Peer-to-Peer Traced Execution of Composite Services

Marie-Christine Fauvet[1]**, Marlon Dumas[2], Boualem Benatallah[1], and Hye-Young Paik[1]

[1] School of Computer Science & Engineering, The University of New South Wales, Sydney NSW 2052, Australia
[2] Cooperative Information Systems Research Centre, Queensland University of Technology, GPO Box 2434, Brisbane QLD 4001, Australia
{mcfauvet,boualem,hpaik}@cse.unsw.edu.au,m.dumas@qut.edu.au
http://www.cse.unsw.edu.au/~{mcfauvet,boualem,hpaik},
http://www.fit.qut.edu.au/~dumas

Abstract. The connectivity generated by the Internet is opening unprecedented opportunities of automating business-to-business collaborations. As a result, organisations of all sizes are forming online alliances in order to deliver integrated value-added services. Unfortunately, due to a lack of tools and methodologies offering an adequate level of abstraction, the development of these integrated services is currently ad hoc and requires a considerable effort of low-level programming, especially when dealing with coordination, communication, and execution tracing issues. In this paper, we present a framework through which business services can be declaratively composed, and the resulting composite services can be executed in a fully traceable manner. The traces of a composite service executions are collected incrementally through peer-to-peer interactions between the involved providers. Once collected, these traces are stored as linked objects in distributed repositories, which are made available for auditing, customer feedback and quality assessment.

1 Introduction

The rapidly growing number of organisations that are making their services accessible through the web, has resulted in a paradigm shift that is gradually transforming the Internet from a repository of information into a vehicle of services. This phenomenon should in turn generate a shift in focus away from the well-known issue of information integration, to the largely unexplored one of service integration. In particular, as new kinds of business intermediaries emerge, it is expected that the practice of developing new services from existing ones (i.e. service composition) will gain a considerable momentum, both as a means to facilitate Business-to-Consumer interactions, and as a foundation to foster Business-to-Business collaborations.

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

Unfortunately, due to a lack of tools and methodologies offering an adequate level of abstraction, the composition of services is currently done through ad-hoc assemblages of manifold technologies, which often require a considerable effort of low-level programming. As a contribution to the development of higher-level abstractions for service composition, we present in this paper a platform (namely Self-Serv) through which Internet-accessible services provided by different organisations can be declaratively composed, and the resulting *composite services* can be executed in a fully *traceable* way, following a *decentralised* paradigm, and within a *dynamic* environment. By decentralised paradigm, we mean that the providers participating in a composite service orchestrate the overall execution through peer-to-peer interactions, instead of being dependent on a centralised scheduler, which could constitute a bottleneck. By dynamic environment, we mean that a composite service is not bound to a particular set of service providers. Instead, an organisation participating in the provisioning of a composite service, is free to interrupt its participation without blocking the availability of the composite service. Similarly, new organisations may decide to provide a particular constituent of a composite service in future executions of it.

Tracing past and ongoing executions is an essential requirement in the area of business process management. Organisations need to track down their activities in order to ensure explainability in case of failure or auditing, and to revise also their practices both for increasing their efficiency, and for improving their customers' satisfaction. Similarly, as organisations form alliances to deliver composite services, the need for tracing the executions of these services will become an increasingly important issue. Traces of service executions can be queried for the following purposes (among others):

Performance evaluation: to make a report on past service executions. A typical query for this purpose would be *"Retrieve the constituents of a composite service whose execution takes the most time in average"*.

Customer feedback: to explain specific failures. A query for this context would be *"Retrieve the traces of all the service executions that have been triggered for a given client"*.

Quality assessment: to detect services whose executions tend to fail, like for example in *"Retrieve the executions of a given service that were frozen at some point for more than 30 minutes, and were cancelled after this period of inactivity"*.

More generally, the question of knowing who did what, is central in a dynamic environment. By querying the traces, the provider of a composite service can audit executions of its constituent services, in order to check, for example, the validity of the bills that are issued by their providers.

Given that in Self-Serv the executions of a composite service are carried out in a decentralised manner, it would be inconsistent to collect their traces through direct communication between the providers of each constituent service and a centralised entity: an approach which creates a potential bottleneck. Instead, in Self-Serv, the collection of traces is carried out through peer-to-peer exchange of partial traces between the providers participating in a composite service. The

resulting traces are then stored in a decentralized manner: each provider being responsible for storing the traces of its own activities. These distributed traces are connected through universal references (e.g. URLs), in such a way that upon request, it is possible to retrieve all the details of the execution of a composite service.

The remainder of the paper is organised as follows. In section 2 we describe our approach to service composition using statecharts. Section 3 discusses the collaborative execution of services, which is then extended in section 4 to cater for tracing. Finally, section 5 gives an overview of related work and Section 6 provides some concluding remarks.

2 Composite service specification

This section introduces the composition model of Self-Serv. We begin with some definitions, before overviewing the statechart formalism and discussing how it is applied to composite service specification. Finally, we conclude with an example.

2.1 Description of the approach

Each service within Self-Serv, provides an interface enabling its instantiation and the subsequent execution of the resulting *service instance*. In other words, the interface of a service defines operators such as *instantiate*, *start*, *freeze*, *cancel*, etc., and describes the protocol for invoking each operation, passing its input parameters, and collecting its outputs. This protocol can be based on remote method invocation (e.g. Java RMI [15]) or message exchange (e.g. SOAP [13]).

Self-Serv distinguishes *elementary services* from *composite services*. Elementary services are pre-existing (e.g. legacy) services, whose instances' execution are entirely under the responsibility of an entity called *service provider*. The provisioning of an elementary service may involve a complex business process, but its internals are hidden behind the composite service's interface: the user of an elementary service has no information about how it is implemented.

A composite service on the other hand, is an aggregation of elementary and other composite services, which are referred to as its *constituents*. The semantics of this aggregation can be described from at least three perspectives: (i) The *control-flow perspective* establishes the order in which the constituents are invoked, the signals that may interrupt their execution, etc. (ii) The *provider perspective* gives an organisational anchor to the composite service by establishing which entity is responsible for performing which service. (iii) The *data exchange perspective* captures both the flow of data between services, and the conversion of these data between the potentially heterogeneous data models used by the services participating in the composition.

We have chosen to model the control-flow perspective of composite services through statecharts [5], since they provide the basic constructs found in business process modeling tools (e.g. Workflow Management Systems [8]) while still possessing a formal semantics, which is essential for reasoning about composite

service specifications. Moreover, statecharts are becoming a standard process-modeling language as they have been integrated into the UML [11].

A statechart is made up of states and transitions. Transitions are optionaly labeled by ECA rules. The occurrence of an event fires a transition if (i) the machine is in the source state of the transition, (ii) the type of the event occurrence matches the event description attached to the transition, and (iii) the condition of the transition holds. When a transition fires, its action part is executed and its target state is entered. The event, condition and action part of a transition are all optional. A transition without an event part is said to be *triggerless*. States can be simple or compound. In our approach, a simple state corresponds to the execution of a service, whether it is elementary or composite. Accordingly, each simple state is labeled by a description of a service offer, and the set of parameters that are to be passed to this service upon instantiation. When a basic state is entered, the service that labels it is invoked. The state is normally exited through one of its triggerless transitions, when the execution of the service is completed. If the state has outgoing transitions labeled with events, an occurrence of one of these events provokes the state to be exited, even if the corresponding service execution is ongoing (i.e. this execution is cancelled). Compound states on the other hand, are not directly labeled by a service invocation. Instead, they contain one or several entire statecharts within them. A compound state that contains two or more statecharts (separated by dashed lines), is called an AND-state. The statecharts within an AND-state are intended to be executed concurrently.

The reader can find a comprehensive description of statecharts in [5]. The example in section 2.2 provides a few intuitive notions about statecharts.

The provider perspective is modeled by associating an organisational entity to each service offer. In other words, the concept of service offer in Self-Serv encompasses both: what has to be done? and who has to do it? The organisational entity associated with a service can be either an individual provider or a community of providers. In the former case, the designated provider is responsible for executing all the instances of this service. It may eventually partially or totally delegate the execution of these instances to another provider, but this delegation is hidden to the users of the composite service. On the other hand, a community of providers will systematically and *transparently* delegate the execution of a service to its members. This delegation is carried out by the *representative* of the community. The means by which a community's representative chooses a member to execute a request, is specified via a *selection policy*. It can be based on a 1-N negotiation protocol (e.g. an auction), or on any ranking algorithm involving parameters such as the customer's profile, the provider's reliability, etc., as discussed in [2, 1].

2.2 Example

As a working example, we consider the composite service "Travel Solutions" described in Figure 1. This composite service aggregates several independent services like flight booking, car rental, event attendance planner, etc. It starts

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

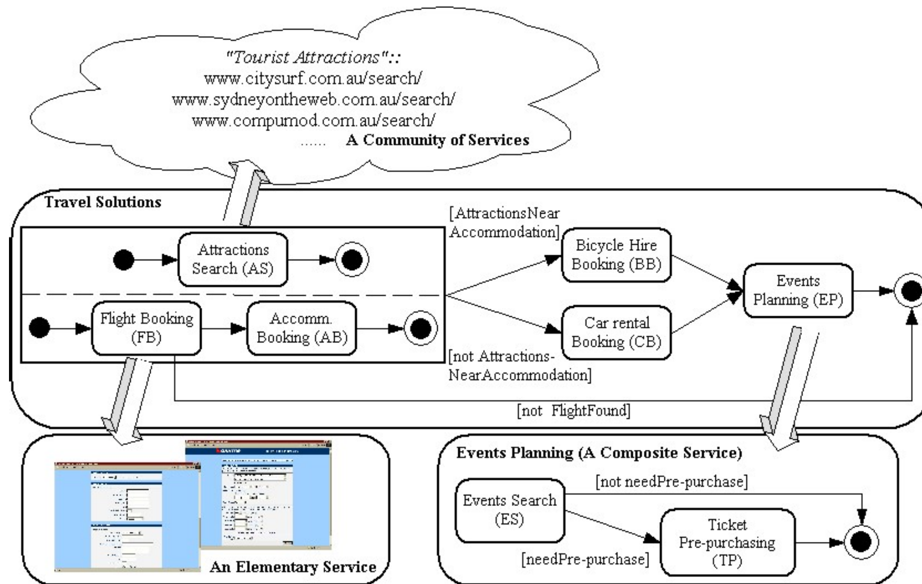

Fig. 1. The "Travel Solutions" composite service.

A constituent of a composite service can be assigned to an individual provider, or to a community of providers. For example, in Figure 1, the flight booking is assigned to a web site, that does not delegate its task to any other entity. Meanwhile, the service that searches for attractions is assigned to a community of providers. This community federates entities such as public tourism offices, and private tourism information sites. When an execution request is addressed to the community, its representative forwards it to one of its members.

3 Composite service execution

In Self-Serv, the coordination between the constituents of a composite service is ensured through peer-to-peer collaboration between software components, hosted by the providers participating in the composition. This approach provides

greater scalability and availability than a centralised one where the execution of a service depends on a central scheduler. In this section, we introduce the two basic concepts of Self-Serv's execution model: service wrappers and state coordinators, and we discuss how they are integrated into Self-Serv's architecture.

3.1 Description of the approach

Service wrappers. Each service, whether elementary or composite, is *wrapped* by a software component hosted by its provider. A service's wrapper provides an implementation of its interface, that is, it implements functions such as *instantiate*, *start*, *cancel*, etc., and it handles conversions between the data model of the service's interface (based on e.g. SOAP [13]), and that of its implementation (based on e.g. a proprietary C++ API). A service's wrapper therefore acts as its entry point, in the sense that it handles requests for executing the service.

State coordinators. Each state ST in a composite service's statechart is represented at runtime by a *coordinator*, which is responsible for: (i) Initiating the execution of the service labeling ST whenever all the preconditions are met. (ii) Notifying the completion of this execution to the coordinators of the states which potentially need to be entered next. (iii) While state ST is active, receive notifications of external events, determine if ST should be exited because of these event occurrences, and if so, interrupt the service execution if it is ongoing, and notify the interruption to the coordinators of the states which potentially need to be entered next.

In other words, the coordinator of a state is a lightweight scheduler which determines: (i) when should a state within a statechart be entered?, (ii) What should be done after the state is entered?, (iii) When should it be exited, and (iv) What should be done after it is exited. The coordinators of a composite service are hosted by the providers of its constituents. The provider of a service is responsible for hosting as many coordinators as there are states which are labeled by it. For instance, in figure 2 service A is associated to three coordinators named Coord.A.1, Coord.A.2 and Coord.A.3 because it is involved in three different service compositions.

Peer-to-peer service execution. When the wrapper of a composite service CS processes a request for executing CS (i.e. when it receives a message `CS.run(...)`) it sends a message to each of the coordinator(s) of the state(s) which need to be entered the first, as indicated by the service's statechart. For the sake of simplicity, let us assume that there is only one such "first" state. The coordinator of this state performs the service invocation which labels its state by sending an invocation message to the corresponding service wrapper. Once the invocation induced by this invocation is completed, the coordinator of the first state sends a notification of completion to the coordinator(s) of the states which need to be entered the next, which in turn perform the service invocation(s) labeling its/their state(s). This peer-to-peer interaction continues until eventually the coordinators of the states which need to be exited the last, send their notifications of completion back to the wrapper of CS, thereby signaling the completion of the overall execution.

Service description. The knowledge required by each coordinator participating in a composite service execution, is statically extracted from the service's statechart by the *service description module* of the Self-Serv system. Specifically, the composite service designer (or service composer) assembles service offers advertised in a repository of services through the service description module. This module then generates and deploys the corresponding composite service state coordinators. Once the service is deployed and assigned to a provider, users, application programs, and even state coordinators belonging to other composite services, can invoke it through its wrapper. Figure 2 summarizes this process.

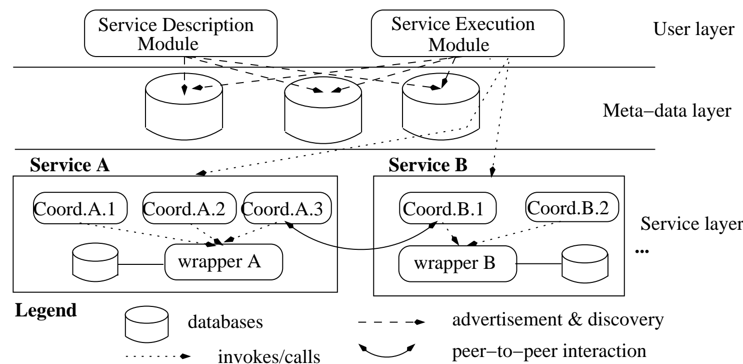

Fig. 2. Partial view of Self-Serv's architecture for composite service management

Extracting the knowledge required by a state coordinator from a composite service's statechart, involves answering the following questions: what are the preconditions for entering a state?, When the execution associated to a state is completed or interrupted by 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 preconditions for entering a state ST are represented by a table whose elements denote rules of the form E[C] such that: (i) E is a conjunction of events of the form ready(ST), meaning that a notification of completion has been received from the coordinator attached to state ST and (ii) C is a conjunction of conditions appearing in the labels of the statechart's transitions. When one of the elements of the preconditions table is triggered, and that its condition evaluates to true, the state is entered, and the service that appears on its label is invoked.

The postprocessings that have to be undertaken when a state is exited are presented as a table whose elements denote rules of the form [C]/A where: (i) C is a conjunction of conditions appearing in the labels of the statechart's transitions and (ii) A is a term of the form notify(ST), meaning that a notification of completion has to be sent to the coordinator associated to the state ST.

The following examples of preconditions and postprocessings tables refer to Figure 1.

– Preconditions(EP) = { ready(CB)[true], ready(BB)[true] }, meaning that the state is entered when a message is received from either the coordinator of the state CB or that BB.
– Preconditions(CB) = { ready(AB) $\wedge$ ready(AS)[not attractions near accommodation] }.
– Postprocessing(ES) = { [need pre-purchase]/notify(TP), [not need pre-purchase]/notify(wrapper) }.
– Postprocessing(AS) = { [true]/notify(CB), [true]/notify(BB) }.

Notice that the condition "attractions near accommodation" is not evaluated before undertaking the postprocessing action notify(BB). This is because evaluating this condition requires the coordinator to know where is the selected accommodation located, and this is only known once the accommodation booking is completed.

Algorithms for deriving the preconditions and the postprocessing for state coordinators of a composite service are detailed in [1].

3.2 Example

To illustrate how the coordinators and the wrappers are deployed, and how they interact, we consider the "Travel Solutions" service described in Figure 1. We assume that this composite service is provided by a company named "Full Tours". The life-cycle of the service starts when a service designer within this company defines the structure of the service using Self-Serv's service description module. This module configures a set of software components implementing the coordinators required to run the composite service. It also assists the designer in deploying the wrapper in one of the servers of "Full Tours", and the coordinators in the dedicated servers provided by the companies referenced in the composite service definition. Once the deployment is completed, the composite service is advertised and can be invoked through its wrapper.

When the wrapper of the "Travel Solutions" service receives an execution request, it sends a message to the coordinators of the states labeled FB and AS (see Figure 1). Upon receiving these messages, these coordinators invoke the services labeling their states. When the service that books a flight completes its execution, the coordinator of the state FB sends a message to that of the state AB. This latter invokes the service that books an accommodation, waits for its completion, and sends a message to the coordinators of the states CB and BB. In the meanwhile, the coordinator of AS sends its completion message to the coordinators of CB and BB too. These completion messages contain the data that must be exchanged between these services, as per the data exchange perspective of the "Travel Solutions" specification. Using these data, the coordinators of BB and CB evaluate the condition "attractions near accommodation" appearing in the labels of their incoming transitions, and accordingly, they decide which state has to be entered. Assuming that the attractions are far from the accommodation, it is the state CB that has to be entered, so the corresponding coordinator invokes the service for renting a car. Once this service completes its execution,

the same coordinator sends a message to the coordinator of the state EP, who sends an execution request to the wrapper of the composite service responsible for searching events. This wrapper initiates the execution of the service that it provides, by sending a message to the coordinator of the state ES, which invokes the service that searches events, waits for its completion, and assuming that tickets for some of the events need to be prepurchased, sends a message to the coordinator of the state TP. This coordinator then invokes the service that purchases tickets, and upon completion, sends a notification to the wrapper of the Event Planning service, which in turn sends a notification to the coordinator of the state EP. Finally, this coordinator sends a message to the wrapper of the "Travel Solutions" service, thereby concluding the overall execution.

4 Service execution tracing

In this section we introduce the mechanisms provided by Self-Serv for keeping trace of past executions of composite services. This functionality is essential for customer support and feedback (i.e. retrieving a given service execution) as well as for detecting deficiencies in the constitution of a composite service (i.e. analysing the past executions of a service in order to retrieve repetitive malfunctionings). First (section 4.1), we describe a model for representing the service execution traces. In section 4.2 we introduce the mechanisms for collecting and storing traces. Section 4.3 illustrates our approach.

4.1 Modeling service execution traces

Simplifying assumptions. For the sake of simplicity, we assume in the sequel that the local coordinators and the wrapper of a composite service share a common time line. This can be achieved using classical clock synchronisation protocols such as NTP [6]. We also assume that all temporal values (instants, durations and intervals), are expressed at the same level of granularity (e.g., the Second or the Minute). Under this assumption, instants and durations are unambiguously designed using integers, while an interval is fully represented as a pair of integers corresponding to its bounds.

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

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

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

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

The information above is modeled by a class named ServiceExecution with two sub-classes ElemServiceExecution and CompServiceExecution. For each execution of an elementary (resp. composite) service, an instance of ElemServiceExecution (resp. CompServiceExecution) is created. For a full description of these classes see [1].

4.2 Collecting and storing execution traces

The responsibility of tracing the executions of a composite service CS is distributed across the wrapper and the local coordinators of this service.

The coordinator of a state ST belonging to the statechart describing CS, is responsible for:

- Receiving information about ongoing executions of CS in the form of collections of references to objects of the class ServiceExecution. The actual content of these objects are stored in repositories managed by the providers participating in the composite service.
- Obtaining a reference to an object of the class ServiceExecution from the wrapper of the service labeling the state ST. This reference is an URL containing both the address of the repository where the value of the object is stored, and the identifier of this object.
- Adding this new reference to the collection of references received from the other coordinators.
- When the state is exited, passing the new collection of references to the coordinators of the states that need to be entered next, or to the wrapper of CS if no state needs to be entered next.

More precisely, when an instance of a composite service CS starts its execution, the wrapper of CS sends the identifier of this instance to the coordinators of the states that need to be entered first. Let us suppose here that there is only one initial state that we call ST, and that this state is labeled by an invocation to a service called S. The coordinator of ST then contacts the wrapper of S, asking it to perform the required invocation. The wrapper of S performs the invocation, and collects information about the parameters passed, the start and

end time of the execution induced by this invocation, and the identity of the *individual* provider that carried out this execution (in the case where the service is offered by a community). With this information, the wrapper of S creates an object of the class ServiceExecution that it stores in a repository maintained by the provider of service S (or the representative, if S is provided by a community). A universal reference (e.g. an URL) to this newly stored object is then created, and passed to the coordinator of ST, which then adds it to an empty collection and passes this collection (with one reference) to the coordinator(s) of the state(s) that need(s) to be executed the next (which is determined using the postprocessing table as discussed in section 3). The coordinator(s) to which this reference is passed, perform(s) a similar operation. On the end, the coordinator(s) of the final states of the composite service, pass(es) its/their collection of references to the wrapper of CS, which stores this collection in a repository maintained by the provider of CS. When a user wishes to query the traces of a composite service, (s)he send her/his query to the wrapper of CS. Should the query need some of the universal references to be resolved, the wrapper of CS will contact the repositories where the data is stored, and poll the actual values of the referenced objects.

As an optimisation aiming at reducing the number of messages exchanged for collecting traces, when an AND-state needs to be entered, the data about the execution trace *before* entering this state is not sent to all the initial states of all of the concurrent threads, but rather to the coordinators of the states that will be entered after the AND-state is exited. Indeed, duplicating this partial execution trace is useless, since when this AND-state will be exited, the traces collected by all its threads will be merged anyway. Let us consider the composite service described by the statechart depicted in figure 3. When S1 finishes, instead of passing the partial trace to each initial state of the AND-state, it sends it straight to S2.

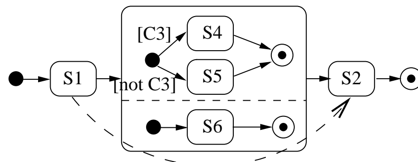

Fig. 3. S1 sends its partial trace to S2

4.3 Back to the example

To illustrate how traces are collected, let us consider the composite service "Travel Solutions" described in Figure 1. We show in Figure 4 one execution of this service. We only show the sequence of statuses through which the service instance goes during its execution, thereby omitting details about their actual parameters and providers. Without loss of generality, we assume that the execution TS_e starts at instant 1.

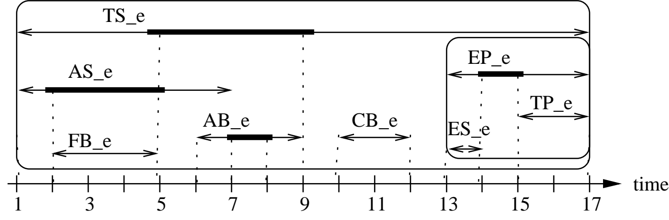

Fig. 4. Execution details of one instance of the service "Travel Solutions".

The following notations are used in Figure 4. TS_e denotes a particular execution of the composite service "Travel Solution" (TS), AB_e a particular execution of service "Accommodation Booking" (AB), and so on. In other words, for each service TS, AS, FB, AB, CB, BB, ES and TP, the suffix _e is added to its name in order to denote one of its executions. A double-arrow $\longleftrightarrow$ is used to denote the interval during which the associated service was running, and finally completed. A thicker part of the arrow means that during the underlying period, the service is frozen.

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

Time	From coordinator of	To coordinator of	Content
5	FB	AB	{FB_e}
7	AS	BB and CB	{AS_e}
9	AB	BB and CB	{FB_e, AB_e}
12	CB	EP	{FB_e, AB_e} ∪ {AS_e} ∪ {CB_e}
17	EP	wrapper	{FB_e, AB_e, AS_e} ∪ {EP_e}

Table 1. Messages between coordinators during the execution of TS_e.

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

The 2nd and 3rd lines of the table 1 can be read as follows. At time 7 (resp. 9) AS's coordinator (resp. AB) sends the partial trace to both BB's coordinator and CB's coordinator. Because the boolean expression [Attractions near from accommodation] is evaluated to false, BB is not required to be executed, so the coordinator of the state that it labels discards the partial traces that were sent to it by the coordinators of AS and AB. Because the state labeled by the EP is the last one to be entered, at time 17 its coordinator sends the whole trace to the wrapper of the service TS.

5 Related work

The issue of service composition, and the related field of inter-organisational workflows, have been the subject of intensive attention in the last years. Here, we focus on those efforts dealing with the aspects addressed in this paper, that is, coordination between services, and execution tracing.

CMI [12] and eFlow [3] are two pioneering systems for specifying, enacting, and monitoring composite services. In both of these systems, 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. Clearly, this centralised approach leads to potential bottlenecks, that are avoided in Self-Serv through the use of a peer-to-peer coordination paradigm.

Closer to the decentralised spirit of Self-Serv is CPM [4]. This platform supports the execution of inter-organisational business processes through peer-to-peer collaboration between a set of workflow engines. The major difference between CPM's and Self-Serv's execution models, is that in CPM, the number of messages exchanged between the workflow engines is not optimised. Instead, each time that a process terminates a given task, it must send a notification to all its other peer processes. Moreover, CPM requires that all the players participating in an inter-organisational process, deploy the same workflow engine, since they all need to interpret a single global process specification. Meanwhile, in Self-Serv the inter-service coordination is entirely handled by the state coordinators.

Self-Serv's execution model has also some similarities with that of Mentor [10], although this latter proposal is targeted to intra-organisational workflow management. Specifically, the problem addressed in [10] is that of distributing the execution of workflows expressed as state and activity charts. Mentor's approach differs from Self-Serv's, in that it is only applicable when the assignment of activities to their executing entities is known at the definition of the workflow, which is a 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.

None of the above proposals explicitely addresses the issue of tracing the executions of a composite service. Actually, we are not aware of any concrete proposal in the area of composite service execution tracing, except for [7] and [9] which address a similar issue: that of tracing the executions of a workflow. [7] assumes that the workflows are executed in a distributed environment, and that each node within this environment (in our context: each provider), maintains the history of its task executions (in our context: its service executions). Within this context, the authors present several strategies for evaluating queries such as "retrieve the history of a given process instance". Unlike our proposal, the set of entities participating in the execution of a workflow is assumed to be fixed.

Our approach also differs from the above in that in Self-Serv, universal references are used to link the abstractions of the trace maintained by the composite service wrapper, and the actual details of these traces which are maintained by the providers participating in the composition. Meanwhile, in [7], there are no equivalent concepts to those of "composite service wrapper" and "universal

reference". Consequently, the traces are entirely distributed among the entities participating in a workflow, and they are only linked through logical references (i.e. foreign keys). This imposes a considerable overhead during query evaluation, since the resolution of these logical references requires the distributed computation of expensive joins.

In [9] the context is that of centralised workflows expressed as statecharts. The authors focus on demonstrating that the process of tracing a workflow's execution can itself be seen as a workflow. Consequently, by merging a workflow **W**, with the workflow dedicated to maintaining the history **W**'s executions, one obtains a "self-traceable workflow". Contrarily to our proposal however, [9] does not discuss how these results can be extended to a distributed and interorganisational workflow, neither does it address the issue of distributedly storing the execution traces.

6 Conclusion and future work

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

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

The above collaboration model has been extended so that the state coordinators are able to incrementally collect the execution trace of each composite service instance. These traces are then stored as distributed objects linked through universal references, in such a way that each participant in a composite service is responsible for storing the trace of its own activities, while the provider of the composite service stores an abstracted view of the overall execution. We plan to extend this first attempt to model and query traces in order to address issues such as querying ongoing execution of e-services (i.e., querying the traces of service instances while they are still running).

We are currently developing an implementation of Self-Serv in which the service wrappers and the state coordinators generated by the service description module are packaged as Enterprise JavaBeans (EJB) [14], which interact through a communication layer based on SOAP [13]. Execution traces will be stored and queried as XML documents. References between the execution trace of a

composite service and the traces of its triggered constituents can be modeled through URLs.

Our next step will be to examine how modifications in the constitution of a composite service can be smoothly handled in Self-Serv. We also plan to investigate how to augment state coordinators with data integration mechanisms so as to handle explicit data-flow dependencies.

References

1. B. Benatallah, M. Dumas, M.-C. Fauvet, and H.-Y. Paik. Self-coordinated and self-traced composite services with dynamic provider selection. Technical report, The University of New South Wales, School of Computer Science & Engineering, 2001. Available at http://www.cse.unsw.edu.au/ mcfauvet/selfserv.ps.gz.
2. B. Benatallah, B. Medjahed, A. Bouguettaya, A. Elmagarmid, and J. Beard. Composing and maintaining web-based virtual enterprises. In *Workshop on Technologies for E-Services*, Cairo, Egypt, September 2000.
3. 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.
4. Q. Chen and M. Hsu. Inter-enterprise collaborative business process management. In *Proc. of the Int. Conference on Data Engineering (ICDE)*, Heidelberg, Germany, April 2001. IEEE Press.
5. D. Harel and A. Naamad. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, October 1996.
6. Internet RFC-1305. Network Time Protocol Specification Version 3. http://www.landfield.com/rfcs/rfc1305.html.
7. P. Koksal, S.N. Arpinar, and A. Dogac. Workflow history management. *SIGMOD Record*, 27(1):67–75, January 1998.
8. F. Leymann and D. Roller. *Production Workflow: Concepts and Techniques*. Prentice Hall, Upper Saddle River, NJ, USA, 2000.
9. P. Muth, J. Weissenfels, M. Gillmann, and G. Weikum. Workflow history management in virtual enterprises using a lightweight workflow management system. In *Proc. of the Workshop on Research Issues in Data Engineering (RIDE)*. IEEE Press, March 1999.
10. 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.
11. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language reference manual*. Addison-Wesley, 1999.
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. SQLData. Simple Object Access Protocol. http://www.soapclient.com.
14. Sun Microsystems Inc. Enterprise JavaBeans Specifications. http://www.javasoft.com/products/ejb/.
15. Sun Microsystems Inc. Java RMI. http://java.sun.com/products/jdk/rmi.