coordinators are deployed, and how they interact among them and with the wrapper of the composite service to which they belong, we consider the example of the “Travel Solutions” service described in Figure 1. We assume that this composite service is provided by a company named “Full Tours”. The life-cycle of the service starts when a service designer within this company defines the structure of the service using the service composition module of the Self-Serv architecture (see Figure 2.2). This module generates a set of software components that implement the wrapper and the state coordinators required to run the composite service. It also assists the service designer in deploying the wrapper in one of the servers of the company “Full Tours”, and the state coordinators in the dedicated servers provided by the companies referenced in the composite service definition. Once the deployment is completed, the composite service is

[2]The discussion in this section applies whether the provider is an individual or a community.

advertised. This advertisement contains, among other things, the name of the server where the wrapper is deployed, and the details on how to invoke the service (i.e. its interface).

When the wrapper receives a request for executing the “Travel Solutions” service, it sends a message to the coordinators of the states labeled FB and AS (see figure 1). Upon reception of these messages, these coordinators invoke the services labeling their states. When the service that books a flight completes its execution, the coordinator of the state FB sends a message to that of the state AB. This latter invokes the service that books an accommodation, waits for its completion, and sends a message to the coordinators of the states labeled CB and BB. In the meanwhile, the coordinator of AS sends its completion message to the coordinators of CB and BB too. These completion messages contain the data that have to be exchanged between these services, as per the data exchange perspective of the “Travel Solutions” specification (which is not detailed in this paper). Using these data, the coordinators of BB and CB evaluate the condition “attractions near accommodation” which appears in the labels of their incoming transitions, and accordingly, they decide which state has to be entered. Assuming that the attractions are far from the accommodation, it is the state CB that has to be entered, so the corresponding coordinator invokes the service for renting a car. Once this service completes its execution, this same coordinator sends a message to the coordinator of the state EP, who sends a execution request to the wrapper of the composite service responsible for searching events. This wrapper initiates the execution of the service that it provides by sending a message to the coordinator of the state ES, which invokes the service that searches events, wait for its completion, and assuming that the tickets for some of the events need to be prepurchased, sends a message to the coordinator of the state TP. This coordinator then invokes the service that purchases tickets, and upon completion, it sends a notification to the wrapper of the Event Planning service, which in turn sends a notification to the coordinator of the state EP. Finally, this coordinator of EP sends a message to the wrapper of the “Travel Solutions” service, thereby concluding the overall execution.

4.2 Preconditions and postprocessings

The example above shows that building the coordinators from a specification of a composite service as a statechart, is a non-trivial problem. Specifically, it involves answering the following questions:

- What are the preconditions for entering a state?
- When the service execution associated to a state is completed (whether successfully or because an event occurrence provoked its cancellation), what are the states that may potentially need to be entered next? The process by which a coordinator sends notifications of completion to other coordinators after the service invocation that labels its state has completed, is subsequently called *postprocessing*.

The behaviour of a state coordinator can therefore be captured using two tables: one containing a set of *preconditions* such that the state is entered when one of these preconditions is met, and another containing a set of *postprocessing actions* that indicate which coordinators need to be notified about the fact that a state is being exited. In the following we show how to represent the elements of these tables.

Definition 1 (*Preconditions table of a state*). The preconditions for entering a state ST are represented by a table with two attributes (namely “event” and “action”) whose elements denote rules of the form E[C] such that:

- E is a conjunction of events of the form ready(ST), meaning that a notification of completion has been received from the coordinator attached to state ST. The conjunction of two events e1 and e2 is noted e1 $\wedge$e2 and the semantics is that if an occurrence of e1 and an occurrence of e2 are registered in any order, then an occurrence of e1 $\wedge$ e2 is generated.

- C is a conjunction of conditions appearing in the labels of the statechart's transitions.

□

When one of the elements of the preconditions table is triggered, and that its condition evaluates to true, the state is entered, and the service that appears on its label is invoked.

The following examples of preconditions tables relate to the example of Figure 1, page 6.

- The preconditions of state EP are { ready(CB)[true], ready(BB)[true] }, meaning that the state is entered when a message is received from either the coordinator of the state CB or that of the state BB.

- The only precondition of state CB is { ready(AB) $\wedge$ ready(AS)[not attractions near accommodation] }.

Definition 2 (*Postprocessings table of a state*). The notification actions (or postprocessings) that have to be undertaken when a state is exited are presented as a table with two attributes (namely "condition" and "action") whose elements denote rules of the form [C]/A such that:

- C is a conjunction of conditions appearing in the labels of the statechart's transitions.

- A is a term of the form notify(ST), meaning that a notification of completion has to be sent to the coordinator associated to the state ST.

□

When a service labelling a state completes its execution, the coordinator of this state evaluates each of the entries appearing in its postprocessing specification. For each entry whose condition evaluates to true, the corresponding notification action is undertaken.

The following examples of postprocessings tables relate to the example of Figure 1, page 6.

- The postprocessing actions of state ES are { [need pre-purchase]/notify(TP), [not need pre-purchase]/notify(wrapper) }.

- The postprocessing actions of state AS are { [true]/notify(CB), [true]/notify(BB) }. Notice that the condition "attractions near accommodation" is not evaluated before undertaking the postprocessing action notify(BB). This is because evaluating this condition requires the coordinator to know where is the selected accommodation located, and this is only known once the accommodation booking is completed.

4.3 Generating the preconditions and postprocessings tables

In order to derive the preconditions table of a state, its incoming transitions are analysed, and one or several preconditions are generated for each of these transitions. Similarly the postprocessing table of a state is generated by analysing its outgoing transitions.

Let us discuss for instance how an outgoing transition is used to generate a set of postprocessing actions. The simplest case is that when this transition leads to a basic state (say ST), and it is labelled with a condition C. The postprocessing action "[C]/notify(ST)" is included in the table of postprocessings, meaning that if condition C is true, a notification must be sent to the coordinator of state ST.

If an outgoing transition points to a compound state CS, then one postprocessing action is generated for each of the initial transitions of CST. This process is carried out recursively, that is, if one of these initial transitions points to another compond state CS', then one postprocessing action is generated for each initial transition in CS', and so on.

More complex cases arise when the outgoing transition points to a final state of a compound state (say CST). In this case, each of the outgoing transitions of CST is considered in turn, and one or several postprocessing actions are generated from it, depending on whether this transition points to a basic state or to another composite state.

The most delicate case is that where an AND-state has to be exited, and immediately after, another AND-state has to be entered, as illustrated in Figure 5. The postprocessing of state S1 in example is { [C1]notify(S2), [not C1]notify(S4), [not C1]notify(S5), [not C1]notify(S6) }. Notice that S1 cannot evaluate conditions C2 and C3, since they may involve data obtained from S3, which is not necessarily completed at the time that S1 does. On the other hand, the preconditions for entering the state S4 are { ready(S1) $\wedge$ ready(S3) }[C2 and C3], { ready(S2) $\wedge$ ready(S3)[C2 and C3] }.

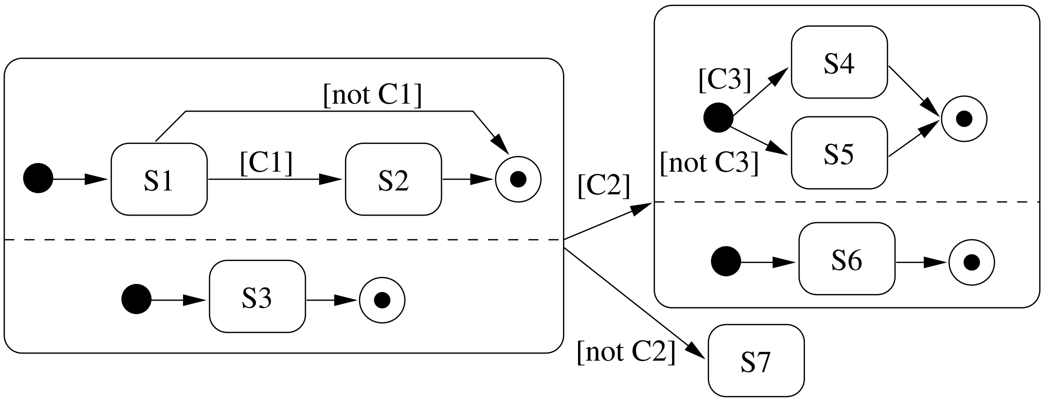


Figure 5: A composite service showing a situation where an AND-state is directly connected to another AND-state.

Appendix A details the recursive algorithms for deriving the preconditions and the postprocessing for each state of a composite service specification.

5 Self-traceability

Self-Serv provides mechanisms for keeping trace of past executions of composite services, and for querying these traces. This functionality is essential for customer support and feedback (i.e. retrieving a given service execution) as well as for detecting deficiencies in the constitution of a composite service (i.e. analysing the past executions of a service in order to retrieve repetitive malfunctionings).

In this section, we propose a model for a database of execution traces, and we discuss how such a database can be populated with an acceptable overhead, that is, how the traces of a composite service's executions can be collected and stored, in a way compatible with the peer-to-peer execution model of Self-Serv. We also present some useful queries over a database of execution queries.

5.1 Modeling service execution traces

Simplifying assumptions. For the sake of simplicity, we assume in the sequel that the local coordinators and the wrappers of a composite service, share a common time line. This can be achieved using classical clock synchronisation protocols such as NTP [11].

We also assume that all temporal values (instants, durations and intervals), are expressed at the same level of granularity (e.g., the Second or the Minute). Under this assumption, instants and durations are unambiguously designed using integers, while an interval is fully represented as a pair of integers corresponding to its bounds.

Life-cycle of a service instance. At a given instant, an instance of a service can be in one of the following statuses: *running*, *frozen*, *completed*, *cancelled* and so on. The life-cycles of a service instances are controlled by a statechart which describes possible statuses and allowed transitions between them. Transitions are only labelled by events. Life-cycle statecharts are hosted by wrappers and may be customised in order to capture particularities of the service. This customisation is operated by the "service composer".

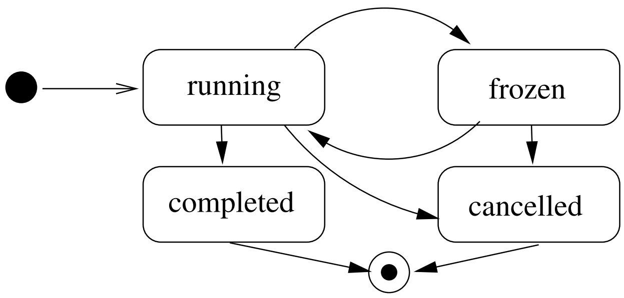


Figure 6: A statechart modeling the life-cycle of a service instance.

As an example, we show in figure 6 a statechart which applies to all of the services involved in "Travel Solutions" service. When an instance of a service is created, it enters the running state. Whilst on this state, the service can be frozen (i.e. suspended) due to an external request, or because a resource required for the service execution is temporarily unavailable. From the frozen state, the service instance can subsequently move back to the running state, and from the running state, it can move to the completed state. A service instance can also move from either the running or the frozen state, to the cancelled one, due to an external event.

Status history. A status history is a trace of the life-cycle of a service instance, that is, the statuses through which this instance went through, and the times of the transitions. At an abstract level a status history is defined as a function from a set of instants to a set of status values. At a concrete level a status history can be effectively represented by an ordered set of interval-timestamped statuses.

Service execution. A service execution models the information about a particular service instance that is made persistent by the wrapper of the service after the instance has been executed (i.e. after it has attained its "completed" or its "cancelled" state). Concretely, a service execution is composed of (i) a status history, (ii) a set of effective input and output parameters, and (iii) the individual provider to whom the instance's execution was assigned. This last information is essential when the provider specification of a service refers to a community.

In addition to the above three properties, a composite service execution is associated with the set of other service executions that it triggered. For example, if a composite service CS involves the execution of two services S1 and S2 one after the other, then each of the service executions of CS is associated with a service execution of S1, and a service execution of S2.

The UML class diagram in figure 11 (appendix B) summarises the above discussion.

5.2 Collecting and storing execution traces

The responsibility of tracing the executions of a composite service is distributed across the wrapper and the local coordinators of this service. This feature is referred to as *self-traceability*.

The execution trace of a composite service CS is modeled by an instance of the class ServiceExecution defined in figure 11, appendix B). The coordinators of the states belonging to the statechart describing CS, is responsible for:

- Receiving information about ongoing executions of CS from other coordinators, in the form of a set of objects of the class ServiceExecution.
- Gathering information about the service instantiations that it performs, and encapsulating this information into objects of the class ServiceExecution.
- When the state is exited, passing the instances of class ServiceExecution that it has collected, to the coordinators of the states that need to be entered next, or to the wrapper of CS if no state needs to be entered next.

More precisely, when an instance of a composite service starts its execution, the wrapper sends this instance identifier to the coordinators of the states that need to be entered the first. Each of these coordinators performs the service invocation that appears in the label of its state. It then collects information about the parameters passed, the start and end time of the execution induced by this invocation, and the identity of the *individual* provider that carried out this execution (in the case where the service is offered by a community). With this information, it creates an object of the class ServiceExecution, and it passes this object to the coordinator(s) of the state that needs to be executed the next (which is determined using the postprocessing table as discussed in section 4). The coordinator(s) to which this execution is passed, perform a similar operation. On the end, the coordinator(s) of the final states of the composite service, pass(es) its/their trace data to the wrapper, which stores these data in a repository. When users wish to query the traces of a composite service, they send their queries to the wrapper.

As an optimisation aiming at reducing the number of messages exchanged for collecting traces, when an AND-state needs to be entered, the data about the execution trace *before* entering this state is not sent to all the initial states of all of the concurrent threads, but rather to the coordinators of the states that will be entered after

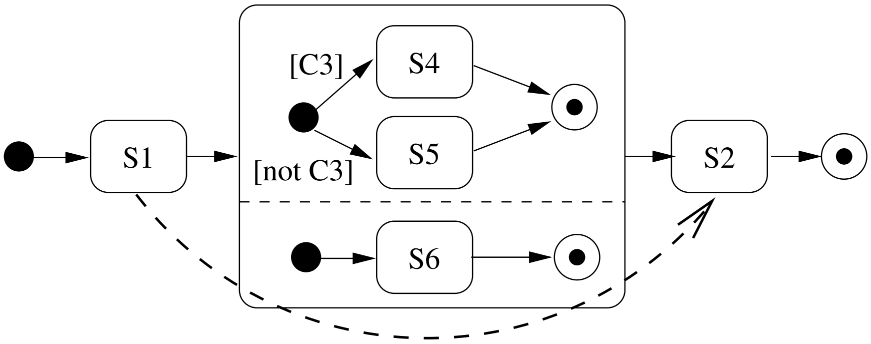


Figure 7: S1 sends its partial trace to S2

the AND-state is exited. Indeed, duplicating this partial execution trace is useless, since anyway when this AND-state will be exited, the traces collected by all its threads will be merged. Let us consider the composite service described by the statechart depicted in figure 7. When S1 finishes, instead of passing the partial trace to each initial state of the AND-state, it sends it straight to S2.

5.3 Back to the example

To illustrate how traces are collected, let us consider the composite service "Travel Solutions" described in Figure 1. We show in Figure 8 one execution of the service "Events Planning" (EP) and two executions of the service "Travel Solutions" (TS) together with the two executions of service EP that they generate. We only show the sequence of statuses through which a service instance goes during its execution, thereby omitting details about their effective parameters and their effective providers. Without loss of generality, we assume that the execution TS_e1 starts at instant 1.

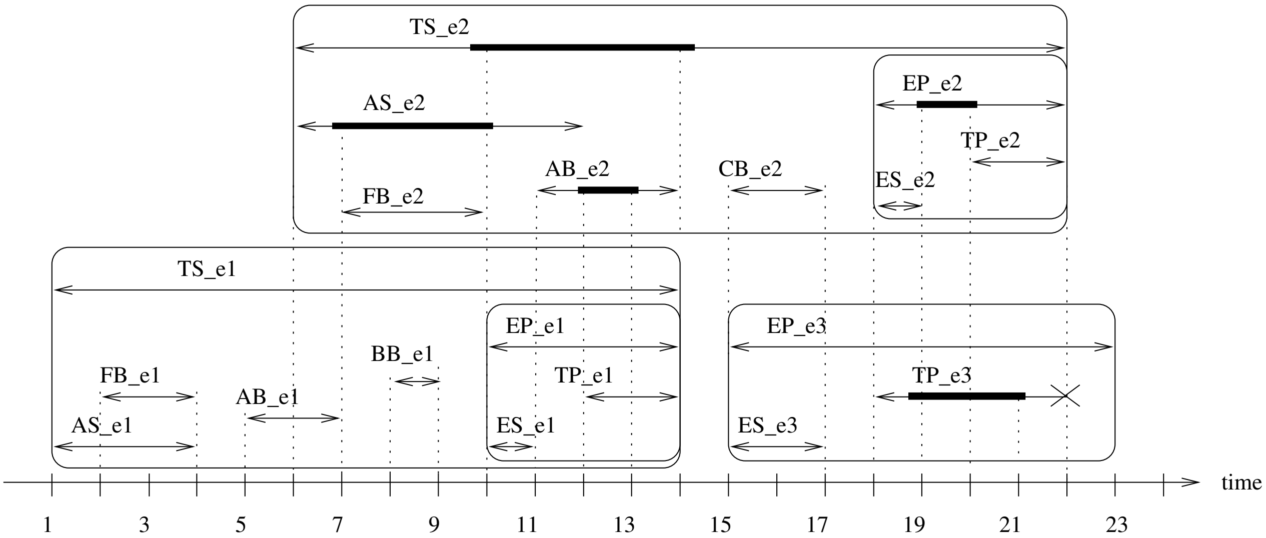


Figure 8: Execution details of two instances of the service "Travel Solutions" and three instances of "Events Planning".

The following notations are used in Figure 8:

- TS_e1 denotes a particular execution of the service TS, AB_e1 a particular execution of service "Accommodation Booking" (AB), and so on. We assume that AS, FB, AB, CB, BB, ES and TP are elementary services. For each service, the suffix _ei ($1 \leq i \leq 3$) is added to its name, in order to denote each of its executions.
- A double-arrow $\longleftrightarrow$ is used to denote the interval during which the associated service was running, and finally completed. A thicker part of the arrow means that during the underlying period, the service is frozen.
- A broken double-arrow $\longleftarrow\!\!\times$ means that the service was running, possibly frozen then unfrozen and finally cancelled. The execution of the service TS, denoted TS_e3, was cancelled. In spite of that, the execution EP_e3 was completed (the cancellation was handled by the coordinator).

A detailed view of the messages exchanged during the execution of TS_e2 is given in table 1. Each line in the table contains the time at which a message was sent, the sender and the recipient, and the contents of the message.

Time	From coordinator of	To coordinator of	Content
10	FB	AB	{FB_e2}
12	AS	BB and CB	{AS_e2}
14	AB	BB and CB	{FB_e2, AB_e2}
17	CB	EP	{FB_e2, AB_e2} $\cup$ {AS_e2}
22	EP	wrapper(TS)	{FB_e2, AB_e2, AS_e2} $\cup$ {EP_e2}

Table 1: Messages between coordinators during the execution of TS_e2.

For the sake of simplicity, we assume that there is no delay between the moment when a service finishes and the moment when the associated coordinator sends the partial trace to the next one. A symbol of the form X_e2 (X $\in$ {FB, AS, AB, CB, EP}) denotes an instance of the class ServiceExecution, that describes an execution of service X). X_e2 is created by X's coordinator at the beginning of X's execution. Coordinators are responsible for maintaining status_Hist attribute's value of the instance to which they are associated.

The 2nd and 3rd lines of the table 1 can be read as follows. At time 12 (resp. 14) AS's coordinator (resp. AB) sends the partial trace to both BB's coordinator and CB's coordinator. Because the boolean expression [Attractions near from accommodation] is evaluated to false, BB is not required to be executed, so the coordinator of the state that it labels discards the partial traces that were sent to it by the coordinators of AS and AB.

Table 2 shows the value of object TS_e2, which describes the execution of an instance of composite service TS. The notation [i..j] ($i \leq j$) denotes an interval containing all the instants greater than i and less than j. The ordered sequence of pairs [$\langle$[6..6],running$\rangle$, $\langle$[7..10],frozen$\rangle$, $\langle$[11..11],running$\rangle$, $\langle$[12..12],completed$\rangle$], can be read as follows: during the interval [6..6] (which contains only one instant), the service was running; then during [7..10] it was frozen, and so on, until finally, it was completed at instant 12.

Identifier	TS_e2	
Status Hist.	[⟨ [6..9],running ⟩,⟨ [10..14],frozen ⟩, ⟨ [15..21],running ⟩,⟨ [22..22],completed ⟩]	
Triggered Constituents	**Ident.**	**Status Hist.**
	AS_e2	[⟨ [6..6],running ⟩, ⟨ [7..10],frozen ⟩, ⟨ [11..11],running ⟩, ⟨ [12..12],completed ⟩]
	FB_e2	[⟨ [7..9],running ⟩, ⟨ [10..10],completed ⟩]
	AB_e2	[⟨ [11..11],running ⟩, ⟨ [12..13],frozen ⟩, ⟨ [14..14],completed ⟩]
	CB_e2	[⟨ [15..16],running ⟩, ⟨ [17..17],completed ⟩]
	EP_e2	[⟨ [18..18],running ⟩, ⟨ [19..20],frozen ⟩, ⟨ [21..21],running ⟩, ⟨ [22..22],completed ⟩]

Table 2: Trace details of an execution of the service “Travel Solutions” (TS_e2).

To summarise, we can say that the execution trace of a composite service is collected incrementally through peer-to-peer collaboration between the local coordinators, and at the end of this execution, it is stored in a repository managed by the wrapper. One may argue that storing the traces in a single repository is contrary to Self-Serv’s philosophy of distributing the responsibility of managing a composite service among several nodes. Instead, one could think of an alternative approach in which each of the local coordinators (which, we recall, are hosted by the providers of the constituent services) stores the objects of the class ServiceExecution that it creates. The wrapper would then act as a query mediator: upon receiving a query from a user, it would decompose it into smaller queries, submit these smaller queries to each service provider, and merge their answers into a single one that would be sent back to the user.

However, this “distributed storage” approach has the following inconvenients over the “centralised storage” approach:

- Overhead: to evaluate a query over a composite service’s executions, the wrapper has to send a message to each of the providers of the constituent services. This leads to a network overhead, and potentially to a poor response time. Indeed, the time for answering a query is proportional to the slowest of the connections with the providers.
- Information unavailability: if one of the providers is temporarily unavailable, the query cannot be answered fully. Worst, let us consider a provider who does not want to participate anymore in a service composition. We can expect that he is not willing to maintain data about the constituent service executions in which he has participated, so he destroys these data. Subsequently, it is no longer possible to fully reconstitute the traces of the composite service executions in which this provider participated.

Moreover, since the wrapper is hosted by the provider of the composite service, it is quite normal that it is this entity that manages the repository of service execution traces.

5.4 Queries on execution traces

We list below some potentially useful queries over a database of execution traces of service TS. For each of them, we give the result (following symbol →) assuming that the database only contains the execution traces depicted in Figure 8 and Table 2. Depending on the data model supported by the DBMS chosen for storing execution traces, these queries may be expressed in either SQL, OQL, XQuery, or a temporal extension of these languages.

Q.1: **Domain and range restriction**

For each execution of the service "Travel Solutions", give its identifier and its status history when at least one of its constituent is "frozen".

→ The result is a bag of couples where the first component is a service execution object identifier and the second one is a status history object identifier; the value is (in this case, the bag is a singleton): $\{\langle$ TS_e2, [$\langle$ [8..9], running $\rangle$, $\langle$ [10], frozen $\rangle$, $\langle$ [12..13], frozen $\rangle$, $\langle$ [19..20], running $\rangle$] $\rangle\}$

Q.2: **aggregation on execution duration**

Within the executions of the service "Travel Solutions", retrieve the service constituents whose execution takes the most time (in average).

→ EP (it takes 5.5 in average)

Q.3: **aggregation on the number of invocations**

Retrieve the elementary service(s) within the executions of the service "Travel Solutions" that are invoked the less frequently (resp. the most frequently).

→ BB and CB

Q.4: **reasoning about succession in time**

Retrieve the first time when an execution of the service "Travel Solutions" was "cancelled".

→ 22

Q.5: **pattern matching on sequences of statuses**

Retrieve the elementary services whose execution had the status "running" then "frozen" during more than 30mn then finally "cancelled".

→ TP

6 Related work

The issue of service composition, and the related field of inter-organisational workflows, have been the subject of intensive developments in the last few years. In this section, we focus on those efforts dealing with the aspects addressed in this paper, that is, coordination between services, provider selection, and execution trace collection.

eFlow [5] is a platform for specifying, enacting, and monitoring composite services. The execution model is based on a centralised process engine, which is responsible for scheduling, dispatching, and controlling the execution of the composite services. Clearly, this centralised process engine represents a potential bottleneck. The same remark applies to a similar platform called the Collaboration Management Interface (CMI) [19]. Both E-Flow and CMI support dynamic provider selection, although the concept of "community" which is the basis of Self-Serv's provider selection approach, is not explicitly supported. This concept of community is described in the WebBIS proposal [3], where it is called *push community*. Another difference between eFlow and CMI on the one hand, and Self-Serv on the other, is that the composite service description language of Self-Serv is based on a well-known process-modeling formalism (statecharts), whereas eFlow and CMI rely on proprietary notations.

ADEPT [12] is a multi-agent platform designed to support inter-organisational business process definition and enactment. In ADEPT, a workflow can be recursively decomposed into smaller sub-workflows, leading to a tree-like structure similar to the one induced by the relationship between composite services and their constituents in Self-Serv. Each sub-workflow in ADEPT, is assigned to an autonomous agent. When the agent responsible for a workflow, wishes to invoke a sub-workflow, it has to negotiate with the agent(s) which provide it. This negotiation is done using a 1 to 1 negotiation protocol. In contrast, thanks to the concept of community, the selection of a provider for a given service in Self-Serv, can be performed using a 1 to N or an N to M negotiation protocol such as an auction or a double auction, thereby leading to more efficient allocation mechanisms.

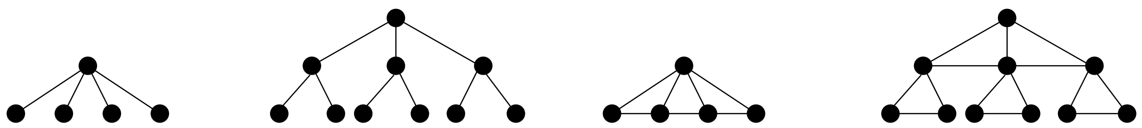
Self-Serv's execution model has some similarities to that of the Collaborative Process Manager (CPM) [6]. CPM supports the execution of inter-organisational business processes through peer-to-peer collaboration between a set of workflow engines, each representing a player in the overall process. An engine representing a player P, schedules, dispatches and controls, the tasks of the sub-process that P is responsible for. In order to support dynamic changes in a business process, such as replacing one of the players, the communication layer is based on a dynamic software agent platform called E-carry. The major difference between CPM and Self-Serv, is that in CPM, the number of messages exchanged between the players is not optimised. Instead, each time that a process terminates a given task, it must send a notification to all the other players. Hence, if a business process execution involves N tasks and M players, its execution requires the exchange of N $\times$ M messages: far more than it is actually required. Moreover, CPM requires that all the players participating in an inter-organisational process deploy the same workflow engine since they all need to interpret the "global" workflow specification. Meanwhile, in Self-Serv the providers are free to choose any means for implementing their services since the coordination between entities is handled through dedicated software components (i.e. the state coordinators) which are generated by the composite service specification module (see section 2.2).

Our coordination approach has some similarities with that developed in the context of the Mentor project [17], although the scope of this latter proposal is that of intra-organisational workflows. Specifically, the problem addressed in [17] is that of distributing the execution of workflows expressed as state and activity charts. The idea is to partition the overall workflow specification into several sub-workflows, each encompassing all the activities that are to be executed by a given entity within an organisation (assuming that this information is statically known). Each of these sub-workflows is itself specified as a statechart, made up of several concurrent regions. The authors prove that their decomposition technique preserves the semantics of the original workflow specification. In addition, some optimisation techniques are developed, that reduce the number and the size of the messages exchanged by the sub-workflows, leading to a "weak synchronisation" model close to that of Self-Serv. MENTOR's approach differs from Self-Serv's, in that it is only applicable when the assignment of activities to their executing entities is known at the definition of the workflow, which is a quite restrictive assumption in the context of service composition. Moreover, as in CPM, MENTOR imposes that each organisation participating in a distributed workflow deploys a full-fledged execution engine, capable of interpreting state and activity charts.

From the perspective of the coordination model, the approaches discussed above can be classified into four main categories:

- The flat centralised approach (e.g. [5] and [19]) in which the constituents of a composite service, are coordinated by a centralised scheduler. The coordination model behind this approach is depicted in figure 9(a), which shows a single node (the provider of the composite service) linked to several other nodes (the providers of the constituents).

- The hierarchically decomposed centralised approach (e.g. [12]), is similar to the previous one, except that the model explicitly takes into account delegation: a service provider SP may delegate the execution of part of its service to other providers (say SP1 and SP2). Still, SP1 and SP2 do not communicate directly among them, but only through SP. The above process is carried out recursively: SP1 and SP2 may delegate the execution of their services to other providers, thereby leading to a tree-structure as depicted in 9(b).
- The flat distributed approach (e.g. [6] and [17]) in which the entities participating in a composite service execution coordinate through peer-to-peer communitication as depicted in figure 9(c).
- The hierarchically decomposed distributed approach (Self-Serv being the only representative that we know) which is similar to the previous one, except that a constituent of a composite service may itself be a composite service, thereby leading to a tree-structure in which siblings are inter-connected as shown in figure 9(d).



(a) Flat centralised (b) Hierarchical centralised (c) Flat distributed (d) Hierarchical distributed

Figure 9: Coordination models for composite service execution

Regarding service execution tracing, we are not aware of any previous concrete proposal, except for [13] and [16] which address a similar issue: that of tracing the executions of a workflow. [13] assumes that the workflows are executed in a distributed environment, and that each node within this environment (in our context: each provider), maintains the history of its task executions (in our context: its service executions). Within this context, the authors present several strategies for evaluating queries such as “retrieve the history of a given process instance”. Contrarily to our proposal, the set of entities participating in the execution of a workflow is assumed to be fixed. Our approach also differs from the above one, in that the traces are collected in a distributed way, but they are stored by a single entity (the entity that hosts the wrapper), thereby avoiding the burden of having to contact all the entities which participate in a composite service, each time that a user wishes to query its execution traces.

In [16] the context is that of centralised workflows expressed as state-charts. The authors focus on demonstrating that the process of tracing a workflow’s execution can itself be seen as a workflow. Consequently, by merging a workflow **W**, with the workflow dedicated to maintaining the history **W**’s executions, one obtains a “self-traceable workflow”. Contrarily to our proposal however, [16] does not discuss how these results can be extended to a distributed and inter-organisational context.

Finally, the issue of providing a transactional layer for inter-organisational workflows and composite service execution, which is skirted in this paper, is currently being studied in several projects, among which WISE [14].

7 Conclusion

In this paper, we presented an approach to model service composition, in which a composite service is defined as an aggregation of other composite and elementary services, whose dependencies are described through a statechart. The provider of a service, whether elementary or composite, can be either a well-identified entity, or a community of entities. In this latter case, the choice of the individual entity within the community which is in charge of executing a given instance of the service, is delayed until run-time, thereby catering for dynamic provider selection.

We then proposed an execution model for composite services, in which the providers of the services participating in a composition, collaborate in a peer-to-peer fashion in order to ensure that the control-flow dependencies expressed by the schema of the composite service are respected. Specifically, the responsibility of coordinating the providers participating in a composite service execution, is distributed across several lightweight software components hosted by the providers themselves. In this way, the execution of a composite service is not dependent on a central scheduler, which could constitute a potential bottleneck.

The above collaboration model has been extended so that the state coordinators are able to incrementally collect the execution trace of each composite service instance. These traces are then sent to an entity known as the wrapper of the composite service, which stores them in a repository and makes them accessible to the service administrator. These traces are particularly useful for customer feedback, and for detecting malfunctionings in the constitution of a composite service.

We are currently designing an implementation of Self-Serv in which the wrappers and the state coordinators generated by the service description module are packaged as Enterprise JavaBeans (EJB) [21] instances, which interact through a communication layer based on the Simple Object Access Protocol (SOAP) [20]. We believe that distributed components is a suitable technology for implementing the state coordinators, since this technology is specifically designed to support the middle-tier within a 3-tier architecture, and the state coordinators can be precisely seen as mediators between the users of a composite service, and the providers of its constituents. In addition, distributed components platforms offer many system facilities such as naming services, transaction management, recovery, etc., and they are designed so as to support platform independence and rapid deployment.

Our next step will be to examine how changes in the constitution of a composite service (i.e. modifications of its schema), can be handled by the service description module, in such a way that the running instances of the service are not affected by this modification.

References

[1] W.M.P. van der Aalst. Three good reasons for using a Petri-net-based workflow management system. In T. Wakayama, editor, *Information and Process Integration in Enterprises: Rethinking documents*, pages 161–182. Kluwer Academic Publishers, Norwell MA, USA, 1998.

[2] W.M.P. van der Aalst, A.P. Barros, A.H.M. ter Hofstede, and B. Kiepuszewski. Advanced workflow patterns. In *Proc. of the 5th IFCIS Int. Conference on Cooperative Information Systems*, Eilat, Israel, September 2000. Springer Verlag.

[3] B. Benatallah, B. Medjahed, A. Bouguettaya, A. Elmagarmid, and J. Beard. WebBIS: a system for building and managing Web-based virtual enterprises. In *Proc. of the 1st workshop on Technologies for E-Services, in cooperation with VLDB2000*, Cairo, Egypt, September 2000.

[4] M. Brodie. The B2B E-comerce revolution and its technology requirements. Presented at the National Science Foundation's Information and Data Management Workshop, Chicago, Illinois, March 2000. GTE Corporation.

[5] F. Casati, S. Ilnicki, L.-J. Jin, V. Krishnamoorthy, and M.-C. Shan. Adaptive and dynamic service composition in eFlow. In *Proc. of the Int. Conference on Advanced Information Systems Engineering (CAiSE)*, Stockholm, Sweden, June 2000. Springer Verlag.

[6] Q. Chen and M. Hsu. Inter-enterprise collaborative business process management. In *Proc. of the Int. Conf. on Data Engineering (ICDE)*, Heidelberg, Germany, April 2001.

[7] WorkFlow Management Coalition. Terminology and glossary. Technical Report WFMS-TC-1011, Workflow Management Coalition, Brussels - Belgium, 1996.

[8] M. Dumas, J. O'Sullivan, M. Heravizadeh, D. Edmond, and A. ter Hofstede. Towards a semantic framework for service description. In *Proc. of the 9th Int. Conf. on Database Semantics*, Hong-Kong, April 2001. Kluwer Academic Publishers.

[9] D. Harel and A. Naamad. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, October 1996.

[10] Forrester Research Inc. Emarketplaces boost b2b trade. Technical report, Forrester Research Inc., Cambridge, USA, 2000.

[11] Internet RFC-1305. Network Time Protocol Specification Version 3. `http://www.landfield.com/rfcs/rfc1305.html`.

[12] N.R. Jennings, T.J. Norman, P. Faratin, P. O'Brien, and B. Odgers. Autonomous agents for business process management. *Journal of Applied Artificial Intelligence*, 14(2):145–189, 2000.

[13] P. Koksal, S.N. Arpinar, and A. Dogac. Workflow history management. *SIGMOD Record*, 27(1):67–75, January 1998.

[14] A. Lazcano, G. Alonso, H. Schuldt, and C. Schuler. The WISE approach to electronic commerce. *Journal of Computer Systems Science and Engineering*, 15(5), September 2000.

[15] F. Leymann and D. Roller. *Production Workflow: Concepts and Techniques*. Prentice Hall, Upper Saddle River, NJ, USA, 2000.

[16] P. Muth, J. Weissenfels, M. Gillmann, and G. Weikum. Workflow history management in virtual enterprises using a leight-weight workflow management system. In *Proc. of the Workshop on Research Issues in Data Engineering (RIDE)*. IEEE Press, March 1999.

[17] P. Muth, D. Wodtke, J. Weissenfels, A.K. Dittrich, and G. Weikum. From centralized workflow specification to distributed workflow execution. *Journal of Intelligent Information Systems*, 10(2), March 1998.

[18] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.

[19] H. Schuster, D. Georgakopoulos, A. Cichocki, and D. Baker. Modeling and composing service-based and reference process-based multi-enterprise processes. In *Proc. of the Int. Conference on Advanced Information Systems Engineering (CAiSE)*, Stockholm, Sweden, June 2000. Springer Verlag.

[20] SQLData. Simple Object Access Protocol. `http://www.soapclient.com`.

[21] Sun Microsystems Inc. Enterprise JavaBeans Specifications. `http://www.javasoft.com/products/ejb/`.

[22] Sun Microsystems Inc. Java RMI. `http://java.sun.com/products/jdk/rmi`.

A Building the coordinators for a composite service

This appendix describes the algorithms for deriving the preconditions and the postprocessing required to build the coordinator attached to each basic state of a composite service specification. For the sake of simplicity, we first deal with the case where the transitions' labels do not have an event nor an action part. Later, in section A.3 , we discuss how events and actions in the transitions can be accommodated. We also assume that only one instance of a composite service is performed at a time. Concurrent instances can be handled by attaching the instance identifier to each message exchanged by the coordinators. In this way, when a coordinator receives a notification or completion, it knows which instance does it concern.

The following terms and notations are used in the sequel:

- The initial transitions (resp. final transitions) of an OR-state state, are the transitions stemming from its initial state (resp. leading to its final state). Notice that if there are several transitions stemming from an initial state, they should be labeled by disjoint conditions so that the choice between them is deterministic. Similarly, the final transitions of an OR-state must either stem from different source states, or be labeled by disjoint conditions.

- The initial transitions (resp. final transitions) of a concurrent region of an AND-state, are the transitions stemming from the initial state of this region (resp. leading to the final state of this region). Analog remarks as for the previous definition apply.

- The set of initial transitions (resp. final transitions) of an AND-state is the union of the sets of initial transitions (resp. final transitions) of its concurrent regions.

- If t denotes a transition, then source(t), target(t), and cond(t), respectively denote its source state, its target state, and the condition part of its label (or "true" if this label has no condition).

- If s is a state, then superstate(s) denotes its superstate.

A.1 Preconditions

We recal that the precondition for entering a state ST of a composite service is expressed as a set of rules of the form E[C] such that:

- E is a conjunction of events of the form ready(ST), meaning that a notification of completion has been received from the coordinator attached to state ST.
- C is a conjunction of conditions appearing in the statechart's transitions.

The semantics of each of these rules is that when an occurrence of each of the events in E has been received by the coordinator, if C evaluates to true, then the coordinator should invoke the service that it is responsible for.

The operational semantics of statecharts [9] ensures that no two transitions with the same target state can fire simultaneously. Hence, when a coordinator receives a completion message from another coordinator, at most one of the rules in its precondition set may be enabled (meaning that all the notifications of completion required for that rule have been received). When one of the rules is enabled, whether the condition part of the rule evaluates to true or false, the coordinator erases from its memory all the completion notifications that it has received. In this way, it is ready for another round of notification receptions, which can occur if the state to which the coordinator is attached is located within a loop.

The algorithm for computing the preconditions of a state ST is presented below as a set of functions defined through recursive equations.

Function PreCond takes as paramater a state S and yields a set of e[c] rules expressing the preconditions that have to be met before the coordinator attached to S invokes the service labeling this state. Its definition involves an auxiliary function PreCondTrans which operates on transitions instead of states.

```
PreCond(st) =
    let {t1, t2, ... , tn} be the incoming transitions of st
        PreCondTrans(t1) ∪ PreCondTrans(t2) ∪ ...  ∪ PreCondTrans(tn)
PreCondTrans(t) =
if source(t) is a basic state then
   {ready(source(t))[cond(t)]}
else if source(t) is an initial state then
     let sup = superstate(source(t))
         if sup = topmost state of the statechart then
            {ready(wrapper)[cond(t)]}
         else AddCond(cond(t), PreCond(sup))
else /* source(t) is a compound state */
     if source(t) is an OR-state then
        let {ft1, ft2, ... , ftn} be the final transitions of source(t)
          AddCond(cond(t), PreCondTrans(ft1) ∪ ...  ∪ PreCondTrans(ftn))
     else /* source(t) is an AND-state */
          let {cr1, ... , crn} be the concurrent regions of source(t),
          let {ft1_cr1, ... , ftn_crn} be the final transitions of cr1,
          let {ft1_cr2, ... , ftn_cr2} be the final transitions of cr2,
          ...
          let {ft1_crn, ... , ftn_crn} be the final transitions of crn
```

```
          AddCond([cond(t)],
               PreCondTrans(ft1_cr1) ∪ ... ∪ PreCondTrans(ftn_cr1)) ×
               PreCondTrans(ft1_cr2) ∪ ... ∪ PreCondTrans(ftn_cr2)) ×
               ...
               PreCondTrans(ft1_crn) ∪ ... ∪ PreCondTrans(ftn_crn)))
```

Function AddCond takes as parameter a condition C and a set of e[c]/a rules SR, and yields a set of e[c]/a rules obtained by adding C as a conjunct to the condition part of each rule appearing in SR. We assume that the event and the action parts of each rule are optional, so that this function also applies to e[c] rules and to [c]/a rules.

```
AddCond(c, {e1[c1]/a1, e2[c2]/a2, ... , en[cn]/an}) =
       {e1[c and c1]/a1, e2[c and c2]/a2, ... , en[c and cn]/an}
```

The binary operator × takes as parameters two sets of e[c] rules (say SR1 and SR2) and generates a set of e[c] rules by "combining" each element of SR1 with each element of SR2, as in a cartesian product. The "combination" of a rule e1[c1] with another rule e2[c2] is e1∧e2[c1∧c2]. Therefore:

```
{e1[c1], e2[c2], ... , en[cn]} × {e1'[c1'], e2'[c2'], ... , en'[cn']} =
    {e1∧e1'[c1∧c1'], e1∧e2'[c1∧c2'], ... , e1∧en'[c1∧cn'],
     e2∧e1'[c2∧c1'], e2∧e2'[c2∧c2'], ... , e2∧en'[c2∧cn'],
     ...
     en∧e1'[cn∧c1'], en∧e2'[cn∧c2'], ... , en∧en'[cn∧cn']}
```

A.2 Postprocessing

We recall that the postprocessing that has to be performed by the coordinator associated to a state ST, is expressed as a set of rules of the form [C]/A such that:

- C is a conjunction of conditions appearing in the labels of the statechart's transitions.
- A is a term of the form notify(ST), meaning that a notification of completion has to be sent to the coordinator associated to the state ST.

When a service finishes its execution, its coordinator evaluates each of the rules appearing in its postprocessing specification. For each rule whose condition evaluates to true, the corresponding notification action is undertaken.

The algorithm for computing the postprocessing of a state ST is presented below as a set of functions defined through recursive equations.

Function PostProc takes as paramater a state and yields a set of [c]/a rules expressing the postprocessing that the coordinator of this state is required to perform. Its definition involves an auxiliary function PostProcTrans which operates on transitions instead of states.

```
PostProc(st) =
    let {t1, t2, ... , tn} be the outgoing transitions of st
        PostProcTrans(t1) ∪ PostProcTrans(t2) ∪ ...  ∪ PostProcTrans(tn)
PostProcTrans(t) =
```

```
if target(t) is a basic state then
   {[cond(t)]/notify({target(t)})}
else if target(t) is a compound state then
        let {it1, it2, ... , itn} be the initial transitions of target(t)
         AddCond(cond(t), PostProcTrans(it1) ∪ ... ∪ PostProcTrans(itn))
else if target(t) is a final state then
     let sup = superstate(target(t))
         if sup = topmost state of the statechart then
            {[cond(t)]/notify(wrapper)}
         else if sup is an OR-STATE
              AddCond(cond(t), PostProc(sup))
         else /* sup is an AND-state.
                 The transitions that stem from this AND-state, as well
                 as any subsequent transitions, may involve conditions
                 which cannot be evaluated by the coordinator that
                 performs the post-processing */
              {[cond(t)]/notify(s) such that s ∈ Succs(sup)}
```

Function Succs takes as paramater a state S and yields the set of basic states which are successors S (intermediate pseudo-states are bypassed). T his function is defined in terms of an auxiliary function SuccsTrans which opera tes on transitions instead of states.

```
Succs(st) = let {t1, t2, ... , tn} = outgoing transitions of st
               SuccsTrans(t1) ∪ SuccsTrans(t2) ...  ∪ SuccsTrans(tn)
SuccsTrans(t) =
   if target(t) is a normal state then {target(t)}
   else if target(t) is a final state
           let sup = superstate(target(t))
               if sup = topmost state of the statechart then {wrapper}
               else Succs(sup)
   else /* target(t) is a compound state */
        let {it1, it2, ... , itn} be the initial transitions of target(t)
            SuccsTrans(it1) ∪ ... ∪ SuccsTrans(itn)
```

A.3 Accommodating transitions with events and actions

In the previous paragraphs, we assumed that the transitions had no events and no actions.

In order to accomodate transitions with actions, the algorithm for computing the preconditions of a state, must also compute the actions that have to be executed before the state is entered. This can be done through some simple modifications to the algorithm “PreCond”. Specifically, the rules generated by this algorithm need to have an action part, in addition to an event and a condition part.

Similarly, in order to accommodate transitions with events, the “PostProc” algorithm requires some modifications. Specifically, it needs to generate rules which potentially have an event part, in addition to the compulsory condition and action parts. When an event occurrence is received by a coordinator, it checks whether it matches the event part of one of the rules in its postprocessing table, and if it does, it evaluates the condition part of the rule, and executes the action part (i.e. the notification actions) if required.

B UML class diagrams

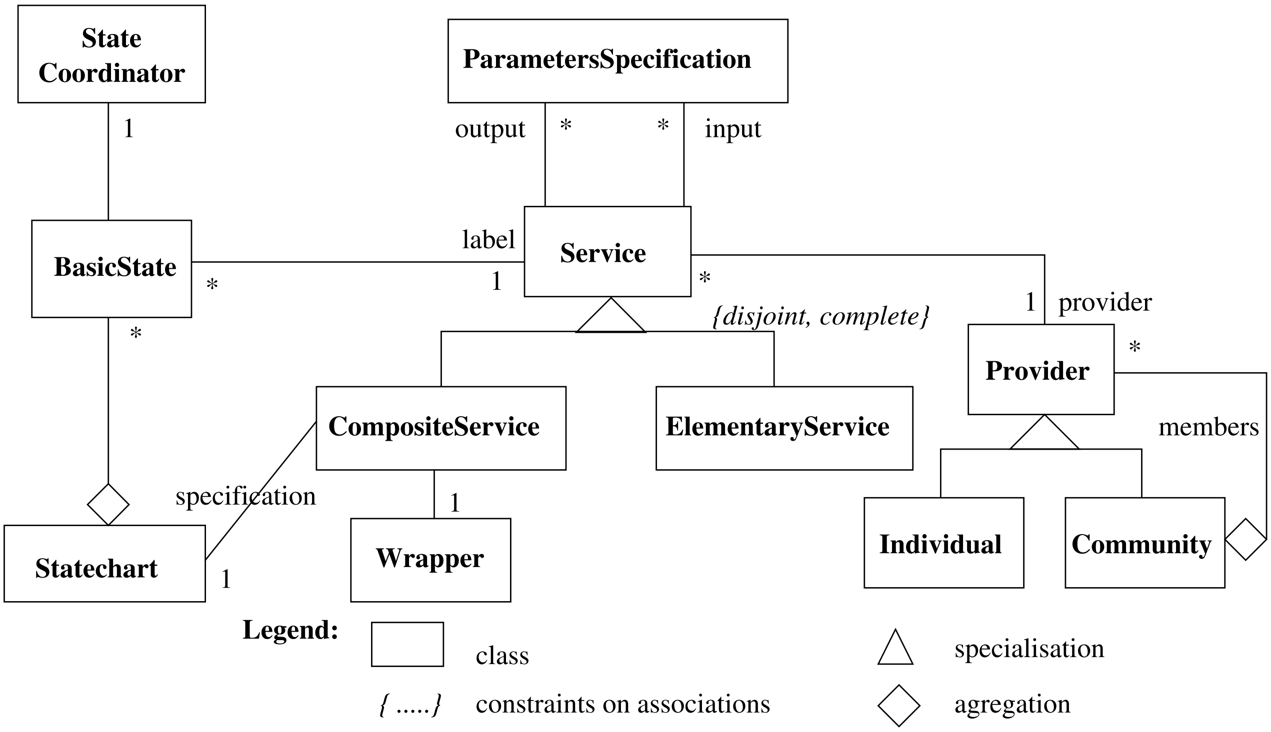


Figure 10: UML class diagram for service composition.

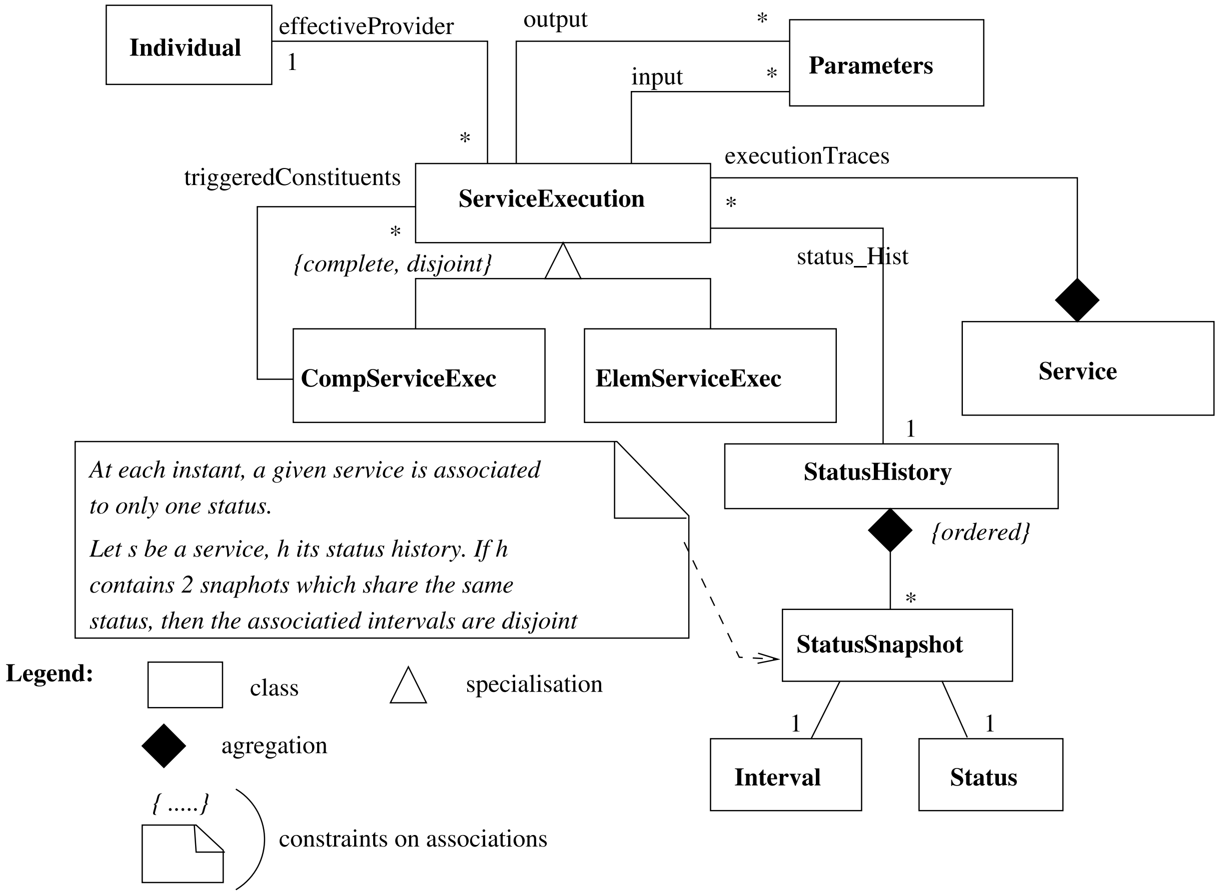


Figure 11: UML class diagram for service execution traces. This class diagram refer to two classes defined in the previous diagram.