HIERARCHICAL WEB SERVICES COMPOSITIONS: VISIBILITY, COMPENSATION AND MONITORING

DEBMALYA BISWAS

Hierarchical Web Services Compositions: Visibility, Compensation and Monitoring

by

©Debmalya Biswas

A thesis submitted to the
School of Graduate Studies
in partial fulfillment of the
requirements for the degree of
Master of Science

Department of Computer Science
Memorial University of Newfoundland

Aug 2005

St. John's Newfoundland

Abstract

Industry and researchers acknowledge Web services as being at the heart of next generation distributed systems. The most promising feature of the Web services platform is its ability to form new services by combining the capabilities of already existing services, i.e., its composability. The existing services may themselves be composed of other services, leading to a hierarchical composition. In this work, we focus on the visibility, compensation and monitoring aspects for hierarchical compositions.

Most works on mechanisms to provide extended functionalities like transactions, monitoring, security, etc. for Web services compositions consider single-level compositions with an implicit assumption that they can be straightforwardly extended to hierarchical compositions. As such, they fail to appreciate an important and unique aspect of hierarchical compositions, the visibility aspect. For example, a service provider may not be aware of any providers in the hierarchy other than its parent and children. On the other hand, a service provider may be aware of all other providers in the hierarchy. Towards this end, we introduce the notion of Spheres of Visibility (SoV). Basically, SoV provides an abstraction to capture the upward/downward visibility aspects in a hierarchical composition. We expect other compositional aspects like transactions, monitoring, security, etc. to build on this abstraction.

We discuss in detail what "compensation" means in a Web services context, analyze proposed models and show how compensation can be implemented efficiently in hierarchical compositions. We identify two aspects, Cost of Compensation (CoC) and End User Involvement, missing from most of the proposed models. Current transaction models also

constrain the act of compensation to the original service provider. Given the ability to bind with a service provider at run-time (dynamic binding), we believe that this is an unnecessary restriction and outline a mechanism for provider independent compensation. We also introduce the notion of side-effects in a hierarchical composition to determine if provider independent compensation is possible for a given scenario. Finally, we outline a compensation mechanism for hierarchical compositions incorporating the above aspects while conforming to the visibility restrictions modeled as SoV.

With respect to monitoring, we focus on capturing the state of a hierarchical Web services composition at any given point of time (snapshot). Such information is useful not only for reporting the current status to the end-user but also for answering specific queries related to the execution. Analogous to distributed systems, capturing the state of a hierarchical Web services composition is difficult because of the absence of a global observer, inherent non-determinism, unexpected communication delays, etc. In addition, for Web services compositions, the "components" of the distributed system may not be known in advance (due to dynamic binding). We discuss in detail how some of the snapshot algorithms proposed in literature can be adapted for hierarchical Web services compositions. Snapshots usually reflect a state of the system which "might have occurred". We outline algorithms to acquire a state that "actually occurred", from such snapshots. Finally, we discuss how the acquired snapshots help us in answering execution status related queries.

Acknowledgements

First and foremost, I would like to thank my supervisor, Dr. K. Vidyasankar. This work would not have been possible without his guidance, support and ideas. I am grateful to him for introducing me to the world of research and for always being there whenever I had difficulties. I would like to thank him for the countless hours he spent in discussions, coming up with suggestions to improve the work and proofreading drafts of this work.

I would like to thank my family (parents, sister and nephew) for their endless love, understanding and faith which allowed me to stay focused during the course of my work.

I am thankful to my friend, Mahantesh, whose smiles and companionship have kept me relatively sane for the last two years. I am grateful to Ms. Usha Vidyasankar for easing my transition to a new country and for providing the homely atmosphere away from home. I am thankful to Dr. Ananthanarayana V. S. for the lively discussions on both technical and non-technical aspects. I would also like to thank the reviewers, Dr. Munindar P Singh and Dr. Jeffrey Parsons, for their comments which helped to improve the work in this thesis considerably.

Special thanks to everyone in the Computer Science Department at Memorial University for providing the perfect study environment, especially, Dr. Wolfgang Banzhaf, Dr. Ashoke Deb, Ms. Radha Gupta and Ms. Elaine Boone for their encouragement and support. Last but not the least, I would like thank the various sources within and outside the University for partially sponsoring my studies at the University and participation at ICWS 2004 and ICDCIT 2004.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Web Services

Web services are recognized as the next generation of distributed computing by both academicians and industrial bodies. *The World Wide Web Consortium (W3C) defines Web Services as "a software application identified by a URI, whose interfaces and bindings are capable of being defined, described, and discovered as XML artifacts. A Web service supports direct interactions with other software agents using XML-based messages exchanged via Internet-based protocols".* Web services, also known in a broader context as Service Oriented Architectures (SOA), are based on the assumption that the functionality provided by an enterprise/provider is exposed as a service. The middleware aspects currently addressed by Web services standards are:

- Service Description: To be machine understandable, the description should describe what a Web service can do, where it resides and how to invoke it. Basically, there are three ways of describing a Web service: (1) Input/Output signatures: A Web service is described in terms of the Input/Output signatures of its operations. This approach is analogous to W3C's Web Service Description Language (WSDL) [WSDL]. A WSDL description is characterized by an abstract part, which is similar to the Interface Definition Languages (IDL) of conventional middleware and a concrete part responsible for defining the protocol binding among other things. (2) Pre and post conditions: Here also, Web services are described in terms of their operations. However, each operation description is annotated with pre and post

conditions which allow capturing the semantic effects of the operations. The specification synonymous with this approach is Web Ontology Language for Services (OWL-S) [OWL-S] Service Profile. (3) Signatures with behavior: None of the above approaches consider the relationships which might exist between operations belonging to a Web service. For example, a payment receipt cannot be generated before an order has been received. As such, it is essential to consider the behavioral aspects while describing a Web service. Such descriptions are usually defined with the help of a state based formalism. In this work, we follow the behavioral approach and assume that a Web service WS is defined as (based on the notion of I/O automaton [LT89]): *WS = (Q, Σ_{INP}, Σ_{OUT}, Σ_{INT}, i, F, δ), where Q is the set of states, i is the initial state and F is the set of final states.* Each state s ϵ Q is defined as a set of first order predicates which hold when the system is in state s. The actions are categorized as internal and external actions. External actions are further classified as input and output actions. Thus, Σ_{INP}, Σ_{OUT}, Σ_{INT} represent the sets consisting of input, output and internal actions respectively. Finally, $\delta \epsilon Q \times \Sigma \times Q$ defines the set of labeled transitions where $\Sigma = \Sigma_{INP} \cup \Sigma_{OUT} \cup \Sigma_{INT}$. Basically, each action moves the system from a state s to a state s', also referred to as the effect of the action.

- Service Discovery: To allow services to be used by others, the service descriptions are published in a service directory. Web services support service discovery both at design time and at run-time (using dynamic binding techniques). Universal Description, Discovery, and Integration (UDDI) [UDDI] is the current de-facto standard for service discovery. UDDI provides a set of data structures and APIs for publishing and querying services in a service directory.

- Service Interactions: Once the services have been described and discovered, the next step consists of actually invoking the services and passing XML messages/data amongst them. Current Web services implementations use the Simple Object Access Protocol (SOAP) [SOAP] for service interactions. Basically, SOAP allows Remote Procedure Call (RPC) style interactions by providing standardized mechanisms to turn a service invocation to an XML message, exchange the message, and turn the XML message back to an invocation. SOAP is often referred to as XML over HTTP. However, this is not entirely correct as the SOAP specification does not specify any particular transport protocol. While the current specification supports HTTP and SMTP, bindings to other transport protocols will probably be defined in future versions of the specification.

Let us summarize the above discussion with the help of a simple Web services usage scenario (Fig. 1.1). The steps below refer to the step numbers in Fig. 1.1.

1. The service provider prepares a WSDL document describing the services it provides. The provider publishes (registers) the WSDL document with an UDDI registry.

2. The client (whenever it needs to get some work done) queries the UDDI registry. The registry returns not only descriptive information about the service provider but also information regarding where (endpoint URI) and how (protocol) the service can be invoked.

3. The client uses the above information to interact with the provider and get the work done.

1.2 Web Services Compositions

Web Services Composition (WS-Composition) relates to the assembly of autonomous components so as to deliver a new service out of the components' primitive services. Fig. 1.2

shows a typical Web services composition scenario. From the client's perspective, a composite Web service implemented by invoking other primitive Web services is the same as a basic Web service and can be described, discovered and invoked the same way. Thus, a composite Web service can act as a basic Web service for further compositions leading to a hierarchical composition. In this work, we focus on hierarchical compositions of Web services.

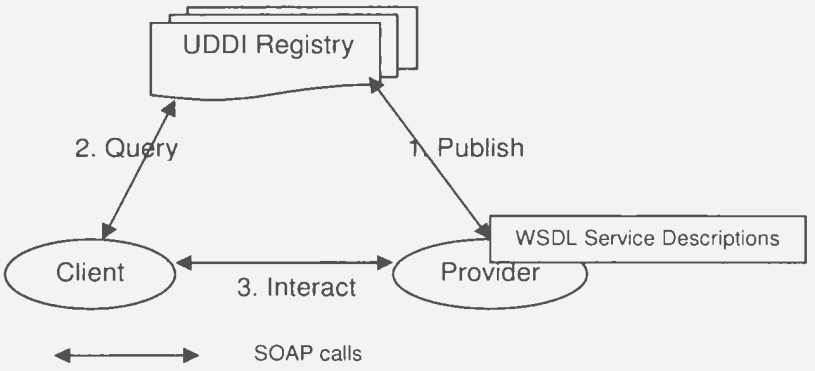

Figure 1.1 Web services usage scenario

Basically, there are two approaches to forming a composite Web service from a set of primitive Web services:

- Dynamic approach: Given a complex user request, the system comes up with a plan to fulfill the request depending on the capabilities of available Web services at run time [NM02], [WPSHN03], [RKM04]. [NM02] outlines a mechanism for the automated composition of a set of services described using DAML-OIL (an earlier version of OWL-S). The operational semantics (behavior) of the set of services are defined as Petri-nets. Given this [NM02] states the problem of automated composition as follows: "Let A be a set of

atomic Web services and let N = (P;T;F;M) be the net that depicts the behavior of all the services in A. Further, let φ represent the user's goal, and let M' be the marking that depicts this goal in N. Then, a1;a2,…;an is a sequential composition of atomic services that achieves user goal φ iff a1;a2,…;an is an occurrence sequence in the reachability analysis of M' in N". [WPSHN03] and [RKM04] consider automated composition of Web services using Hierarchical Task Planning (HTN) techniques and Linear Logic (LL) theorem proving respectively.

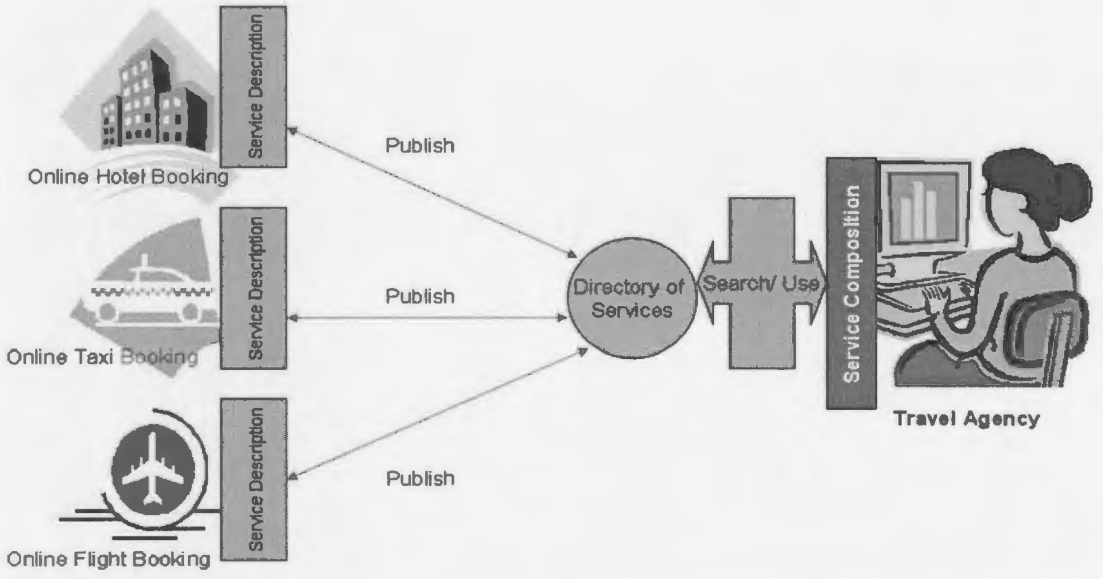

Figure 1.2 Typical Web services composition scenario

- Static approach: Given a set of Web services, composite Web services are defined manually combining their capabilities. Business Process Execution Language for Web Services (BPEL) [BPEL] provides constructs for a composition designer to manually specify the process model (composition schema) of a composite service in terms of the interactions between already existing services (defined using WSDL).

In the static approach, the composite Web services are not defined in response to any specific user request. Rather, they are defined to provide recurring, general services which are usually requested by users. This is in contrast to the dynamic approach which is more suitable for ad-hoc complex user requests. As evident, both the approaches have their own pros, cons and research challenges. In this work, we consider a mix of the above approaches where the composite services are defined statically but the binding with providers is performed at run-time depending on the user request [CIJKS00]. In [CIJKS00], a composite Web service is defined as a graph which may include service, decision and event nodes. [CIJKS00] allows run-time binding by attaching a search recipe to the definition of the nodes. *Thus, we assume the existence of a composition schema and the capability to select and bind with service providers at run-time.* Most of the work in this thesis deals with the execution of a particular invocation of an existing (pre-defined) composition schema. The services (providers), which do not depend on any other services (providers) for their execution, are referred to as primitive services (providers). We also assume a WS-Coordination [WS-C] like infrastructure where there is a coordinator associated with each service provider. The coordinator is responsible for all non-functional aspects related to the execution of the provider such as monitoring and transactions. Whenever a provider A invokes an action of another provider B, B's coordinator registers itself as a sub-ordinate coordinator with A's coordinator.

1.3 Objectives of the Thesis

Web services have been widely accepted as a platform to provide interoperability between heterogeneous systems. To make them suitable for mission critical applications, recent work

has focused on mechanisms to provide extended functionalities like transactions, monitoring, security, etc. for Web services compositions. In this work, we focus on the following aspects of hierarchical Web services compositions:

1. Visibility: The visibility aspect relates to the knowledge, a provider has, about the rest of the composition. Providers, in a hierarchical composition, vary in their visibility over the rest of the composition. On one hand, the privacy and autonomy requirements of the service providers restrict their visibility. On the other hand, functionalities such as transactions, monitoring, security, end-user involvement, etc. call for ancestors having visibility over some of their descendants and vice versa. Thus, we need a framework which allows efficient specification and management of upward/downward visibility in a hierarchical composition. Towards this end, we introduce the notion of Spheres of Visibility (SoV) as an abstraction to formally define and capture the upward/downward visibility aspects of the providers in a hierarchical composition. Given that visibility is an inherent trait of hierarchical compositions, we expect other compositional aspects like transactions, monitoring and security to build on this abstraction. In particular, we show how SoV can be used in conjunction with the compensation mechanism. This part of the work appears in the proceedings of the 3rd IEEE European Conference on Web Services (ECOWS 2005), Växjö, Sweden.

2. Compensation: Transactions have long been considered as the preferred mechanism for handling failures in distributed systems. Basically, transactions allow grouping individual operations into logical units of work having the following ACID (Atomicity, Consistency, Integrity, Durability) properties. Due to their heterogeneous, autonomous and long-lived

nature, traditional ACID based models are not sufficient for providing transactional guarantee to Web services compositions. To overcome this limitation, many extended transaction models have been proposed based on the concept of compensation (semantically undoing the effects of an execution). However, current compensation based models fail to acknowledge the following aspects:

(a) Cost of Compensation (CoC): They assume the existence of a pre-defined compensating action (for each action) which is invoked in case the effects of the original execution need to be canceled. They do not acknowledge the fact that there might be multiple options capable of compensating the effects of the original execution with different associated costs (CoC). This is especially true for a hierarchical composition where compensation may be possible at different levels.

(b) End-user involvement: We argue that the whole compensation process including selection of the optimal option is complex enough to warrant end-user involvement.

(c) Provider independent compensation: Most models consider compensation as a provider dependent activity, i.e., compensation is the responsibility of the original service provider. We believe that this is an unnecessary restriction given the presence of a state based formalism (which allows us to dynamically determine the effects to be compensated) and the ability to select a service provider (the compensation provider) at run-time.

Finally, we show how compensation can be realized in a hierarchical composition incorporating the above aspects (compensation options at different levels, provider independent compensation and end-user involvement), while conforming to the visibility restrictions modeled as SoV. Part of the work on compensation appears in the proceedings of

the First International Workshop on Semantic Web Services and Web Process Composition (SWSWPC 2004), CA, USA, LNCS Volume 3387, pages 69 - 80.

3. Monitoring: Monitoring is an inherent requirement of any distributed system. The need for a monitoring mechanism is even more critical for Web services compositions because of their complexity and long running nature. Their complexity makes them prone to failures leading to the need for a monitoring mechanism to detect and report failures. The fact that a composition may be running for a long time (even days) calls for the ability to report its intermediate status. Monitoring Web services compositions, similar to distributed systems, is difficult because of the following reasons:

- No global observer: In fact, due to their privacy and autonomy requirements, even the invoking service provider may not have visibility over the internal processing of the invoked service providers.
- Non-determinism: Web services compositions allow parallel composition of processes. Also, Web services usually depend on external factors for their execution. Therefore, it may not be possible to predict their behavior before the actual execution. For example, whether a flight booking will succeed or not depends on the number of available seats (at the time of booking) and cannot be predicted in advance.
- Unpredictable communication delays: Communication delays make it impossible to record the states of all the involved providers instantaneously. For example, let us assume that provider A initiates an attempt to record the state of the composition. Then, by the time the request (to record its state) reaches provider B and B records its state, provider A's state might have changed.

- Dynamic configuration: Dynamic binding allows the providers to be added incrementally as the execution progresses. Although we assume static composition, aspects such as provider independent compensation may require adding new services at run-time. Thus, the "components" of the distributed system may not be known in advance.

OWL-S states the need for execution monitoring as follows "the ability to find out where in the process the request is and whether any unanticipated glitches have appeared". In this work, we only consider the first part, i.e., providing information about the current state of the execution. The problem of "capturing the state of a system" has been studied extensively in the area of distributed systems and the solutions are usually categorized as snapshot algorithms. We discuss in detail how some of the snapshot algorithms proposed in literature can be extended in a Web services context. Snapshots usually reflect a state of the system which "might have happened". Snapshot algorithms capable of capturing a state which "actually happened" depend on real-time timestamps or fully synchronized clocks. Towards this end, we show how we can acquire a state of the composition which actually occurred from a state which might have occurred. We conclude by discussing the different types of execution related queries and how we can answer them using the captured snapshots. This part of the work appears in the proceedings of the 6th VLDB Workshop on Technologies for E-Services (TES-05), Trondheim, Norway.

1.4 Structure of the Thesis

The rest of the thesis is structured as follows. Chapter 2 presents a brief overview of the challenges and related work with respect to each of the three aspects (visibility,

compensation and monitoring). In Chapter 3, we focus on the visibility aspect and introduce the notion of SoV. We give a framework for the specification and implementation of SoV in a hierarchical Web services composition. Chapter 4 deals with the compensation aspect. We analyze the compensation based transaction models (proposed in literature) with respect to the capabilities and limitations of a hierarchical Web services composition and propose a mechanism for their efficient implementation. The monitoring aspect is considered in Chapter 5. We provide extensions of some of the well-known snapshot algorithms to capture the state of a hierarchical Web services composition (which might have occurred) and show how we can acquire a state which actually occurred from such snapshots. Chapter 6 concludes the work and provides directions for future work.

Chapter 2

Related Work

We divide this chapter into three sections corresponding to the objectives of the thesis.

2.1 Visibility

SoV extends the concept of Spheres of Control (SoC) [D78] initially proposed by Davies. A SoC encapsulates entities sharing a similar set of properties or having a dependency relation. Some of the dependency relations considered in [D78] are atomicity, commitment, resource allocation, recovery, auditing and consistency. SoV logically groups the providers (and their details) visible to a provider in a hierarchical composition. Mechanisms to create the SoC for atomicity, commitment, recovery and so on, as outlined in [D78], assume tightly coupled and non-autonomous systems. Given that the visibility aspect is an inherent trait of loosely coupled and autonomous systems, SoV can be considered as a complementary sphere to extend the work in [D78] to loosely coupled and autonomous systems.

Later works have extended the initial concept to Spheres of Atomicity [AH00] and Commitment [SY01]. [AH00] utilizes the properties of the processes (pivot, compensatable and retriable) in a Sphere of Atomicity to determine if the sphere, as a whole, guarantees atomicity. [SY01] applies the concept of SoC to Multi-Agent Systems (MAS) to structure agents based on their commitment guarantees. However, the above works are not directly related to the work presented in this thesis and we mention them for the sake of completeness.

No other work that we are aware of has attempted to formalize the visibility aspect in a hierarchical composition. Some of the works which have touched upon this aspect are: [M98] identifies real-life scenarios where there might be a need to deviate from the inheritance of access rights upwards through the hierarchy in a role-based access control. [CD96] discusses the visibility aspect with respect to the visibility of the results of a sub-transaction in a nested transactional system. [CD96] advocates the provision to be able to expose the results to a particular ancestor to improve performance.

2.2 Compensation

2.2.1 Background - Transactions

A transaction can be considered as a group of operations encapsulated by the operations Begin and Commit/Abort having the following properties (ACID):

- Atomicity: Either all the operations are executed or none of them are executed. In case of failure (abort), the effects of any operation belonging to the transaction are canceled (roll-back).
- Consistency: Each transaction moves the system from one consistent state to another.
- Isolation: To improve performance, often several transactions are executed concurrently. Isolation necessitates that the effects of such concurrent execution are equivalent to that of a serial execution. This is achieved by ensuring that the intermediate results of a transaction are not externalized until it completes successfully (commits).
- Durability: Once a transaction commits, its effects are durable, i.e., they should not be destroyed by any system or software crash.

For example, let us consider the classic bank transaction t_b which involves transferring money from an account A to another account B. The transaction consists of two operations - the first operation withdraws money from account A and the second deposits it into account B. Needless to say, any partial execution of the transaction would result in an inconsistent state. The atomicity property ensures that either both withdraw and deposit operations succeed or both fail. The isolation property ensures that the changes to both accounts A and B are not visible to other transactions until t_b commits. The atomicity and isolation property together ensure the consistency of the system (accounts A and B).

The software responsible for implementing transactions is often referred to as a Transaction Manager (TM). We can divide the functionality of a TM into the following parts:

- Concurrency Control Manager (CCM): The CCM ensures the isolation property. Concurrency control mechanisms allow sharing the resources between concurrent transactions in a controlled manner.
- Recovery Manager (RM): The RM is responsible for providing the atomicity and durability properties in the event of a failure or software/hardware crash.
- Log Manager (LM): The LM is responsible for writing the execution details to stable storage. While the log plays an important role in recovery, it is also used to analyze and improve the system performance and efficiency.

While transactions have been accepted as the standard means to provide fault-tolerance and reliability, new challenges arise when we try to apply them in a distributed setting. Distributed transactions consist of operations which are executed at different sites connected by a communication network. Distributed transactions originate at a site (also known as the

home/root site) gradually involving other sites where operations belonging to the transaction are forwarded for execution. The main differences between transaction processing in a centralized and distributed system are as follows:

1. Decision making: The decision to commit/abort a transaction is not restricted to a single TM. Rather, a collective decision needs to be taken based on the decisions of the TMs of all the involved sites.

2. Multiple points of failure: With centralized systems, the system is either working or not working. However, in a distributed system we can have partial failures in the sense that some of the involved sites fail while others are still working.

As such, we need a protocol which ensures that the same decision (commit/abort) is consistently carried out at all the involved sites irrespective of partial failures. Two phase commit (2PC) protocol (commercially standardized as the XA interface specification) is probably the most widely accepted solution for the above problems. The TM at the home site acts as the coordinator while the TMs at all other involved sites assume the role of participants. As suggested by the name, 2PC protocol consists of two phases. In the first phase, the coordinator TM sends a PREPARE message to all the participant TMs. Each participant TM votes Yes/No depending on whether it wants to commit/abort. If the coordinator TM receives "Yes" from all of its participant TMs, then it begins the second phase of the protocol by sending COMMIT messages to all of them. However, if it receives "No" from at least one of the participant TMs, then it initiates the second phase by sending ABORT messages to all the participant TMs. Finally, the coordinator TM waits for acknowledgement from the participant TMs to complete the second phase.

While the above protocol works well for tightly coupled distributed applications, its applicability to long running, loosely coupled and cross-organizational applications is limited. To ensure ACID properties (in a centralized scenario), locks need to be held until the transactions commit. With distributed transactions, the locks would have to be held until all the involved (coordinator and participant) sites are ready to commit. The above scenario can be easily extended to a hierarchy of TMs as shown in Fig. 2.1. Such a hierarchy arises when a site, invoked to process part of the execution, invokes another site to process part of its own execution in a recursive fashion. As such, all the TMs in the hierarchy are responsible for executing operations associated with the global transaction initiated by the topmost (also known as the root coordinator) TM. All the non-root TMs except the leaves (also known as subordinate coordinators) are responsible for coordinating operations executed by the corresponding sub-tree of participating TMs. Given such a setting, locks at each site would have to be held until all the TMs in the hierarchy are ready to commit. Obviously, this is not a desirable situation performance wise especially for long running transactions.

An elegant solution to the above limitation is the concept of Nested Transactions [M81]. Nested Transactions allow the TMs at the involved sites to release their locks as soon as the transaction completes locally by externalizing intermediate results in a controlled manner. Basically, the global transaction (submitted to the root TM) is divided into a number of subtransactions that may be executed concurrently. While ACID properties are guaranteed for the global transaction, subtransactions are not fully isolated as their results are exposed to their parents. Even the durability of subtransactions is not guaranteed as its effects might need to be canceled (after it has been committed) if its parent aborts.

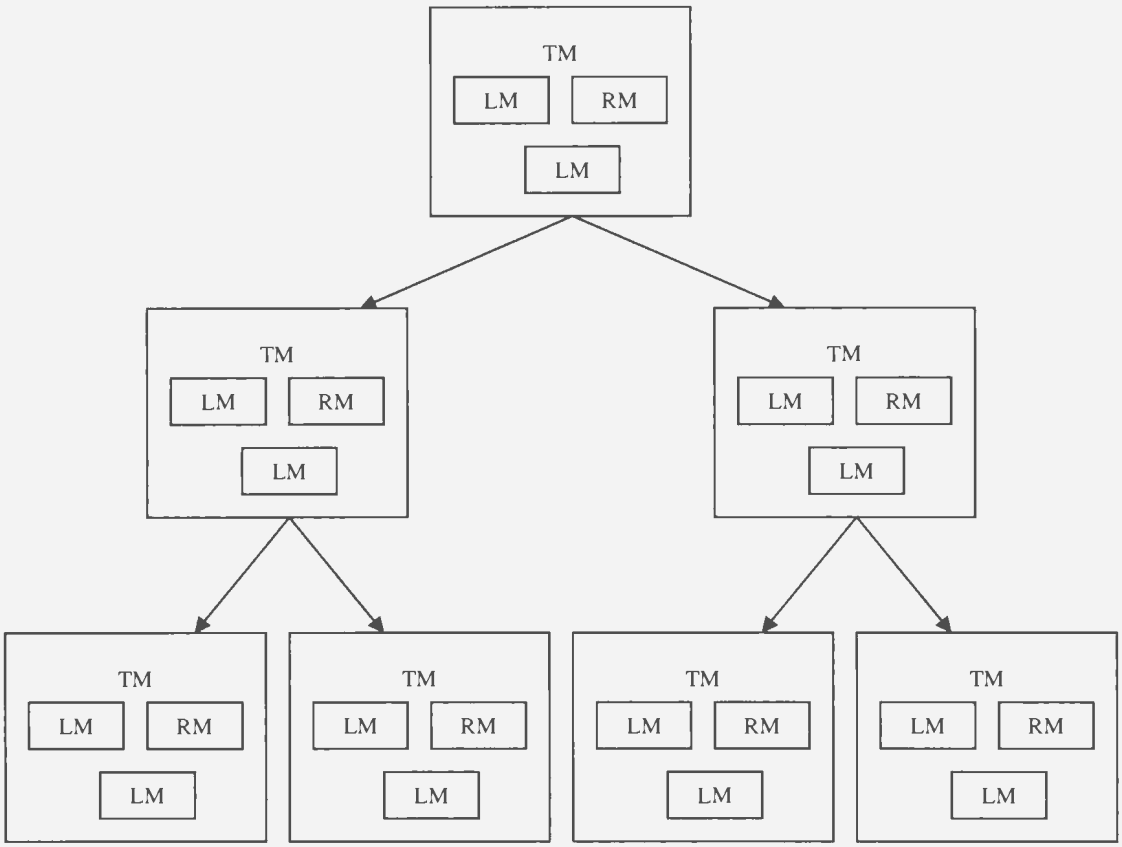

Figure 2.1 Hierarchical transaction processing infrastructure

While the concept of Nested Transactions resolves the performance issue to some extent, it still requires certain guarantees from the involved TMs. However, such guarantees may not always be feasible considering the autonomy and heterogeneity requirements of loosely coupled distributed systems. Sagas [GMS87] or Open Nested Transactions [WDSS93] alleviate this problem by allowing intermediate results produced by the subtransactions to be exposed without any restrictions. Sagas rely on the concept of compensating transactions to ensure atomicity in case of a failure. Basically, for each transaction t, a compensating transaction t_c capable of semantically undoing the effects of the transaction t is specified. In

case of failure, atomicity is guaranteed by executing the compensating transactions in the reverse order of the original execution sequence of their respective transactions. Classic examples of compensating transactions are "Cancel reservation" or "Withdrawal" capable of undoing the effects of a "Reserve ticket" or "Deposit" transaction respectively. Here, it helps to keep in mind that compensation may not always be possible, especially for real-life transactions.

2.2.2 Related Work

Web services transactions are also characterized by their long running and loosely coupled nature. As such, compensation based transaction models are typically well suited for Web services compositions. Below, we take a look at some of the compensation based transaction models proposed for Web services.

[KS03] describes how compensating transactions can be modeled based on the active database concept of triggers basically as Event-Condition-Action (ECA) rules. [TIRL03] presents a forward recovery based transaction model. It applies the concept of co-operative recovery in the context of Web services. Co-operative recovery allows the participating service providers (affected by failure) to coordinate a recovery solution. Towards this end, they introduce the notion of Web Service Composition Actions (WSCA) as an extension to the notion of Coordinated Atomic Action (CA). They also acknowledge the importance of end user interaction in the recovery process. However, in their scenario, the end user is involved only as the last resort. For example, when it is not possible to get flight or hotel reservation then the end user is consulted to suggest another travel date. They do not consider the use of end user interaction for taking intermediate decisions, selecting among possible

compensation options, etc. In [PMB02], Pires et. al. propose a framework (WebTransact) for building reliable Web services compositions. Their framework, based on the concept of forward recovery, allows for the specification of composition properties like atomicity and guaranteed termination. According to [PMB02], "the WebTransact framework defines four types of transaction behaviors of remote services, which are: compensable, virtual-compensable, retriable, or pivot. A remote operation is compensable if, after its execution, its effects can be undone by the execution of another remote operation. The *virtual-compensable* remote operation represents all remote operations whose underlying system supports the standard 2PC protocol. These services are treated like compensable services, but, actually, their effects are not compensated by the execution of another service, instead, they wait in the prepare-to commit state until the composition reaches a state in which it is safe to commit the remote operation. A remote operation is *retriable*, if it is guaranteed that it will succeed after a finite set of repeated executions. A remote operation is *pivot*, if it is neither retriable nor compensable". [VV04] presents a conceptual, multi-level service composition model, which extends the above model [PMB02] to allow specification of atomicity and guaranteed termination properties at different levels of abstraction. The work in this thesis can be considered complimentary to [VV04] as we consider the practical aspects involved in implementing transactions once a multi-level (hierarchical) composition has been defined satisfying certain properties. [B04b] proposes a formal model based on π-calculus to capture the behavior of nested long running transactions in a Web services context. Of particular interest are the different modes of failure propagation: up-propagation, down-propagation, down-specific propagation and non-propagation. Basically, they argue that a transaction's

failure should trigger not only the compensation of the enclosed transactions (up-propagation) but also of the nested transactions (down-propagation). According to [B04b], if a node P fails then the global compensation process is created by composing the local compensation processes of all the nodes in the sub-tree of P. Our work can be considered as an extension to the above model where the visibility aspect is also considered while computing the possible compensation options (globally). An excellent survey of the extended transaction models which have been proposed for Web services compositions is provided in [JG03].

On the standards front, the BPEL specification provides a feature called Long Running Transactions (LRT) which addresses the problem of the order in which the compensating operations need to be invoked. However, the notion of LRT is purely local and does not provide distributed coordination among multiple-participant services to reach a consensus regarding the outcome. The problem of distributed agreement for a business process spanning multiple vendors and platforms is left to coordination protocols like WS-Transaction (WS-T) [WS-T] or the more recent WS Transaction Management (WS-TXM), part of the WS Composite Application Framework (WS-CAF) [WS-CAF]. Both, WS-T and WS-TXM provide support for compensation based long running activities (called Business Activities in the WS-T context and Long Running Actions in WS-TXM terminology). The WS-TXM framework also proposes a Business Process (BP) transaction protocol which uses a set of interposed coordinators to provide transactional guarantee across business domains. Basically, the overall business activity is split into domain specific tasks where each domain might be using a different transaction model.

2.3 Monitoring

2.3.1 Related Work

Research in the area of Web services monitoring [PBBST04], [LAP03] has focused on monitoring as a mechanism for detecting and handling failures. [PBBST04] uses monitoring to detect and signal if the invoked providers are behaving according to the specified protocols. In [LAP03], the monitor is responsible for the entire execution process starting from requesting the planner to come up with a plan for the user request to ensuring that the execution is proceeding as per plan (once execution starts).

As mentioned earlier, we are primarily interested in capturing the state of a hierarchical Web services composition at any given point of time (snapshot). The problem of "capturing the state of a system" has been studied extensively in the area of Distributed Systems. A distributed system is usually modeled as an undirected graph G = (V, E) where V represents the set of nodes and E is the set of communication channels connecting them. Thus, if we can freeze the computation at some instant t, the snapshot would consist of the states of the nodes and the contents of the channels at t. The contents of the channels are usually the messages sent but not yet received by the nodes on that channel. However, freezing the execution may not always be possible (and even if possible should be avoided). Also, snapshot algorithms should not obstruct the actual execution and need to be superimposed over the underlying computation. Snapshot algorithms try to record the node states and channel contents in such a way that they form a complete and consistent state of the system. Consistency is usually defined by the condition "if the receive event of a message has been recorded in the state of a

node then the corresponding send event should also have been recorded in the state of the sender node".

It is easy to observe that we do not need to physically freeze the system to take a snapshot. We can achieve the same effect if the clocks of all the nodes in the system are synchronized. However, it is impossible to perfectly synchronize the clocks in a distributed setting. Towards this end, several clock synchronization protocols have been proposed based on the notion of logical [L78], [M89] and physical clocks [M91]. In [L78], Lamport presents an algorithm to partially order the events across the system based on the assumption that the clocks are monotonically increasing. An interesting property (or limitation) of Lamport's algorithm is: "If event a happened before event b then $T(a) < T(b)$ (where T(a) and T(b) are the timestamps associated with events a and b respectively). However, if $T(x) < T(y)$, it is not possible to determine if event x causally happened-before y or if they are concurrent". [M89] overcomes the above limitation by attaching vector timestamps to the events. Basically, it ensures that the timestamps of two concurrent events are incomparable. Network Time Protocol (NTP) [M91] extends clock synchronization for large networks connected over the internet. NTP provides skews in the range of 1-30 ms, even for wide area networks. NTP is based on physically synchronizing the clocks in a distributed system with an external clock such as a GPS clock or other radio clocks. We show how we can capture a snapshot of the hierarchical Web services composition based on the assumption that the clocks of the providers are synchronized using one of the techniques discussed above.

However, all the above approaches require considerable coordination among the participants which may not always be feasible in Web services environment (given the

autonomy requirements of the providers). Below, we discuss a more loosely coupled approach, the Distributed Snapshots Algorithm (DSA) [CL85]. Their algorithm requires the channels to preserve the FIFO property. In addition to the messages belonging to the underlying computation, the DSA assumes a special type of message called the marker. The markers do not have any effect on the underlying computation. The algorithm can be initiated by one or more processes, each of which records its state, without receiving markers from other processes. The DSA can be divided into two phases: 1) the recording phase and 2) the collection phase. The recording mechanism given by [CL85] is as follows:

Marker-Sending Rule for a Process p. For each channel c, incident on, and directed away from p:

p sends one marker along c after p records its state and before p sends further messages along c.

Marker-Receiving Rule for a Process q. On receiving a marker along a channel c:

if q has not recorded its state then

begin q records its state;

q records the state c as the empty sequence

end

else q records the state of c as the sequence of messages received along c after q's state was recorded and before q received the marker along c.

Once the states have been recorded by all the nodes (the recording phase has terminated), they need to be collected to get a snapshot of the system. The collection phase is context dependant and [CL85] does not give any specific mechanisms to collect the recorded states. For example, all the nodes may send their recorded states to a previously agreed upon node or flood the recorded states through the system so that each node can determine the snapshot of the system.

Later works have tried to optimize the above algorithm by minimizing the time/message complexity or removing the requirement for FIFO channels. An excellent survey of snapshot algorithms (including the ones discussed above) can be found in [KRS95]. We outline how the DSA can be optimized to capture the snapshot of a hierarchical Web services composition.

Chapter 3

Spheres of Visibility

3.1 Introduction

Current hierarchical composition frameworks restrict a service provider's visibility to its parent and children. However (as we discuss later), a provider might require information about its ancestors and descendants. We consider the hierarchical composition as shown in Fig. 3.1, where the end-user is the top-level ancestor (above the root provider).

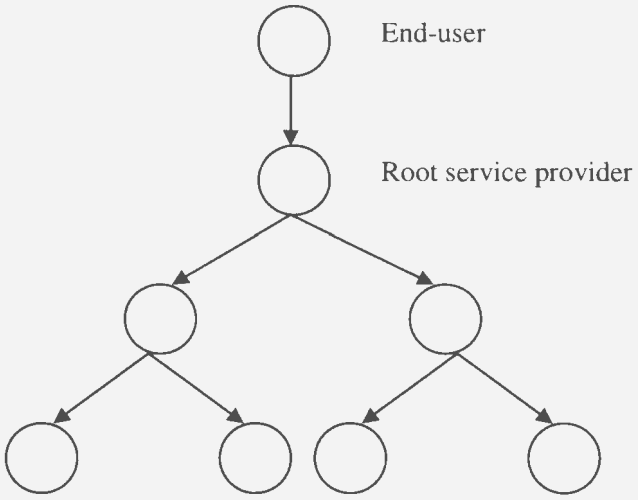

Figure 3.1 Typical hierarchical composition

Below, we outline a few real-life scenarios where a service provider might require information about its ancestors and descendants. Information about an ancestor might be required for getting input and for sending notifications/output:

- Getting input: Due to security reasons, many online shoppers (buyers) prefer to give their credit-card or bank account information directly to the financial institution handling the

payment. As such, the financial institution which might have been invoked somewhere down the hierarchy by a seller (service provider) needs information about its ancestor (the buyer) to get his/her payment details. Please note that the ancestor (to contact) does not always have to be the end-user. It is easy to envision a scenario where there might be a need to contact an intermediate ancestor like a manager (to get his approval) or agent/broker (to whom the end-user has delegated responsibility).

- Sending notifications/output: In a typical (hierarchical) project team, members are usually required to send project related notifications to everyone concerned (not only their immediate supervisors). Similarly, a shipping company delivering goods directly to the buyer (of an e-transaction) is also a classic example where ancestor information might be required.
- Sometimes, non-functional aspects like performance might require exposing results directly to an ancestor. As discussed earlier, [CD96] proposes exposing intermediate results to an ancestor directly to achieve higher concurrency.

Lower level information (about descendants) is usually required for recovery/auditing/analysis. For example, in a supply chain management scenario [LTS98], lower level information is analyzed (by the higher level entities) to increase the order fulfillment rate while higher level information is required to reduce the inventory cost.

Thus, we need a mechanism which allows upward/downward access in a controlled manner. Towards this end, we introduce the notion of Spheres of Visibility (SoV). The SoV_P of a provider P consists of references to the providers visible to P in the hierarchical composition. As such, the SoV_P would vary in range from the entire hierarchy to only its children and parent. For each provider in the SoV, we are interested in the following

attributes: provider (URI, physical address) and services provided (such as execution sequence, input/output values and effects) details. Thus, a provider A might have visibility over provider B's provider details only, services provided details only, both or none. The reason behind considering the two attributes separately is to share only as much information as required (principle of least privilege). Roughly, provider details are sufficient for invoking an action of the provider to get input/feedback or send notifications while service details are required for recovery/analysis. We allow for two types of references to providers in the SoV:

1. Strong reference: A strong reference to a provider implies the inclusion of references to all the intermediate providers (in the hierarchy) in the SoV. Basically, it implies knowledge of the hierarchy up to a certain lower/higher level. For example, the existence of a strong reference from A to G (Fig. 3.2) implies that A has strong references to B and D also. We can define the *strong reference property* as follows: For a pair of parent-child providers (say, P-Q), if A is the highest ancestor to which P is visible, i.e., max-Up_P = A then max-Up_Q has to be A or a lower provider (it cannot be an ancestor of A). Similarly, if Z is the lowest descendent to which Q is visible, i.e., max-Down_Q = Z then max-Down_P cannot be a descendent of Z.

2. Weak reference: A weak reference is a direct reference to a provider without any knowledge about the intermediate providers in the hierarchy.

3.2 Definition

We define $SoV_{P\text{-}prov}$ and $SoV_{P\text{-}service}$, the SoV's of provider P corresponding to the provider and services provided details (hereafter, referred to as the service details) respectively. To

keep the definition simple and since $SoV_{P\text{-}prov}$ and $SoV_{P\text{-}service}$ can be defined similarly, we drop the suffix (prov/service) and simply mention SoV_P below. The SoV_P of a provider P is defined as the combination of its visibility due to strong ($Strong\_references_P$) and weak ($Weak\_references_P$) references. Further, we divide the strong references of P ($Strong\_references_P$) into its upward ($Up\text{-}SoV_P$) and downward ($Down\text{-}SoV_P$) visibility. Formally,

$$SoV_P = Strong\_references_P \cup Weak\_references_P = (Down\text{-}SoV_P \cup Up\text{-}SoV_P) \cup Weak\_references_P$$

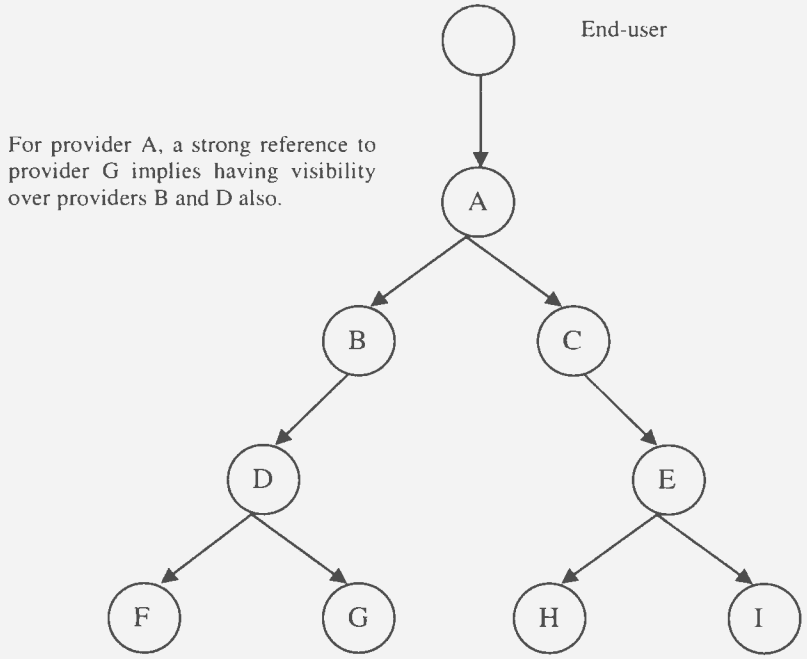

Figure 3.2 Strong reference example

Below, we discuss $Down\text{-}SoV_P$, $Up\text{-}SoV_P$ and $Weak\_references_P$ in detail:

- $Down\text{-}SoV_P = \cup_{R \in Children_P} [Down_R] \cup Children_P$, where $Children_P$ consists of P's children. Thus, $Down\text{-}SoV_P$ defines the downward visibility of P (recursively) in terms of the

downward visibilities of its descendants. Please note that, in the above definition, we use $Down_R$ to refer to the downward visibility of child R (instead of Down-SoV_R). $Down_R$ allows the child R to restrict visibility over some of the providers in its downward SoV (Down-SoV_R) to its parent P (and recursively, to its ancestors). Thus, we define the $Down_R$ of a provider R as follows: $Down_R = (Down\text{-}SoV_R - Down\_restrict_R)$, where $Down\_restrict_R$ is defined such that the strong reference property is not violated. Fig. 3.3 shows an example scenario highlighting the use of Down-SoV_P and $Down_P$. The above definition leads to the following visibility pattern (Fig. 3.4): the downward visibility over a set of providers decreases as we go higher up in the hierarchy.

- Up-SoV_P = $Up_{Parentp\text{-}P} \cup Parent_P$, where $Parent_P$ is P's parent. Thus, Up-SoV_P defines the upward visibility of P (recursively) in terms of the upward visibilities of its parent (and ancestors). Similar to downward visibility, in the above definition, we use $Up_{Parentp\text{-}P}$ to refer to the upward visibility of $Parent_P$ (instead of Up-$SoV_{Parentp\text{-}P}$). Basically, $Up_{Parentp\text{-}P}$ allows $Parent_P$ to restrict visibility over some of the providers in its upward SoV (Up-$SoV_{Parentp}$) to its child P (and recursively, to its descendants). Let $Parent_P$ = M, then we can define the $Up_{M\text{-}P}$ of provider M with respect to its child P as follows: $Up_{M\text{-}P} = (Up\text{-}SoV_M - Up\_restrict_{M\text{-}P})$, where $Up\_restrict_{M\text{-}P}$ is defined such that the strong reference property is not violated. The need for the suffix P in $Up_{M\text{-}P}$ and $Up\_restrict_{M\text{-}P}$ arises from the fact that provider M might like to specify different upward visibilities for each of its children (P's siblings). Fig. 3.5 shows an example scenario highlighting the upward visibility aspect (as discussed above). The above definition leads to the following visibility pattern (Fig. 3.6): the upward visibility over a set of providers decreases as we go lower down in the hierarchy.

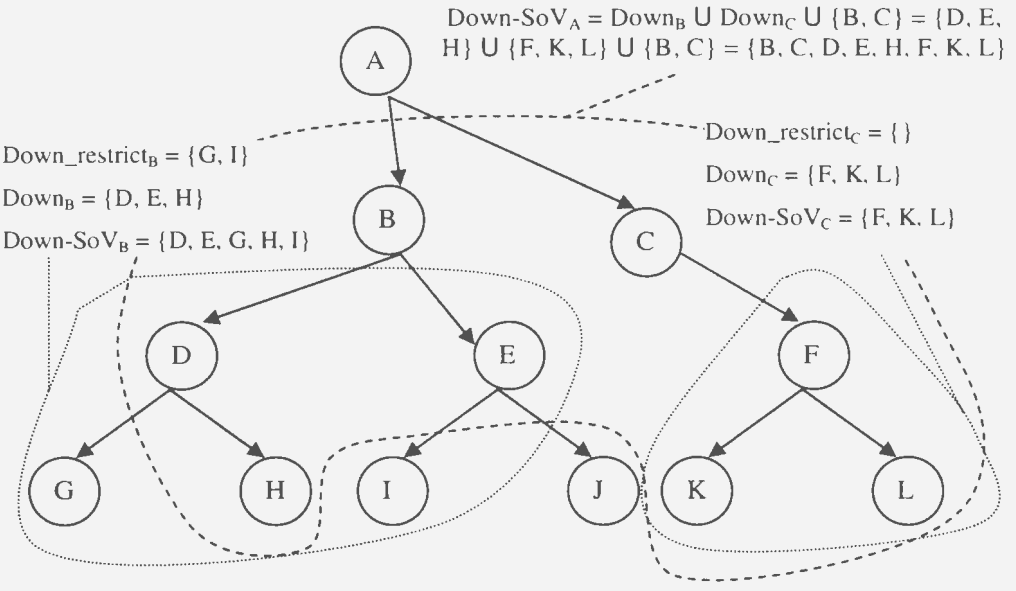

Figure 3.3 Example composition scenario highlighting the downward visibility of A

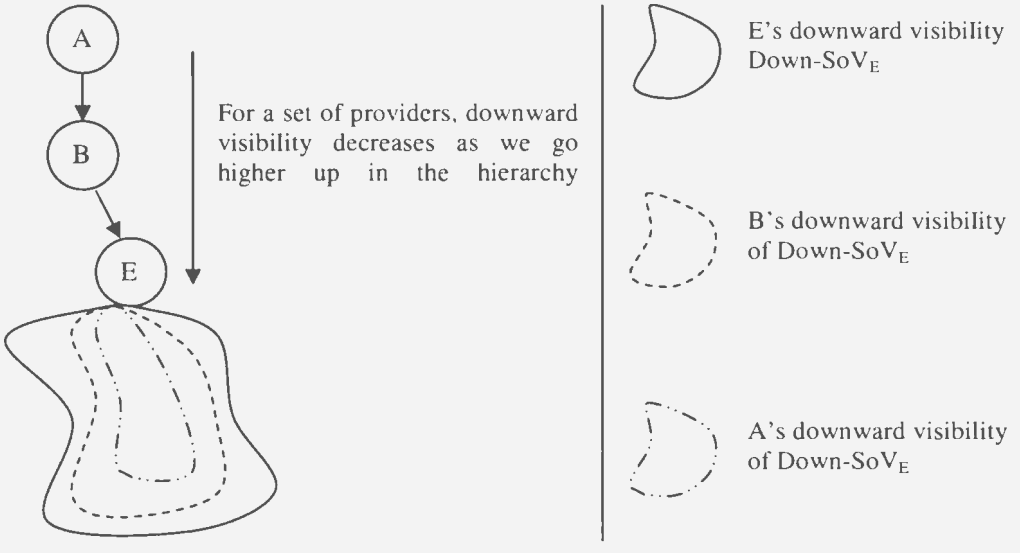

Figure 3.4 Downward visibility pattern

- Weak_references$_P$ is the set of weak references to the providers visible to P. While information about the hierarchical structure among the providers referenced by strong

references is evident from their recursive definition, no such knowledge is available for the providers referenced by weak references. For example (Fig. 3.7), provider A may have weak references to providers D and G. However, it does not imply that A is aware of the hierarchical relationship (parent-child) between providers D and G (Fig. 3.8). A corresponding real-life scenario would be: In an online shopping scenario, the buyer needs visibility over the online store to track his order and later over the courier company to track his/her shipment. Basically, the buyer needs weak references to both the online store and courier company. However, it does not need to be aware of the hierarchical relationship between the online store and the courier company, i.e., whether the courier company was invoked by the online store directly or by a 3rd party seller (which in turn was invoked by the online store).

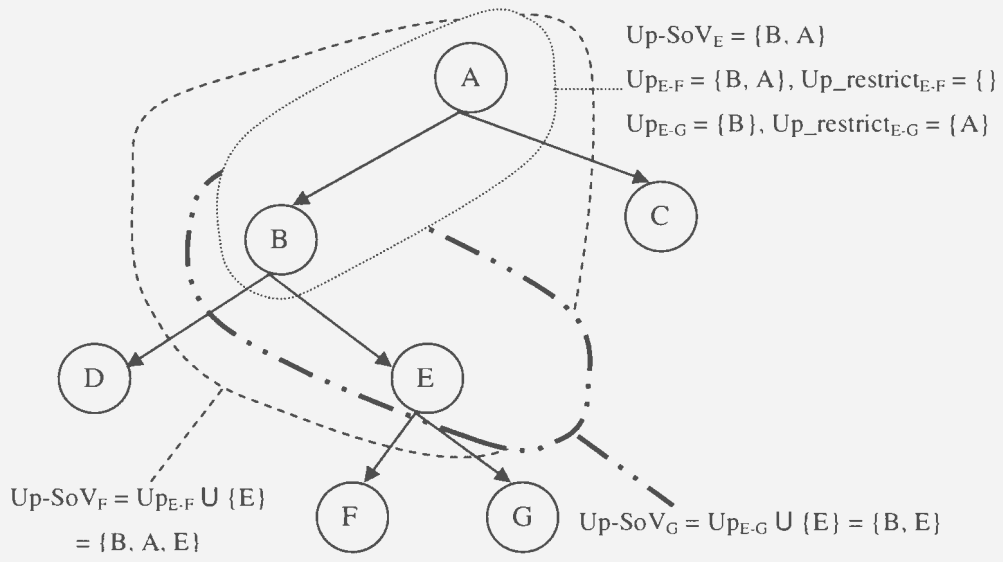

Figure 3.5 Example composition scenario highlighting the upward visibilities of F and G

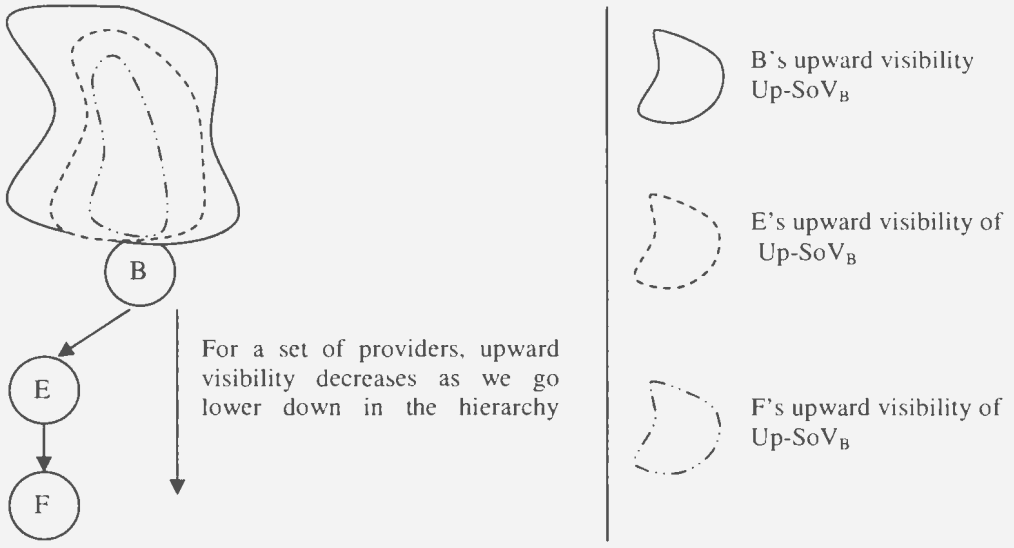

Figure 3.6 Upward visibility pattern

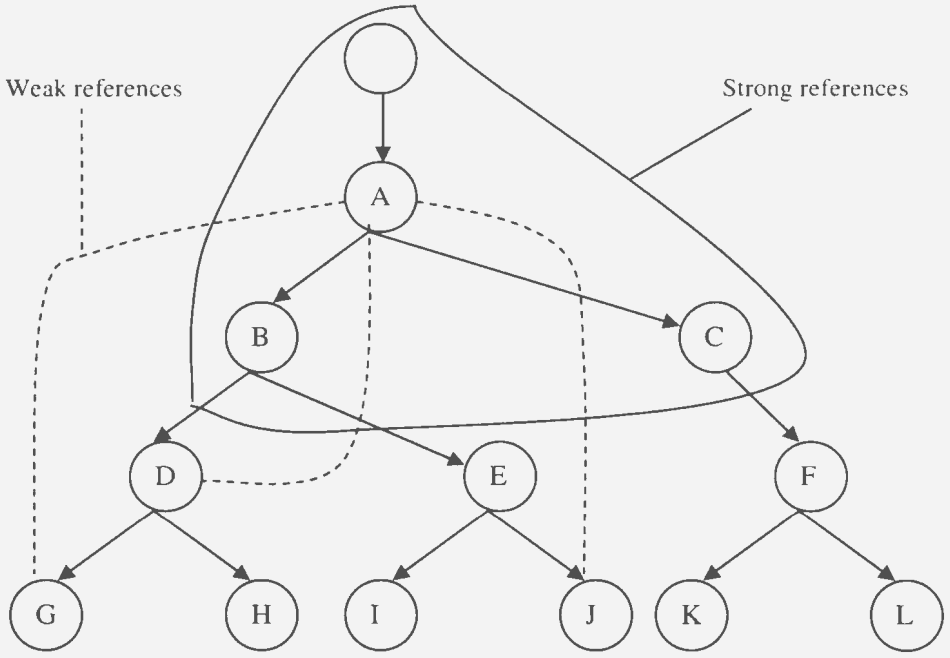

Figure 3.7 Example composition scenario highlighting the SoV$_A$ of provider A

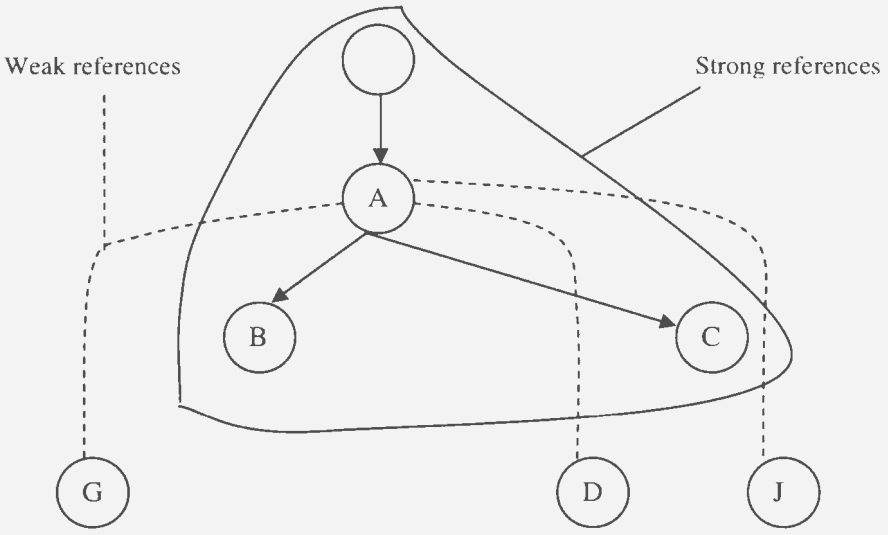

Figure 3.8 Graphical representation of the SoV_A of provider A as given in Fig. 3.7

Depending on the range of visibility, we can classify the SoV's as follows:

- Up-partial: Provider P has visibility over some of its ancestors.
- Down-partial: Provider P has visibility over some of the providers in the sub-tree rooted at P.
- Down-complete: Provider P has downward visibility over the entire sub-tree rooted at P. It is easy to observe that SoV_P is down-complete iff for each provider Q in the sub-tree rooted at P, $Down\_restrict_Q = \{\}$.
- Up-complete: Provider P has upward visibility up to the end-user. Similar to the definition of down-complete, SoV_P is up-complete iff for each ancestor (including parent) Q of P, $Up\_restrict_{Q\text{-}R} = \{\}$ where R is a child of Q in the path from Q to P.

Combinations of the above categories are obviously possible, beginning from "up-partial;down-partial" to "up-complete;down-complete".

3.3 Implementation

In this section, we discuss how we expect the SoV's to be specified. Here, we take a cue from the traditional Role Based Access Control (RBAC) implementations. With RBAC, the parents acquire the permissions of the children while the children inherit the prohibitions of their parents. In this case, the providers inherit the upward (downward) visibilities of their parent (children) with some restrictions (as applicable).

While strong references are ideal for scenarios which require visibility across levels in a hierarchical composition, weak references are more suited for ad-hoc visibility requirements. As mentioned earlier, the assignment of providers to SoV's is based on the principle of least privilege (assign only as much visibility as needed). The privacy and autonomy requirements may dictate hiding the details about all the providers in the upward and downward visibility of a provider P from its children and parent respectively, i.e., $Up\_restrict_{P\text{-}R} = Up\text{-}SoV_P$ for each child R and $Down\_restrict_P = Down\text{-}SoV_P$. Addition and deletion of the providers in $Up\_restrict_{P\text{-}R}$ and $Down\_restrict_P$ are as dictated by the visibility requirements of the functional and non-functional aspects. By functional requirements, we refer to the need for getting input, sending results/notifications to ancestors/descendants as specified by the composition schema. Different non-functional aspects like transactions, monitoring, user-interaction, etc. may have their own visibility requirements specified as Service Level Agreements (SLA) or policies (WS-Policy). For example, performance evaluation may require monitoring (visibility over) a sub-tree of the hierarchy. Similarly, compensation (as we discuss in the next chapter) may require direct visibility over the end-user. We do not rule out the possibility of conflicts between the visibilities requirements of the functional and

non-functional aspects. However, resolving the conflicts is application/context dependent and beyond the scope of this work. As such, we assume the existence of a non-conflicting set of visibility requirements for the hierarchical composition.

Below, we outline a simple scheme for the assignment of strong references. Let us assume that provider F is selected by E to execute part of its functionality (an action) - Fig. 3.9. Basically, for each selected provider (F), we need to specify the upward visibility of F (Step 2) and the downward visibility of its ancestors (B and A) with respect to F (Steps 3 and 4). The actual steps are as follows:

1. F sets $Parent_F$ = E and E adds F to $Children_E$.

2. E sets $Up_{E\text{-}F}$ to regulate the upward visibility of F.

3. If E needs to expose F to its parent B, it simply sends a message to its parent B, notifying B about F. If not, add F to $Down\_restrict_E$.

4. If B needs to expose F to its parent A, it simply forwards the information about F to A. If not, add F to $Down\_restrict_B$. Basically, this upward propagation of information about F continues until some provider (say, X) adds it to $Down\_restrict_X$ or it reaches the root provider (end-user).

With weak references, the situation is slightly different as the providers are not aware of the upward (downward) visibility requirements of their descendants (ancestors). With reference to Fig. 3.10, let us assume that provider A needs downward visibility over F (although, B does not have downward visibility over F). Given this, the visibility requirement can be propagated downwards (recursively via child providers) until F or a provider which has downward visibility over F is reached. Once the destination provider is reached, its

reference can be added to the weak references set ($Weak\_references_A$) of the source provider (A). On the same lines, if F needs upward visibility over A (although, its parent E does not have upward visibility over A) then the visibility requirement can be propagated upwards (recursively via parent providers) until provider A or a provider which has upward visibility over A is reached. Since weak references do not have any hierarchical relationship, we can extend the above scheme to specify weak references between providers which are not ancestors/descendants (such as F and C).

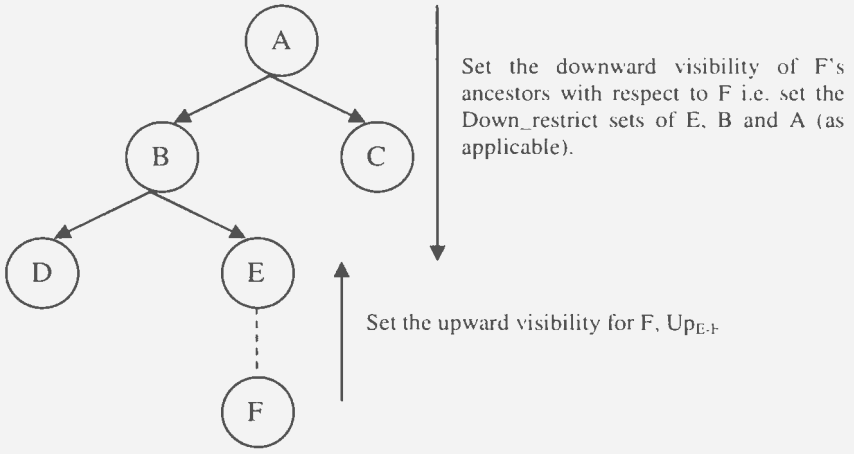

Figure 3.9 Assignment of strong references with respect to newly selected provider F

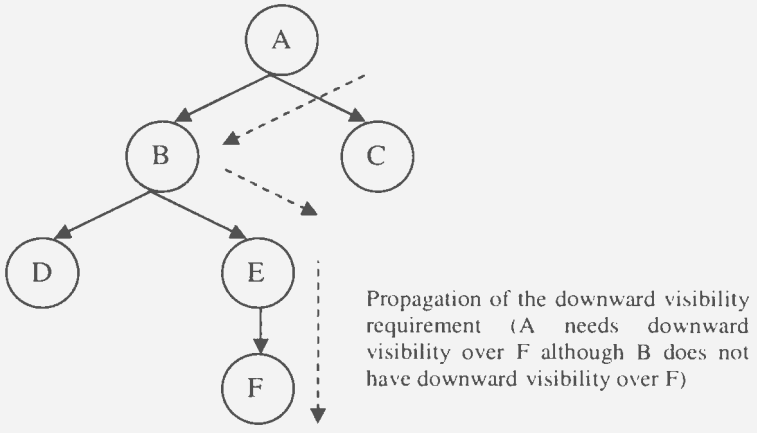

Figure 3.10 Weak reference assignment using visibility requirement propagation

Although for strong references assignment, the scheme (outlined above) updates the SoV's of all the affected providers with each provider selection, we would like to stress that this is not a requirement. Other more efficient schemes can be designed. It is easy to see that the above schemes can be extended to handle the deletion of references (from SoV's) too. Finally, we address the problem of specifying the visibility requirement that provider P needs visibility over provider Q even before Q has been selected (esp. with downward visibility). An example, in the online-shopping scenario, is where the buyer would like to have visibility over the courier company to track his/her order. To overcome this problem, we assume that all visibility requirements are specified in terms of "the provider of action x (of the composition schema)" rather than "provider y". Basically, the visibility requirement in the example online shopping scenario would be specified as "the buyer needs visibility over the service provider handling shipping" rather than DHL/FedEx. This is similar to the concept of "roles" in RBAC where permissions are assigned to roles irrespective of the actual users assigned to the roles at run-time.

3.4 Modeling Real-Life Scenarios Using SoV

In this section, we illustrate how some real-life scenarios can be modeled using SoV. First, let us consider a typical e-transaction scenario as shown in Fig. 3.11. Let the visibility requirements be as follows:

- The courier companies need visibility over the contact details of the customer to deliver the goods: The requirement can be accommodated by adding the customer's reference to the weak references sets of the courier companies. Now, their parents (suppliers A and B)

may not have provider details visibility over the customer. Thus, the visibility requirement needs to be propagated via parent providers (as discussed earlier).

- To track the status of his/her order (at all times), the customer needs visibility up to the courier companies' levels for each of the purchased products: The above visibility requirement can be accommodated by providing strong references to the courier companies used to deliver the products (implying that the customer has visibility over any intermediate suppliers).

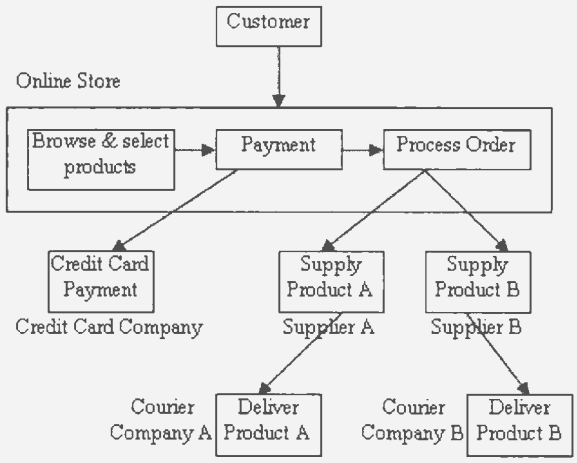

Figure 3.11 Typical e-transaction scenario

The earlier (e-transaction) scenario illustrated the use of SoV with respect to provider details visibility. We refer to the supply management scenario given in [LTS98] to illustrate the utility of service details visibility. [LTS98] states the need for upward and downward information sharing as "the information acquired by downstream entities are mainly material and capacity availability information from their suppliers. The information acquired by an upstream entity is information about customer demand and orders. The depth of information penetration can be specified in various degrees, e.g., isolated, upward one tier, upward two

tiers, downward one tier, downward two tiers, and so forth." The above visibility requirement can be elegantly accommodated by assigning strong service details visibility up to the desired level.

Chapter 4

Compensation

Due to their long running and loosely coupled nature, compensation based models have been widely accepted as the preferred mechanism to implement Web services transactions. Compensation based models, in the event of a failure, try to undo the effects of the currently executing activities (using roll-back) and terminated ones (using compensation) to preserve atomicity. Below, we consider some aspects required to ensure satisfactory compensation for hierarchical Web services compositions.

4.1 Cost of Compensation (CoC)

Most models assume the existence of a predefined compensating action, which is invoked in case of failure. They fail to appreciate the fact that there may be multiple ways of compensating the same actions (effects of the action). Real-world activities often have a cost associated with them. As such, the different options may have an associated cost, the Cost of Compensation (CoC). *The actual CoC may be in terms of the time, money, effort, etc. required to perform compensation.* The compensating options may also vary depending on the extent of compensation provided. Below, we describe a simple classification for compensating actions:

- Fully compensatable: Such actions are capable of negating the effects of the original transaction completely. In OWL-S terminology, the "effect" of a fully compensating action would be the exact negation of the "effect" of the action (for that particular invocation) it compensates. The only exception is probably the noop action which does not have any effect

on the execution (the compensating action corresponding to a noop should also be a noop action).

- Conditionally compensatable: By conditionally compensatable, we refer to those compensating actions which allow full compensation, but have a condition associated with them. For example, often vendors instead of refunding money allow the customer to purchase something equal in worth to the returned item. Studies [GLC99] have shown that most refund policies can be described as a conditional relationship between price, quantity and time. We find the factor customer relationship/history (premium member, credit rating, etc.) as being equally important and assume all compensation options as described in terms of the factors: price, quantity, time and customer relationship. The effects of a conditionally compensatable action can be expressed as a percentage of the effects of the action it compensates. For example, if the effect of a booking action is "Ticket booked and $x paid" then the effect of a conditionally compensating action satisfying the condition "quantity (>5 tickets) & time (>14 days in advance) -> money (refund 90%)" would be "Ticket canceled and 90% of $x received".

- Partially compensatable: As the name suggests, such actions are capable of compensating the effects of the original action partially. For example, if the effect of a trip planning action is "Flight booking St. John's - London, London - Bahrain and Bahrain - Delhi, Hotel booking at London and Bahrain" then the effects of a partially compensatable action (for the above action) might be "Flight booking London - Delhi, Hotel booking at London canceled" or "Flight booking London - Bahrain and Bahrain - Delhi, Hotel booking at Bahrain canceled".

In a nested composition scenario, compensation may be possible at different levels with different costs. For example, let us consider the classic travel booking scenario (Fig. 4.1). If the hotel and flight booking need to be compensated then it can be achieved by either invoking the compensating (cancellation) operations of the hotel and flight booking providers respectively or by invoking the compensating operation (Cancel Travel) of the composite travel booking provider. Now, if we assume that the user is a premier member of the composite travel booking provider and as such gets a 15% discount on all cancellation charges then it makes sense to invoke the Cancel Travel operation of the composite travel booking provider, i.e., perform compensation at a higher level. On the same lines, most travel agents consider cancellation as a separate activity (similar to booking) and would charge their commission in addition to the cancellation charges applicable (Fig. 4.2). Given this scenario, it makes sense to compensate by invoking the Cancel Hotel and Cancel Flight operations directly, i.e., perform compensation at a lower level.

Please note that the above concept is different from the upward propagation of unhandled faults allowed by most transaction models. For instance, the "throw" mechanism followed by BPEL allows transferring the responsibility for compensation to the enclosing scope in case the current scope (where the failure has occurred) does not have an appropriate handler or the compensation fails. While the mechanism allows compensation at different levels, it is still based on the concept of trying to perform the first possible compensation rather than acknowledging the possibility of multiple options and trying to perform the most optimum one. Also, there is no downward propagation of faults and therefore the compensation options at lower levels are not considered.

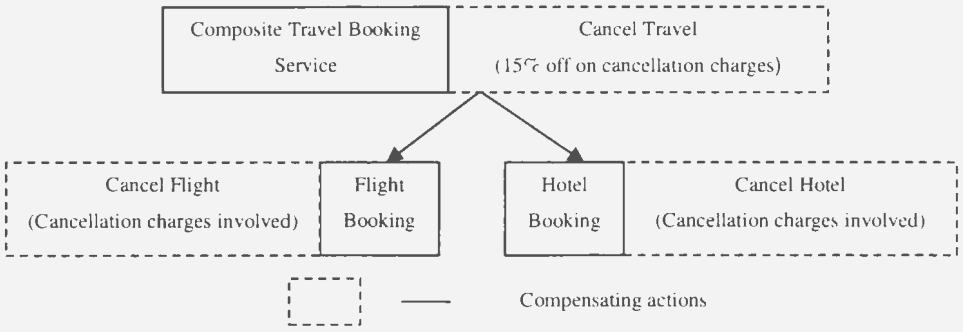

Figure 4.1 Travel booking scenario A

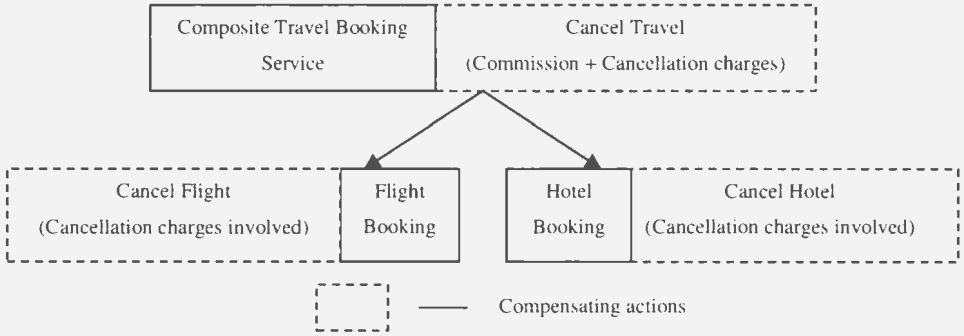

Figure 4.2 Travel booking scenario B

4.2 End-User Involvement

In the previous section, we discussed how there might be multiple compensation options available to recover from a failure. Given this, we need a mechanism to select the compensation option and more often than not it is the end-user who has the required information and intelligence to take such a decision. Please note that people often equate compensation selection with fault (exception) handler selection. However, there is a subtle difference between the two. Fault handler selection depends on the type of fault that has occurred and as such can be done by the system. Compensation selection, on the other hand, is much more complex. During compensation we are trying to semantically undo the effects

of an action which has earlier completed successfully (without failure). There are many variable factors like current state of the system, extent to which the results need to be undone (for forward recovery), CoC, etc. to be considered which make the process of compensation selection quite complex.

Also, we have the other extreme (with respect to multiple available compensation options) where compensation is not possible. In such a case, the end-user might help by suggesting alternate acceptable outcomes. For example, if the flight reservation cannot be compensated (non refundable tickets) then the user might change his initial requirement of both flight and hotel reservation to just flight tickets saying he can get the hotel reservation from some other travel agency. Researchers have also proposed forward recovery schemes as a means of overcoming the limitations posed by non-compensatable actions. However, forward recovery does not mean that there is no need for compensation. In fact, [VV04] acknowledges that forward recovery might not be always possible and it might be required to compensate up to some extent and then try forward recovery. [TIRL03] relies on the concept of co-operative recovery. For example, in the case where either the flight or hotel booking fails, the component service raises an exception that is cooperatively handled at the next higher level. While conceptually absolutely feasible, the question we need to ask here is "Are current software systems intelligent enough to take such decisions on their own?" Probably the only systems capable of showing such intelligence are MAS. Even if we assume MAS, it is probably more feasible to have an agent acting on behalf of the end-user [KS02] rather than two agents trying to negotiate a recovery. The reason is that an agent acting on behalf of the end-user needs to keep only the user's interests in mind while two agents trying to negotiate

would involve a conflict between the interests of the respective parties the agents represent (which is definitely much more complex).

4.3 Side-Effects and Provider Independent Compensation

As mentioned earlier, compensation is defined as semantically undoing the effects of an execution. These effects are usually the externally visible or advertised effects of an execution. We show that in addition to such effects, there might be some effects which are not always evident especially in a hierarchical composition. However, they would still need to be canceled to achieve full compensation.

Thus, by side-effects, we refer to those effects of an execution which are in addition to the advertised (expected) effects of an execution. For example, let us assume that a shipping service provider S uses trucks provided by transportation provider T for its operations. As such, a booking with S would also involve a booking with T. However, there is no mention of T's involvement in the booking confirmation receipt provided by S to the customer. Thus, the booking with T is a side-effect of the above execution and would need to be canceled along with S's booking for a successful compensation. Below, we take a look at the different ways side-effects arise in a hierarchical composition and the information about them required for compensation:

- Internal actions: Side-effects usually arise as a result of the internal actions involved. In a hierarchical composition, the input/output actions of the providers become internal actions of their parents. For example, in the shipping scenario discussed above, the arrangement with transportation provider T would map to an internal action of S. Thus,

information about the internal actions also needs to be recorded. In addition to recording the effects of internal actions, we also need to capture their outputs (if any). Such outputs might be needed to undo their effects during compensation. For example, with respect to the shipping scenario (discussed above), we would need T's booking reference number to cancel the booking with T. However, this information is not in the booking confirmation receipt given to the customer. Thus, S would need to store this information somewhere so that it can be used for compensation later (if required).

- Output and input action concatenation: Composition involves connecting the output action of a provider to the input action of another provider. However, exact matches are rare and often the output actions result in effects (output values) which are not required (utilized) by the next input action in sequence. For example, a credit card payment (output action) may initiate delivery (input action) in an online shopping scenario. Now, let us assume that the credit card used has an associated reward points system. The award of credit points is clearly a side-effect and not required for delivery (the next input action). However, a successful compensation would require not only crediting the paid amount to the credit card account but also deducting the awarded credit points. As in the previous mechanism, here also we require that any additional output be recorded so that it can be used for compensation later.

Please note that all invoked providers do not necessarily produce side-effects. Also, side-effects may not always be relevant for compensation. For example, registering with a web site may give discounted rates (as applicable) with partner sites which remain valid even if the original registration is canceled. Another scenario would be side-effects canceling each other.

With the above discussion about side-effects in the background, let us consider the notion of provider independent compensation. Most transaction models consider compensation as a provider dependent activity, i.e., compensation is the responsibility of the original service provider. We argue that this is an unnecessary restriction and propose provider independent compensation as an alternative. Basically, we would like to consider compensation as any normal activity (whose provider is selected dynamically at run-time) rather than constrain its execution to the original service provider. Towards this end, we identify the information which would be required as well as the conditions which determine if provider independent compensation is feasible.

The information required for provider independent compensation of the effects of a provider P is basically the services provided details (execution sequence, input/output values, effects, etc.) of P. Due to side-effects, we need the services provided details of all the providers in the sub-tree rooted at P which produce side-effects. An XML schema which can be used to capture and store the above (compensation) information appears in Appendix A. We assume that the original service providers retain such information until their expiry (flight departure time for a flight booking action). It is similar to saying that the original service providers enter the stage *"Completed but Compensatable"* for a limited period after completing execution. In case of failure, the compensation information (maintained by the original service providers) is used to select a provider for compensation (dynamic binding). Once selection is over, the stored (compensation) information is passed to the selected provider (required to perform the compensation).

Until now, we have assumed that the original providers are ready to share their services provided details with other providers (at least, with the compensation providers). While it may be a valid assumption given that we are dealing with static composition (can be specified as part of the contract), it may not always be feasible due to the security and autonomy requirements of the providers. As such, let us assume that each provider P has visibility over the services provided details of a limited set S of providers in the sub-tree rooted at P. Let S_s denote the providers in the sub-tree rooted at P which produce side-effects. Given such a scenario, provider independent compensation of P is possible iff S is a superset of S_s. We consider the partial visibility aspect in detail in the next section.

4.4 Sphere of Control for Compensation (SoCC)

For compensation, we first need to compute the possible compensation options. The compensation options consist of the provider independent options and the compensation possible at different levels. For example (with reference to Fig. 4.3), let us assume that provider D has failed and as a result the effects of provider C need to be compensated. Given this, some of the possible compensation options are {C}, {E, F}, {E, I, J}, {F, G, H}, {G, H, I, J}. For each option, the effects of each provider can be compensated either in a provider independent fashion or by invoking a compensating action of the original provider (provider dependent compensation). For example, we can expand the option {E, I, J} as provider independent compensation for E and invoke the compensating actions of I and J.

Thus, to compute the compensation options, the providers affected by the failure are first brought into a *SoCC, called the Sphere of Control for Compensation. The affected providers*

are usually the terminated providers (which have finished executing their actions) in the sub-tree rooted at the parent of the failed provider. Once we have the SoCC, the compensation options can be computed as follows: If we consider the SoCC as a set of providers, each compensation option is a subset of the SoCC with the following properties (with reference to Fig 4.3):

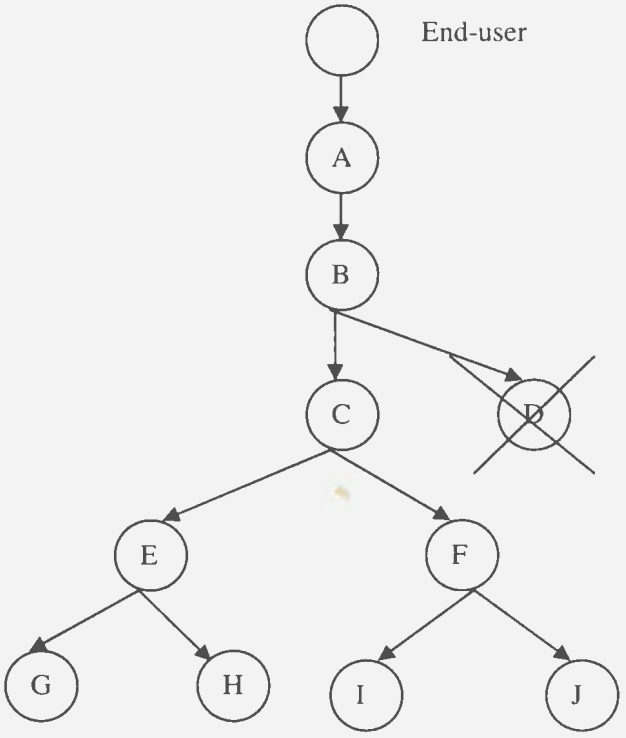

Figure 4.3 Example hierarchical composition scenario

- Consistency: The subset cannot contain providers having an ancestor-descendant (including parent-child) relation between them. For example, the subset {E, F, I} is not consistent because F and I have a parent-child relationship. The consistency criterion is based on the fact that provider independent compensation or invoking the compensating action of any provider (F) in the hierarchy also compensates the effects of any providers in the sub-tree

rooted at F. Thus, there is no need to compensate the effects of any of its descendants (I) separately.

- Completeness: The providers in the subset should compensate the effects of the original execution completely. For example, the subset {G, H, I} is not complete because the effects of J won't be compensated.

Finally, to perform the actual compensation, we need the provider details of the providers to invoke their compensating actions and the service details of the providers for provider independent compensation. Until now, we have assumed that there are no visibility restrictions in the hierarchical composition. Now, let us consider the visibility restrictions modeled as SoV. With SoV's defined, only those affected providers can be brought into the SoCC which are in the Down-SoV_P of the parent P of the failed provider. Since we are interested in both the provider and service details, we consider the providers in both Down-$SoV_{P\text{-}prov}$ and Down-$SoV_{P\text{-}service}$. With the visibility restrictions, we need to consider the following conditions (in addition to the consistency and completeness properties) to compute the compensation options.

- Compensation by invoking the compensating actions of the original provider: A provider's (Q) effects can be compensated by invoking the compensating actions of either Q or of all of Q's children. However, for the above to be possible, Down-$SoV_{P\text{-}prov}$ should be a superset of $Children_Q$ (P has downward provider details visibility over all of Q's children).

- Provider independent compensation: Provider independent compensation of any provider Q is possible iff Down-$SoV_{P\text{-}service}$ is a superset of the providers in the sub-tree rooted at Q *which produce side-effects*. Since it is not possible for P to determine if a lower

level provider R actually produces side-effects unless R belongs to Down-$SoV_{P\text{-}service}$ (or R is willing to provide such information itself), we change the requirement for provider independent compensation as follows: "Provider independent compensation of any provider Q is possible iff Down-$SoV_{P\text{-}service}$ is a superset of *all* the providers in the sub-tree rooted at Q (down-complete with respect to Q)".

Below, we outline a simple algorithm to compute the compensation options with the visibility restrictions (as specified by the SoV's) in place. The algorithm is initiated by the coordinator corresponding to the parent of the failed provider (hereafter, referred to as the initiating coordinator and provider respectively). The algorithm could be initiated by other ancestors too (discussed later). Initially, the algorithm traverses downwards, as permitted by the downward visibility of the initiating provider, to compute the providers in the SoCC. Once the SoCC has been computed, it uses the conditions outlined above to determine the compensation options. To keep the algorithm simple, we consider the parts to determine the provider independent and dependent options separately. However, the two parts can easily be interleaved and there is no need to traverse the sub-tree (rooted at the initiating provider) twice.

/ Part I: Algorithm to determine the possible provider dependent compensation options at different levels and acquire the provider details of the providers (as applicable) */*

1. The initiating coordinator (say, C-P) sends the provider details seeking message (hereafter, referred to as the PM) to the coordinators of its affected children providers.

2. /* Downward traversal */ Each child coordinator (say, C-Q), on receiving the PM, forwards it to the coordinators of the providers in (Down-$SoV_{P\text{-}prov} \cap Children_Q$), i.e., to Q's

children whose provider details are visible to the initiating provider P. /* The request is forwarded to only the children at a level to maintain the hierarchical structure of the acquired information. */

If the PM is received by a primitive coordinator, it simply returns a blank message to its parent /* based on the assumption that any parent is aware of the provider details of all its children. As such, the provider details of the children can be returned by their parents (Step 3) */.

3. Once the coordinator C-Q has received replies from all its children (to which it had sent the PM), it checks if (Down-$SoV_{P\text{-}prov}$ is a superset of $Children_Q$). /* Check if P has downward provider details visibility over all of Q's children */

If so, C-Q appends the provider details of all its children to the information returned by its children and sends it to its parent.

Otherwise, C-Q appends the message "Provider dependent compensation is not possible at the level of Q's children" and the provider details of its children in (Down-$SoV_{P\text{-}prov} \cap Children_Q$) to the information returned by its children and sends it to its parent. This upward propagation of information continues until the initiating coordinator has received replies from all its children coordinators.

/ Part II: Algorithm to determine the provider independent compensation options and acquire the service details of the providers (as applicable). */*

1. The initiating coordinator (C-P) sends the service details seeking message (hereafter, referred to as the SM) to the coordinators of its affected children providers.

2. Each child coordinator (say, C-Q), on receiving the SM, forwards it to the coordinators of the providers in (Down-$SoV_{P\text{-}service} \cap Children_Q$), i.e., to Q's children whose service details are visible to the initiating provider P.

If the SM is received by a primitive coordinator, it returns a blank message to its parent.

3. Once the coordinator C-Q has received replies from all of its children (to which it had sent the SM), it checks if (Down-$SoV_{P\text{-}service}$ is a superset of $Children_Q$) or if it has received a "Provider independent compensation is not possible at R" message from one of its children R. /* Check if P has downward service details visibility over all the providers in the sub-tree rooted at Q. */

If so, C-Q appends the service details of all its children to the information returned by its children and sends it to its parent.

Otherwise, C-Q appends the message "Provider independent compensation is not possible at Q" and the service details of its children in (Down-$SoV_{P\text{-}service} \cap Children_Q$) to the information returned by its children and sends it to its parent. This upward propagation of information continues until the initiating coordinator has received information from all its children coordinators (to whom it had sent messages).

Please note that the above algorithm requires the lower level providers to have access to the Down-$SoV_{P\text{-}prov}$/Down-$SoV_{P\text{-}service}$ of the initiating provider P. We assume that the Down-$SoV_{P\text{-}prov}$ (Down-$SoV_{P\text{-}service}$) is propagated along with the PM (SM). Further (to protect the confidentiality of P's visibility information), we assume that each coordinator Q forwards only the subset (Down-$SoV_P \cap Down_R$) to its child R (rather than the complete set Down-SoV_P). For example, with reference to Fig. 4.3, if Down-$SoV_{B\text{-}service}$ = {C, D, E, F, G, H, I}

then C-B sends {E, F, G, H, I} along with the SM to C-C. Similarly, C-C would send {G, H} and {I} along with the SM (while forwarding the SM) to C-E and C-F respectively. Also, note that selecting provider dependent compensation for a provider X implies passing the responsibility of compensation (of the effects of X) to X. The corresponding provider X could very well choose provider independent compensation for a descendent Y (provided, of course, that the downward visibility of X is down-complete with respect to Y).

Let us walk through the above algorithm with the help of an example scenario.

Scenario (Fig. 4.4): We assume that provider D has failed and as a result the effects of provider C need to be compensated. Thus, C-B is the initiating coordinator for the algorithm. Further, we assume that B has visibility as shown in Fig. 4.4. Basically, Down-$SoV_{B\text{-}service}$ = {C, F, I, J} and Down-$SoV_{B\text{-}prov}$ = {C, E, G, H}. Given this, a trace of the algorithm would be as follows:

Part I:

1. C-B sends the PM to C-C.

2. Since Down-$SoV_{B\text{-}prov} \cap Children_C$ = {E}, C-C forwards the PM to C-E. Since Down-$SoV_{B\text{-}prov} \cap Children_E$ = {G, H}, C-E forwards the PM to both C-G and C-H. Both G and H are primitive providers, as such C-G and C-H simply return blank messages to C-E.

3. On receiving replies from C-G and C-H, C-E returns the provider details of G and H to C-C since Down-$SoV_{B\text{-}prov}$ is a superset of $Children_E$. On receiving the reply from C-E, C-C appends the message "Provider dependent compensation is not possible at the level of C's children" and the provider details of E to the information returned by C-E and sends it to C-B

(because Down-$SoV_{B\text{-}prov}$ is not a superset of $Children_C$). C-B terminates the algorithm on receiving the information from C-C.

Part II:

1. C-B sends the SM to C-C.

2. Since Down-$SoV_{B\text{-}service} \cap Children_C = \{F\}$, C-C forwards the SM to C-F. Since Down-$SoV_{B\text{-}service} \cap Children_F = \{I, J\}$, C-F forwards the SM to both C-I and C-J. Both I and J are primitive providers, as such C-I and C-J simply return blank messages to C-F.

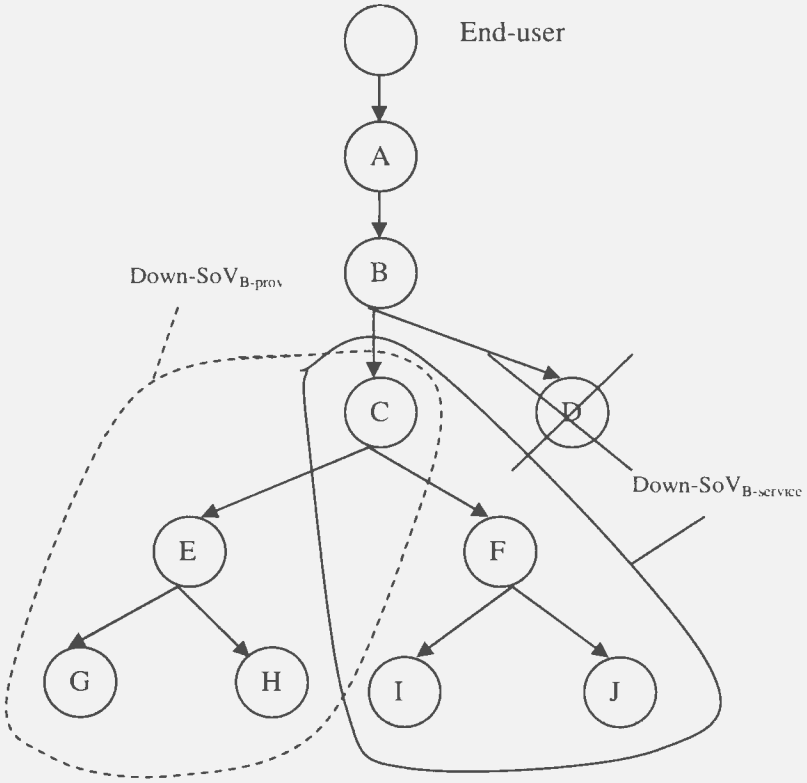

Figure 4.4 Example composition scenario of Fig. 4.3 with visibility restrictions

3. On receiving replies from C-I and C-J, C-F returns the service details of I and J to C-C since Down-$SoV_{B\text{-}service}$ is a superset of $Children_F$ and C-F hasn't received a "Provider

independent compensation is not possible at I (J)" message from C-I (C-J). On receiving the reply from C-F, C-C appends the message "Provider independent compensation is not possible at C" and the service details of F to the information returned by C-F and sends it to C-B (because Down-$SoV_{B\text{-}service}$ is not a superset of $Children_C$). C-B terminates the algorithm on receiving the information from C-C with the conclusion that provider independent compensation is possible only up to F.

The initiating coordinator uses the information received from its children (to which it had sent the PM/SM) in accordance with the consistency and completeness properties (discussed earlier) to compute the possible compensation options. For the example scenario (Fig. 4.4), the computed compensation options would be:

- Option 1: Invoke the compensating action of C.
- Option 2: Provider independent compensation of F and invoke the compensating action of E.
- Option 3: Provider independent compensation of F and invoke the compensating actions of G, H.
- Option 4: Provider independent compensations of I, J and invoke the compensating action of E.
- Option 5: Provider independent compensations of I, J and invoke the compensating actions of G, H.

Finally, we need to select the best compensation option. Let us assume that we need the end-user's feedback (end-user involvement to select among the available compensation options) or approval from a manager higher up in the hierarchy.

/ Algorithm to get end-user feedback */*

1. The initiating coordinator (say, C-P) checks if it has contact (provider) details visibility over the end-user, i.e., if End_User ε Up-$SoV_{P\text{-}prov}$. If so, C-P can contact the end-user directly to get his/her feedback. If not, the initiating coordinator sends a request for feedback message (hereafter, referred to as the RFM) to its parent, identifying itself and the end-user as the ultimate source and destination respectively.

2. The parent coordinator, on receiving the RFM, checks if it has provider details visibility over the end-user. If so, it forwards the RFM directly to the end-user. Otherwise, forwards the RFM to its parent. This upward propagation continues until the RFM reaches the end-user.

3. The end-user, on receiving the RFM, provides his/her feedback to the initiating coordinator. ¬

Example Scenario (Fig. 4.5): We assume that provider D has failed and as a result the effects of provider C need to be compensated. Thus, C-B is the initiating coordinator for the algorithm. Further, we assume that C-B has already computed the SoCC and now needs end-user feedback to select the optimum compensation option. Given the SoV of provider B as shown in Fig. 4.5, a trace of the above algorithm would be as follows:

1. Since End_User is not a member of Up-$SoV_{B\text{-}prov}$, it sends a RFM to its parent (C-A) identifying itself and the end-user as the ultimate source and destination respectively.

2. C-A forwards the RFM to the end-user (End_User belongs to Up-$SoV_{A\text{-}prov}$).

3. The end-user, on receiving the RFM, provides its feedback to C-B. ¬

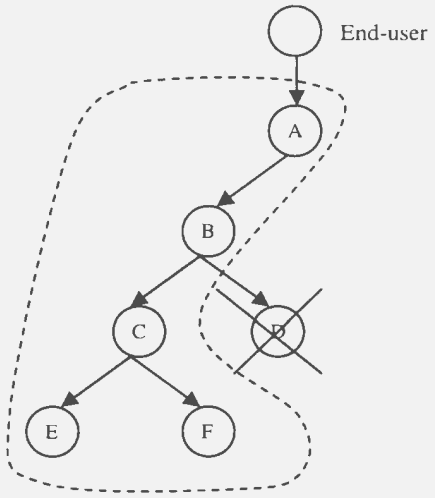

Figure 4.5 Example composition scenario

The above message sending pattern, i.e., sending messages to an ultimate destination via intermediaries is supported by SOAP/WS-Routing [WS-R]. Although, we consider user-interaction with respect to compensation above, it is easy to see that the mechanism can be directly applied to support any user (or ancestor) interaction requirements of the service providers.

In the discussion until now, we have considered that the parent of the failed provider is (the initiating provider) responsible for computing the compensation options. As such, the computed options are limited to the level of the parent. However, compensation may be possible at higher levels too. For example (with reference to Fig. 4.4), the effects of C can be compensated by compensating the effects of higher level providers A/B (the effects of A/B are a superset of the effects of C). The above can be accommodated by making a slight modification to the end-user feedback algorithm. The initiating coordinator sends the computed compensation options to its parent (irrespective of whether it has provider details

visibility over the end-user or not). The parent coordinator, on receiving the information, assumes the role of the initiating coordinator and executes the same algorithm (discussed earlier) to compute the provider dependent and independent compensation options at its level. The parent coordinator passes the computed compensation options along with the received options (from its child) to its parent. This upward propagation continues until the end-user receives the accumulated compensation options. For example (again, with reference to Fig. 4.4), let us assume that A had invoked another provider B' before B and the downward visibility of A is as shown in Fig. 4.6. Thus, we need to compensate both B and B' to compensate the effects of A.

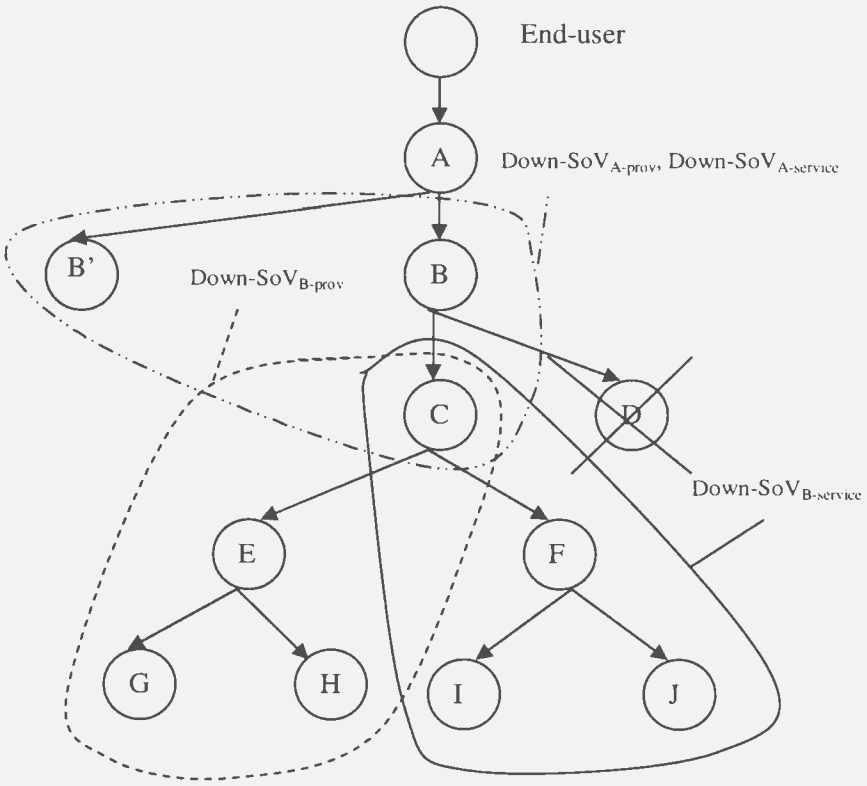

Figure 4.6 Extended composition scenario with visibility restrictions

Given this, the compensation options (available to the end-user) would be as follows:

/* compensation by compensating the effects of C */

- Option 1: Invoke the compensating action of C.
- Option 2: Provider independent compensation of F and invoke the compensating action of E.
- Option 3: Provider independent compensation of F and invoke the compensating actions of G, H.
- Option 4: Provider independent compensations of I, J and invoke the compensating action of E.
- Option 5: Provider independent compensations of I, J and invoke the compensating actions of G, H.

/* compensation by compensating the effects of B */

- Option 6: Invoke the compensating action of B.

/* compensation by compensating the effects of A */

- Option 7: Invoke the compensating action of A.
- Option 8: Invoke the compensating actions of B' and B.
- Option 9: Provider independent compensation of B' and invoke the compensating action of B.
- Options 10-14: Invoke the compensating actions of B' and (Option 1 or Option 2 or Option 3 or Option 4 or Option 5).
- Options 15-19: Provider independent compensation of B' and (Option 1 or Option 2 or Option 3 or Option 4 or Option 5).

Chapter 5

Monitoring

In a hierarchical composition, it is essential to be able to provide information about the state of execution *at the right granularity (level)* to the user. For example, while a user might be interested in execution status messages of the form "The bedroom is being painted", he might not be interested in messages of the form "The house is being built" or "1000th bedroom brick is being laid". Thus, we consider the problem of capturing the states of all the providers (which have been invoked until now) at different levels of the hierarchical composition. Before discussing the actual algorithm, let us have a look at the underlying state transition and composition model.

5.1 State Transition and Composition Model

We consider the composition model as shown in Fig. 5.1. In addition to the coordinator (mentioned earlier), each service provider has a log manager associated with it. The log manager logs information about any state transitions as well as any messages sent/received by the provider (protocols view of services). The state transitions and messages considered are as outlined in Fig. 5.2.

- Not - Executing (NE): The provider is waiting for an invocation.
- Executing (E): On receiving an Invocation message (IM), the provider changes its state from NE to E.
- Suspended (S) and Suspended by Invoker (IS): A provider, in state E, may change its state to S due to an internal action (Suspend) or IS on the receipt of a Suspend message (SM).

Conversely, the transition from S to E occurs due to an internal action (Resume) or from IS to E on receiving a Resume message (RM) respectively.

- Canceling (CI), Canceling due to invoker (ICI) and Canceled (C): A provider, in state E or S or IS, may change its state to CI due to an internal action (Cancel) or to ICI on the receipt of a Cancel message (CM). Once it finishes cancellation, it changes its state to C and sends a Canceled message (CedM) to the invoking provider. Please note that cancellation may require canceling the effects of some of its children (discussed later).
- Terminated (T) and Compensating (CP): The provider changes its state to T once it has finished executing the action. On termination, the provider sends a Terminated message (TM) to the invoking provider. A provider may be required to cancel an action even after it has finished executing the action (Compensation). A provider, in state T, changes its state to CP on receiving the CM. Once it finishes compensation, it moves to C and sends a CedM to the invoking provider.

To keep the discussion simple, we assume that each provider is responsible for executing a single action (composite/primitive). The state of a provider at time t is its execution history until t. For simplicity (and where there is no scope for confusion), we represent the state of a provider at t in terms of the state of its executing action at t and sometimes, also as a combination of the states of its executing and invoked actions at t. For example, if the execution history of a provider P_A until t is "(Receive IM of a_1 from User, E_1) (E_{11}, Send IM of a_{11} to P_B) (E_{12}, Send IM of a_{12} to P_C) (Receive TM of a_{11} from P_B, T_{11})" then the state of P_A at t can be represented as E_1 or E_1 (T_{11}, E_{12}). Due to the support for solicit-response invocations, the terms "parent" and "child" with respect to the hierarchical composition need

a little clarification here. Usually, the invoking and invoked providers are referred to as the "parent" and "child" respectively. However, the above definition is clearly not suitable for solicit-response invocations where a "child" invokes an action of the "parent". Here, it helps to recall that any solicit-response invocation of provider P by Q implies an earlier request-response invocation of provider Q by P. Thus, we use the terms "parent", "child", "ancestor" and "descendant" corresponding to the initial request-response invocation.

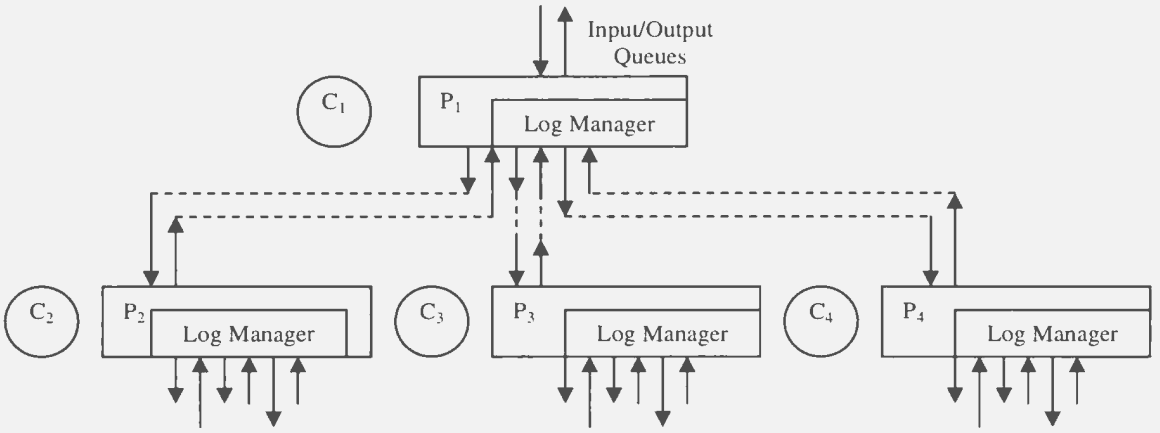

Figure 5.1 Composition infrastructure

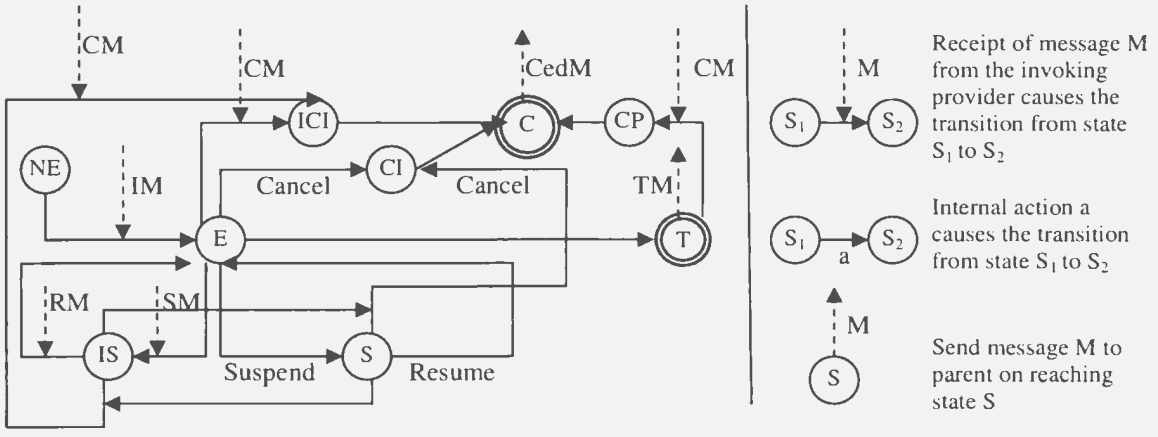

Figure 5.2 Invoked provider lifecycle

The lifecycles of an invoking and invoked provider are not independent. In fact, the discussion until now (Fig. 5.2) can be considered as the lifecycle of an invoked provider with respect to its invoker. Below, we consider a few characteristics of the lifecycle of an invoking provider with respect to its invoked providers (Fig. 5.3). Please note that a composite provider can invoke many providers. Thus, Fig. 5.3 shows the lifecycle stages of a composite provider with respect to an invoked provider. The same cycle would be repeated for other invoked providers. We augment the discussion below with an example scenario where a composite service provider P invokes an action a_{11} of provider Q.

- Normal execution: Once P starts executing an action (E), it is capable of invoking actions of other providers. To invoke action a_{11}, P sends the corresponding IM to Q.
- Suspension: Provider P may decide to suspend any of its invoked actions (which are still executing). For example, if P is currently in state E_1 (E_{11}) and it decides to suspend the action a_{11} then it sends the SM to Q and changes its state to E_1 (IS_{11}). Whenever P decides to resume action a_{11}, it sends the corresponding RM to Q and changes its state back to E_1 (E_{11}).
- Cancellation: We allow for two types of cancellation. (1) Provider P decides to cancel one of its invoked actions. For example, if P is currently in state E_1 (E_{11}) or E_1 (T_{11}) and it decides to cancel the action a_{11} then it sends the CM to Q and changes its state to E_1 (ICI_{11}) or E_1 (CP_{11}) accordingly. Please note that the same message CM is used for both cancellation and compensation. We do not differentiate between the two because of synchronization problems between parent-child providers. To illustrate the problem, let us assume that we have separate messages for cancellation (CM) and compensation (say, CpM). Consider a situation where the child has terminated (T) but its TM has not yet reached the parent. Now,

if the parent had to cancel the execution of the child, it would send a CM to the child (since the state of the child is still E at the parent's site). However, the child has already terminated and requires a CpM to cancel its effects. (2) Provider P needs to cancel its execution (due to its internal action Cancel or on receiving a CM from its parent), implying cancellation for all the actions invoked by P. For example, if the current state of P is E_1 (T_{11}) and it receives a CM then it sends a CM to Q and changes its state to ICI_1 (CP_{11}). Please note that the above state transition is not evident from Fig. 5.3.

- Termination: Provider P changes the state of a_{11} to T (C) on receiving the TM (CedM) from Q. Needless to say, P can change the state of its action a_1 to T (C) only after it has received the TM (CedM) from the providers of all the actions invoked by P.

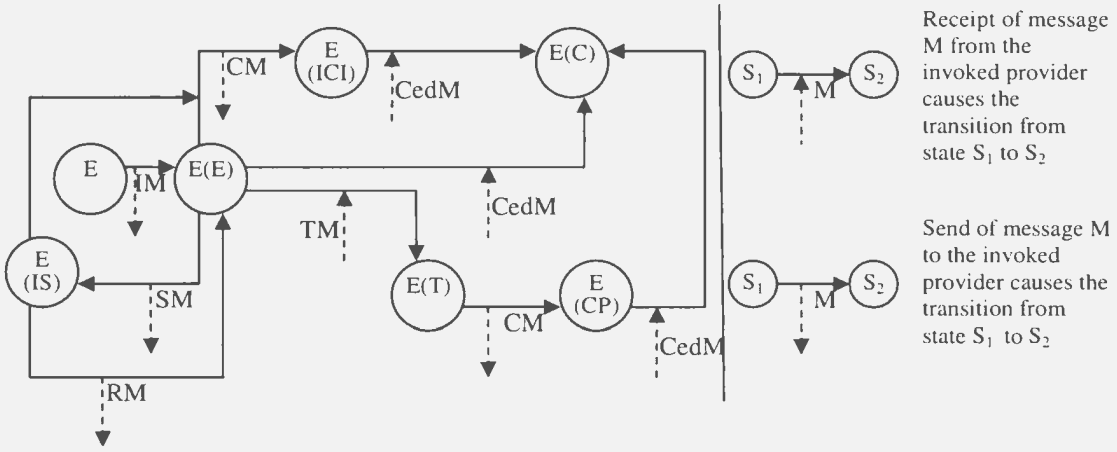

Figure 5.3 Invoking provider lifecycle with respect to one invoked action

The allowed state transitions are summarized in Tables 1, 2 and 3. We assume that the composition schema (static composition) specifies a partial order for the actions invoked by a provider. We define the happened-before relation between the actions invoked by a provider as follows:

An action a happened-before action b invoked by the same provider (a --> b) iff one of the following holds: (1) There exists a control/data dependency between actions a and b such that a needs to terminate before b can start executing. (2) There exists an action c invoked by the same provider such that a --> c and c --> b.

The term "partial order" in literature is synonymous with the partial ordering between events in a distributed system as determined by Lamport's happened-before relation [L78]. Lamport's happened-before relation between events assumes that local events are totally ordered. In comparison, the composition schema defines a partial order on the actions invoked by a provider (local actions).

Table 1 Allowed states of the invoked action (at the invoking provider site) with respect to the state of the invoking action

State of the invoking action	State of the invoked action (at the invoking provider site)
E/S/IS	E, T, ICI, CP, C, IS
CI/ICI	ICI, CP, C
CP	CP, C
T	T
C	C

Table 2 Allowed states of the invoked action (at the invoked provider site) with respect to the state of the invoking action

State of the invoking action	State of the invoked action (at the invoked provider site)
E/CI/ICI/S/IS	E, T, CI, ICI, CP, C, IS, S
T/CP	T, CP, C
C	C

Table 3 Allowed states of the invoked action at the invoked provider site with respect to its state at the invoking provider site

State of the invoked action (at the invoking provider site)	State of the invoked action (at the invoked provider site)
E/IS	E, T, CI, C, IS, S
ICI	E, T, CI, ICI, CP, C, IS, S
CP	T, CP, C
C	C
T	T

Note that a particular invocation (of a provider) may have already terminated by the time execution details of that invocation are required for reporting the state, answering execution status related queries, etc. To accommodate the above scenario, we assume that the log entries are persistent, i.e., execution details of a particular invocation are available even after it has terminated. Finally, to accommodate asynchronous communication, we assume the presence of Input/Output (I/O) queues. Basically, each provider has an I/O queue with respect to its parent and children (as shown in Fig. 5.1). The Input (Output) queue of a provider p corresponding to provider q is referred to as I_{pq} (O_{pq}). Although, each provider might only have a single I/O queue in a practical implementation, it really does not matter for the rest of discussion as long as there is some way of differentiating between messages sent by different providers. We assume that the status of the I/O queues and logs are updated in an atomic manner. With respect to Fig. 5.2 and Fig. 5.3, for any message M whose send (receipt) causes the provider to change its state, the details of the state transition are written to the log and M inserted into (deleted from) the Output (Input) queue in an atomic fashion.

For example, as soon as an action terminates, the state of the action in the log is updated (to T) and the corresponding message TM inserted into the Output queue atomically.

5.2 Synchronized Clock - Snapshot

In this section, we discuss snapshot algorithms based on the assumption that the clocks of the providers are synchronized using one of the techniques discussed earlier [L78], [M89], [M91]. Please note that the above approaches require considerable coordination among the participants (providers), which may not always be possible in a Web services scenario due to the autonomy requirements of the providers. A more loosely coupled approach for clock synchronization is the use of a timestamp element as advocated by WS standards like WS-Security [WS-S] for SOAP messages. Basically, a timestamp element consists of the creation time and transmission delays. Given this, we can calculate the skew (drift) between the invoking and invoked providers' clocks as follows: *skew = receiver's processing time - sender's creation time - transmission delay*. The transmission delay, in this case, corresponds to the time spent waiting in the I/O queues before the message is processed. Although the synchronization achieved with message timestamps may not be as accurate as with NTP (Network Time Protocol), we believe that it would still be acceptable for Web services compositions given their long-lived nature.

Given synchronized clocks and logging (as discussed earlier), a snapshot of the hierarchical composition at time t would consist of the logs of all the "relevant" providers until time t (provider P's log until time t is hereafter referred to as log_{Pt}). The relevant providers can be determined in a recursive manner (starting from the root provider) by

considering the providers of the invoked actions recorded in the parent provider's log until time t. In case clock synchronization protocols like NTP are not possible and message timestamps are used, then we need to consider the skew while recording the logs. Basically, if a parent provider's log is recorded until time t, its children providers' logs need to be recorded until (t + skew). The states of the I/O queues can be determined as follows. Initially, we consider that all the queues are EMPTY.

For each invoking (P) - invoked (Q) pair of providers until time t, if for an action a_t (*the last entry for action a_t in log_{Pt} denotes its state at t*)

- log_{Qt} does not contain an entry for a_t while its state is denoted as E/IS/ICI in log_{Pt} then add IM/SM/CM corresponding to a_t to I_{QP}.
- log_{Pt} denotes the state of a_t as E while log_{Qt} denotes its state as T/C then add TM/CedM corresponding to a_t to I_{PQ}.
- log_{Qt} denotes the state of a_t as E/IS/S while its state is denoted as ICI in log_{Pt} then add CM corresponding to a_t to I_{QP}.
- log_{Qt} denotes the state of a_t as T while its state is denoted as CP in log_{Pt} then add CM corresponding to a_t to I_{QP}.
- log_{Pt} denotes the state of a_t as ICI/CP while log_{Qt} denotes its state as C then add CedM corresponding to a_t to I_{PQ}.
- log_{Qt} denotes the state of a_t as E while its state is denoted as IS in log_{Pt} then add SM corresponding to a_t to I_{QP}.
- log_{Qt} denotes the state of a_t as IS while its state is denoted as E in log_{Pt} then add RM corresponding to a_t to I_{QP}.

Please note that the above list is not exhaustive as it does not consider some of the more implementation dependent scenarios. For example, if log_{Pt} denotes the state of a_t as IS while log_{Qt} denotes its state as S then whether Q changes its state to IS (and waits for RM from P) or it simply ignores the SM (and the corresponding RM from P) is implementation dependent and we would not like to impose any constraints on the same.

5.3 Distributed Snapshot Algorithm for Web services (DSW)

In this section, we do not assume synchronized clocks and outline an extension of the DSA to capture the state of the composition.

Distributed Snapshot algorithm for Web services (DSW):

Assumption: The I/O queues maintain the FIFO order of the messages. For example, if provider P inserts a message m_1 before message m_2 in its Output queue corresponding to Q, then Q receives m_1 before m_2.

The algorithm is initiated by the root provider, which atomically records its state (as of the time of recording) and sends markers to its children providers. By recording its state at time t, we mean that a provider records the contents of its local log at t, i.e., its execution history until t.

Child providers, on receiving the markers, do the same, i.e., atomically record their states (as of the time of recording) and send markers to their children providers. This downward propagation of the markers continues until leaf providers are reached.

The states of the I/O queues are computed as outlined for the synchronized clock scenario.

The above algorithm differs from the original DSA as follows:

- In DSA, markers are sent along all the outgoing channels. Basically, DSA assumes that the network topology is static (fixed in advance). With Web services compositions, due to dynamic binding, a provider at any point of time is only aware of the providers of the actions it has invoked until then. A provider may invoke other providers after it has recorded its state. Thus, the set of providers, whose states are recorded, may vary from one snapshot to the next.

- Our algorithm does not require the providers to record the states of their I/O queues explicitly. The contents of the I/O queues can be determined from the local states of the providers as discussed earlier.

Correctness:

As with the DSA, here also we show that the above algorithm captures a state of the hierarchical composition which "might have happened" (is consistent with the state transitions discussed earlier - Tables 1, 2 and 3). More precisely, we show that the recorded states preserve the causality of the messages sent/received, i.e., if the reception of a message is recorded then its transmission has also been recorded.

Intuitively, the proof follows from the fact that messages are exchanged only between parent-child providers and that the state of a parent is always recorded before any of its children. Thus,

- for messages recorded as received by any parent: If the receive event is recorded, then its corresponding send event (by the child) will also get recorded as the state of the child is recorded later.

- for messages recorded as received by any child: The FIFO nature of the I/O queues ensures that the parent sent the corresponding message before sending the marker. And, since the recording of state and sending of markers is done in an atomic fashion, the corresponding send event would have been recorded by the parent.

5.4 Actual State of the Composition

The snapshot acquired by the DSW highlights a global state of the composition which "might have happened". For example let us consider a single-level composition. Now the algorithm might record the states of the providers in the composition as shown in Fig. 5.4 namely, that actions a_1, a_{11} and a_{12} are all executing. However the execution might have happened as shown in Fig. 5.5 where action a_{11} had terminated before a_{12} started executing, i.e., actions a_{11} and a_{12} were never executing simultaneously. On the other hand, the global state as shown in Fig. 5.4 might have actually occurred too. In the absence of a global observer it is impossible to deduce whether a recorded global state actually occurred or not.

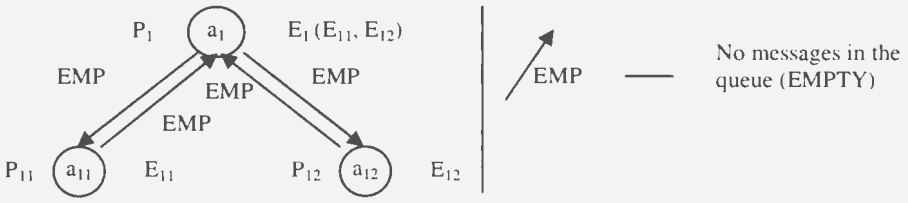

Figure 5.4 Sample Snapshot showing "what might have happened"

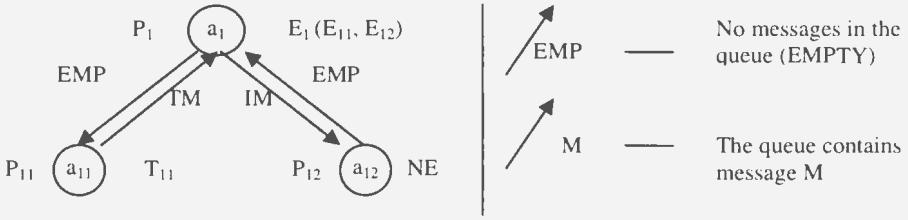

Figure 5.5 Execution showing what "might have actually happened"

Let P_S denote the set of providers whose states were recorded as part of a DSW snapshot S. For a pair of parent-child providers in P_S, if the state of the parent was recorded at time t then the child's state would have been recorded at a later time, say (t + L). Thus the providers in P_S may never have been in their recorded states simultaneously. We can still infer the following about the states of the providers in P_S at t based on the state of a provider P in P_S recorded at t.

- *Observation 1*: The presence of an action a_t in the recorded state of P implies that all the actions having a happened-before relation with a_t have terminated by t (their states are T at t).
- *Observation 2*: If the recorded state of P is E/S/IS/CI/ICI then its ancestors cannot be in the states T/CP/C at t.
- *Observation 3*: If the recorded state of P is T(C) then all the providers in the sub-tree rooted at P are in the state T (C) at t.
- *Observation 4*: If the recorded state of P is CP then all the providers in the sub-tree rooted at P are in the state CP (or would be at a later time).

In the rest of the section, we use the above observations to acquire a state of the composition which actually occurred. We define an actual state of the composition as follows:

A global state represents the actual state of the composition at time t if it reflects the states at t of all and only those providers invoked until t.

The concept of actual states is similar to the notion of Strongly Consistent Global States (SCGS) [B04a] in literature. While [B04a] defines SCGS in terms of the local states of all the

providers in the system, we define the actual state at a time t in terms of the local states of the providers invoked until t (due to dynamic binding). Algorithms to detect SCGS in [B04a] are based on real-time timestamps (similar to our algorithm based on synchronized clocks).

Given a (DSW) snapshot S initiated at time t, we can acquire an actual state of the composition at some point t_p in the past ($t_p <= t$) as given below.

Algorithm 1:

/* Intuitively, we can simulate "freezing the execution" if we can determine a time t_p at which none of the providers invoked until t_p are executing, i.e., they are in the state T/C at t_p. Thus, the algorithm tries to determine the latest set of providers which have definitely been canceled or terminated until t. The algorithm achieves this by determining the most recent time t_p when all the invoked providers at the root level are in the state T/C (implying all the lower level providers invoked until t_p are also in the state T/C - by Observation 3). We illustrate the steps with the help of an example scenario (Fig. 5.6 and 5.7). Fig. 5.6 depicts a sample snapshot (only shows the current states of the recorded actions) and Fig. 5.7 shows the recorded state of the root provider (contents of its log until t).*/

1. Let P_{IN} denote the set of invoked providers in the recorded state of the root provider. If the state of each provider P in P_{IN} is either C or T, then terminate the algorithm /* the recorded snapshot represents an actual state of the composition */.

2. From the recorded state of the root provider, determine the last provider P_l in P_{IN} to terminate/cancel before the invocation of the first provider P_f in P_{IN} which is still executing (in state E/IS/ICI) /* the "last" part (of the above statement) helps us in acquiring the latest set while the "first" part ensures that the acquired set consists only of terminated/canceled

providers. $P_{IN} = \{P_B, P_C, P_D, P_E, P_F\}$, $P_l = P_E$ and $P_f = P_F$ */. *Given this, t_p corresponds to the time just after P_l terminated/canceled.*

/* The following steps determine the providers invoked (at all levels) until t_p and their states at t_p. Recall that the recorded states of the providers reflect their states at a later time t. As discussed earlier, the states of all the providers (invoked until t_p) would be C/T at t_p. A small complication arises due to the possibility of compensation. A provider which was in state T at time t_p may have been compensated before t (after t_p) leading to its state being recorded as C. For example, the recorded state of the provider P_E is C (Fig. 5.6). However, from the log (Fig. 5.7) it is clear that the state of P_E was T at t_p. As such, we may need to adjust the recorded states of some of the providers (invoked until t_p) so that they reflect their states at time t_p. */

3. Let S_{AP} denote the set of invoked providers (at all levels) until t_p. Initially, S_{AP} = Root provider.

4. Adjust the recorded state of the root provider so that it reflects its state at t_p (contents of its log until t_p- Fig. 5.7). Use the newly adjusted state to determine the set of providers P_{TP} invoked by the root provider until t_p and their states at t_p. Adjust the recoded states of providers in P_{TP} accordingly (if required). Add the providers in P_{TP} to S_{AP} i.e. $S_{AP} = S_{AP} \cup P_{TP}$ /* the adjusted recorded state of the root provider denotes the state of P_E as T, so adjust the recorded state of P_E accordingly (trim its log until t_p - Fig. 5.8). */.

5. Repeat Step 4 recursively for each provider in P_{TP} (determined at each stage) until leaf providers are reached. /* $S_{AP} = \{P_A, P_B, P_C, P_D, P_E, P_G, P_H, P_I, P_J, P_K, P_L, P_M, P_N, P_P, P_Q, P_R, P_S, P_T\}$ */

The global state G, consisting of the recorded states of the providers in S_{AP}, represents the actual state of the composition at time t_p (just after the provider P_I terminated/canceled) - Fig. 5.9.

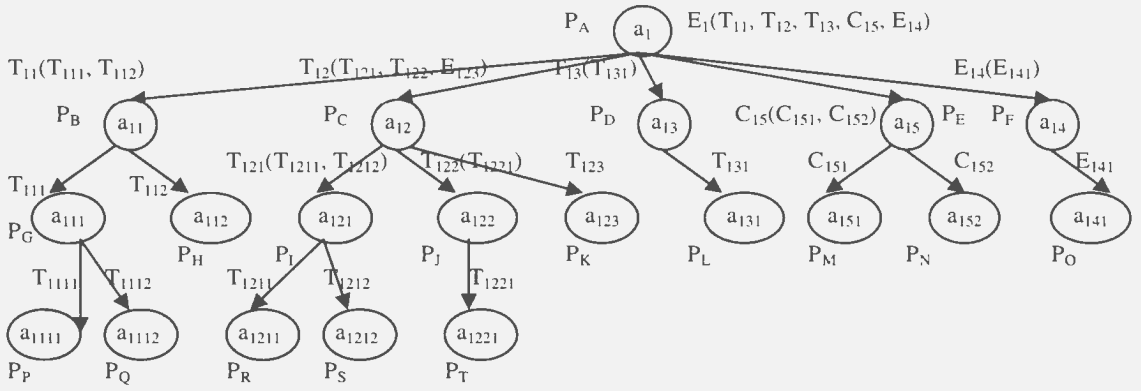

Figure 5.6 Sample Snapshot

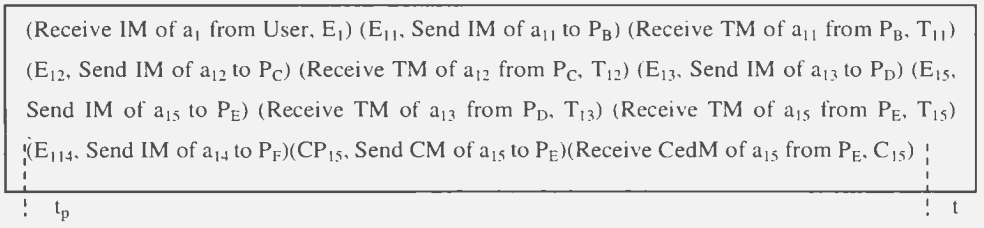

Figure 5.7 Recorded (t) and adjusted (t_p) state of the root provider (Algorithm 1)

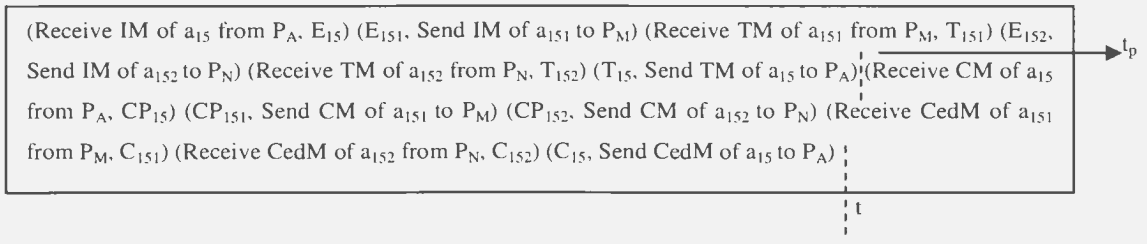

Figure 5.8 Recorded (t) and adjusted (t_p) state of the provider P_E

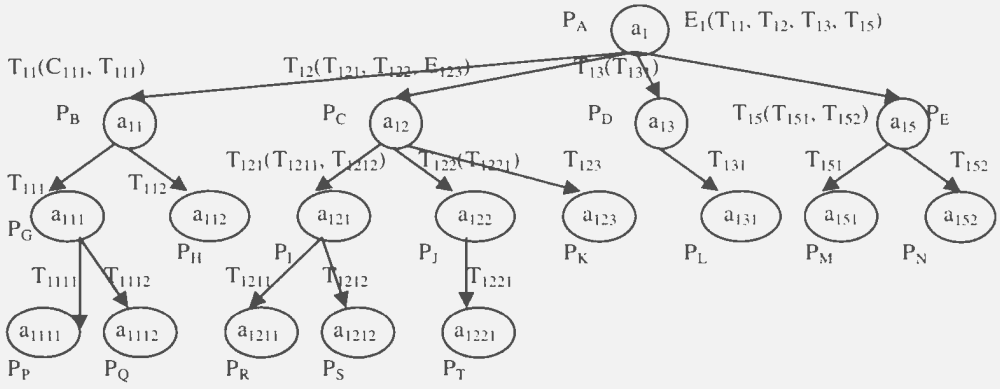

Figure 5.9 Actual state corresponding to the snapshot in Fig. 5.6 (Algorithm 1)

It is easy to observe that the above mechanism can be used to acquire an actual state for any sub-tree belonging to a snapshot acquired by the DSW. For example, if we add another level of nesting to the composition in Fig. 5.6 (Fig. 5.10) then the above mechanism can be used to acquire an actual state for the sub-tree rooted at provider P_A.

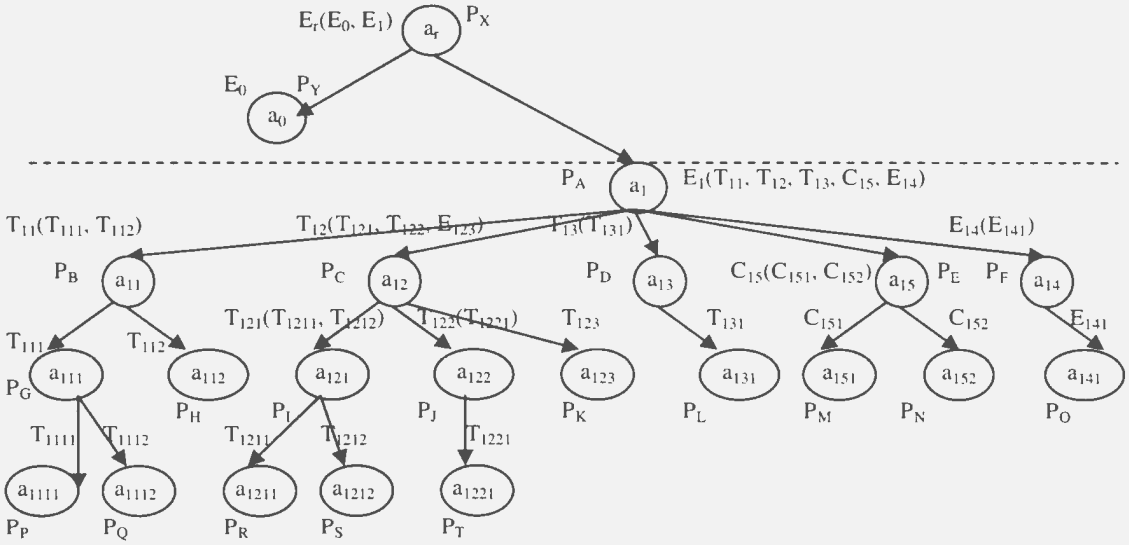

Figure 5.10 Acquiring actual states of sub-trees

An obvious limitation of the above algorithm is that the states of all the providers (except the root provider) in any actual state acquired by the algorithm would always be T/C. To understand the problem in capturing the state of a provider as E, we revisit the single level composition scenario considered in Fig. 5.4. Let us assume that the states of the providers P_1, P_{11} and P_{12} are recorded by the DSW at times t_1, t_2 and t_3 respectively. We know for sure that t_2 and t_3 occurred after t_1. It is easy to see that the above information is not sufficient to determine the actual state of the composition at t_1, t_2 or t_3. Fig. 5.4 reflects the difficulty with respect to a scenario where the states of more than one invoked provider are recorded as E by the invoking provider. As such, let us restrict the scenario further so that the state of only one child is recorded as E (Fig. 5.11). With this restriction, let us try to determine the actual state of the composition at t_3. We can say that provider P_{11} would be in its recorded state T at t_3 if there exists a happened-before relation between a_{11} and a_{12}. On the same lines, the recorded state of P_1 would change between t_1 and t_3 iff one of the following occurs: (1) an already invoked action of P_1 changes its state or (2) P_1 invokes a new action a_{13}. The above two scenarios won't occur (P_1 would be in its recorded state at t_3) if for each action a_t ($a_t \neq a_{12}$) belonging to the composition schema either a_t --> a_{12} (all actions invoked before a_{12} have terminated) or a_{12} --> a_t (no new actions can be invoked before a_{12} terminates). Extending the above scenario to a hierarchical composition, *we can simulate "freezing the execution" if a_{12} belonged to the lowest level of a recorded snapshot* (Fig. 5.12). When a_{12} started executing, its ancestor (including parent) actions (a_R and a_1) and their corresponding providers (P_R and P_1) would also have been executing (Observation 2), i.e., we have an executing action (provider) at each level. However (as explained above), the executing ancestor actions

(providers) at each level cannot change their states until a_{12} (P_{12}) terminates. With reference to Fig. 5.12, P_1 cannot changes its state until P_{12} terminates. All the providers (P_{11}) invoked by P_1 before P_{12} have already terminated and it cannot make any new invocations until P_{12} terminates. On the same lines, P_R cannot change its state until P_1 (and recursively, P_{12}) terminates. We use the above logic in the algorithm given below to acquire an actual state of the composition at time t_p' given a (DSW) snapshot S initiated at time t ($t >= t_p$'):

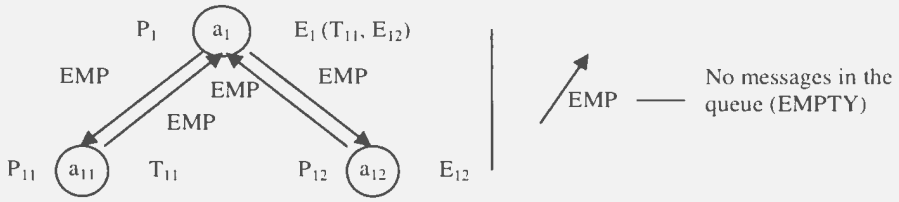

Figure 5.11 Sample Snapshot

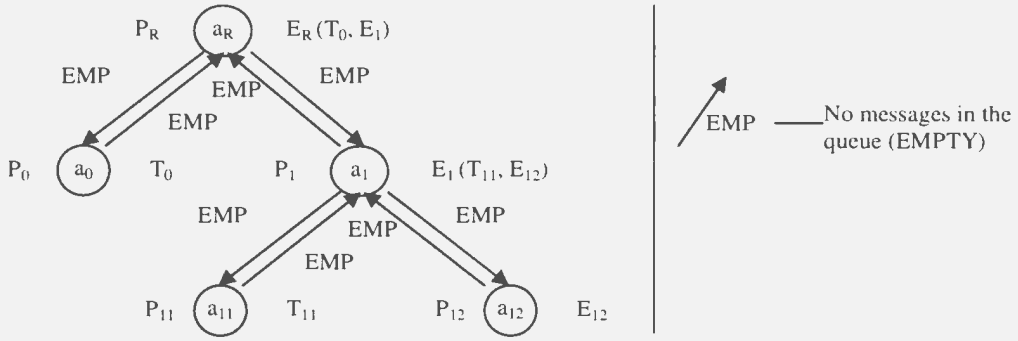

Figure 5.12 Hierarchical extension of the sample snapshot in Fig. 5.11

Algorithm 2:

/* We illustrate the steps with the help of the earlier scenario (Fig. 5.6 and 5.7). */

1. Let S_{AP} denote the set of invoked providers (at all levels) until t_p'. Initially, S_{AP} = Root provider.

2. Let CS_r denote the projection of the composition schema with respect to the root provider. Determine the last action a_l invoked by the root provider to terminate such that $\nexists a_t$: $a_t \varepsilon CS_r$ and $a_t \neq a_l$: a_t --> a_l or a_l --> a_t. Let P_{lt} be the provider invoked to execute a_l. Adjust the recorded state of the root provider so that it reflects the state when P_{lt} was executing. $S_{AP} = S_{AP} \cup P_{lt}$ /* Fig. 5.13 shows part of the composition schema with respect to the root provider. Thus, $a_l = a_{12}$, $P_{lt} = P_C$. Adjust the recorded state of the root provider so that it reflects the state when P_C was executing, i.e., (E_{12}, Send IM of a_{12} to P_C) is the last entry in its log - Fig. 5.14. */.

3. Repeat Step 2 for the provider P_{lt} (instead of the root provider). Keep repeating Step 2 recursively for the provider P_{lt} (determined at each stage) until leaf providers are reached. /* Let us assume that P_C is sequential, i.e., a_{121} --> a_{122} --> a_{123}. Given this, $a_l = a_{123}$, $P_{lt} = P_K$ and adjust the recorded state of P_C such that (E_{123}, Send IM of a_{123} to P_K) is the last entry in its log. */

4. The steps for determining the providers invoked before P_{lt} at each level, appending them to S_{AP} and performing any required adjustments to their recorded states (due to compensation) is similar to Algorithm 1 and as such have been skipped. /* Please note that the states of the providers invoked before P_{lt} at each level would be C/T.*/

The global state G, consisting of the recorded states of the providers in S_{AP}, represents the actual state of the hierarchical composition at time t_p' (Fig. 5.15), where t_p' corresponds to the time when the provider P_{lt} (with respect to the lowest level) started executing. With reference to the example scenario (Fig. 5.6), t_p' corresponds to the time when provider P_K started executing.

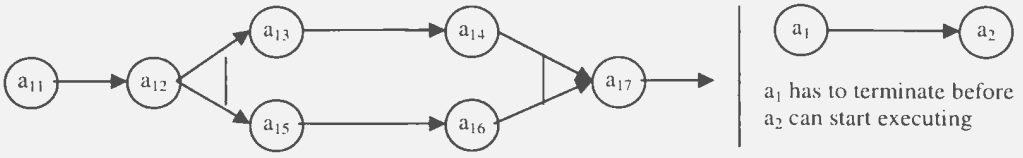

Figure 5.13 Part of the composition schema with respect to the root provider

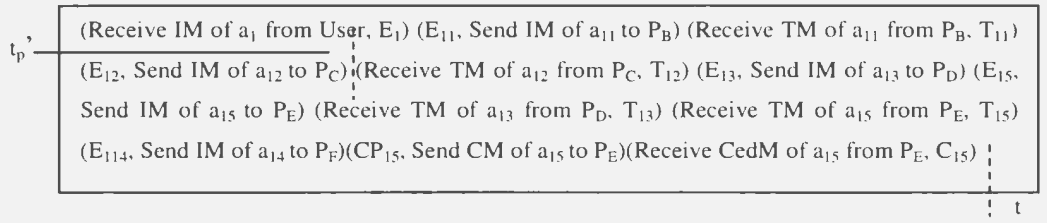

Figure 5.14 Recorded (t) and adjusted (t_p') state of the root provider (Algorithm 2)

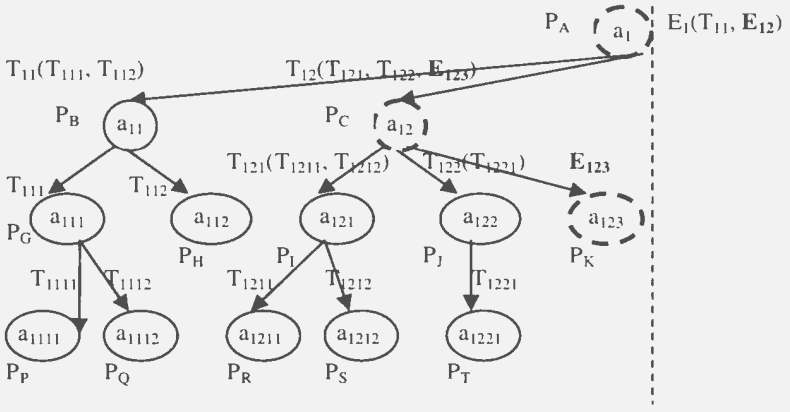

Figure 5.15 Actual state corresponding to the snapshot in Fig. 5.6 (Algorithm 2)

Now, there exists an exception scenario where the above algorithm is not guaranteed to return an actual state. Algorithm 2 is based on Observation 2, i.e., if a provider is in state E at time t then its ancestors cannot be in the states T/CP/C at t (they are also executing at t). Given this, the complication arises due to the possibility of suspension. Let us consider the problem with reference to the actual state in Fig. 5.15. Let the providers P_A, P_C and P_K

(executing providers at each level) be in state E between the time intervals (t_{A1} - t_{A2}), (t_{C1} - t_{C2}) and (t_{K1} - t_{K2}) respectively such that $t_{A1} < t_{C1} < t_{K1} < t_{K2} < t_{C2} < t_{A2}$. Given this, Algorithm 2 returns the actual state at time t_p' (just after P_K started executing), i.e., $t_{K1} < t_p' < t_{K2}$. Now, let us assume that provider P_A was in the state S during the entire interval (t_{K1} - t_{K2}) as shown in Fig. 5.16. Given this scenario, the state acquired by Algorithm 2 does not represent the actual state of the composition at t_p' (P_A was in state S at t_p'). Again, it is not possible to determine if such a scenario (as described above) actually occurred or not. The best we can do is to examine the contents of the log (recorded state) of the executing provider at each level to determine if it was ever suspended (in state S/IS). If so, we conclude that Algorithm 2 is not guaranteed to return an actual state and Algorithm 1 is the only option.

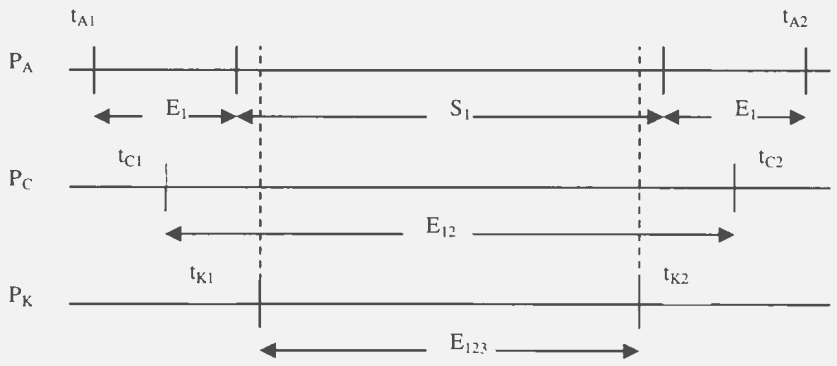

Figure 5.16 Exception scenario for the actual state in Fig. 5.15

5.5 Taking Stock of the Situation

We started this chapter with the objective of providing information about the current (intermediate) state of execution of a hierarchical composition. Towards that end we have outlined three algorithms to capture a snapshot of the ongoing execution: (1) Algorithm

based on the assumption of synchronized clocks (synchronized algorithm), (2) DSW and (3) Algorithms to acquire an actual state of the composition from a DSW snapshot (actual state algorithms). Providing intermediate information encapsulates two aspects:

- Reporting the state: The snapshots acquired by both the synchronized and actual state algorithms can be used to report the current state of the execution. It is easy to see that a synchronized snapshot also represents an actual state of the composition. Please recall that a snapshot acquired by the actual state algorithms represents the state of the composition at some point in the past. While a synchronized snapshot answers the question "what is currently going on", a snapshot acquired by the actual state algorithms provides information about "what has happened".
- Answering state related queries: Sometimes a diagram representing the state of the entire composition may contain too much information for the end-user to comprehend. As such, it should be possible to answer specific queries related to the state of execution. We discuss the capabilities and limitations of the different snapshots (acquired using the algorithms mentioned earlier) with respect to answering different types of queries. We use [CHK01] as a basis while determining the set of execution related queries which a hierarchical system might be required to answer. [CHK01] discusses a query model (WSQM) for workflows specified and enacted using XML. We divide the queries into the following categories:

1. Local queries: Queries which can be answered based on the local state information of a provider. For example, queries such as "What is the current state of provider P?" or "Has provider P reached a specific state?". As obvious, we do not need a snapshot of the

composition to answer such queries. Local queries can be answered by directly querying the concerned provider as long as it provides a query interface such as WSDM [WSDM].

2. Status queries: Queries expressed over the states of several providers (complex queries). We assume that any query related to the status of a composition is expressed as a conjunction of the states of individual providers. We do not consider disjunction for the simple reason that any disjunctive query can be expressed as a local query. Examples of status queries: "Have providers A, B and C reached states T, E and E respectively?", "Have providers A, B, C and D started executing?". It is easy to see why such queries cannot be answered by querying the concerned providers separately. Status queries can be answered using snapshots acquired by any of the above given algorithms. Such queries have been referred to as stable predicates in literature. *Stable predicates are defined as predicates which do not become false once they have become true.* To check for a stable predicate, we simply need to keep taking snapshots regularly until the predicate holds. Common examples of stable predicates are deadlock and termination. In our case, the stability of the query is reflected by the fact that we do not have to capture the state of a provider as E to conclude that it has started executing. We can infer the same even if the state of the provider is recorded as T/CI/ICI/CP/C/IS/S.

3. History queries: Queries related to the execution history of the composition. For example, "How many times have A and B been suspended?" or "How many providers have been canceled until now?". Both synchronized and actual state snapshots can be used to answer execution history related queries. A DSW snapshot cannot be used directly because the recorded states of the providers reflect their states at different times. If the query is

answered using a snapshot acquired by the actual state algorithms then it needs to be mentioned that the statistics are with respect to a time t_p in the past.

4. Relationship queries: Queries based on the relationship between states. For example, "What was the state of provider A when provider B was in E" or "Did provider A start executing before provider B?". Unfortunately snapshot based mechanisms do not guarantee answers for such queries. For example we would not be able to answer the first query unless we have a snapshot which captures the state of provider B when it was in state E. B could have been in state NE when a snapshot was taken and in state T when the next snapshot was taken. Such predicates have been referred to as unstable predicates in literature. *Unstable predicates keep alternating their values between true and false.* As discussed above if a predicate becomes true between successive snapshots it won't get detected. While unstable predicates are in general very difficult to detect, researchers have studied some special classes of unstable properties:

(a) Strong unstable predicates [GW96] or predicates which will "definitely" hold [CM91]: A predicate is called a strong predicate iff the global state over which it holds is guaranteed to occur for any execution (irrespective of the execution speeds, communication delays and other variable parameters in a distributed setting). Intuitively, strong unstable predicates allow us to verify that a desirable state will always occur.

(b) Strong unstable linked predicates [GW96], [MC88]: A linked predicate is expressed as a sequence of local predicates and is called a strong linked predicate if the corresponding local states occur in the same sequence for every execution. Such predicates are useful in detecting a sequence of states in a distributed setting.

In a hierarchical composition scenario, the concepts of strong and linked predicates can be used to answer relationship queries as long as there exists a parent-child (ancestor-descendant) relationship between the concerned providers. For example, a global state where parent and child providers are executing simultaneously will definitely occur or an ancestor will always starting executing before any of its descendants. Researchers have also considered weak unstable [GW94] or "possibly" true predicates [CM91] and predicates defined over atomic sequences [HPR93]. We point the reader to [SM92] for a survey of the unstable predicates studied in literature.

Finally we consider some parameters which allow us to get a rough estimate of the accuracy of a DSW snapshot:

- The time taken to record the snapshot (duration): It is easy to see that the accuracy of the algorithm decreases as the duration increases. The duration of the algorithm depends on the communication delay between the providers (at different levels) and the number of (nesting) levels. Assuming the communication delay between providers is specified as part of the SLA and we can get the number of (nesting) levels at any point of time from the composition schema, it is possible to estimate the duration of the algorithm. In fact, given the long running nature of complex business process, if the duration of the algorithm is very small (in secs) as compared to the total execution time of the business process (in days), it might even be practical to actually freeze the execution.
- Degree of concurrency: This factor arises from the fact that it is possible to get an "actual" snapshot if all the involved providers are sequential. In fact, the higher the concurrency the lower the accuracy of the algorithm. Given the composition schema, we can

compute the degree of concurrency in terms of the number of actions capable of running concurrently. As mentioned before, two actions are capable of executing concurrently if they do not have a happened-before relation between them.

5.6 Customizations and Optimizations

DSW captures the state of the hierarchical composition until the lowest level (leaf providers are reached). The algorithm can be customized as follows to record the state of the composition up to a certain granularity:

- Capture the state up to level n: Append a counter with the marker. Each child provider, on receiving the marker, increments the counter by 1 and forwards it to its children (if any) if the value of the counter <= n.
- Capture the state until a certain condition holds: The condition may be time based (for example, capture the state of as many providers as possible within a time frame) or any predicate which can be evaluated locally. Similar to the above case, we can accommodate this requirement by appending the predicate to the marker. Each child provider, on receiving the marker, evaluates the predicate and forwards it to its children if the predicate holds.

Snapshot algorithms are primarily used to capture an intermediate state of the execution. As such there might be a need to run it multiple times for the same execution. However we cannot apply the idea of incremental snapshots [V89] directly here. A provider cannot decide to forward the marker to only those children to (from) which it has sent (received) messages after the last snapshot. Although the provider may not have exchanged any messages with its children since the last snapshot, the state of the lower level providers may have changed.

Exceptions include scenarios where the children providers have either terminated or canceled. While lower level terminated child providers may be canceled (compensation), it would involve the send of a CM (a provider cannot decide to compensate itself).

Even without the idea of incremental snapshots, it may not be required to traverse the entire hierarchy for each snapshot. A provider P may proactively take snapshots of the sub-tree rooted at P and return the same whenever it receives a marker from its parent. To preserve consistency, the above should be allowed only if the recorded states of P and its parent are consistent with the earlier state transition discussion (Tables 1, 2 and 3). For example, if the state of an invoked action a_i is recorded as T at the invoking provider's site (P's parent) then a_i's state should be T at the invoked provider's (P's) site too. In case P does not have visibility over the entire sub-tree rooted at P, it can still take snapshots of the providers in its downward visibility (Down-SoV_P).

Chapter 6

Conclusion

The major contribution of this thesis is with respect to the visibility, compensation and monitoring aspects of hierarchical Web services compositions. Web services compositions are characterized by their "loosely coupled" nature. For a single-level composition, it translates to the composite provider not having visibility over the internal (processing) logic of the primitive providers. For a hierarchical composition, it translates to a provider not having visibility over the other providers in the composition. We introduced the notion of Spheres of Visibility (SoV) to capture the visibility aspects of a hierarchical composition. We also provided implementation details related to populating the SoV's of the involved service providers. While the concept of compensation has been around for quite some time, new challenges arise when we try to apply the same to hierarchical Web services compositions. Most of the transaction based compensation models do not acknowledge the fact that compensation may be possible at different levels in a hierarchical composition with different costs (CoC). The dynamic aspect of Web services compositions, esp. dynamic binding, also allows for the possibility of provider independent compensation. Earlier works have considered end-user involvement as a last resort for unhandled (or ad-hoc) faults. We stressed that end-user involvement may be required for selecting the most optimum compensation option too. We showed how the above aspects could be implemented in a hierarchical Web services composition with visibility restrictions (modeled as SoV). On the monitoring front, we outlined algorithms to capture the state of a hierarchical Web services

composition. More precisely, we discussed how the Distributed Snapshots Algorithm (DSA), algorithms based on the assumption of synchronized clocks and incremental snapshots algorithm could be adapted to capture the state of a hierarchical Web services composition. Next, we showed how we can acquire an "actual" state of the composition from such snapshots. Finally, we discussed using the captured state information to answer execution status related queries.

Some aspects which we would like to explore in the future are as follows:

- In this work, we showed how compensation and end-user involvement can be achieved in a hierarchical composition taking the visibility aspect of the providers in consideration. In the future, we would like to consider how some of the other compositional aspects especially security can be implemented in conjunction with SoV.
- We would like to extend the monitoring algorithms to consider failure detection including deadlock, livelock, etc.
- We would like to consider the notion of "constraints" for hierarchical compositions. Constraints are used to specify the functional and non functional limitations of an action. For example, a painting service might be available only for a particular paint type/color (functional) and cost/duration (non functional). The challenge arises when we try to reason about the constraints of a composite service based on the constraints of its component services (constraint composition).
- The aspects discussed in this thesis assume a static composition and dynamic binding environment. It would be interesting to try and apply the same to fully dynamic compositions.

Bibliography

[AH00] Gustavo Alonso, Claus Hagen. Exception Handling in Workflow Management Systems. IEEE Transactions on Software Engineering, vol. 26, no. 10, pp. 943-958, Oct 00.

[B04a] Janusz Borkowski. Hierarchical Detection of Strongly Consistent Global States. In Proceedings of the 3rd International Symposium on Parallel and Distributed Computing, 2004, pp:256-261.

[B04b] Laura Bocchi. Compositional Nested Long Running Transactions. In Proceedings of the Fundamental Approaches to Software Engineering 2004, LNCS 2984, pp:194--208.

[BPEL] Specification: Business Process Execution Language for Web Services (BPEL4WS). http://www-106.ibm.com/developerworks/library/ws-bpel/.

[CD96] Qiming Chen, Umeshwar Dayal. A Transactional Nested Process Management System. In Proceedings of the 12th International Conference on Data Engineering, 1996, pp. 566-573.

[CHK01] V. Christophides, R. Hull, A. Kumar. Querying and Splicing of XML Workflows. In Proceedings of the 6th International Conference on Cooperative Information Systems (CoopIS 2001), 2001, pp:386-402.

[CIJKS00] Fabio Casati, Ski Ilnicki, LiJie Jin, Vasudev Krishnamoorthy, Ming-Chien Shan. Adaptive and Dynamic Service Composition in eFlow. HP Technical Report, HPL-2000-39, March, 2000.

[CL85] K. M. Chandy, L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. ACM Transactions on Computer Systems, 3(1):63--75, February 1985.

[CM91] R. Cooper, K. Marzullo. Consistent detection of global predicates. In Proceedings of the ACM/ONR Workshop on Parallel Distributed Debugging 1991, pp:163-173.

[D78] C. Davies, Jr. Data processing spheres of control. IBM Systems Journal, 17(2):179--198, 1978.

[GLC99] B. Grosof, Y. Labrou, H. Chan. A Declarative Approach to Business Rules in Contracts: Courteous Logic Programs in XML. In Proceedings of the 1st ACM Conf. on Electronic Commerce (EC-99), 1999, ACM Press (1999), pp:68-77.

[GMS87] H. Garcia-Molina and K. Salem. SAGAS. In Stonebraker, M. ed. Readings in database systems, San Francisco, California, 1987, 290-300.

[GW94] V.K. Garg, B. Waldecker. Detection of Weak Unstable Predicates in Distributed Programs. IEEE Transactions on Parallel and Distributed Systems, vol. 05, no. 3, pp. 299-307, March 1994.

[GW96] V.K. Garg, B.Waldecker. Detection of Strong Unstable Predicates in Distributed Programs. IEEE Trans. Parallel and Distributed Systems, Dec. 1996, pp. 1323-1333.

[HPR93] M.Hurfin, N.Plouzeau, M.Raynal. Detecting Atomic Sequences of Predicates in Distributed Computations. In Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, 1993, pp:32-42.

[JG03] Tao Jin, Steve Goschnick. Utilizing Web Services in an Agent Based Transaction Model (ABT). In Proceedings of the 1st International Workshop on Web Services and Agent based Engineering (WSABE'2003), 2003.

[KRS95] A.D. Kshemkalyani, M. Raynal, M. Singhal. An Introduction to Snapshot Algorithms in Distributed Computing. Distributed Systems Eng. J., vol. 2, no. 4, pp. 224-233, Dec. 1995.

[KS02] H. Kuno, A. Sahai. My Agent Wants to Talk to Your Service: Personalizing Web Services through Agents. HPL-2002-114.

[KS03] Randi Karlsen, Thomas Strandenæs. Trigger-Based Compensation in Web Service Environments. In Proceedings of the International Conference on Enterprise Information Systems 2003 (ICEIS03), pp:487-490.

[L78] L. Lamport. Time, Clocks and Ordering of Events in Distributed Systems. Comm. ACM, vol. 21, no. 7, pp. 558-565, July 1978.

[LAP03] A. Lazovik, M. Aiello, M. Papazoglou. Planning and Monitoring the Execution of Web Service Requests. In Proceedings of the 1st International Conference on Service-Oriented Computing (ICSOC'03), 2003, LNCS 2910, pp:335--350.

[LT89] N. A. Lynch, M. R. Tuttle. An Introduction to Input/Output Automata. CWI Quarterly 2, 3 (1989), 219 – 246.

[LTS98] Fu-Ren Lin, Gek Woo Tan, M.J. Shaw. Modeling Supply-Chain Networks by a Multi-Agent System. In Proceedings of the 31st International Conference on System Science, 1998, pp:105 - 114.

[M81] T.E.B. Moss. Nested Transactions: An Approach to Reliable Distributed Computing. Ph.D. Thesis, MIT Laboratory for Computer Science, 1981.

[M89] F. Mattern. Virtual time and global states of distributed systems. In Parallel and Distributed Algorithms, Elsevier Science Publishers B.V. (North-Holland), 1989, pp:215-226.

[M91] D. L. Mills. Internet time synchronization: the Network Time Protocol. IEEE Trans. Communications 39, 10 (October 1991), 1482--1493.

[M98] J. D. Moffet. Control principles and role hierarchies. In Proceedings of the 3rd ACM Workshop on Role-Based Access Control, 1998, pp:63-69.

[MC88] B.P. Miller, J.D. Choi. Breakpoints and Halting in Distributed Programs. In Proceedings of the 8th Int'l Conf. Distributed Computing Systems, 1988, CS Press, pp. 316-323.

[NM02] S. Narayanan, S. A. McIlraith. Simulation, Verification and Automated Composition of Web Services. In Proceedings of the 11th International World Wide Web Conference (WWW-11), 2002.

[OWL-S] Specification: Web Ontology Language for Services (OWL-S). http://www.daml.org/services/owl-s/.

[PBBST04] M. Pistore, P. Bertoli, F. Barbon, D. Shaparau, P. Traverso. Planning and Monitoring Web Service Composition. In Proceedings of the Workshop on Planning and Scheduling for Web and Grid Services, 2004.

[PMB02]Paulo F. Pires, Marta L.Q. Mattoso, Mário Roberto F. Benevides. Building Reliable Web Services Compositions. In Proceedings of the Web, Web-Services, and Database Systems 2002, LNCS 2593, pp:59-72.

[RKM04] Jinghai Rao, Peep Kungas, Mihhail Matskin. Logic Based Web Services Composition: From Service Description to Process Model. In Proceedings of the 2nd Intl. Conference on Web Services (ICWS 04), 2004, IEEE, pp:446-453.

[SM92] R. Schwartz, F. Mattem. Detecting causal relationships in distributed computations: In search of the holy grail. Tech. Rep. SFB124- 15/92, Univ. of Kaiserslautern, Germany, 1992.

[SY01] Munindar P. Singh, Pinar Yolum. Commitment Machines. Revised Papers from the 8th International Workshop on Intelligent Agents VIII, 2001, pp:235-247.

[TIRL03] Ferda Tartanoglu, Valérie Issarny, Alexander Romanovsky, Nicole Levy. Coordinated Forward Error Recovery for Composite Web Services. In Proceedings of the 22nd Symposium on Reliable Distributed Systems (SRDS'2003), 2003.

[UDDI] Speciation: Universal Description, Discovery and Integration (UDDI). http://www.uddi.org/specification.html.

[V89] S. Venkatesan. Message-optimal incremental snapshots. In Proceedings of the 9th International Conference on Distributed Computing Systems, 1989, pp:53--60.

[VV04] K. Vidyasankar, Gottfried Vossen. Multi-level Model for Web Service Composition. In Proceedings of the 2nd Intl. Conference on Web Services (ICWS 04), 2004, IEEE, pp: 462-471.

[WDSS93] G. Weikum, A. Deacon, W. Schaad, H.-J. Schek. Open Nested Transaction in Federated Database Systems. IEEE Data Engineering Bulletin, 16(2):4--7, June 1993.

[WPSHN03] Dan Wu, Bijan Parsia, Evren Sirin, James Hendler, Dana Nau. HTN planning for web service composition using SHOP2. Web Semantics, 1(4):377--396, 2004.

[WS-C] Specification: WS-Coordination (WS-C). http://msdn.microsoft.com/library/en-us/dnglobspec/html/ws-coordination.asp.

[WS-CAF] OASIS Web Services-Composite Application Framework (WS-CAF) Primer. http://www.webservices.org/index.php/article/articleview/1297/1/24/.

[WSDL] Specification: Web Services Description Language (WSDL). http://www.w3.org/2002/ws/desc/.

[WSDM] Specification: Web Services Distributed Management (WSDM). http://devresource.hp.com/drc/specifications/wsdm/index.jsp.

[WS-R] Specification: WS-Routing. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnglobspec/html/wsroutspecindex.asp.

[WS-S] WS Security Addendum. http://msdn.microsoft.com/library/en-us/dnglobspec/html/ws-security-addendum.asp.

[WS-T] Specification: WS-Transaction. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnglobspec/html/wstxspecindex.asp.

Appendix A

XML Schema for Provider Independent Compensation

In this section, we outline an XML schema which can be used to capture and store the information required for provider independent compensation. The XML schema follows:

```
<!-- Input name/value pairs required for compensation -->

<xs:complexType name = "inputType">
  <xs:sequence>
    <xs:element name = "inputParName" type = "xs:string"/>
    <xs:element name = "inputValue" type = "xs:string"/>
  </xs:sequence>
</xs:complexType>

<!-- An action element, basically, consists of four elements: the action name, its effects, input
values and the original service provider. We use the WS-Addressing EndpointReference
XML data type to store the provider details. -->

<xs:complexType name = "action">
  <xs:sequence>
    <xs:element name = "actionName" type = "xs:string"/>
    <xs:element name= "effects" type = "xs:string" maxOccurs = "unbounded"  minOccurs
      = "1"/>
    <xs:element name = "inputValues" type = "inputType" maxOccurs = "unbounded"
      minOccurs = "0"/>
    <xs:element name = "provider" type = "wsa:EndpointReference"/>
  </xs:sequence>
</xs:complexType>

<!-- A composite action consists of a partial order of actions (which may themselves be
composite). To accommodate this hierarchical composition, we need to be able to specify a
composite action within another composite action (in a recursive manner) -->

<xs:complexType name = "compositeAction">
  <xs:all>
```

```
    <!-- Composite action details -->
    <xs:element name = "compositeActionName" type = "xs:string" maxOccurs = "1"/>
    <xs:element name = "compositeEffects" type = "xs:string" maxOccurs = "unbounded"
      minOccurs = "1"/>
    <xs:element name = "compositeInputValues" type = "inputTypes" maxOccurs =
      "unbounded" minOccurs = "0"/>
    <xs:element name = "compositeProvider" type = "wsa:EndpointReference"/>
    <!-- Information about invoked actions -->
    <xs:element name = "invokedAction" type = "action" maxOccurs = "unbounded"
      minOccurs = "1"/>
   <!-- Recursive composite actions -->
   <xs:element name = "recSeq" type = "compositeAction" maxOccurs = "unbounded"
      minOccurs = "0"/>
    </xs:all>
</xs:complexType>
```

The figure below shows a hypothetical travel scenario which involves booking flight tickets from St. John's - Delhi and have them delivered to the customer. The letters in brackets () correspond to the actions in the figure. We assume that the travel agent (A) deals directly with airline (B), credit card (C) and courier (D) companies. Further, the airline company B uses another airline company G to provide for part of the journey (London-Delhi). Given this scenario, the figure shows the XML compensation information which would be maintained by the different service providers.

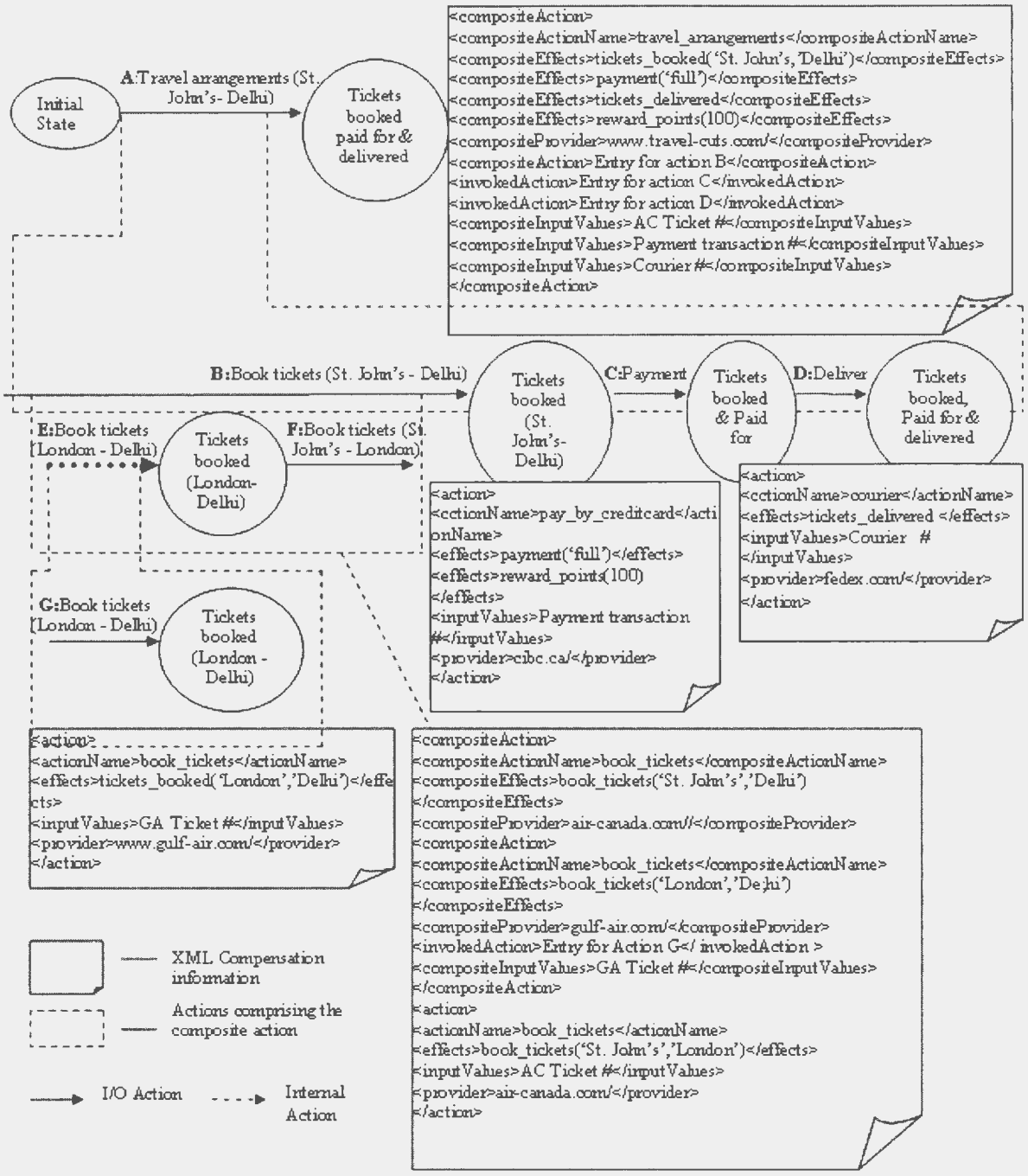

Figure A. 1 XML Compensation information maintained by the service providers at different levels of a hierarchical composition