Declarative Composition and Peer-to-Peer Provisioning of Dynamic Web Services

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

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

[2] *School of Information Systems*
Queensland University of Technology
GPO Box 2434, Brisbane QLD 4001, Australia
m.dumas@qut.edu.au

Abstract

The development of new services through the integration of existing ones has gained a considerable momentum as a means to create and streamline business-to-business collaborations. Unfortunately, as Web services are often autonomous and heterogeneous entities, connecting and coordinating them in order to build integrated services is a delicate and time-consuming task. In this paper, we describe the design and implementation of a system through which existing Web services can be declaratively composed, and the resulting composite services can be executed following a peer-to-peer paradigm, within a dynamic environment. This system provides tools for specifying composite services through statecharts, data conversion rules, and provider selection policies. These specifications are then translated into XML documents that can be interpreted by peer-to-peer inter-connected software components, in order to provision the composite service without requiring a central authority.

1. Introduction

The automation of *Web services* interoperation is gaining a considerable momentum as a paradigm for effective Business-to-Business (B2B) collaboration [13, 3]. Established enterprises are continuously discovering new opportunities to form alliances with other enterprises, by offering value-added integrated services. By Web service (also called *e-service*)[1], we mean a semantically well defined abstraction that allows users to access functionalities offered by Web applications. A typical example of a Web service is booking an airline ticket through an HTML-based interface.

The main goal of our work is to enhance the fundamental understanding of how to facilitate large-scale interoperation of Web services. In this paper, we distinguish the following key issues when composing and executing Web services:

- **Fast composition:** The "why" part of Web services composition is now widely understood [13, 11]. However, the technology (i.e, the "how" part) to compose and execute Web services in appropriate time-frame, has not kept pace with the rapid growth and volatility of available opportunities. Indeed, the development of integrated Web services is still largely ad-hoc, time-consuming and requiring a considerable effort of low-level programming. This approach is clearly tedious and hardly scalable because of the volatility and size of the Web. The need for fast composition and deployment of Web services, will require a high-level declarative service composition language.
- **Scalable composition:** The number of services to be integrated may be large. Consequently, approaches where the development of an integrated service requires the understanding of each of the underlying services are inappropriate. In addition, Web services may need to be composed as part of a short term partnership, and then disbanded when the partnership is no longer profitable. This form of partnership does not assume any *a priori* defined relationships between services. Thus, the integration of a large number of dynamic Web services, requires scalable and flexible techniques.
- **Distributed execution:** The execution of a composite service in existing techniques is usually centralised, whereas the participating services are distributed and autonomous. A centralised execution model incurs sever problems including, scalability, availability, and security problems [5]. Given the highly dynamic and distributed nature of Web services, we believe that novel techniques involving peer-to-peer execution of services will become increasingly attractive. Peer-to-peer computing is gaining a considerable momentum,

[1] In the remainder, we will use the terms e-service, Web service, and service interchangeably

Proceedings of the 18th International Conference on Data Engineering (ICDE'02)
1063-6382/02 $17.00 © 2002 **IEEE**

as it naturally exploits the distributed nature of the Internet [14].

In this paper, we overview the design and implementation of *SELF-SERV* (compoSing wEb accessibLe inFormation & buSiness sERVices): a framework for dynamic and peer-to-peer provisioning of Web services. In SELF-SERV, Web services are *declaratively* composed, and the resulting composite services are executed in a *decentralised* way within a *dynamic* environment. In a nutshell, the salient features of SELF-SERV are:

- A *declarative* language for composing services based on statecharts [8]: 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 communities* to architect the composition of a potentially large number of dynamic services. Service communities are essentially containers of alternative services. They provide descriptions of desired services (e.g., providing flight booking interfaces) without referring to any actual provider (e.g., UA flight-booking Web service). Actual providers can register with any community of interest to offer the desired service. They can leave these communities at any time.
- A *peer-to-peer* service execution model, whereby the responsibility of coordinating the execution of a composite service, is distributed across several peer 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 manage service execution. The knowledge required at runtime by each of the coordinators involved in a composite service (e.g. location, peers, and control flow routing policies) is statically extracted from the service's statechart and represented in a simple tabular form. In this way, the coordinators do not need to implement any complex scheduling algorithm.

The remainder of this paper is organised as follows. Section 2 describes SELF-SERV's approach to service composition. Section 3 discusses SELF-SERV's peer-to-peer execution model for composite services. Section 4 presents SELF-SERV's system architecture and describes the implementation of this architecture using Java and XML-oriented tools. Finally, section 5 gives a brief overview of related work and section 6 provides some concluding remarks.

2 Composing Web Services

In this section, we describe those concepts and functionalities of SELF-SERV which are intended to provide a high-level language for composing pre-existing services.

2.1. Types of services

SELF-SERV distinguishes 3 types of services: *elementary services*, *composite services*, and *service communities*.

An elementary service is an individual Internet-accessible application (e.g., a Java program) that does not rely on another Web service to fulfill user requests. An example of an elementary service might be a Web form-based interface to a weather information source.

A composite service aggregates multiple Web services, which are referred to as its *components*. An example of a composite service would be a Web-accessible travel preparation service, integrating autonomous 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). In order to ensure that all services provide a uniform interface, each service in SELF-SERV is *wrapped* by a software component hosted by its provider. A service's wrapper acts as its entry point, in the sense that it handles requests for executing the operations provided by the service.

2.2 Elementary Services

The operations of an elementary service are realized in terms of calls to proprietary/legacy applications. Specifically, each operation of an elementary service is associated with a *translator*. A *translator* is mainly used to map the SELF-SERV operation into the format understood by the underlying proprietary/legacy applications. For instance, assume that `Travel Insurance (TI)` is an elementary service that provides an operation called `getInsurance`. A corresponding translator, say `TI_translator`, associates `getInsurance` with a routine that calls, e.g.,

a Java class method from the underlying application. The development of translators is left to the responsibility of the provider of the service.

2.3. Composite Services

The operations of a composite service are expressed as a composition of operations of other Web services using statecharts [8]. Encoding the flow of operation invocations as statecharts have several advantages. First, statecharts possess a formal semantics, which is essential for analysing composite service specifications. Next, statecharts are becoming a standard process modeling language as they have been integrated into the Unified Modeling Language (UML). Finally, statecharts offer most of the control-flow constructs found in existing workflow specification languages (branching, concurrent threads, structured loops).

2.3.1 Overview of Statecharts

A statechart is made up of states and transitions. Transitions are labeled by *ECA* (Event Condition Action) rules. 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 the execution of a service, whether elementary or composite. Accordingly, each basic state is labeled 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 contain one or several entire statecharts within them. Compound states come in two flavors: OR and AND states. An OR-state contains a single statechart, while an AND-state contains several statecharts (separated by dashed lines) which are intended to be executed concurrently. Each of these statecharts is called a *concurrent region*. When a compound state is entered, its initial state(s) become(s) active. The execution of a compound state is considered to be completed when it reaches (all) its final state(s). Initial states are denoted by filled circles, whereas final states are denoted by two concentric circles.

2.3.2 Data Flow and Conversion Rules

An operation of a composite service can be seen as having input parameters, output parameters, consumed and produced events, and a statechart glueing these elements together. The input and output parameters 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, and the produced events can be generated by the actions of the statechart. Moreover, the statechart contains a set of invocations of component services. Each of these invocations is described by the name of the service, the name of the operation, the effective input parameters, and the variables to which the output of the operation are assigned.

In addition to input and output parameters, the conditions and the actions of a statechart implementing a composite service operation may refer to other variables, namely *internal* variables. Specifically, an internal variable is a data item that affects the outcome of a service execution, but which is not an input nor an output parameter of the operation that the statechart implements. An internal variable can be used in, e.g., one of the branching conditions of statecharts.

To summarise, a variable appearing 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 variable. The value of an internal variable may be: (i) obtained from the output of a service invocation, (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:

S::m(Q1, ..., Qn, &V1, ..., &Vn)

The semantics of this expression is an invocation of the operation `m` provided by service `S`, with input parameters provided by queries `Q1`, ..., `Qn`, and such that the outputs of the invocation are assigned to variables `V1`, ..., `Vn`. A query `Qi` can be simply a variable name or any other query. SELF-SERV adopts XPath [6] 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`.

2.3.3 Example

Figure 1 contains the statecharts of two composite services, namely `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 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.

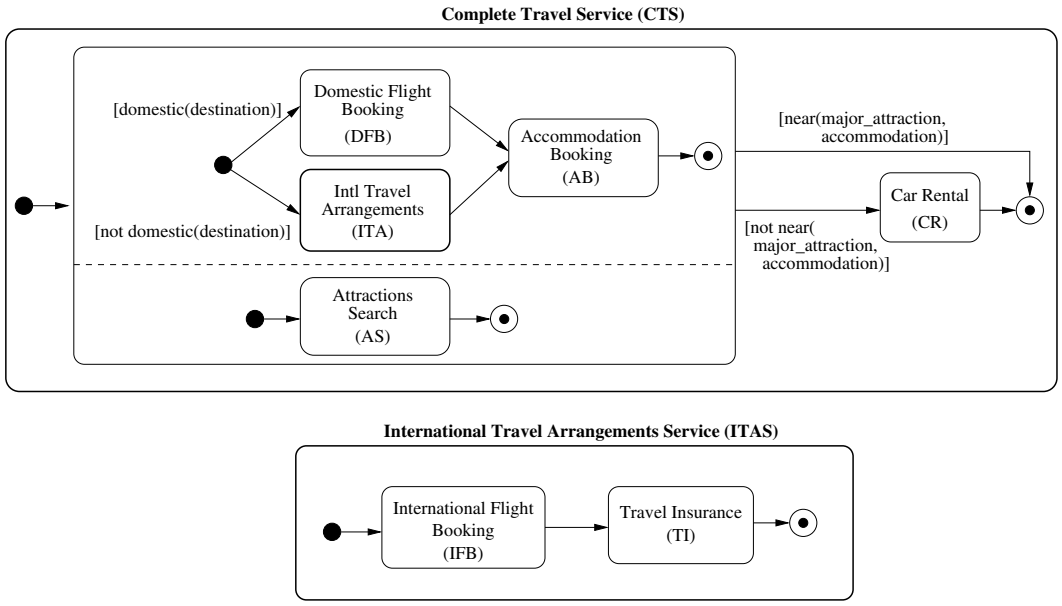

Figure 1. The Travel Solution composite service.

Table 1 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 2 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 2 are input parameters of `CTS` (e.g., `minDepartureDate`, `maxDepartureDate`, `destination`), while others are internal variables (e.g., `departureDate`, `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`.

Table 1. Signature of the operation prepareTrip **of CTS, and signatures of the service operations that it invokes (see Figure 1).**

```
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 accommDetails, 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 numberOfStars,
        out float price, out XMLDoc accommDetails)
ASS::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)
```

We also note that 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

Table 2. Invocation table of CTS::prepareTrip

State	Invocation
AS	ASS.getAttractions(destination, &attractions)
DFB	DFBS.booking(minDepartureDate, maxDepartureDate, minReturnDate, maxReturnDate, starRating, 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, &accommPrice, &accommDetails)
CR	CRS.booking(destination, name, departureDate, returnDate, &rentalPrice, &rentalDetails)

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`.

2.4 Service Communities

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). Service providers may use services platforms of their choice (e.g., Sun Jini or HP e-speak) to advertise and locate Web services including communities [2].

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 `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`.

The means by which a community chooses a member to execute an operation is specified via a *selection policy*. A selection policy can be based, e.g., on an auction, or any ranking algorithm involving parameters such as customer's profile, Web Service's reliability, etc. If the selection is based on auctions, the community essentially works as an auction house in which the members bid for executing operations. A selection strategy is implemented by a program which takes as parameter relevant data such as the operation's input parameter values, the user's profile, past execution logs, input from humans, etc. The low-level specification and implementation of selection policies is out of the scope of this paper.

3. Peer-to-Peer Provisioning of Web Services

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

SELF-SERV's execution model is based on the idea that each state `ST` appearing in a composite service specification is represented by a *state coordinator* responsible for:

- Receiving notifications of completion from other state coordinators and determining from these notifications when should state `ST` be entered.
- Invoking the service labelling `ST` whenever all the preconditions for entering `ST` are met. This invocation is

[2] In the current implementation of our system, we use the service discovery engine of the *AgFlow* prototype [15]).

done by sending a message to the service's wrapper and waiting for a reply.

- Notifying the execution's completion to the coordinators of the states which may need to be entered next.
- While state ST is active, receive notifications of external events (such as a cancellation), determine if ST should be exited because of these event occurrences, and if so, interrupt the service execution and notify the interruption to the coordinators of the states which potentially need to be entered next.

In other words, the coordinator of a state is a lightweight scheduler which determines (i) when should a state within a statechart be entered?, (ii) what should be done after the state is entered?, (iii) when should the state be exited?, and (iv) 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 wrappers. 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 C1 to a coordinator C2 expresses the fact that the execution of the state represented by C1 has completed, and that C1 believes that the state represented by C2 needs to be entered. The notification message contains the input parameters of the composite service execution, as well as the up-to-date values of all the internal variables of the statechart that C1 needs to transmit to C2.

On the other hand, the messages exchanged between the coordinators of a state and the wrapper of the service labelling this state are called *service invocations/completions*. A service invocation message contains the name of the service operation that is being invoked, as well as the values of the input parameters. A service completion message contains the values of the return parameters.

3.2. Example

The diagram in Figure 2 shows the messages exchanged by the coordinators and the wrappers during a particular execution of service CTS. 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 wrapper (message 1). Assuming that the trip is international, the wrapper 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 service ITAS (2a.2) and ASS (2b.2) through their wrappers. When the wrapper of ASS returns an output (2b.3), the coordinator of AS sends a control-flow notification both to the wrapper 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 or not until thread 2a is completed. Meanwhile, the wrapper of ITAS starts this 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 wrapper of ITAS (2a.9). This wrapper 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 wrapper of CTS (2a.14a) and to the coordinator of CR (2a.14b). The coordinator of CR and the wrapper 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 wrapper of CTS, and the overall execution is completed.

3.3. Preconditions and postprocessings tables

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

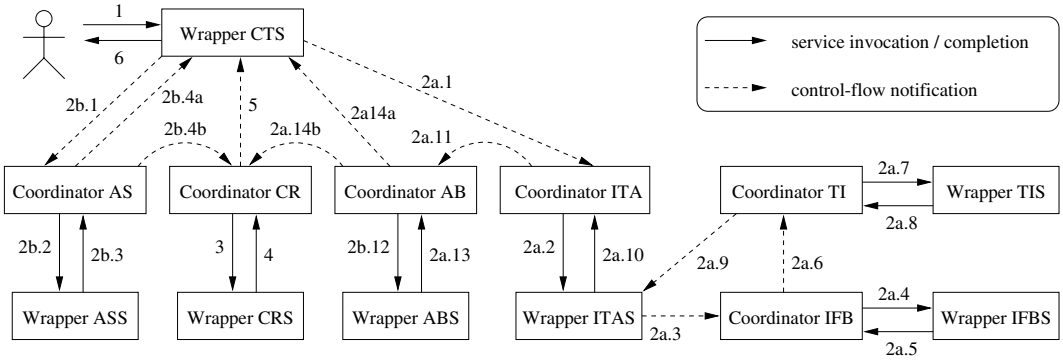

Figure 2. Interactions between the coordinators and the wrappers during an execution of CTS. The acronyms used here are detailed in Figure 1.

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[3] 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)*[4] *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(tn), i.e., source(CT) = source(t_1) and target(CT) = target(tn). 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)]`.

[3]Notice that the definition of compound transition that we adopt, is slightly different from that of [8].

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

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.* □

For the example of 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 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* e1 *and* e2 *is noted e1 ∧ e2 and the semantics is that if an occurrence of* e1 *and an occurrence of* e2 *are registered in any order, then an occurrence of e1 ∧ e2 is generated.*
- *There exists a compound transition* CT *from* ST' *to* ST *such that* C ⊆ ST.
- *If Conjuncts(C) ≠ Conjuncts(Cond(CT)), then the conditions in* CT \ C *are exactly those which cannot be evaluated by the coordinator of* ST'. □

In the example of Figure 1, we have that Postprocessing(AS) = { [true]/notify(CR), [true]/notify(wrapper) }. 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 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.

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 [1], 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, 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. By observing that the postprocessing table of a state ST is the union of the postprocessing actions associated to the outgoing transitions of ST, we deduce that:

PostProc(st) =
let {t1, t2, . . . , tn} are the outgoing transitions of st in
  PostProcTrans(t1) ∪ PostProcTrans(t2) ∪ . . . ∪ PostProcTrans(tn)

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

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, that is, 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.

When 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. A distinction should be made here between the case where CST has no successors (in which case the notification of completion is sent to the wrapper), the case where CST is an OR-STATE, and the case where it is an AND-state. In this latter case, the conditions emanating from CST are not included in the postprocessing table, since their evaluation may require information which is not available when the transition T is taken. For instance, in the case of Figure 1, the condition attractions far from accommodation should not appear in the postprocessing table of state AS, since it cannot be evaluated until state AB is exited, and state AB executes in a region concurrent to that of AS.

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) 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.

PreCond(st) =
let {t1, t2, . . . , tn} be the incoming transitions of st
  PreCondTrans(t1) ∪ PreCondTrans(t2) ∪ . . . ∪ PreCondTrans(tn)

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 and it is therefore taken when the composite service's wrapper 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.

Detailed descriptions of PreCondTrans and PostCondTrans are given in [1]. It can be proven by structural induction (the proof is omitted for space reasons) 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 C1 to another coordinator C2, only if there is a compound transition from the state of C1 to that of C2, and either the state of C2 needs to be entered, or it is impossible for C1 to determine whether the state of C2 needs to be entered or not.

4. Implementing SELF-SERV

In this section, we discribe a prototype implementation of SELF-SERV architecture. This implementation has shown that the ideas behind SELF-SERV fit together, are consistent with one another, and are realizable using existing technologies.

4.1. Architecture

The SELF-SERV's prototype architecture (see Figure 3) is composed of an *interface*, a *service manager*, and a *pool of services*. All these components have been implemented in Java. Currently, three types of proprietary/native services are supported: Domino-based workflows, Java applications that provide access to relational databases via JDBC, and Web-accessible programs. As mentioned earlier in the paper, services communicate through XML documents containing data and control flow information. These documents are exchanged through Java sockets. Oracle's XML Parser 2.0 is used for parsing XML documents.

The *service manager* consists of three modules, namely the *service discovery engine*, the *service editor*, and the *service deployer*. In the current implementation, we use the service discovery engine of the *AgFlow* prototype [15][5]. Service descriptions (e.g., location and properties) are stored in an XML-based meta-data repository called the *service catalogue* implemented on top of the Oracle8i DBMS. The *service editor* provides facilities for defining new services and editing existing ones. A service is edited through a visual interface, and translated into an XML document for subsequent analysis and processing by the service deployer.

Any service wishing to participate in the SELF-SERV platform, needs to register with the service discovery engine. For this purpose, the administrator of the service has to download and install a pre-existing class, namely *Coordinator*, implementing the concept of coordinator (see Section 4.2). An administrator registering a service is also required to build a wrapper of this service, by downloading and configuring a pre-existing class *Wrapper* provided by the SELF-SERV platform (see Section 4.2). The only infrastructure required to install and configure these classes are Java and the XML parser. 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 *service deployer* is responsible for generating the preconditions 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, while the outputs are routing tables formatted in XML as well. 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. It also assists the composer in the deployment of the wrapper of the composite service. At present, security issues related to uploading tables are not considered.

Notice that the statechart implementing a composite service is not uploaded/downloaded into the hosts of the component services. Instead, a host providing a service S will only receive the routing tables of the states where an invocation to S is made. In this way, it is not required that each participant of the SELF-SERV platform deploys the whole system. Instead, the only parts of the system which are required by all the participants are the classes implementing the coordinators and the wrappers.

We note that a service may be involved in several compositions (e.g., S1 may be a component of both CS1 and

[5]The presentation of this module is out of the scope of this paper.

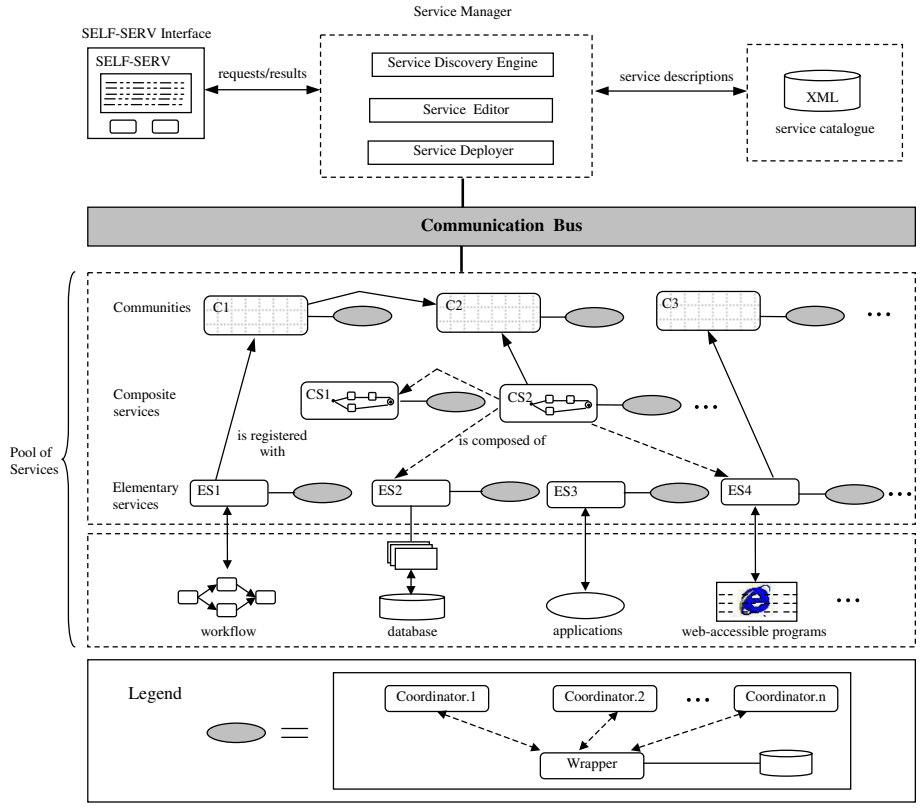

Figure 3. Architecture of the SELF-SERV prototype.

CS2), and may even be referenced more than once in a given composition (e.g., the state ST1 of CS invokes the operation op1() of S, and the state ST2 of CS invokes the operation op2() of S). Consequently, the host of a service S, may need to store several preconditions and several post-processing tables (one table of each type per state in which S is invoked). In order to ensure that a coordinator is able to retrieve the right table at the right moment, each preconditions and each postprocessings table is identified by a pair (cs-id, state-id), where cs-id is the identifier of a composite service and state-id is the identifier of a state of cs-id's statechart.

4.2. The *Coordinator* and *Wrapper* classes

Message Contents. The messages exchanged between the coordinators (i.e. the control-flow notifications) are identified by tuples of the form (senderState_id, receiverState_id, compositeSVC_id, instance_id), where senderState_id is the identifier of the state that is being exited, receiverState_id is the identifier of the state that potentially needs to be entered, compositeSVC_id is the identifier of the composite service, and instance_id is the identifier of the instance of this composite service to which the message relates.

On the other hand, the invocation messages sent by the coordinators to their underlying wrappers are identified by tuples of the form (compositeSVC_id, composite-Instance_id, service_id, operation_id), such that compositeSVC_id is the identifier of the composite service, compositeInstance_id is the identifier of the instance of the composite service, service_id is the identifier of the invoked service, and operation_id is the identifier of the invoked operation within this service.

The Class *Wrapper*. The concept of service wrapper is mapped into an abstract class called *Wrapper*, which defines (among others) methods for (i) invoking an operation of the service (method start_service), and (ii) collecting the outputs of a service instance (method get_service_result). To account for the three kinds of services supported in SELF-SERV, the abstract class *Wrapper* is specialised into three concrete subclasses: *ElementaryWrapper*, *CompositeWrapper*, and *Community-Wrapper*. Each of these subclasses realises the above methods in a different way.

In the *ElementaryWrapper* class, the method start_service loads the translator program corresponding to the invoked operation, and invokes the

underlying application program. The outputs of this application program are translated back into the format of SELF-SERV using the translator, and made accessible through the method `get_service_result`.

In the *CommunityWrapper* class, the method `start_service` begins by invoking a program implementing the selection policy. This program returns the identifier of one of the members registered with the community. Then the selected service is invoked through its wrapper, which can be implemented by an instance of the class *ElementaryWrapper*, *CompositeWrapper* or even *CommunityWrapper*.

Finally, in the *CompositeWrapper* class, the method `start_service` begins by accessing the postprocessings table of the initial state of the 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 composite wrapper. Once all the control-flow notifications are received, the outputs of the service invocation are made available through the `get_service_results` method.

The classes *ElementaryWrapper*, *CommunityWrapper* and *CompositeWrapper*, may need to dynamically load external programs. For example, the *ElementaryWrapper* may need to load the translator programs that realise the service operations, while the class *CommunityWrapper* may need to load and execute the programs implementing the selection policies. This is done using the reflection capabilities of the Java platform.

The Class *Coordinator*. The functionalities of the coordinators are realised by a class called *Coordinator*, which provides methods for receiving, processing, generating and sending control-flow notifications, service invocation, and service completion messages.

More precisely, the class *Coordinator* implements a software component made up of a *container* and a *pool of objects*.[6] The container is a process that runs continuously, listening to a socket through which control-flow notification messages from other coordinators are received. When the container 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 delegated 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 pool of objects 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 delegated to the coordinator object that was created when the first message related to that instance was received. This object is retrieved from the pool of objects 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 the wrapper of this service. Once the corresponding completion message is received, the object polls the result parameters from the service's wrapper, generates one or several control-flow notification messages (according to the information contained in the relevant postprocessings table), and dispatches these messages through sockets. From there on, the coordinator object is no longer needed, so it is removed from the pool and destroyed.

5. Related Work

Service composition is a very active area of research and development [13, 3]. In this section, we focus on research efforts that are closely related to our work, namely *eFLOW* [4], *CMI* [12], *CrossFlow* [7], *WISE* [9], *CPM* [5], and *Mentor* [10, 3].

CMI [12] and eFlow [4] 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 is in contrast 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 [2]. WebBIS however does not provide a means for specifying a global view of a composite service.

[6] This pool of object is different from the pool of services discussed previously.

CrossFlow and WISE are inter-organisational workflow management platforms that focus on loosely coupled processes. They consider important requirements of B2B applications such as dependability and external manageability. However, the dynamic and peer-to-peer provisioning of services is not explicitly supported.

CPM supports the execution of inter-organisational business processes through peer-to-peer collaboration between a set of workflow engines, each representing a player in the overall process. 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. 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).

In Mentor, the problem addressed is that of distributing the execution of workflows expressed as state and activity charts. The idea is to partition the overall workflow specification into several sub-workflows, each encompassing all the activities that are to be executed by a given entity within an organisation. Mentor differs from SELF-SERV, in that it assumes that the assignment of activities to entities is known at workflow definition time: a restrictive assumption in the context of service composition. Moreover, as in CPM, Mentor imposes that each organisation participating in a distributed workflow deploys a full-fledged execution engine, capable of interpreting state and activity charts.

6. Conclusion

In this paper, we have presented the design and the implementation of SELF-SERV, a framework for declarative Web services composition using statecharts, where the resulting services can be executed in a decentralized way within a dynamic environment. The concept of communities is used to form alliances among a potentially large number of dynamic services. The underlying execution model allows services participating in a composition, to collaborate in a peer-to-peer fashion in order to ensure that the control and data flow dependencies expressed by the schema of a composite service are respected. So far, we have implemented a prototype that realises the execution of composite service in a peer-to-peer manner. We illustrated the viability of the proposed approach. On-going work includes the assessment of the performance and scalability of SELF-SERV.

References

[1] B. Benatallah, M. Dumas, M. Fauvet, and H. 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.

[2] B. Benatallah, B. Medjahed, A. Bouguettaya, A. Elmagarmid, and J. Beard. Composing and maintaining web-based virtual enterprises. In *Proc. of the 1st VLDB Workshop on Technologies for E-Services*, Cairo, Egypt, September 2000.

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

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

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

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

[7] CrossFlow Project web page. http://www.crossflow.org/.

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

[9] 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.

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

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

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

[13] G. Weikum editor. Special issue on infrastructure for advanced e-services. *IEEE Data Engineering Bulletin*, 24(1), March 2001.

[14] B. Yang and H. Garcia-Molina. Comparing Hybrid Peer-to-Peer Systems. In *Proc. of 27th Int. Conference on Very Large Data Bases*, Roma, Italy, 2001.

[15] L. Zeng, B. Benatallah, A. Ngu, and P. Nguyen. AgFlow: Agent-based Cross-Enterprise Workflow Management System. In *Proc. of 27th Int. Conference on Very Large Data Bases*, Roma, Italy, 2001.