Facilitating the Rapid Development and Scalable Orchestration of Composite Web Services*

Boualem Benatallah[1], Marlon Dumas[2], Quan Z. Sheng[1]

[1] School of Computer Science & Engineering
The University of New South Wales
Sydney NSW 2052, Australia
{boualem,qsheng}@cse.unsw.edu.au

[2] Centre for Information Technology Innovation
Queensland University of Technology
GPO Box 2434, Brisbane QLD 4001, Australia
m.dumas@qut.edu.au

Abstract

The development of new Web services through the composition of existing ones has gained a considerable momentum as a means to realise business-to-business collaborations. Unfortunately, given that services are often developed in an ad hoc fashion using manifold technologies and standards, connecting and coordinating them in order to build composite services is a delicate and time-consuming task. In this paper, we describe the design and implementation of a system in which services are composed using a model-driven approach, and the resulting composite services are orchestrated following a peer-to-peer paradigm. The system provides tools for specifying composite services through statecharts, data conversion rules, and multi-attribute provider selection policies. These specifications are interpreted by software components that interact in a peer-to-peer way to coordinate the execution of the composite service. We report results of an experimental evaluation showing the relative advantages of this peer-to-peer approach with respect to a centralised one.

Index Terms: Web service, Web service composition, Web service orchestration, dynamic provider selection, peer-to-peer interaction, statechart.

1 Introduction

Web services are gaining a considerable momentum as a means to architect and implement integrated enterprise applications, business-to-business collaborations [4, 9], and e-Government

*This research has been partly supported by an Australian Research Council (ARC) Discovery Grant number DP0211207. *Correspondence to:* Boualem Benatallah, School of Computer Science and Engineering, The University of New South Wales, Sydney NSW 2052, Australia

systems [20]. A Web service is essentially a semantically well defined abstraction of a set of computational and/or physical activities involving a number of resources, intended to fulfill a customer need or a business requirement. A Web service allows applications and/or other services to programmatically interact with, for example, information sources, application programs, and business processes. An example of a Web service is a flight booking system accessible through SOAP [36].

Web services can be composed with each other in the context of inter-organisational business processes, leading to *composite (Web) services* [10]. Composite services allow organisations to form alliances, to outsource functionalities, and to provide one-stop shops for their customers. An example of a composite service is a travel booking system integrating flight booking, accommodation booking, travel insurance, and car rental Web services.

The aim of the work reported in this paper is to enhance the fundamental understanding of how to facilitate the rapid and scalable composition of Web services. Specifically, it addresses the following key issues related to service composition:

- **Rapid composition:** The *why* part of Web services composition is widely understood [30]. However, the technology (i.e, the *how* part) to compose Web services in appropriate time-frames has not kept pace with the growth and volatility of available opportunities. Indeed, the development of integrated Web services is still largely adhoc, time-consuming and requiring a considerable effort of low-level programming. This approach is hardly applicable because of the volatility and size of the Web. More agile approaches (e.g. model-driven) to service composition are therefore required.
- **Adaptation to large and dynamic environments:** The set of services to be composed may be large and continuously changing. Consequently, approaches where the development of a composite service requires the understanding of each of the underlying services are inappropriate. Instead, a divide-and-conquer approach should be adopted, whereby services addressing similar customer needs (i.e. *substitutable services*) are grouped together, and these groups take over some of the responsibilities of service composition.
- **Distributed orchestration:** The orchestration of composite services in existing techniques is usually centralised, whereas the participating services are distributed and autonomous. A centralised orchestration model has several drawbacks with respect to scalability and availability [11]. Given the highly dynamic and distributed nature of Web services, we believe that novel techniques involving peer-to-peer orchestration of ser-

vices will become increasingly attractive. In a peer-to-peer execution model, distributed service components of similar capacity collaborate directly with each other without the need to establish a hierarchical relationship between them. Peer-to-peer computing is gaining a considerable momentum, as it naturally exploits the distributed nature of the Internet [40].

The *Self-Serv* system [6, 7] described in this paper addresses these issues by providing middleware and tool support for facilitating the composition and orchestration of Web services. In Self-Serv, Web services are composed using a model-driven approach, and the resulting composite services are executed in a *decentralised* way within a *dynamic* environment. The salient features and contributions of Self-Serv are:

- A language for process-based composition of Web services based on statecharts [24]: a widely used formalism in the area of reactive systems, which is emerging as a standard for process modeling as it has been integrated into the Unified Modeling Language (UML). Statecharts support the expression of control-flow dependencies such as branching, merging, concurrency, etc. They also provide an implicit style for expressing data-flow dependencies through the use of global variables.
- A concept of *service community* which provides a means to compose a potentially large number of services in a flexible manner. Service communities are essentially containers of substitutable services. They provide descriptions of desired services (e.g., providing flight booking interfaces) without referring to any actual provider. Actual providers can register with any community of interest to offer the desired service. At run-time, the community is responsible for selecting the service offer that best fits a particular user profile in a specific situation.
- A *peer-to-peer* orchestration model, whereby the responsibility of coordinating the execution of a composite service is distributed across several software components called *coordinators*. Coordinators are attached to each involved service. They are in charge of initiating, controlling, monitoring the associated services, and collaborating with their peers to orchestrate the service execution. The knowledge required at runtime by each of the coordinators in a composite service (e.g. location, peers, and routing policies) is statically extracted from the service's statechart and represented in a tabular form. In this way, the coordinators do not need to implement any complex scheduling algorithm.

The rest of the paper is organised as follows. Section 2 describes Self-Serv's approach to service composition, from the specification of control-flow and data-flow, to that of selection

policies. Section 3 discusses the peer-to-peer orchestration approach and analytically compares it to a centralised one. Section 4 describes the system architecture and implementation of Self-Serv. Section 5 describes an experimental setup in which the deployment and execution costs of the peer-to-peer approach are evaluated and compared to a centralised one. Finally, Section 6 provides an overview of related work and Section 7 draws some conclusions.

2 Service Composition Model

Self-Serv distinguishes three types of services: *elementary services*, *composite services*, and *service communities*. An elementary service is an access point to an application that does not rely on another Web service to fulfill user requests. In the Self-Serv system, it is assumed that every elementary service provides a programmatic interface based on SOAP and WSDL [32]. This does not exclude the possibility of using Self-Serv to integrate legacy applications, such as those written in CORBA. However, for such applications to be composed with others using Self-Serv, appropriate adapters should first be developed.

A composite service is an umbrella structure that brings together other composite and elementary services that collaborate to implement a set of operations. The services brought together by a composite service are referred to as its *component services*. An example of a composite service would be a travel preparation service, integrating services for booking flights, booking hotels, searching for attractions, etc.

The concept of service community is a solution to the problem of composing a potentially large number of dynamic Web services. A community describes the capabilities of a desired service without referring to any actual Web service providers. In other words, a community defines a request for a service which makes abstraction of the underlying providers. In order to be accessible through communities, pre-existing Web services can register with them. Services can also leave and reinstate these communities at any time. At runtime, when a community receives a request for executing an operation, it selects one of its current members, and delegates the request to it.

Whether elementary, composite, or community-based, a Web service is specified by an *identifier* (e.g., URL), a set of *attributes*, and a set of *operations*. The attributes of a service provide information which is useful for the service's potential consumers (e.g., public key certificates). We do not consider the specification of richer abstractions such as service conversations. For details about service conversation support in our approach, we refer the reader to [3].

2.1 Community Services

A community is an aggregator of service offers with a unified interface. It is intended as a means to support the composition of a potentially large number of dynamic Web services. The description of a community contains a set of operations that can be used to interact with the community and its underlying members. These operations are described without referring to the definitions of local services (i.e., members).

2.1.1 Service Registration

The registration of a service with a community requires the specification of *mappings* between the operations of the service and those of the community. The following is an example of a mapping:

> **source service** `Qantas Airway QAS` **target community** `Flight bookings FBS`
> **operation mappings operation** `FBS.search_flight()` **is** `QAS.search_ticket();`
> **operation** `FBS.book_flight()` **is** `QAS.book_ticket()`

In this example, the operation `search_flight` (resp., `book_flight`) of the community `Flight bookings` is mapped to the operation `search_ticket` (resp., `book_ticket`) of the service `Qantas Airway`. A registration may concern only a subset of the operations of a community. Thus, Web services have the flexibility to register only for the operations that they can provide. For instance, the community `Flight bookings` provides operations for searching (i.e, `search_flight`) and buying (i.e., `book_flight`) flight tickets. If a Web service provides only one of these operations, then it will register only for the operation that it provides.

A Web service can register with one or several communities. A community can be registered with another community. For example, the Web services `Qantas Airway` and `Cathay Pacific` are registered with the community `Flight bookings` which is itself registered with the community `Intl Travel Arrangements`.

2.1.2 Multi-Attribute Service Selection Policies

A community is associated with a scoring service that interprets a *selection policy.* A selection policy specifies preferences over services of a community. It consists of a *multi-attribute utility function* which has the form

$$U(s) = \sum_{i \in SA} w_i.Score_i(s)$$

where:

- $Score_i(s)$ is an attribute scoring function, which, given a value of an attribute i of the service s, returns a score (a positive integer value). SA is the set of selection attributes.
- w_i is the weight assigned to the attribute i.

The scoring service computes the weighted sum of service attribute scores using the weight property of each selection attribute. It then selects the service which produces the higher overall score according to the multi-attribute utility function. Selection attributes belong to one of three categories: *advertised*, *provider-supplied*, and *community-supplied*. Advertised attributes are the attributes that a service provider makes available at registration time (i.e., when the service becomes member of the community). For example, a service provider may advertise the expected duration of an operation invocation. Provider-supplied attributes are available from service providers only upon request. For instance, the price of a product can be defined as being a provider-supplied attribute. Finally, community-supplied attributes are the attributes that can be derived at the community level from past execution logs. For instance, values of service quality attributes such as reliability and availability can be estimated based upon past execution logs. Self-Serv supports a predefined set of generic attributes, and allows developers to introduce new attributes. Table 1 lists the predefined attributes and their corresponding scoring functions.

Quality Attribute	Quality Functions
Execution Price	$q_{price}(s, op)$ represents the amount of money that a service requestor has to pay for executing operation op of service s.
Execution Duration	$q_{du}(s, op)$ measures the expected delay in seconds between when a request to s for executing operation op is sent and when results are received.
Reputation	$q_{rep}(s)$ is a measure of its trustworthiness which depends on end-users experiences of using the service s.
Reliability	$q_{rel}(s)$ is the probability that a request to s is correctly processed within a maximum expected time frame.
Availability	$q_{av}(s)$ is the probability that a request to s is accessible.

Table 1: Service Quality Attributes.

A scoring function is provided for each selection attribute. For simplicity, we assume that the value of each scoring function has been scaled to the interval [0..1] and that a higher value indicates a better quality. For instance, the scoring function associated with the attribute `Execution Duration` (*ed* for short) is $Score_{ed}(s, op) = 1/q_{du}(s, op)$ (i.e, the higher the execution duration is, the lower is the score). It should be noted that the method of

estimating the value of a community-supplied attribute is not unique, neither is the set of selection attributes.

Selection policies are provided by communities. Consumers can customise these policies by providing weights for the selection attributes. A community may provide several alternative multi-attribute utility functions which correspond to different selection policies. Consumers choose policies depending on their preferences.

The automatic construction of attribute scoring functions is not addressed by Self-Serv. This issue is in fact addressed by work in the area of preference-based product recommendation systems [37]. In these systems, consumers specify their preferences via questionnaires. The information extracted from these questionnaires is then used to construct attribute scoring functions. Note that the focus of this paper is not on specifying selection policies. We mention this aspect here for the sake of completeness. Our work on service selection is described elsewhere [41].

2.2 Composite Services

The operations of a composite service are expressed as compositions of operations offered by other Web services using statecharts [24]. The choice of statecharts as the language for capturing the flow of operation invocations in Self-Serv is motivated by several reasons. First, statecharts possess a formal semantics, which is essential for analysing composite service specifications. Next, statecharts are a well-known and well-supported behaviour modelling notation, and they are part of the Unified Modeling Language (UML). Furthermore, statecharts offer most of the control-flow constructs found in existing process description languages: sequence, branching, concurrent threads, and cycles. [19] shows that statercharts are suitable for expressing typical control-flow dependencies. Specifically it is shown that statecharts can capture a relatively high number of the workflow patterns identified in [1].

However, like any other process modelling languages (e.g. languages based on Petri nets, process algebra, or transaction logic), statecharts have their relative advantages and drawbacks. In particular, statecharts do not provide direct support for modelling the so-called *multiple instances patterns* [1], that is, situations where multiple copies of the same activity are executed simultaneously and these copies need to synchronize upon completion. Nonetheless, this characteristic is shared by several other proposals in this area, like the Business Process Execution Language for Web Services (BPEL4WS) [14], which is currently emerging as an implementation-level standard in the area of Web service composition. In any case, the fundamental ideas behind our approach (e.g. peer-to-peer orchestration and late service

selection through communities) can be applied to other process modelling languages than statecharts, although the algorithms for composite service orchestration will differ depending on the language chosen.

2.2.1 Overview of Statecharts

A statechart is made up of states and transitions. Transitions are labeled by *ECA* (Event Condition Action) 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. The event, condition, and action parts of a transition are all optional. A transition without an event is said to be *triggerless*.

States can be basic or compound. In Self-Serv, a basic state corresponds to an invocation of a service operation, whether an elementary service, a community, or a composite service. Accordingly, each basic state is labelled with an invocation to a service operation. When the state is entered, this invocation is performed. The state is normally exited through one of its triggerless transitions when the execution induced by this invocation is completed. If the state has outgoing transitions labeled with events, an occurrence of any of these events causes the state to be exited and the ongoing execution to be cancelled.

Compound states provide a mechanism for nesting one or several statecharts inside a (larger) statechart. There are two types of compound states: OR and AND states. An OR-state contains an arbitrary statechart nested inside it, which is executed when the compound state is entered. An AND-state on the other hand contains several statecharts (separated by dashed lines) which are all executed concurrently when the compound state is entered. Each of the statecharts contained in an AND-state is usually called an *AND component* or an *orthogonal component*, but in this paper we choose the term *concurrent region* instead, to avoid confusion with the term *component service* introduced earlier.

The origin of the terms OR-state and AND-state can be explained as follows. The states of the statechart contained in an OR-state (i.e., the substates of the OR-state) are related by an *exclusive or* relationship, in the sense that being in an OR-state is equivalent to being in *exactly one* of its substates. Meanwhile, being in an AND-state is equivalent to being in *all* the concurrent regions contained in the AND-state.

From an operational perspective, when a compound state is entered, the initial state(s) of the statechart(s) nested in it become(s) active. The execution of a compound state is

considered to be completed when it has reached (all) its final state(s). Initial states are denoted by filled circles whereas final states are denoted by two concentric circles.

2.2.2 Data Flow and Conversion

An operation of a composite service is described by its input parameters, output parameters, consumed and produced events, and a statechart glueing these elements together. The input and output parameters are mapped to variables of the statechart and can be referenced in any of the conditions and actions of the statechart. Similarly, the consumed events can appear in any of the event parts of the statechart's transitions, and the produced events can be generated by the actions of these transitions. Moreover, the statechart contains a set of invocations to component services. Each of these invocations is described by the name of the service, the name of the operation, an expression to compute the effective input parameters, and the variables of the statechart to which the outputs of the operation are assigned.

Data flow between states in the statechart is therefore specified through the use of variables. A variable in the statechart of a composite service operation can be: an input parameter of the composite service operation, an output parameter of the composite service operation, or an internal (or local) variable. The value of an internal variable may be: (i) obtained from the output of a service invocation as mentioned above, (ii) requested from the user during the execution of the composite service, or (iii) derived from the input parameters of the composite service operation and/or other internal variables through a query. To cater for the first of these cases, we adopt the following syntax for invoking service operations:

$$\mathsf{S::m}(Q_1, ..., Q_n, \&V_1, ..., \&V_n)$$

The semantics of this expression is an invocation of the operation `m` provided by service `S`, with input parameters provided by queries Q_1, ..., Q_n, and such that the outputs of the invocation are assigned to variables V_1, ..., V_n. A query Q_i can be simply a variable name or any other query. Self-Serv adopts XPath [12] as the query language. To cater for the second and third cases above, Self-Serv recognizes in the action parts of the statechart the following types of expressions: (i) `X := USER`: the value of the internal variable `X` is supplied by the user, and (ii) `X := Q`: the value of `X` is the result of query `Q`.

2.2.3 Example

Figure 1 contains the statecharts of two composite services: `Complete Travel Services (CTS)` and `Intl Travel Arrangements Service (ITAS)`. The latter is invoked within the former. The statechart of `CTS` is composed of an AND-state, in which a search for attractions

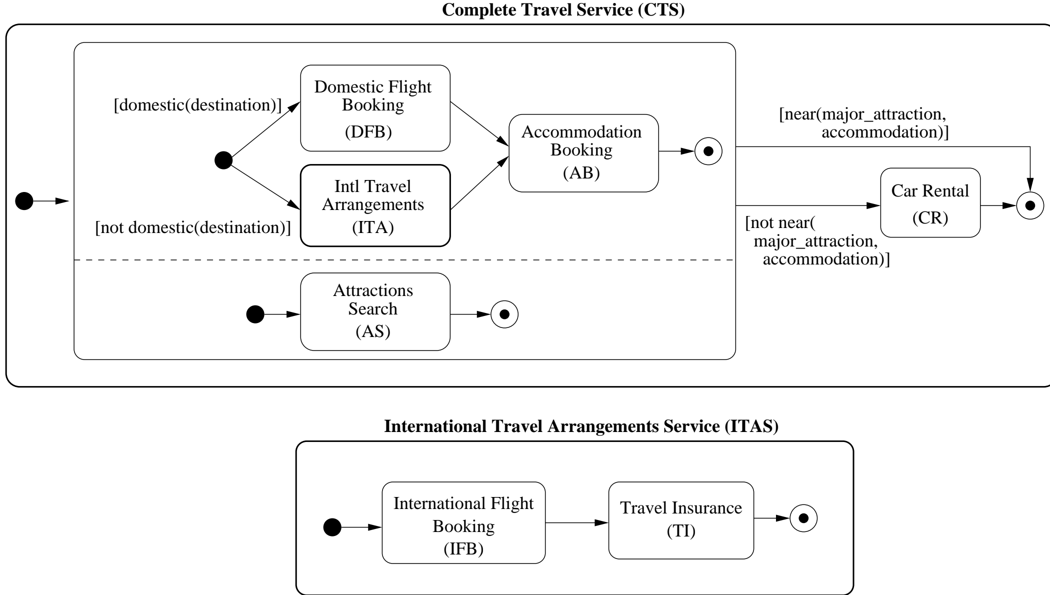

Figure 1: The Travel Solution composite service.

is performed in parallel with the bookings of the flight and the accommodation. When both of these threads complete, a car rental booking is performed if the major attraction is far from the booked accommodation.

Table 2 describes the signature of `CTS` and the signatures of the services that it invokes. To describe the signatures of the services, the following notations are used:

- CTS::prepareTrip denotes an invocation of the operation `prepareTrip` provided by the service `CTS`.
- The keyword `in` indicates that a parameter is passed by value. For instance `in Date minDepatureDate` indicates that the parameter `minDepartureDate` of type `Date` is passed by value.
- The keyword `out` indicates that a parameter is passed by variable. For example `out float totalPrice` means that the service operation returns a value of type `float`, and that this value is assigned to the variable given in place of this parameter.

Table 3 details the invocations that are made in each of the states of the composite service `CTS`. For a given row, the left column of the table contains the name of a state, e.g., `AS`, and the right column provides the name of the service operation that is invoked when that state is entered, followed by the effective parameters. Some of the variables appearing in Figure 1 and in the associated Table 3 are input parameters of `CTS` (e.g., `minDepartureDate`, `maxDepartureDate`, `destination`), while others are internal variables (e.g., `departureDate`,

```
CTS::prepareTrip(in Date minDepatureDate, in Date maxDepartureDate, in Date minReturnDate,
                 in maxReturnDate, in string destination, in string name,
                 out float totalPrice, out XMLDoc flightDetails,
                 out XMLDoc accommodationDetails, out XMLDoc rentalDetails)
CRS::booking(in string city, in string name, in Date rentalDate, in Date returnDate,
             out float price, out XMLDoc rentalDetails)
ABS::booking(in string city, in string name, in Date arrivalDate, in Date departureDate,
             in int starRating, out float price, out XMLDoc accommodationDetails)
AS::getAttractions(in string city, out XMLDoc attractions)
DFBS::booking(in Date minDepatureDate, in Date maxDepartureDate, in Date minReturnDate,
              in maxReturnDate, in string destination, in string name,
              out Date actualDepartureDate, out actualReturnDate,
              out float price, out XMLDoc flightDetails)
ITAS::booking(in Date minDepatureDate, in Date maxDepartureDate, in Date minReturnDate,
              in maxReturnDate, in string destination, in string name,
              out Date actualDepartureDate, out actualReturnDate,
              out float totalPrice, out XMLDoc flightDetails)
```

Table 2: Signatures of the operation prepareTrip provided by CTS, and signatures of the service operations that it invokes.

flightDetails). All of the internal variables involved in this example, are used to store the outputs of the component services invocations. In addition, the values of some internal variables are used as input parameters to component services invocations. For example, the variable departureDate is used to store one of the outputs of the invocation of the operation DFBS::booking, and it is later on used to provide the value of an input parameter for operations AB::booking and CRS::booking.

State	Invocation
AS	AS::getAttractions(destination, &attractions)
DFB	DFBS.booking(minDepartureDate, maxDepartureDate, minReturnDate, maxReturnDate, destination, name, &departureDate, &returnDate, &flightPrice, &flightDetails)
ITA	ITAS.arrangeTrip(minDepartureDate, maxDepartureDate, minReturnDate, maxReturnDate, destination, name, &departureDate, &returnDate, &flightPrice, &flightDetails)
AB	ABS::booking(destination, name, departureDate, returnDate, starRating &accommodationPrice, &accommodationDetails)
CR	CRS::bookCar(destination, name, departureDate, returnDate, &rentalPrice, &rentalDetails)

Table 3: Table of invocations of the CTS::prepareTrip composite service operation.

The statechart in Figure 1 features four conditions in its transitions. Conditions are modeled as calls to boolean functions, which take as parameters queries involving input parameters of the composite service as well as internal variables. For example, the condition

domestic(destination) is a function call whose parameter is directly obtained from one of the inputs of service CTS. Meanwhile, near(major_attraction, accommodation) is a function call whose parameters are given by the values of internal variables. Although not shown in the statechart for clarity reasons, the value of the variable major_attraction is derived from the value of the variable attractions (which is an XML document) through an XPath expression. Also not shown in the statechart, is the fact that the value of the internal variable starRating (which is used as an input parameter in the invocation ABS.booking) is requested from the user at runtime, just after the flight booking is completed. This situation should be expressed through the action starRating := USER.

3 Peer-to-Peer Orchestration

This section starts with an overview and illustration of the basic concepts of the service execution model of Self-Serv. After this overview and illustration, a formal description of the concepts and algorithms is given.

3.1 Overview

The execution model of Self-Serv is based on the idea that each state ST appearing in a composite service specification is represented by a *state coordinator* which is responsible for:

- Receiving notifications of completion from other state coordinators and determining from these notifications when should state ST be entered. These notifications of completion include the relevant data items (i.e. the values of variables) which have been gathered by the previous states visited during the execution.
- Invoking the service operation labelling ST whenever all the preconditions for entering ST are met. This invocation is done by sending a message to the service and waiting for a reply.
- Upon completion of the service execution started in the previous step, notifying this completion to the coordinators of the states that may need to be entered next. These notifications of completion contain all data items that need to be passed on to the next state coordinators.
- While state ST is active, receiving notifications of external events (e.g., a cancellation) and determining if ST should be exited due to these event occurrences. If so, the state coordinator will interrupt the ongoing service execution and will send notifications of

"completion" to the coordinators of the states which potentially need to be entered next.

- If user input is needed in order to determine the value of a variable used within the state, requesting this value from the initial coordinator which will then perform the necessary user interaction.

In essence, the coordinator of a state is a lightweight scheduler which determines when should a state be entered, and what should be done after the state is exited. The knowledge needed by a coordinator in order to answer these questions at runtime is statically extracted from the statechart describing the composite service operation, and represented in the form of *routing tables* as detailed later.

A composite service execution is orchestrated through peer-to-peer message exchanges between the coordinators of the states of the service's description, and through message exchanges between the coordinators and the component services. The messages exchanged between the coordinators for the purpose of notifying that a given state should/may be entered are called *control-flow notifications*. A control-flow notification sent by a coordinator C_1 to a coordinator C_2 expresses the fact that the execution of the state represented by C_1 has completed, and that C_1 believes that the state represented by C_2 needs to be entered. The notification message contains the up-to-date values of all the internal variables of the statechart that C_1 needs to transmit to C_2. On the other hand, the messages exchanged between the state coordinators and the component services are called *service invocations/completions*. Service invocations are performed using SOAP, since every service in Self-Serv, whether elementary, composite, or community-based, provides a SOAP entry point.

The *initial coordinator* of a composite service is a special type of coordinator which acts as the entry point to the service. When a composite service is invoked, its initial coordinator sends a control-flow notification to the coordinators of the states which need to be entered in the first place. From that point on, the execution of the composite service is orchestrated through peer-to-peer interactions between the state coordinators. At the end, the initial coordinator receives back the control-flow notifications indicating that the execution of the service instance is terminated. The initial coordinator can then return a completion message to the invoker.

In addition, the initial coordinator of a composite service is responsible for detecting and handling failures, and for processing external events. Specifically, the initial coordinator receives and processes failure notifications issued by the state coordinators when a control-

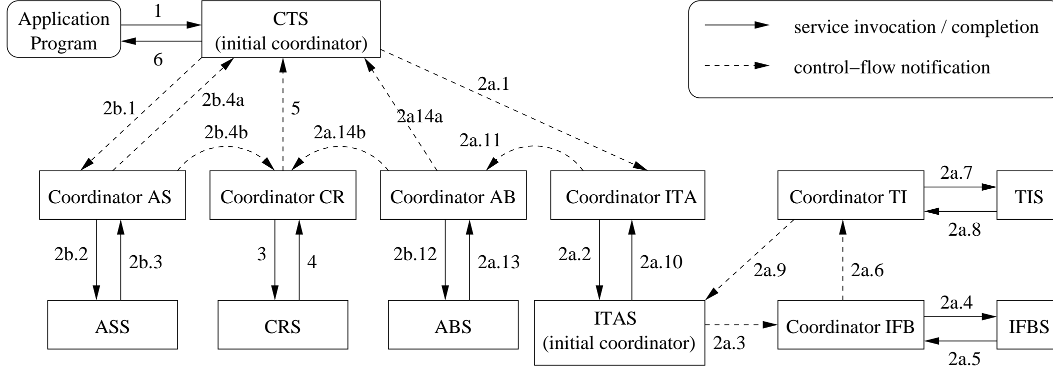

Figure 2: Interactions between the coordinators and the component services during an execution of CTS.

flow notification that needs to be sent has not been delivered after several retries. Also, the initial coordinator detects timeouts that may indicate that the composite service execution is stalled at a given node. When such timeouts occur, the initial coordinator identifies the point of failure (if any), selects an alternative service if possible, and notifies the failure to the invoker. Finally, external events directed to an instance of a composite service (e.g. for suspending its execution), are handled by the initial coordinator, who forwards these events to the appropriate state coordinators.

3.2 Orchestration Example

The diagram in Figure 2 shows the messages exchanged by the coordinators and the component services during a particular execution of service CTS (see Figure 1). The layout of the arrows indicate the type of the message (control-flow notification or service invocation/result) as explained in the legend of the figure. The numbers labelling the arrows capture the temporal relationships between the messages. For instance, message 3 is sent after message 2 which is sent after message 1. Some messages are exchanged as part of concurrent threads. In this case, the messages are given the same serial number, followed by a character. For instance, the messages starting with 2a and 2b in Figure 2 (e.g., 2a.1 and 2b.1) are sent within concurrent threads. Messages sent within the same thread are identified by serial numbers within that thread. For instance, message 2a.1 and 2a.2 are sequential messages exchanged within thread 2a.

The execution in this diagram starts when a user or application invokes the service CTS through its initial coordinator (message 1). Assuming that the trip is international, the initial coordinator of CTS sends a control-flow notification to the coordinators of ITA (message 2a.1) and to the coordinator of AS (message 2b.1). These coordinators trigger the services ITAS

(2a.2) and ASS (2b.2) respectively. When ASS returns an output (2b.3), the coordinator of AS sends a control-flow notification both to the initial coordinator of CTS (2b.4a) and to the coordinator of CR (2b.4b), since it is not possible to determine whether the major attraction is near the accommodation until the accommodation has been booked, and this is done in a separate concurrent thread. Meanwhile, the initial coordinator of ITAS starts the execution of this composite service by sending a control-flow notification to the coordinator of IFB (message 2a.3), which invokes IFBS (2a.4 and 2a.5). The execution of ITAS continues its course (2a.6, 2a.7 and 2a.8) until eventually a termination message is sent to the initial coordinator of ITAS (2a.9). This service returns a result to the coordinator of ITA (2a.10), which sends a notification to the coordinator of AB (2a.11). After invoking ABS (2a.12 and 2a.13), the coordinator of AB sends notifications both to the initial coordinator of CTS (2a.14a) and to the coordinator of CR (2a.14b), since again, the condition near(major_attraction, accommodation) cannot (always) be evaluated at that point in time. The coordinator of CR and the initial coordinator of CTS then evaluate the condition near(major_attraction, accommodation). If this condition is true, the coordinator of CR invokes the service CRS (messages 3 and 4). Once this invocation is completed, a notification is sent to the initial coordinator of CTS, and the overall execution is completed.

3.3 Preconditions and Postprocessing Actions

Extracting the knowledge required by a state coordinator from the statechart implementing a composite service operation, involves answering the following questions:

- What are the *preconditions* for entering a state? That is, what are the source states of the transitions leading to a given state, and what are the conditions that need to be satisfied for this transition to be taken.
- When the execution of a state is completed (whether successfully or because of a signal), which are the states that may need to be entered next? The process by which a coordinator notifies that its state is being exited to the relevant peer coordinators is called *postprocessing*.

The behavior of a state coordinator can therefore be captured through two sets: (i) a set of *preconditions* such that the state is entered when one of these preconditions is met, and (ii) a set of *postprocessing actions* indicating which coordinators need to be notified when a state is exited. Preferably, these sets of preconditions and postprocessing actions should be defined in a way to ensure minimal communication overhead. In other words, when a state

is exited, only those states that potentially need to be entered are notified. The following definitions formalize what is meant by a state potentially needed to be entered.

First of all, in order to identify the states which are accessible from another one in a single step, we introduce the concept of *compound transition*. Intuitively, a compound transition[1] is any path (i.e. list of linked transitions), going from a basic state to another basic state without passing through any other basic state.

Definition 1 (**Compound transition**). *A compound transition CT is a sequence of transitions t_1, t_2, ..., t_n belonging to a given statechart, such that:*

- *source(t_1)[2] is a basic state,*
- *target(t_n) is a basic state, and*
- *for all i in [1..n-1], either target(t_i) is the final state of a region belonging to the compound state source(t_{i+1}), or source(t_{i+1}) is the initial state of a region belonging to the compound state target(t_i).*

Under these conditions, `CT` *is said to connect source(t_1) with target(t_n), i.e., source(CT) = source(t_1) and target(CT) = target(t_n). The condition part of CT, noted Cond(CT), is the conjunction of the conditions labelling t_1, ..., t_n.* □

For example, in Figure 1 there is a compound transition with two elements, going from state `AS` to state `CR`, and another going from `AB` to `CR`. In both cases, the condition of the compound transition is `[true` ∧ `not near(major_attraction, accommodation)]`.

When a state is exited, the states which potentially need to be entered next are those which are target of a compound transition for which either: (i) the condition part is true, or (ii) the condition part cannot be fully evaluated, but the part that can be evaluated is true.

Definition 2 (**Minimal postprocessing table of a state**). *The minimal postprocessing table of a state ST, is a set of rules of the form [C]/ST' such that:*

- *There exists a compound transition* `CT` *such that source(CT) = ST and target(CT) = ST'.*
- *Conjuncts(C) ⊆ Conjuncts(Cond(CT)), where Conjuncts($c_1 \wedge \ldots \wedge c_n$) = $\{c_1, \ldots c_n\}$.*
- *If Conjuncts(C) ≠ Conjuncts(Cond(CT)), then the elements of Conjuncts(Conds(CT)) \ Conjuncts(C) are exactly those that cannot be evaluated at the time the state ST is exited. Here, \ stands for the set difference operator.* □

[1] Notice that the definition of compound transition that we adopt, is slightly different from that of [24].

[2] Here, source(t) denotes the source state of transition t, while target(t) denotes the target state of t.

In the example of Figure 1, we have that `Postprocessing(AS) = { [true]/notify(CR), [true]/notify(I) }`, where I is the identifier of the initial coordinator of the composite service CTS. Notice that the condition near(major_attraction , accommodation) cannot be evaluated by the coordinator of AS, since it involves information which is only known once the accommodation has been selected, and this is done in a separate concurrent region.

When a service labelling a state completes its execution, the coordinator of this state evaluates the condition part of each of the entries appearing in its postprocessing table. For each entry whose condition evaluates to true, it sends a notification message to the coordinator of the state referenced in that entry. The constraints imposed in the Definition 2 ensure that a state ST', will receive a notification of completion from another state ST, if and only if either (i) the state ST' needs to be entered, or (ii) it is not possible for ST to determine whether the state ST' should be entered or not. In this latter case, the decision on whether ST' should be entered or not, is made by the coordinator of ST' based on its preconditions table as defined below.

Definition 3 (**Minimal preconditions table of a state**). *The (minimal) preconditions table of a state ST of a composite service specification is a set of rules E[C] such that:*

- E *is a conjunction of events of the form ready(ST'). The event ready(ST') is generated when a notification of completion is received from the coordinator attached to state* ST'*. The conjunction of two events* e_1 *and* e_2 *is noted* $e_1 \wedge e_2$ *and the semantics is that if an occurrence of* e_1 *and an occurrence of* e_2 *are registered in any order, then an occurrence of* $e_1 \wedge e_2$ *is generated.*
- *There exists a compound transition* CT *from* ST' *to* ST *such that* $C \subseteq ST$.
- *If Conjuncts(C)* $\neq$ *Conjuncts(Cond(CT)), then the conditions in* $CT \setminus C$ *are exactly those which cannot be evaluated by the coordinator of* ST'. □

With respect to Figure 1, `Preconditions(AB)={ ready(ITA)[true],ready(DFB)[true] }`, meaning that the state AB is entered when a message is received from either the coordinator of the state ITA or that of DFB. Similarly, `Preconditions(CR)={ ready(AB)∧ready(AS)[not near(major_attraction, accommodation)] }`.

When a rule in the preconditions table of a state ST is triggered (i.e. an event occurrence matches the event part of the rule), if the rule's condition evaluates to true, state ST is entered, and the service that labels it is invoked by the coordinator of ST. The third item in Definition 3 ensures that the coordinator of ST will only evaluate those conditions which have not been previously evaluated by the coordinators referenced in the event part of the rule.

```
PostProc(ST) = let {T_1, T_2,..., T_n} are the outgoing transitions of ST in
               PostProcTrans(T_1) ∪ PostProcTrans(T_2) ∪ ...∪ PostProcTrans(T_n)

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 {IT_1, IT_2, ..., IT_n} be the initial transitions of target(T) in
        AddCond(cond(T), PostProcTrans(IT_1) ∪...∪ PostProcTrans(IT_n))
   else if target(T) is a final state then
      let SUP = superstate(target(T)) in
        if SUP is the topmost state of the statecharts then { [cond(T)]/notify(initial_coordinator) }
        else if SUP is an OR-STATE AddCond(cond(T), PostProc(SUP))
        else { [cond(T)]/notify(S) such that S ∈ CTargets(SUP) }

   The above equations make use the following two auxiliary functions.
      AddCond(c, { e_1[c_1]/a_1, ..., e_n[c_n]/a_n }) = { e_1[c and c_1]/a_1, ..., e_n[c and c_n]/a_n }
      CTargets(ST) = { target(CT) | CT is a compound transition ∧ source(CT) = ST }
```

Figure 3: Algorithm for the generation of postprocessing actions.

3.4 Routing Tables Generation

We describe in turn the algorithms for generating the postprocessing and the preconditions tables of a state. For the sake of simplicity and for space reasons, we restrict our presentation to the case where the transitions are only labeled with conditions (i.e., they do not have an event nor an action part). In [5], we discuss how transitions labeled with user-defined events and actions can be accommodated.

3.4.1 Postprocessings Table Generation

In order to derive the postprocessing table of a state, its outgoing transitions are analyzed, and one or several postprocessing actions are generated for each of them. The algorithm for generating the postprocessing table of a state, namely `PostProc` (see Figure 3), relies on an auxiliary algorithm `PostProcTrans` which takes as input a transition `T`, and returns a set of postprocessing actions that need to be undertaken if transition `T` is taken.

Let us now discuss how an outgoing transition `T` is used to generate a set of postprocessing actions. The simplest case is that when this transition leads to a basic state (`target(T)`), and it is labeled with a condition (`cond(T)`). The postprocessing action `[cond(T)]/notify(targe-t(T))` is included in the postprocessing table, meaning that if condition `cond(T)` is true, a notification must be sent to the coordinator of state `target(T)`.

If an outgoing transition `T` points to a compound state `CST`, then one postprocessing action is generated for each of the initial transitions of `CST`. The condition labelling `T` is then added

as a conjunct to the condition guarding each of these postprocessing actions, since T has to be true for any of these actions to be undertaken. This process is carried out recursively: if one of the initial transitions of CST points to another compound state CST', then one postprocessing action is generated for each initial transition in CST' and so on.

If an outgoing transition T points to a final state of a compound state CST, the outgoing transitions of CST are considered in turn, and one or several postprocessing actions are generated for each of them. Given a transition T' emanating from CST, a distinction is made here between the case where CST is an OR-STATE, and that where CST is an AND-state. In the former case, the condition labelling T' should be included as a conjunct in each of the guards of the postprocessing actions generated from T'. In the latter case, the condition labelling T' should not be included in any of the guards of the postprocessing actions generated from T', since the evaluation of this condition may require information which is not available when ST is exited. For example, in the case of Figure 1, the condition attractions far from accommodation should not appear in the postprocessing table of the state AS, since it cannot be evaluated until state AB is exited, and state AB is in a region concurrent to that of AS.

Finally, if an outgoing transition T points to the final state of the whole composite service, the postprocessing action(s) generated from this transition will involve the initial coordinator of the composite service. In other words, if this transition is taken, a control-flow notification will be sent to the initial coordinator.

3.4.2 Preconditions Table Generation

The preconditions table of a state is generated by determining, for each of the incoming transitions of the state, what are the conditions that should be met for that transition to be taken. The function PreCond(ST) (see Figure 4) which computes the preconditions of state ST can thus be written in terms of an auxiliary function PreCondTrans(T) which computes the preconditions of transition T.

The function PreCondTrans(T) distinguishes the cases where the source of the transition is a basic state, the one in which it is an initial state, the one in which it is an OR-state, and that in which it is an AND-state. In the first case, the only precondition for taking the transition is that the source state is exited, and the condition in the transition is taken. In the second case (the transition T stems from an initial transition), the preconditions for taking the transition T are identical to the preconditions for entering the superstate of T, except that they contain the condition in the transition of T as a conjunct. Notice that if the superstate of T is the topmost state of the statechart, T is an initial transition of the composite service

```
PreCond(ST) let {T_1, T_2,..., T_n} be the incoming transitions of ST
            PreCondTrans(T_1) ∪ PreCondTrans(T_2) ∪ ... ∪ PreCondTr ans(T_n)

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 is the topmost state of the statechart then {ready(initial_coordinator)[cond(T)]}
        else AddCond(cond(T), PreCond(SUP))
    else if source(T) is an OR-state then
        let {FT_1, FT_2, ..., FT_n} be the final transitions of source(T)
         AddCond(cond(T), PreCondTrans(FT_1) ∪ ... ∪ PreCondTrans(FT_n))
    else /* source(T) is an AND-state */
         let {CR_1, ..., CR_n} be the concurrent regions of source(T),
         let {FT_1_CR_1, ..., FT_n_CR_n} be the final transitions of CR_1,
         let {FT_1_CR_2, ..., FT_n_CR_2} be the final transitions of CR_2,
         ...
         let {FT_1_CR_n, ..., FT_n_CR_n} be the final transitions of CR_n
                AddCond([cond(T)],
                PreCondTrans(FT_1_CR_1) ∪...∪ PreCondTrans(FT_n_CR_1)) ×
                PreCondTrans(FT_1_CR_2) ∪...∪ PreCondTrans(FT_n_CR_2)) ×
                ...
                PreCondTrans(FT_1_CR_n) ∪...∪ PreCondTrans(FT_n_CR_n)))
```

The binary operator $\times$ (Cartesian product) used in this algorithm takes as parameters two sets of e[c] rules (say SR_1 and SR_2) and generates a set of e[c] rules by combining each element of SR_1 with each element of SR_2, where the combination of a rule $e_1[c_1]$ with another rule $e_2[c_2]$ is $e_1 \wedge e_2[c_1 \wedge c_2]$.

$$\{e_1[c_1], e_2[c_2], \ldots, e_n[c_n]\} \times \{e'_1[c'_1], e'_2[c'_2], \ldots, e'_n[c'_n]\} = \\ \{e_1 \wedge e'_1[c_1 \wedge c'_1], e_1 \wedge e'_2[c_1 \wedge c'_2], \ldots, e_1 \wedge e'_n[c_1 \wedge c'_n], \\ e_2 \wedge e'_1[c_2 \wedge c'_1], e_2 \wedge e'_2[c_2 \wedge c'_2], \ldots, e_2 \wedge e'_n[c_2 \wedge c'_n], \\ \ldots \\ e_n \wedge e'_1[c_n \wedge c'_1], e_n \wedge e'_2[c_n \wedge c'_2], \ldots, e_n \wedge e'_n[c_n \wedge c'_n]\}$$

Figure 4: Algorithm for the generation of preconditions.

and it is therefore taken when the composite service's initial coordinator sends an order to execute the service.

The case where a transition stems from a compound state `CST` is treated by recursively applying the function `PreCondTrans` to the final transitions of `CST`, and merging the resulting preconditions tables. In the case where `st` is an `OR-state`, the merging is a simple set union. In the case of an `AND-state`, each concurrent region is treated as an `OR-state`, and the preconditions tables obtained for each concurrent region are merged through a Cartesian product, meaning that the `AND-state` is exited if one of the final transitions in each of the concurrent regions is taken.

It can be proven by structural induction that the tables generated by `PreCond` and `PostProc` fulfill the conditions in Definition 3 and Definition 2 respectively. It follows that, at runtime, a control-flow message is sent from a coordinator C_1 to another coordinator C_2, only

if there is a compound transition from the state of $\texttt{C}_1$ to that of $\texttt{C}_2$, and either the state of $\texttt{C}_2$ needs to be entered, or it is impossible for $\texttt{C}_1$ to determine whether the state of $\texttt{C}_2$ needs to be entered or not. It should be noted that changes of composite services (e.g., remove a state from the statechart of a composite service) is not considered in this arcticle. However, one of the authors work in the area of change management for integrated services can be found in [42].

The above algorithms assume that the transitions in the statechart have no *event* and *action* part (i.e. only conditions are considered). In reality, statechart transitions can be labelled with events which may interrupt the execution of the service invocation labelling its source state, and actions which may manipulate the variables of the statechart.

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 a coordinator receives an event occurrence (which are sent by the initial coordinator), it checks whether this event occurrence 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 required notification actions.

Similarly, in order to accommodate 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 by modifying the algorithm `PreCond`, so that the postprocessing rules that it generates have an action part, in addition to an event and a condition part.

3.5 Analysis of Centralised and P2P Orchestration

In the P2P orchestration approach presented above, the coordinator of a state S is placed in the same machine as the service invoked in S. As a result, every control-flow notification potentially entails a *physical message exchange* (i.e., a message exchange between different physical machines). On the other hand, an invocation to a component service does not involve any physical message exchange.

In practice however, a coordinator and the component service that it invokes can be located in separate machines, in which case a message from a coordinator to the component service entails a physical message exchange. Furthermore, several coordinators can be placed in the same physical machine, so that a control-flow notification exchanged between these coordinators does not entail any physical message exchange. In an extreme case, all the coordinators can be placed in the same physical machine. We will subsequently call this

orchestration approach *centralised*, since the set of all the coordinators placed in a single physical machine can be seen as forming a central scheduler.

In the sequel, we compare the P2P and the centralised approaches in terms of physical message exchanges. Specifically, we estimate the maximum number of physical message exchanges required by a composite service execution, in terms of the number of invocations to component services involved by this execution. These physical message exchanges can result either from control-flow notifications or from invocations to component services.

Centralised approach. The worst-case number of physical message exchanges required by an execution of a composite service involving N invocations to component services is $2 \times N$. Indeed, in this approach only the invocations to component services entail physical message exchanges: the control-flow notifications do not. Moreover, each invocation to a component service requires two messages: one from the coordinator to the component service and another from the component service back to the coordinator.

P2P approach. The worst-case number of physical message exchanges required by an execution of a composite service involving N invocations to component services, is bounded by $M \times (N + 1)$, where M is the number of basic states in the corresponding statechart. The reasoning behind this bound is the following. First, we note that only the control-flow notifications require physical message exchanges: the invocations to component services do not require so, since the coordinator that performs this invocation is located in the same machine as the component service that is invoked. Next, we note that each time that a basic state is exited, at most M control-flow notifications are sent by the coordinator of this state: $M - 1$ to its fellow coordinators and 1 to the initial coordinator. Hence, after an invocation to a component service is completed and the corresponding state is exited, at most M physical message exchanges (entailed by control-flow notifications) take place. If the composite service execution involves N invocations to component services, $N \times M$ physical message exchanges take place during it. Moreover, when the composite service begins its execution, at most M messages are sent by the initial coordinator to the coordinators. Overall, the worst-case number of physical message exchanges is thus $M + M \times N = M \times (N + 1)$.

The above is a tight bound as evidenced by the example in Figure 5. In this example, each time that one of the states labelled $S_1, \cdots, S_m$ is exited, the corresponding coordinator must send one control-flow notification to each of the other coordinators, and one to the initial coordinator. Indeed, in the worst-case none of the coordinators is able to fully evaluate the conditions C_1 and C_2: these conditions may involve one data item from each of the

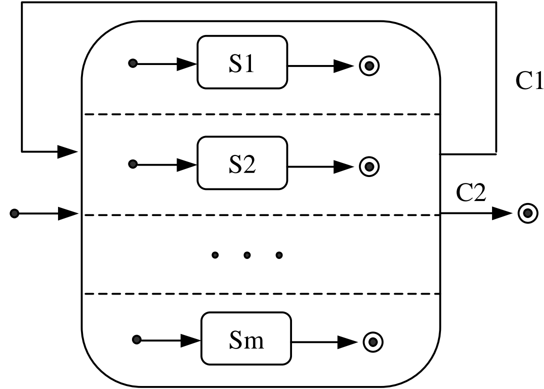

Figure 5: Worst-case scenario for the P2P orchestration approach.

invocations to $S_1, \cdots, S_m$, so that each coordinator must send the data item that it collects to all the other coordinators and let them evaluate these conditions when they have all the data items required. Hence, each invocation to a component service is followed by M control-flow notifications, leading to a total of $M \times N$ physical message exchanges. This, added to the M messages that the coordinator of the composite service needs to send to the coordinators of the states labelled $S_1, \cdots, S_m$, yields exactly the above bound. The above however is an extreme case. In practice, provided that there are no or few AND-states followed by conditional branches such as that of Figure 5, one can expect that the P2P approach requires less physical message exchanges than the centralised one.

4 Implementation of Self-Serv

The Self-Serv system [34] consists of a Service Composition Environment (also called the *Service Manager*) for defining and deploying composite services and communities, and a runtime environment that acts as a middleware for orchestrating composite services and performing dynamic provider selection. Both the service composition environment and the runtime environment have been implemented in Java.

The services registered in Self-Serv form the so-called *pool of services*, and they can be composed with others to form new services (see Figure 6), which are themselves registered and added to the pool. Services all provide a SOAP-based programmatic interface. Elementary services typically wrap application programs, workflows, databases, etc.

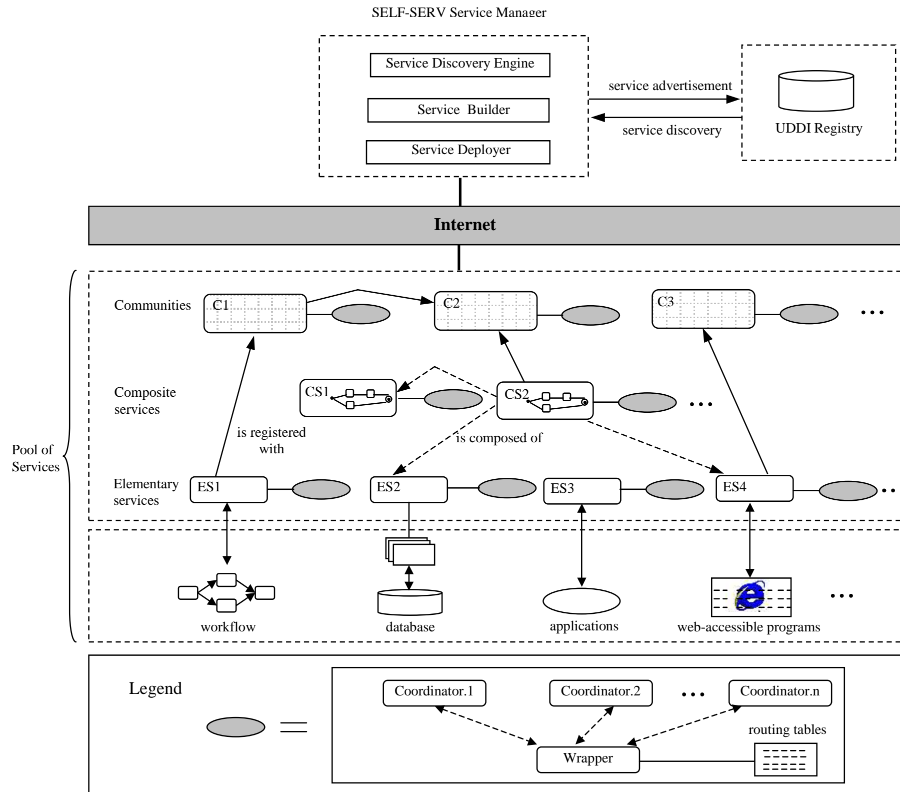

Figure 6: Architecture of the Self-Serv prototype.

4.1 Service Composition Environment

The service composition environment consists of a set of integrated tools that allow service developers and users to create and execute services. It is composed of the following component tools: service discovery engine, service builder, and service deployer.

Service Discovery Engine. The service discovery engine facilitates the advertisement and location of services (see Figure 7). It is implemented using SOAP, WSDL, and UDDI [2]. Service registration, discovery and invocation are implemented by SOAP calls. When a service registers with a discovery engine, a UDDI SOAP request containing the service description in WSDL is sent to the UDDI registry. After a service is registered in the UDDI registry, it can be located by sending the UDDI SOAP request (e.g., business name, service type) to the UDDI registry.

The discovery engine is implemented using the IBM Web Services Toolkit 2.4 (WSTK) [25]. WSTK provides several components and tools for Web service development (e.g., UDDI, WSDL, SOAP). In particular, we used the UDDI Java API (UDDI4J) to access a private UDDI registry (i.e, hosted by the Self-Serv platform), as well as the WSDL generation tool

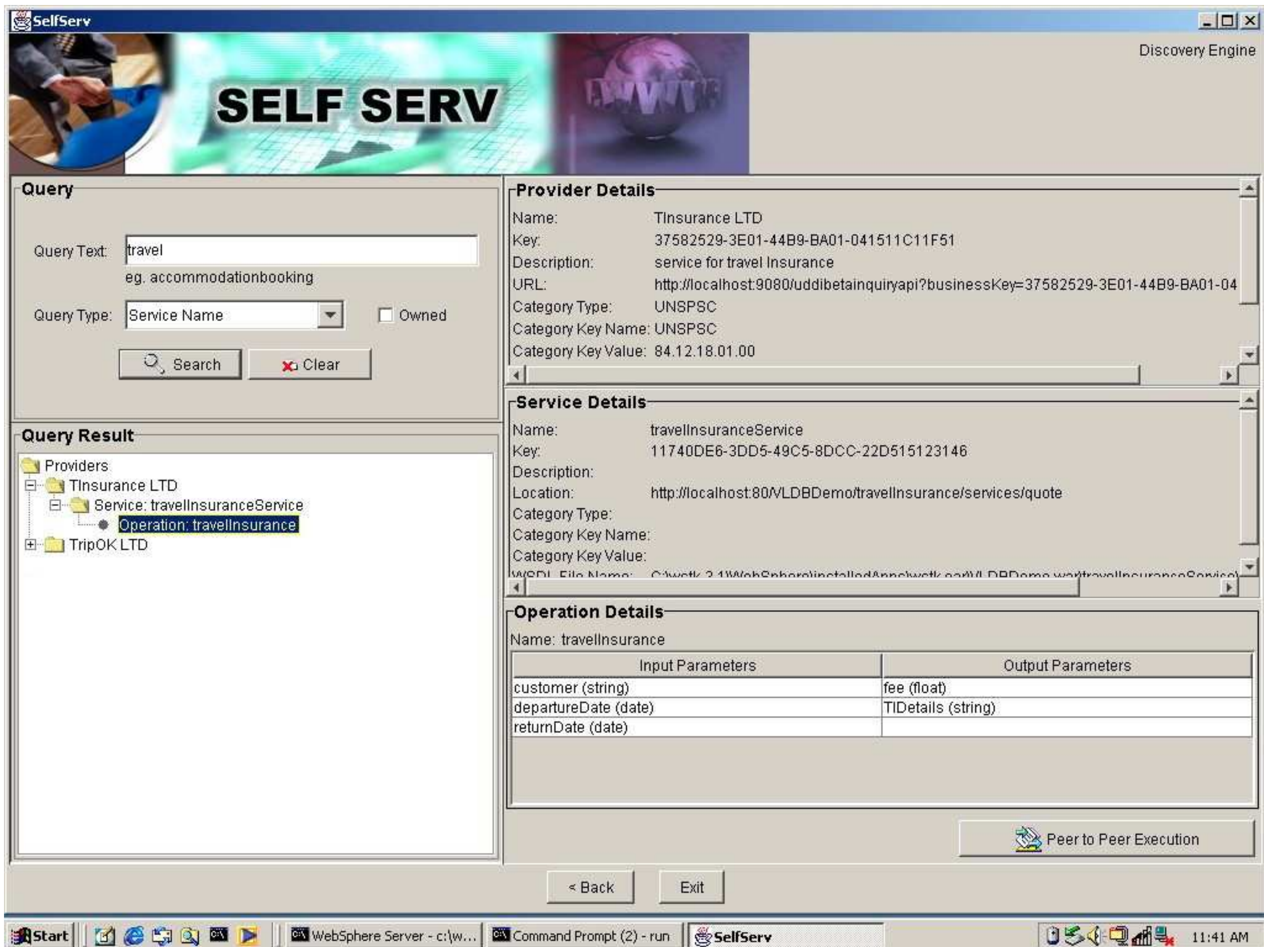

Figure 7: Service Discovery Engine.

for creating the WSDL documents and SOAP service descriptors required by the discovery engine. Details about the implementation of the discovery engine are presented in [35].

Service Builder. The service builder assists developers in the creation and maintenance of communities and composite services. It provides an editor for describing the statechart diagram of a composite service operation, for creating and configuring service communities, and for importing operations from existing Self-Serv services into composite services and communities. A search and browse facility is offered to locate component services using the service discovery engine and import their operations into states.

Service Deployer. The service deployer is responsible for generating the precondition and postprocessing tables of every state of a composite service statechart, using the algorithms presented in Section 3.4. The input of the programs implementing these algorithms are statecharts represented as XML documents (which are generated by the service builder), while the outputs are routing tables formatted in XML.

Once the tables are generated, the service deployer assists the service composer in the process of uploading these tables into the hosts of the corresponding component services, as well as in setting up the initial coordinator of the composite service. At present, security issues related to uploading tables are not considered.

4.2 Runtime Environment

The runtime environment of Self-Serv consists of three classes: *Community*, *InitialCoordinator*, and *Coordinator*. These classes are relatively lightweight, and the only infrastructure that they require are standard Java libraries, a JAXP-compliant XML parser, and a SOAP server. In the current implementation, we use Oracle's XML Parser 2.0 and IBM's Apache Axis 1.0. By default, the XML documents containing the routing tables are stored in plain files, so that there is no need to have a DBMS in the site where the installation is made. However, if the administrator decides to store these documents in a DBMS, (s)he can customize the class *Coordinator* accordingly.

The class *Community* provides a method to invoke a service operation on the community. This method first invokes the scoring service to find the most suitable member for handling the invocation, and then invokes the selected member service. A scoring service is a Java method that takes as input a selection policy, queries the service descriptions, and returns the identifier of one of the members registered with the community. The descriptions of selection attributes are represented in XML. The class *Community* also provides operations for managing the membership of the community (e.g. to join and quit the community).

The class *InitialCoordinator* on the other hand, provides methods for invoking an operation of a composite service. When this method is invoked, the initial coordinator reads the postprocessings table of the initial state of the corresponding statechart, and sends a control-flow notification message to each of the coordinators of the states that need to be entered first. These coordinators then interact in a peer-to-peer way with other coordinators, until eventually the coordinators of the states which are exited the last, send their control-flow notifications back to the initial coordinator. Once all the control-flow notifications are received, the outputs of the service invocation are made available.

The class *Coordinator* provides methods for processing and generating control-flow notifications. This class implements a software component made up of a *container* and an *object pool*. The container is a process that runs continuously, waiting for control-flow notifications from other coordinators. When it receives a message from another coordinator, it first examines the identifier of the composite service instance to which the message relates, and proceeds

as follows:

- If the identifier of the composite service instance is unknown to the container (i.e., this is the first time that a control-flow notification related to that instance is received), a new coordinator object is created, and this object is given access to the routing tables of the receiver state indicated in the message identifier. The task of handling the notification is forwarded to this newly created object by invoking a method `process_notification` on it. If other control-flow notifications related to the same composite service instance are expected to arrive subsequently, the object is temporarily added to the object pool so that it can treat them as they arrive.
- If on the other hand the container has previous knowledge about the composite service instance to which the control-flow notification relates, the notification is forwarded to the coordinator object that was created when the first message related to that instance was received. This object is retrieved from the object pool and the method `process_notification` is invoked on this object.

Each object in the pool is dedicated to a particular composite service instance, and processes all the incoming control-flow notifications related to that instance. By keeping track of these notifications, and by having access to the relevant preconditions tables, the object is able to detect when should a given state of the composite service be entered. When a coordinator object detects that a given state of the composite service needs to be entered, it sends an invocation message to this service. Once the corresponding completion message is received, the object polls the result parameters from the service, generates one or several control-flow notification messages (according to the information contained in the relevant postprocessings table), and dispatches these messages. From there on, the coordinator object is no longer needed, so it is removed from the pool and destroyed (garbage collection).

5 Performance Evaluation

To evaluate the proposed approach, we conducted experiments using the implemented prototype system. This section presents three experimental results. The first experiment shows the cost of deploying composite services in Self-Serv. The second experiment compares the number of messages required to execute composite services using the P2P orchestration approach and the centralised orchestration approach. The experiment also shows the workload distribution of these two models. The third experiment compares the execution time of the two execution models under different message sizes.

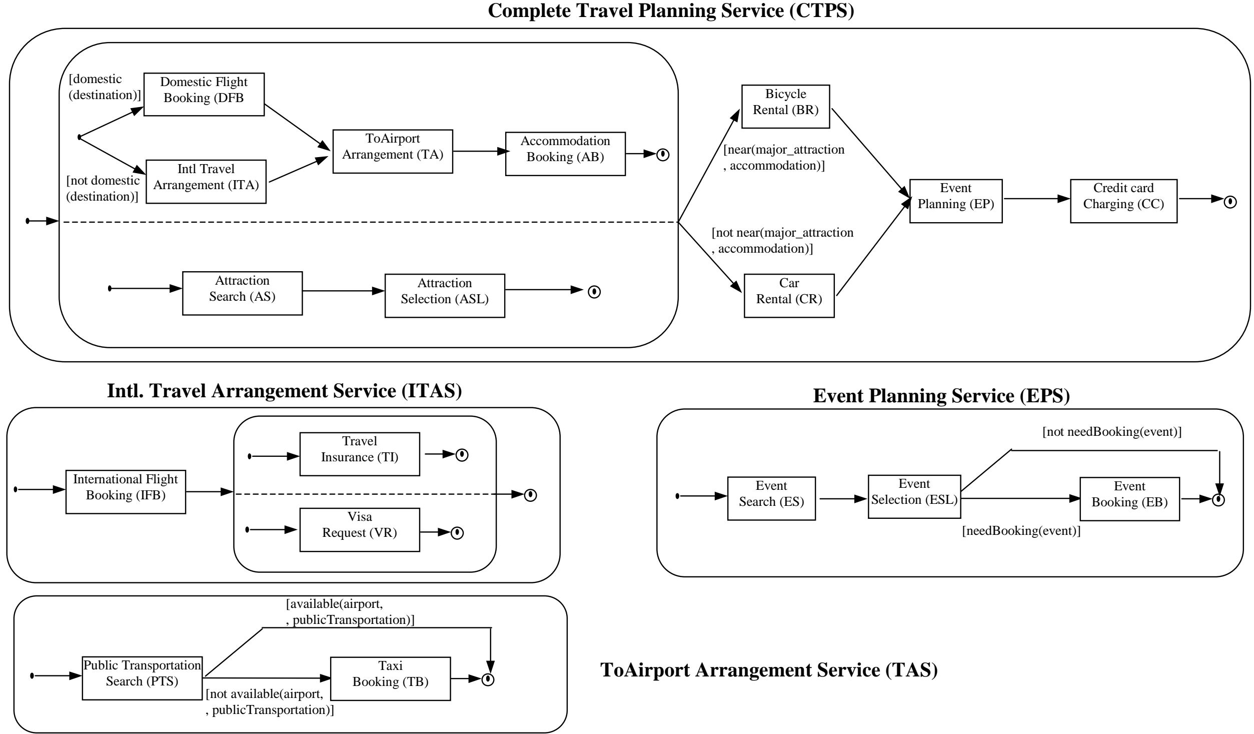

Figure 8: Complete Travel Planning Service (CTPS).

For the experiments, we designed a scenario *Complete Travel Planning Service* (CTPS), shown in Figure 8. We conducted experiments using a cluster of PCs running the prototype system. All PCs have the same configuration of Pentium III 933MHz and 2GB RAM. Each PC runs Debian Linux and the Java 2 Standard Edition V1.3.0. The machines are connected to a LAN through 100Mbits/sec Ethernet cards. One of them is dedicated to `CTPS` while others are servers for component services.

5.1 Deployment Cost

The purpose of this experiment is to measure the deployment cost (i.e., time required to deploy a service) of a composite service using Self-Serv. In the experiment, we created several composite services with different number of component services and recorded the time taken by the deployment of each composite service. The services were created by randomly adding states to the composite service `CTPS`. The deployment procedure includes generating the precondition/postprocessing tables for each component service and uploading the tables to the corresponding host machines. We deployed each composite service 10 times and computed the average deployment time.

Table 4 plots the time for deploying a composite service in terms of the number component

No. of component service	15	20	30	40	50
Deployment cost(second)	8.9	10.1	13.1	15.9	19.0

No. of component service	60	70	80	90	100
Deployment cost(second)	21.8	25.9	30.2	32.9	34.1

Table 4: The deployment cost of composite services.

services. This table shows that for large composite services, the deployment speed tends to be of 3 component services per second.

5.2 Number of Exchanged Messages

The purpose of this experiment is to study and compare the performance of the P2P orchestration model with that of the centralised one. The comparison was done by measuring: (i) the number of overall physical message exchanges, and (ii) the distribution of workload (i.e., number of messages handled at a given host) across participant hosts.

In the implementation of the centralised approach, a *central scheduler* is responsible for sending and receiving messages to and from the component services. The central scheduler is located in the same machine as `CTPS`, while the component services are located in the other machines. The physical message exchanges in the centralised model correspond to the messages exchanged between the central scheduler and the component services.

On the other hand, in the implementation of the P2P approach, the coordinator of a state and the component service invoked in this state are located in the same machine, i.e. each pair (state coordinator, component service) is located in a separate machine, while the initial coordinator of `CTPS` is located in its own machine. The physical message exchanges in this approach correspond to the messages exchanged between the initial coordinator and the state coordinators, and those exchanged between the state coordinators.

We executed `CTPS` and counted the number of messages under these two orchestration models. Note that there are four branches in `CTPS`, so we executed the composite service using all the possible combinations of *truth* values of the branching conditions. Table 5 shows the results of these simulations.

From the table we can see that for every possible combination, the P2P model requires less physical message exchanges than the centralised one. For example, in case 13 of Table 5 (i.e., the flight will be an international one, the customer will rent a bicycle, the events need to be booked, and the customer will book a taxi), 26 messages are exchanged in the centralised model, whereas only 18 messages are exchanged in the P2P model. In other words,

Combination of branch conditions					Number of messages	
	Branching conditions					
No.	domestic	nearAttraction	needBooking	available	P2P	Centralised
1	yes	yes	yes	yes	14	20
2	yes	yes	yes	no	15	22
3	yes	yes	no	yes	13	18
4	yes	yes	no	no	14	20
5	yes	no	yes	yes	14	20
6	yes	no	yes	no	15	22
7	yes	no	no	yes	13	20
8	yes	no	no	no	14	20
9	no	yes	yes	yes	18	24
10	no	no	yes	yes	18	24
11	no	no	no	yes	16	22
12	no	yes	no	no	17	24
13	no	yes	yes	no	18	26
14	no	yes	no	yes	16	22
15	no	no	yes	no	18	26
16	no	no	no	no	17	24
Average					15.625	22.125

Table 5: The number of exchanged messages in the execution of CTPS.

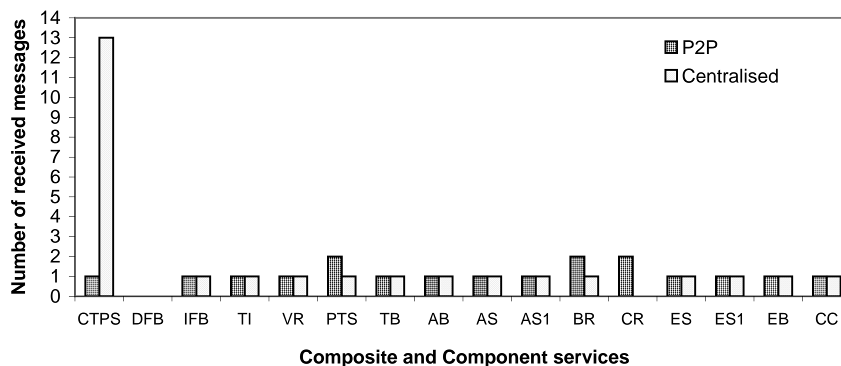

Figure 9: Workload allocation in the execution of CTPS.

the centralised model needs nearly 45% more exchanged messages than the P2P model does. Overall, the average number of physical message exchanges under the P2P model is 15.625, while it is 22.125 for centralised one. This shows that in realistic scenarios, the P2P model does require less physical message exchanges than the centralised one as conjectured in Section 3.5.

We also measured the workload allocation of the participant hosts (i.e., the hosts of CTPS and its components), by counting the number of messages received at each participant host. Figure 9 shows the results for case 13. Other cases yielded similar results.

The results show that in the centralised model, messages are not distributed evenly. In particular, the machine hosting the central scheduler receives many messages (13 messages in the example). On the other hand, other machines only receives 1 message each. Half of the messages are received by the central scheduler. In contrast to the centralised model, the

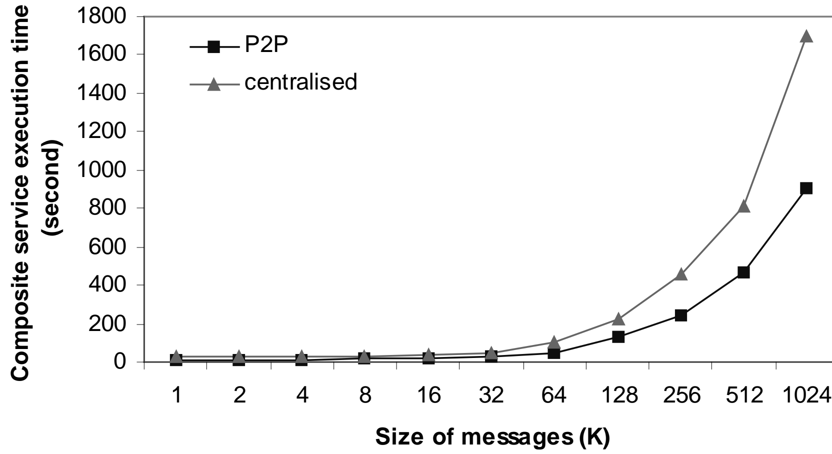

Figure 10: The execution time of the composite service CTPS.

workload of P2P model is distributed gracefully among the machines. CTPS only receives 1 message (i.e., 5.6% of the total messages). The above experiments consider only one execution of CTPS. However, we can estimate that there could be 13,000 messages handled by the central scheduler if 1,000 executions run concurrently. While for P2P model, the wrapper of CTPS would only receive 1,000 messages. As a result, the P2P execution model is more scalable.

5.3 Execution Time versus Message Size

The aim of this experiment is to investigate the execution performance of P2P and centralised execution models, with different size of exchanged messages. In the experiment, we assume that the size of all exchanged messages remains the same during the service execution. The size of messages ranges over the values 1K, 2K, 4K, 16K, 32K, 64K, 128K, 256K, 512K, and 1024K. For each message size, we executed CTPS 10 times and computed the average execution time. The results for case 13 is shown in Figure 10. Similar results were obtained for other cases.

From the figure we can see that, in both the P2P and the centralised approach, the execution time does not change greatly when the size of messages is small. For example, in the P2P approach, the execution time is 12.6 seconds when the message size is 1K and it is 19.8 seconds when the message size reaches 16K. However, the execution time changes from 46.9 seconds to 130.8 seconds when the message size grows from 64K to 128K. We also note that when the size of messages increases, the execution time of the centralised approach increases more sharply than that of the P2P approach, and it is always larger than in the P2P approach. This is due to two reasons. First, the number of exchanged messages in the P2P

approach is less than in the centralised one. Second, the messages in the centralised approach need to systematically transit through a central scheduler which constitutes a bottleneck.

6 Related Work

Service composition is a very active area of research and development [4, 9, 17, 33, 18, 21]. In this section, we examine component-based middleware, and Web services standards. We also look at existing service composition approaches.

6.1 Component-based Middleware

Component-based middleware (e.g., CrossWorlds, IBM SanFrancisco) [17, 13] typically rely on distributed object frameworks such as CORBA, DCOM, EJB, and other state-of-the-art technologies such as Enterprise Application Integration (e.g., IBM MQSeries) and Enterprise Resource Planning suites (e.g., SAP R/3), database gateways and transaction monitors. A component-based middleware provides standard data and application integration facilities (e.g., pre-built application adapters, data transformations, and messaging services among heterogeneous systems) supporting uniform access to heterogeneous applications. In this approach, composite services (e.g., ordering a product) can be assembled from independently developed components (e.g., checking inventory, delivering goods, and payment). However, the composition of services is realised through ad-hoc code development. This static and ad-hoc composition approach would be hardly scalable because of the large number partners that may be involved in a composition, the loosely coupled and possible dynamic nature of Web services. Hard coding the composition flow makes the definition, deployment, and evolution of services difficult and time-consuming [30, 31].

Component-based middleware are thus appropriate for the integration of small numbers of tightly coupled services with stable interfaces. Our approach focuses on the composition and deployment of a potentially large number of loosely coupled and dynamic services. Component-based middleware efforts are complementary to our approach. An intra-enterprise application which developed using a component-based middleware, could be wrapped as an elementary service and then composed with other services using using our approach.

6.2 Web Services Standards

Several standards that aim at providing infrastructure to support Web services interoperability have emerged recently including SOAP, WSDL, UDDI, BPEL4WS, and WSCL[39].

They provide building blocks for service invocation, description, advertisement, discovery, and orchestration. SOAP provides an XML-based protocol for exchanging information and requesting services. WSDL is an XML-based language that can be used to describe service operations. UDDI provides a registry for advertising and discovering Web services. BPEL4WS and WSCL build upon WSDL to support interactions among services. WSCL can be used to describe conversations that a service supports (e.g. the supported operations and the legal order of the their invocations). BPEL4WS can also be used to describe the behavior of a service in terms of supported flows of operation invocations, as well as to specify composite services. Several application development platforms such as Microsoft .NET, IBM WebSphere, Sun ONE, and BEA Weblogic Integrator provide some support for Web services standards.

The above standards and platforms are complementary to our approach. Our approach builds upon the building blocks of these standards (e.g., SOAP, UDDI) and extends them in significant ways. We provide a scalable service mediation framework that provides high level support for defining composite Web services involving a variable number of participants and the resulting composite services can be enacted in a *decentralised* way within a *dynamic* environment. Finally, we note that efforts in B2B interaction standards such as EDI, RosettaNet, and ebXML [17, 15, 16] provide common building blocks (e.g., documents and business processes semantics, syntax for message exchanges) for all classes of B2B applications. The incorporation of B2B interaction standards into Web service infrastructures will enable organisations to use Web services to carry out conversations according to these standards.

6.3 Service Composition Approaches

CMI [31] and eFlow [10] are platforms for specifying, enacting, and monitoring composite services. In both of these platforms, the underlying execution model is based on a centralised process engine, responsible for scheduling, dispatching, and controlling the execution of all the instances of a composite service. This contrasts with Self-Serv's peer-to-peer execution approach. Both eFLOW and CMI support dynamic provider selection, although the concept of community provided in Self-Serv is not explicitly supported. The concept of community in Self-Serv stems from the concept of *push community* sketched in WebBIS [8]. WebBIS however does not provide means for specifying: (i) a global view of a composite service, and (ii) multi-attribute provider selection policies.

CrossFlow [23] and WISE [27] are inter-organisational workflow management platforms that focus on inter-connecting business processes for e-commerce. They consider important requirements of B2B applications such as dependability and external manageability. They

differ from Self-Serv in that they do not consider the issue of multi-attribute provider selection in a dynamic environment (where providers join and leave communities continuously), nor the peer-to-peer orchestration of process definitions.

CPM [11] 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. A 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 task, it must notify it to all the other players. Hence, if a process involves N tasks and M players, its execution requires the exchange of $N \times M$ messages: far more than required as shown in our experiments. Moreover, CPM requires that all the players participating in an inter-organisational process deploy a full-fledged workflow engine to cater for the coordination with the other players, whereas in Self-Serv the coordination between entities is handled through lightweight schedulers (the state coordinators).

Self-Serv's peer-to-peer orchestration model has also some similarities with the one used in the Mentor distributed workflow system [28]. Given a workflow specified as a state and an activity chart, Mentor partitions it into several sub-workflows, each encompassing all the activities that are to be executed by a given entity within an organisation (thereby assuming that this information is statically known). Each of these sub-workflows is itself specified as a statechart. [28] describes some optimization techniques that reduce the number and the size of the messages exchanged by the sub-workflows, leading to a *weak synchronisation* model close (though using different techniques) to that of Self-Serv. Also in the context of the Mentor project, [22] further considers the issue of configuring a distributed workflow system in order to meet performance and availability constraints while minimising the system costs. 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 during the deployment of the workflow, which is a restrictive assumption in the context of service composition where providers can leave and join a community or alter the characteristics of their offers (e.g. the QoS or the price) after the composite service has been defined and deployed.

ADEPT [26] 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 components in Self-Serv. Each sub-workflow in ADEPT, is assigned to an autonomous agent. When the agent responsible for a workflow, needs to invoke a sub-workflow, it has to negotiate with the agent(s) that provide it. This

contrasts with Self-Serv where the selection of the component service within a community is done through the evaluation of a selection policy. In this sense, ADEPT and Self-Serv complement each other.

FUSION [38] is a framework for building and managing service portals. It provides a description model for Web service methods as well as a Web Services Execution Specification Language (WSESL). An optimal execution plan can be automatically generated from the abstract requirements specified in this language. In this sense, Self-Serv and FUSION both aim at facilitating the rapid composition of Web services. However, Self-Serv differs from FUSION in that it considers the peer-to-peer orchestration of composite services. Also, the issue of deriving an optimal execution plan is not considered in Self-Serv, but instead Self-Serv supports dynamic service selection through the notion of service community.

DAML-S [29] focuses on enhancing the descriptions of services with higher level abstractions (e.g., constraints on operations, legal order of invocations) in order to capture the service's behaviour so that users can better understand the service execution semantics and how to interact with it. Specifically, DAML-S aims at defining ontologies for service description that will allow software agents to reason about the properties of services. We also mention the Web Service Modeling Framework (WSMF) [21] which combines Web semantics and Web service technologies into an integrated framework. The issues addressed in this area are complementary to those addressed in Self-Serv.

7 Conclusion

Self-Serv is a framework supporting the model-driven development and decentralised execution of composite Web services. The main features of Self-Serv are:

- An approach to model composite Web services based on statecharts. XML code capturing the logic of the composition is generated directly from the resulting models, thereby facilitating the use of agile software development approaches.
- A divide-and-conquer approach to manage large amounts of services by grouping them into communities; communities are responsible for the management of their membership and for the runtime selection of services against user's profiles.
- A decentralised execution model in which the providers participating in a composite service interact with each other in a peer-to-peer way to ensure that the control and data flow dependencies expressed in the schema of the composite service are respected.

Self-Serv has been implemented as a platform that provides graphical tools for: (i) specifying the schema of composite services; (ii) deploying these specifications over a network; (iii) running instances of the deployed composite services; and (iv) managing communities of services. Together, these tools provide an environment for the rapid development and testing of composite Web services.

The Self-Serv platform has been used to validate the feasibility and benefits of the proposal. In particular, several relatively large composite services have been specified and deployed using the platform. The results have been encouraging: services with over 15 nodes and complex control and data-flow dependencies have been specified in less than an hour, and services with as many as a hundred components have been deployed in seconds. In addition, we have used the platform to experimentally measure the performance of the peer-to-peer orchestration approach with respect to a centralised approach. The results of these experiments show that peer-to-peer orchestration leads to a more even load distribution among the participants, and as a result, to lower execution times.

To handle the case when one or more component services fails or is unavailable, we are considering extending the composition model to integrate transactional semantics for a group of states in a statechart. Another interesting extension to the work reported here is the study of the self-adaptability of composite services. The idea is that the history of past executions of a composite service can be used to dynamically optimise an ongoing execution according to a given set of parameters such as time, price, quality of service, etc. At present, the runtime optimisation of service executions in Self-Serv is done independently at each node of the composite service, i.e. each community decides locally which of its members should execute a given service invocation. A *global* optimisation approach could lead to better execution with respect to the above parameters.

Acknowledgments

The authors would like to thank Marie-Christine Fauvet, Anne Ngu, and The Phuong Nguyen for our fruitful discussions regarding the design of Self-Serv. We also thank Eileen Oi-Yan Mak, Nathan Wong, Alex Yue-Fai Tang and Murray Spork their participation in the implementation of Self-Serv.

References

[1] W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow Patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.

[2] Ariba Inc, Microsoft Co, and IBM Co. Universal Description, Discovery and Integration of business for the web. `http://www.uddi.org`, 2000.

[3] B. Benatallah, F. Casati, F. Toumani, and R. Hamadi. Conceptual Modeling of Web Services Converstations. In *Proc. of the 15th International Conference on Advanced Information Systems (CAiSE'03)*, Klagernfurt/Velden, Austria, June 2003.

[4] B. Benatallah and F. Casati, editors. Special Issue on Web Services. *Distributed and Parallel Databases, An International Journal*, 2002.

[5] B. Benatallah, M. Dumas, M.C. Fauvet, and H.Y. Paik. Self-Coordinated and Self-Traced Composite Services with Dynamic Provider Selection. Technical Report UNSW-CSE-TR-0108, School of Computer Science & Engineering, University of New South Wales, May 2001. Available at `http://www.cse.unsw.edu.au/~qsheng/selfserv.ps.gz`.

[6] B. Benatallah, M. Dumas, Q. Z. Sheng, and A. Ngu. Declarative Composition and Peer-to-Peer Provisioning of Dynamic Web Services. In *Proc. of the 18th IEEE International Conference on Data Engineering (ICDE'02)*, pages 297–308, San Jose, USA, 2002.

[7] B. Benatallah, Q. Z. Sheng, and M. Dumas. The Self-Serv Environment for Web Services Composition. *IEEE Internet Computing*, 7(1):40–48, January/February 2003.

[8] C. Bussler, F. Casati, S. Ceri, D. Georgakopoulos, T. Özsu, and M. Shan, editors. *Proceedings of the 1st VLDB Workshop on Technologies for E-Services*, Cairo, Egypt, May 2000.

[9] F. Casati, D. Georgakopoulos, and M. Shan, editors. Special Issue on E-Services. *VLDB Journal*, 24(1), 2001.

[10] F. Casati and M.-C. Shan. Dynamic and Adaptive Composition of E-Services. *Information Systems*, 26(3):143–162, May 2001.

[11] Q. Chen and M. Hsu. Inter-Enterprise Collaborative Business Process Management. In *Proc. of 17th International Conference on Data Engineering (ICDE'01)*, pages 253–260, Heidelberg, Germany, April 2001. IEEE Computer Society.

[12] J. Clark and S. DeRose. XML Path Language (XPATH) Version 1.0.

[13] E. Cobb. The Evolution of Distributed Component Architectures. In *Proc. of the 9th International Conference on Cooperative Information systems (CoopIS'01)*, pages 7–21, Trento, Italy, September 2001.

[14] F. Curbera, Y. Goland, J. Klein, F. Leymann, D. Roller, S. Thatte, and S. Weerawarana. Business Process Execution Language for Web Services. `http://dev2dev.bea.com/techtrack/BPEL4WS.jsp`.

[15] A. Dogac and I. Cingil. A Survey and Comparison of Business-to-Business E-Commerce Frameworks. *ACM SIGecom Exchanges*, 2(2):14–25, June 2001.

[16] A. Dogac, Y. Tambag, P. Pembecioglu, S. Pektas, G. B. Laleci, G. Kurt S. Toprak, and Y. Kabak. An ebXML Infrastructure Implementation through UDDI Registries and RosettaNet PIPs. In *Proc. of 2002 ACM SIGMOD International Conference on Management of Data*, Madison, Wisconsin, USA, June 2002.

[17] A. Dogac, editor. Special Issue on Electronic Commerce. *ACM SIGMOD Record*, 27(4), December 1998.

[18] A. Dogac, guest editor. Special Section on Data Management Issues in E-commerce. *ACM SIGMOD Record*, 31(1), March 2002.

[19] M. Dumas and A. ter Hofstede. UML Activity Diagrams as a Workflow Specification Language. In *Proc. of the International Conference on the Unified Modeling Language (UML'01)*, Toronto, Canada, October 2001.

[20] A. Elmagarmid and W. J. McIver. The Ongoing March Toward Digital Government. *IEEE Computer*, 34(2), Feburary 2001.

[21] D. Fensel and C. Bussler. The Web Service Modeling Framework WSMF. *Electronic Commerce Research and Applications*, 1(2):113–137, 2002.

[22] M. Gillmann, J. Weißenfels, G. Weikum, and A. Kraiss. Performance and Availability Assessment for the Configuration of Distributed Workflow Management Systems. In *Proc. of the 7th International Conference on Extending Database Technology (EDBT'00)*, pages 183–201, Konstanz, Germany, March 2000.

[23] P. Grefen, K. Aberer, H. Ludwig, and Y. Hoffner. CrossFlow: Cross-Organizational Workflow Management for Service Outsourcing in Dynamic Virtual Enterprises. *Special Issue on Infrastructure for Advanced E-Services, Bulletin of the Technical Committee on Data Engineering*, 24(1), March 2001.

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

[25] IBM WSTK Toolkit. http://alphaworks.ibm.com/tech/webservicestoolkit.

[26] 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.

[27] 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.

[28] 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.

[29] S. Narayana and S. McIlraith. Simulation, Verification and Automated Composition of Web Services. In *Proc. of the 11th International World Wide Web Conference (WWW'02)*, Honolulu, USA, May 2002.

[30] P. O'Kelly. B2B Content and Process Integration. `http://www.psgroup.com/`, November 2000.

[31] 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 12the International Conference on Advanced Information Systems Engineering (CAiSE'00)*, Stockholm, June 2000.

[32] Web Services Description Language (WSDL). `http://www.w3.org/wsdl`.

[33] M. Shan, A. Umar, and Y. Zhang, editors. *Proceedings of the 12th International Workshop on Research Issues on Data Engineering*, San Jose, USA, February 2002.

[34] Q. Z. Sheng, B. Benatallah, M. Dumas, and E. Mak. SELF-SERV: A Platform for Rapid Composition of Web Services in a Peer-to-Peer Environment. In *Proc. of the 28th International Conference on Very Large Data Bases (VLDB'02)*, Hong Kong, China, August 2002.

[35] Q. Z. Sheng, B. Benatallah, R. Stephan, E. Mak, and Y. Q. Zhu. Discoverying E-Services Using UDDI in SELF-SERV. In *Proc. of the International Conference on E-Business (ICEB'02)*, pages 396–401, Beijing, China, May 2002.

[36] Simple Object Access Protocol (SOAP). `http://www.w3.org/TR/SOAP`.

[37] M. Stolze and M. Stoebel. Utility-based Decision Tree Optimization: A Framework for Adaptive Interviewing. In *Proc. of the 8th International Conference on User Modelling*, pages 105–116, Sonthofen, Germany, 2001.

[38] D. VanderMeer, A. Datta, K. Dutta, H. Thomas, K. Ramamritham, and S. Navathe. FUSION: A System Allowing Dynamic Web Service Composition and Automatic Execution. In *Proc. of the IEEE International Conference on E-Commerce(CEC'03)*, pages 399–404, California, USA, June 2003.

[39] Web Services Conversation Language (WSCL). `http://www.w3.org/TR/wscl10`.

[40] B. Yang and H. Garcia-Molina. Comparing Hybrid Peer-to-Peer Systems. In *Proc. of 27th International Conference on Very Large Data Bases (VLDB'01)*, Roma, Italy, 2001.

[41] L. Zeng, B. Benatallah, M. Dumas, J. Kalagnanam, and Q. Z. Sheng. Quality Driven Web Services Composition. In *Proc. of the 12th International World Wide Web Conference (WWW'03)*, Budapest, Hungary, May 2003.

[42] L. Zeng, B. Benatallah, and A. Ngu. On Demand Business-to-Business Integration. In *Proc. of the 9th International Conference on Cooperative Information Systems (CoopIS'01)*, pages 403–417, Trento, Italy, September 2001.