TECHNICAL REPORT #2007-01

Multi-Level Modeling of Web Service Compositions with Transactional Properties*

by

K. Vidyasankar

In conjunction with:
Gottfried Vossen
Department of Information Systems
University of Muenster
Leonardo-Campus 3
D-48149 Muenster, Germany

Department of Computer Science
Memorial University of Newfoundland
St. John's, NF, Canada A1B 3X5

February 2007

Multi-Level Modeling of Web Service Compositions with Transactional Properties[*]

K. Vidyasankar[†]
Dept. of Computer Science
Memorial University
St. John's, Newfoundland
Canada, A1B 3X5

Gottfried Vossen[‡]
Dept. of Information Systems
University of Muenster
Leonardo-Campus 3
D-48149 Muenster, Germany

February 2007

Abstract

Web services have become popular in recent years as a vehicle for the design, integration, composition, and deployment of distributed and heterogeneous software. However, while industry standards for the description, composition, and orchestration of Web services have been under discussion (and development) for quite some time already, their conceptual underpinnings are still not well-understood. Indeed, conceptual models for service specification are rare so far, as are investigations based on them. This paper presents and studies a *multi-level service composition model* that perceives service specification as going through several levels of abstraction: It starts from transactional operations at the lowest level, and then abstracts into activities at higher levels that are close to the service provider or even the end user. We believe that service composition should be treated from a specification and execution point of view at the same time, where the former is about the composition logic and the latter about transactional guarantees. Consequently, our model allows for the specification of a number of transactional properties such as *atomicity* and *guaranteed termination* at all levels. Different ways of achieving the composition properties as well as implications of the model are addressed.

1 Introduction

Web services [2, 7] have become popular as a vehicle for the design, integration, composition, and deployment of distributed and heterogeneous software, based on the hope that

[*]A perliminary version of this paper appeared in Proc. 3rd IEEE International Conference on Web Services (ICWS) 2004, San Diego, USA, 462–469.

[†]Work partially done while visiting the University of Muenster, Spring 2006. This research is supported in part by the Natural Sciences and Engineering Research Council of Canada Discovery Grant 3182.

[‡]Work partially done while visiting The University of Waikato, Hamilton, New Zealand, Summer 2006.

distributed computing can now be made a reality easier than with previous approaches such as RPC, object-orientation, or static middleware. However, while industry standards for the description, composition, and orchestration of Web services have been under development for quite some time already, their conceptual underpinnings are still not well-understood. Indeed, conceptual models for service specification are still rare, as are investigations based on them. This paper tries to make a contribution in this direction. In particular, it presents a *multi-level service composition model* that perceives service specification as a process that goes through several levels of abstraction: It starts from transactional concepts at the lowest level, and then gradually abstracts into activities at higher levels that are close to the service provider or even the end user. Importantly, the model allows for a specification of desirable composition properties such as *atomicity* and *guaranteed termination* at all levels.

Web services and service-oriented architectures (SOAs) are currently seen by software vendors and application developers as a new way of coming across both application and data integration problems. The general vision is twofold: First, software services can be described in an implementation-independent and "semantic" fashion; such descriptions are published in generally accessible repositories which can be queried in standardized ways, and users, customers, or clients can hence find service descriptions, compose them into new services fitting their needs, and finally execute the new services by referring back to the service providers behind their selection. To achieve these goals, a variety of industry standards has been made available in recent years, among them SOAP (*Simple Object Access Protocol*) for transportation purposes [17], UDDI (*Universal Description, Discovery and Integration*)[1] for building and querying service repositories, WSDL (*Web Services Description Language*) for service descriptions [17], and BPEL4WS *Business Process Execution Language for Web Services* [2] for the description of service compositions in the form of graph-based process models.

Second, Web services represent an important way of realizing a so-called *service-oriented architecture* (SOA) [13, 22]. A SOA tries to answer the question of which services are available (within, say, a given enterprise) already, which ones need to be newly implemented, and which ones need to be obtained from a suitable provider. To this end, it is reasonable to assume that, from a top-down development perspective, it makes sense to come up with one or more process models that clarify and fix the goals and procedures a client (or a collection of clients in an enterprise) wants to support by appropriately chosen services. Such models will typically be tied to a particular application domain, such as commerce, banking, the travel industry, etc. and will refer to organizational structures and also incorporate objects as well as resources occurring in processes. The next step would be to determine which portions of the overall "process map" can be grouped together in such a way that they can jointly be supported by a service. The result will then be an architecture fixing the composition and integration details at a conceptual level and beyond service and departmental borders [25].

As has been noted, for example, by Hull et al. [14], the conceptual underpinnings of Web services are still not completely understood. For example, in BPEL4WS it is possible to define choreographies (or service compositions) by defining a flow of control using guarded links between the respective activities (which appear in <flow> tags);

[1]`http://www.uddi.org`

[2]`http://www-106.ibm.com/developerworks/library/ws-bpel/`

yet this is entirely syntactic, and there is no way to argue about the properties of the resulting flow. On the other hand, studies such as those reported in [14] indicate that service composition may be more intricate than what the standardization committees assume. Using models such as Mealy automata, Hull and others have been able to show that undesirable side effects may occur when certain types of services are composed (e.g., the result of composing "regular" services may all of a sudden be a "context-sensitive" service).

The model we are proposing and studying in this paper is based on the perception that service composition is not adequately described as long as flat models are used; indeed, in a *flat* model, be it classical transactions, finite-state automata, or Petri nets, the composition designer has to fix a particular level of abstraction and then will run into difficulties when trying to argue about properties that relate to (lower-level) components or to (higher-level) aggregations and that hence actually span several logical levels of the composition. Opposed to this, our intention is to construct a "bridge" between a low-level model that is based on classical transactions [27], a model that generalizes transactional guarantees to an (intermediate) process level [23], and a high-level model such as the ones used in PARIDE [15] that orchestrates e-services via Petri nets.

1.1 A Service Composition Example

As a motivating example, we consider an electronic shopping scenario, where a customer is hunting for some specific goods (such as a a musical instrument). To this end, the various services he or she plans to compose are the following (in the order given):

1. Initially, the customer starts a *price comparison* by turning to a service such as `dealtime.com`. Individual actions are the inspection of various offers made for the product in question, and comparing them based on price, delivery charge, availability, delivery time, etc. Once the customer decides on the shop he wants to buy from, he can turn to the next service.

2. The second service is provided by the *shop*. We assume that the product (e.g., a digital piano) is available in various versions (e.g., dark or light wood), and that the customer can pick one of these. If availability is not granted, he may change his decision. Once committed, the transaction is handed over to a broker (e.g., `PayPal`) for collecting the payment.

3. The *payment broker* is actually a sub-service of the previous service. If payment is transferred successfully, the supplier of the goods enters terminating actions, in this case packaging and delivery. However, if payment transfer is not successful, a different stream of terminating actions is entered: the customer may pay cash or cancel the order.

4. The final service in this case, to be activated within the sequence of termination actions that follow successful payment, is the *delivery service*, which can be an ordinary furniture mover (who might take up to 10 days until delivery, yet is cheap), an express service delivering within 3 to 4 days, or the customer may decide to pick up the piece himself, so that delivery time is minimized.

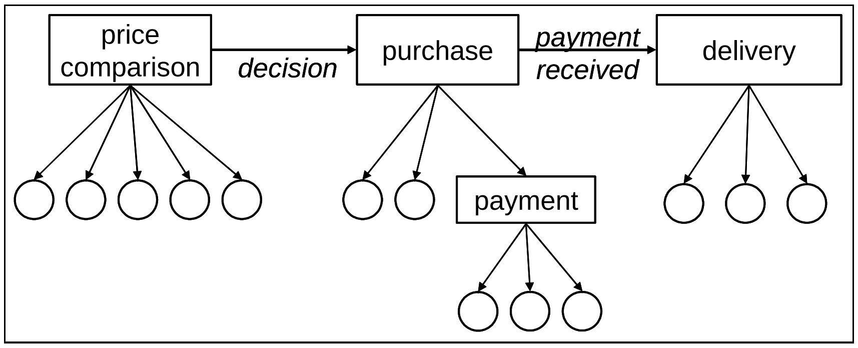

Figure 1: Shopping service composition.

An illustration of this service composition appears in Figure 1. What can be seen from this figure are some of the main ingredients of a Web service: Various individually described and implemented activities or services get combined into a new service. This combination can involve sequencing (price comparison, purchase and delivery in this example), concurrency (getting details from various shops in price comparison), nesting (payment nested within purchase) and, in fact, any complex arrangement that needs to be described using a more sophisticated specification language; we will later assume that a user is capable of providing service compositions at the highest level of abstraction.

The various activities that get composed and combined are different in nature: Some are as simple as a database ACID transaction [27] and can hence be easily *undone* (e.g., the result of a price comparison) or *compensated* for (e.g., overpayment), while others (in our case the payment for the piano in Step 3) mark a decisive point in a service execution which cannot be gone back beyond (at least not easily); following [23], we call such activities "*pivotal*." The occurrence of a pivotal activity has implications for whatever follows in the service composition, since once the pivot has been executed, there should be a guarantee that the "remainder" of the service is also executed and terminated successfully; below we will call this the *guaranteed termination property*. In particular, if a customer has decided on goods to purchase, he or she wants to finish the deal.

1.2 Contributions

The points we are trying to make in this paper, and which extend those made in [26], are the following:

1. An issue such as service composition should be treated from a specification *and* an execution point of view at the same time, where the former is about the composition logic and the latter about transactional guarantees.

2. To remedy the current situation that all activities composed into a service are treated at the same level of abstraction, we present a *multi-level* approach to service composition in this paper: It starts from underlying transactions (in the context of which activities ultimately get executed), and ends at a high level where processes can be abstractly described.

Notice that the latter is in line with previous studies within a variety of contexts; for example, multi-level transaction models [27] have been devised for being able to tolerate non-serializable executions, given the availability of (higher-level) semantic information.

The organization of the remainder of this paper is as follows: In Section 2 we review related work, in particular work on which our approach is built. Next, Section 3 presents our service composition and execution model and discusses different ways of achieving the relevant composition properties. In Section 4 we point out several service issues that can be captured nicely in our model, among which are the *sharing of responsibilities* and *added value*. In Section 5 we first generalize our basic path model from Section 3 to trees of services, and then present our multi-level model. Section 6 puts our model framework in perspective and concludes the paper.

2 Related Work

In this section, we review work that is related to ours, where we restrict our attention to those approaches on which we build, or which we target for extension. Our emphasis in this section is in showing that most *conceptual* models discussed up to now in the literature have been flat models which are limited in their ability to properly describe service compositions. We mention that industry standards such as WSFL can easily establish complex models, by providing the possibility to deliver highly nested XML documents. However, such a form of nesting is purely syntactic, and is unable to associate distinct properties with individual levels of nesting.

An excellent survey of work on modeling individual as well as composite services has recently been delivered by Hull et al. [14]. As far as individual services are concerned, formal models that have been employed include *method signatures* as known from object-oriented programming and *finite-state automata*, mostly in the form of *Mealy* machines. The former approach typically considers a service as a black-box from which only input and output can be seen, whereas the Mealy machine approach considers a service as a "white-box" whose inner structure is visible.

It turns out that such models are not too far from what is happening in industry consortia at the moment. For example, WSDL, the *Web Service Definition Language*, knows I/O signatures and in particular has two categories of message types, *reactive* (where a message is input to a service and can be one-way or of type "request-response") and *proactive* (where a message is output from a service and can be notification or of type "solicit-response"). On the other hand, simple Mealy machines, although capable of reading input and producing output, are hardly suited for handling data as well. To this end, they have been enhanced, for example, by storage capabilities in the style of relational transducers [1].

A major emphasis has recently been put on the specification of service *conversations*, which denote single enactments of a global process. Standards such as WSCL (the *Web Service Conversation Language*) use automata to this end, which from a conceptual perspective are compositions of the Mealy-type of automata mentioned earlier. Indeed, such a composition can proceed in the style common for finite state machines, i.e., they can be composed serially or in parallel, and they can be composed to form loops (corresponding to concatenation, alternatives, and Kleene star in regular expressions, resp.). Compositions are presently formed as peer-to-peer systems with distributed control [11, 5], as hub-and-spoke systems that employ publish-and-subscribe techniques [24], or as systems using mediators like in the WebTransact Architecture [18] or in BPEL4WS.

Our interest is in service compositions and conversations for which certain properties can be specified at design time and verified at run time. Work in this direction is gradually evolving, for example in the verification technique described in [12] which can check for deadlock avoidance or response times. More promising from our perspective are approaches that relate the service composition task to workflow specification, in particular to the specification of workflows and processes that cross organizational boundaries (since individual services typically have distinct providers). Work in this direction has been reported by Colombo et al. [9] as well as in the service orchestration approach used in PARIDE [15] which is based on Petri nets. Finally, Schuldt et al. [23] extend concurrency control and recovery techniques from ordinary transactions to processes and their composition; since this work is the most relevant to ours, we review it in more detail next.

In the model of Schuldt, Alonso, Beeri, and Schek [23], an *activity* corresponds to a conventional (database) transaction or a *transaction program* executed in a transactional application. A *transactional process* is specified in a *process program* which is a set of partially ordered activities. All activities have the atomicity (all-or-nothing) property, that is, every execution will either *commit*, with the intended non-null effect, or *abort*, with the null effect. Next, three important properties of activities are defined in [23]:

1. An activity a is *compensatable* if there exists a compensating activity (that can be executed after a) which semantically undoes the effects of a.

2. An activity a is *assured* or *retriable* if its commit is guaranteed, perhaps after repeated trials (i.e., aborts and restarts).

3. An activity is a *pivot* if it is not compensatable.

Note that compensatability and retriability are orthogonal properties: a compensatable transaction may or may not be retriable, and vice versa. The following is a brief description of a process program:

- A process program is a (rooted) directed tree whose nodes may be of one of the following two types: *singleton* nodes, each corresponding to one activity, or *multi-activity* nodes, each corresponding to a partially ordered set of activities. Two different order constraints may be associated with the activities of a multi-activity node: a partial *strong order* and a partial *weak order*. Activities related by weak order can be executed concurrently but the result of the execution must be equivalent to one where the order is preserved. Those related by strong order must be executed in the given order.

- The edges of the tree correspond to the strong order constraints between the activities of the end nodes.

- Each pivot must be a singleton node. This captures the fact that no other activity of a process may be executed in parallel to a pivotal activity.

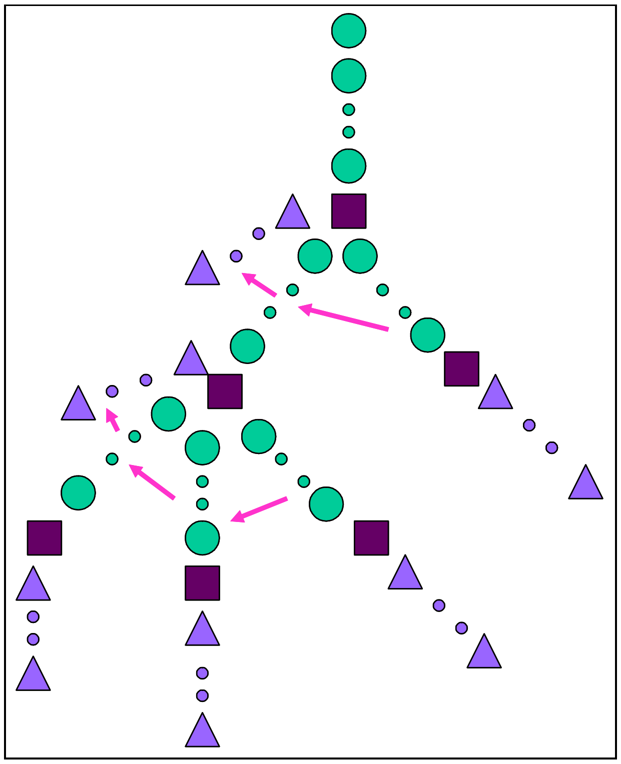

Figure 2: A sample process model.

- A total order, called *preference order,* is defined on the children of a pivot. The last child must be the root of an *assured termination tree*, consisting only of retriable activities.
- The execution of the program starts at the root. A (possibly empty) sequence of nodes with compensatable activities are executed. If any of these activities abort, then all activities executed thus far are compensated. Then a pivot will be executed. If it aborts, again all the activities executed thus far will be compensated and the execution terminates.
- If the pivot commits, the subtree rooted at the first child of the pivot is executed. If that execution terminates with abort, the subtree rooted at the second child will be executed, and so on. As a last resort, the assured termination tree, rooted at the last child of the pivot, will be executed.
- Finally, a process program may not have any pivot. In that case, it has the same properties as a regular transaction, that is, it can be aborted any time prior to its commit.

We illustrate the model just described in the following figures, where we use (green) circles to indicate compensatable activities, (pink) squares for pivots, and (blue) triangles for retriable activities, resp., as shown in Figure 2. Figure 3 shows an execution where something goes wrong in the part of the process that is still compensatable; the

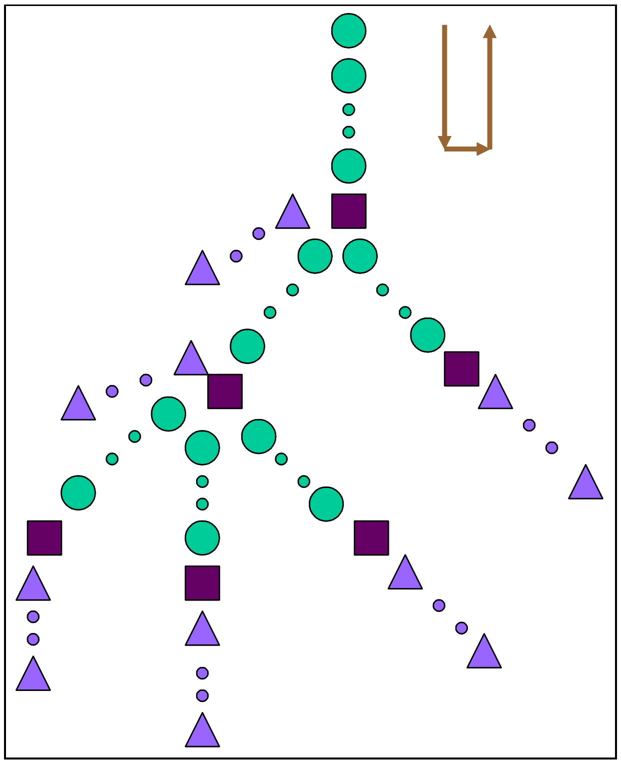
Figure 3: A sample execution with failure and compensation.

necessary compensations are executed, and the process terminates. Figure 4 shows a sample successful termination, where a first pivot is followed by a second one, which then is followed by the first sequence of retriable activities. Next, Figure 5 shows an execution ending somehow in one of possible branches of retriable acitivites.

A process program conforming to the properties listed above will in the sequel be said to have the *guaranteed termination property.* Figure 6 shows four distinct guaranteed terminations. A *process* is an execution of a process program. The execution may contain aborted activities, compensated and compensating activities, aborted activities of sub-processes, etc. However, the actual (net) *effects* of a process are represented by a path in the tree; this path will contain zero or more pivots. Notice that the guaranteed termination property of processes is a generalization of the atomicity property of the traditional transactions.

The emphasis in [23] is put on defining a unified model for process concurrency control and recovery, which essentially extends earlier work by some of the authors [21, 3]; beyond this, they present a dynamic scheduling protocol for the execution of transactional processes that achieves correct executions in the sense defined. Opposed to this, our work considers a model of Web services where processes have services or activities and components, yet we preserve the distinction between compensatable, pivot, and retriable ones. This extension of the model just described is the subject of the next section.

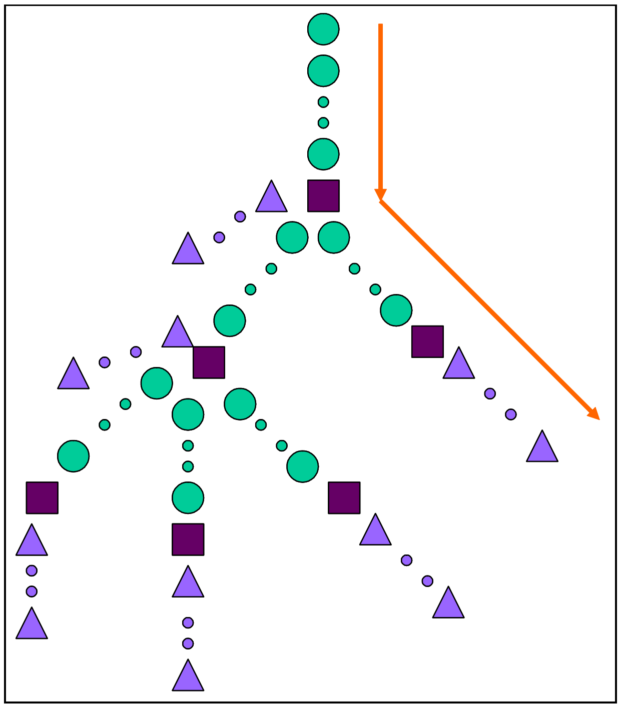

Figure 4: A sample successful termination.

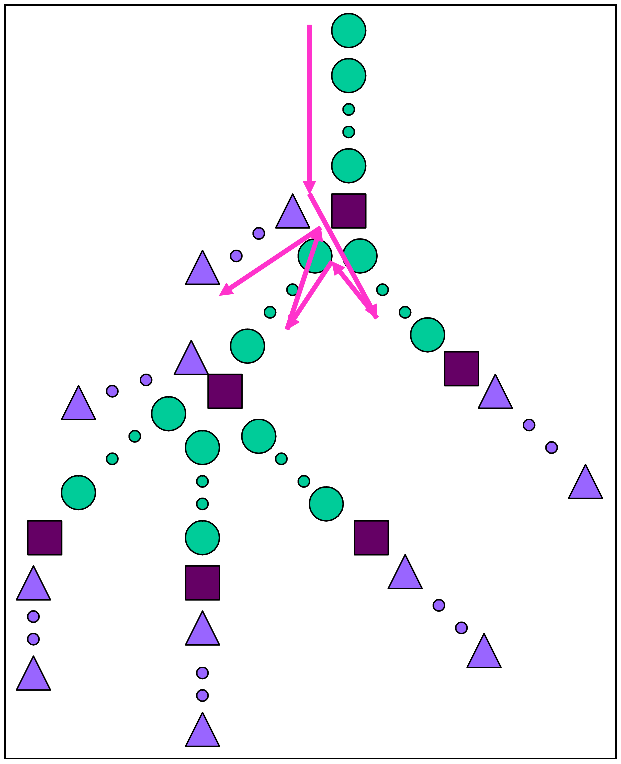

Figure 5: Another sample execution.

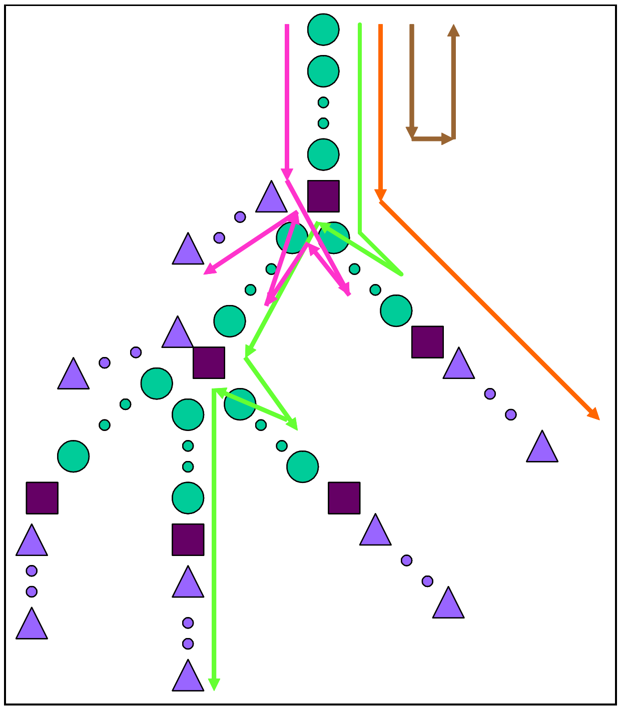

Figure 6: The various guaranteed terminations.

3 A Multi-Level Composition Model

In this section, we present our multi-level model of service composition; in particular, we consider the properties defined for activities in [23], which we have reviewed in the previous section, and extend them to composite activities.

We will consider a process program as a *composition*, denoted $\mathcal{C}$, and an execution of the program, that is, a process, as a *composite activity*, denoted C. We will refer to the activities of the process (that is, the transactions) as *basic activities*. In the following, we extend the transactional properties of the basic activities to composite activities; in other words, we will extend what has above been illustrated in Figures 3–6 to multiple levels of abstraction. We have considered atomicity, and compensatability, pivotal, and retriability properties; we will use the abbreviation *c, p, r* to denote the last three properties, resp.

3.1 Atomicity of Basic Activities

As stated in the previous section, every execution of a basic activity will either *commit*, with the intended non-null effect, or *abort*, with the null effect. In the sequel, we will call the former case the *non-null* termination and the latter case the *null* termination of the activity; we will also denote the two cases as the *successful termination*, called *s-termination*, and the *failed termination*, called *f-termination*, resp. We use the following definition for atomicity.

Definition 3.1 (Atomicity of a basic activity) A basic activity is *atomic* if its execution is guaranteed to result in either the null termination or the s-termination. □

We also assume that the termination properties and hence the atomicity, and the *c, p, r* properties are relative to the composition (and therefore every execution of that composition). Hence, if a is a basic activity of a composite activity C, then the last three properties are denoted $c[C]$, $p[C]$ and $r[C]$.

3.2 Pivot Graphs

As indicated in the previous section, the guaranteed termination property of a given process program facilitates focussing only on the pivots in the program. We define *pivot graphs* for compositions and composite activities as follows. We denote the pivots as p_i for some index i. For convenience, we define a (dummy) *root pivot* $p_\perp$ as an empty activity that is executed first and always successfully. For the process programs (and each such subprocess program) which do not have a pivot, we will associate a (distinct) dummy pivot; this is different from the root pivot.

Definition 3.2 (Pivot graph) A *pivot graph* of a composition $\mathcal{C}$, denoted $pg(\mathcal{C})$, is a directed tree rooted at $p_\perp$ such that

(i) it has at least one node in addition to the root,

(ii) its non-root nodes correspond to the pivots in $\mathcal{C}$,

(iii) the edges correspond to the precedence relation among the pivots in $\mathcal{C}$, and

(iv) the children of each pivot are totally ordered according to the preference order of the subtrees containing them in $\mathcal{C}$. □

Essentially, each node p_i in $pg(\mathcal{C})$ represents the corresponding (real or dummy) pivot p_i in $\mathcal{C}$ together with the compensatable activities preceding p_i in $\mathcal{C}$; the retriable activities in the assured termination path of p_i are ignored. Technically, a different notation, for example $\hat{p_i}$, should be used in the pivot graph to distinguish this node from p_i in $\mathcal{C}$; but, for easy readability, we will use p_i itself.

Example 3.1 Figure 7 shows the pivot graph of a composition. We use this as the running example in this section. The preference order of the children of p_1 is (p_2, p_3), and the order of the children of p_3 is (p_4, p_5, p_6). □

A *pivot graph* of a composite activity C, that is, an execution of $\mathcal{C}$, will be denoted $pg(C)$. Recall that an execution of a process program contains effectively all the nodes from the root to a leaf. Since the assured termination trees of $\mathcal{C}$ that do not contain any pivots will not be represented in $pg(\mathcal{C})$, $pg(C)$ will correspond to a (directed) path from the root to some node in the tree $pg(\mathcal{C})$. We will continue using simply C to denote an *arbitrary* execution of $\mathcal{C}$. To denote a *particular* execution, we will use the sequence of pivots that have been executed in $\mathcal{C}$ as a subscript of C: if $(p_\perp,\ p_i,\ p_j,\ p_k)$ is the sequence, then we will denote C as C_{ijk} omitting $\perp$; if $(p_\perp)$ is the sequence, then we will use $C_\perp$. We will also use the notation $C_{[\perp,m]}$ to denote an execution where all the pivots from the root to p_m in $pg(\mathcal{C})$ have been executed. In this notation, the above two cases

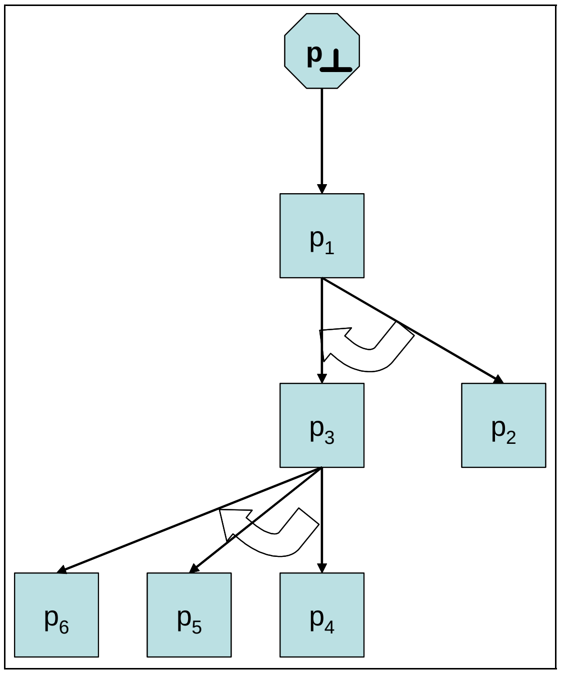

Figure 7: Pivot graph of a composition.

will be represented as $C_{[\perp,k]}$ and $C_{[\perp,\perp]}$. As a concrete example, the execution of Figure 7 where only $p_\perp$, p_1 and p_3 have been successfully executed will be denoted as C_{13} and as $C_{[\perp,3]}$.

We note that, by our convention, $pg(C)$ will always contain $p_\perp$. If it contains *only* $p_\perp$, then C has the null effect and we will call C the *null* termination of $\mathcal{C}$. In all other cases, $pg(C)$ will contain one or more pivots in addition to $p_\perp$, and C will be called a *non-null* termination of $\mathcal{C}$.

3.3 Termination Properties of Composite Activities

Given a composition $\mathcal{C}$, we next consider a higher-level composition $\mathcal{U}$ that contains $\mathcal{C}$, and let U be an execution of $\mathcal{U}$ that contains C. We will associate the transactional properties, namely, atomicity as well as the *c, p, r* properties, to C *relative to* ($\mathcal{U}$ and) U. Since we have categorized the termination possibilities of C as null and non-null, we will assume that the application semantics of the composition $\mathcal{U}$ will determine whether a non-null termination of C is a successful termination or a failed termination relative to $\mathcal{U}$. That is, we assume that, based on the application semantics, each non-null termination of C can be mapped to either an *f*-termination or an *s*-termination relative to the composition $\mathcal{U}$; and each null termination of C will be an *f*-termination relative to $\mathcal{U}$.

Example 3.2 Let us consider the example of Figure 7 again (cf. Example 3.1). We will first associate semantics to the activities. We assume that this composition is for planning a trip from St. John's (Newfoundland) to London (England) to attend a conference. We assume the following details:

1. Air Canada is the only carrier offering direct service between these two cities. (Pivot p_1 is for the purchase of flight tickets with Air Canada.)
2. The conference has arranged special rates with (a hypothetical) Ideal Hotel.
3. The hotel has two locations, called Ideal-A and Ideal-B.
4. The conference will be held in Ideal-A. A small number of rooms in Ideal-A and a substantially large number of rooms in Ideal-B are available at a special conference rate. (Pivot p_2 corresponds to making a reservation in Ideal-A, and pivot p_3 to making one in Ideal-B.)
5. Ideal-B is quite far from Ideal-A.
   (a) The conference organizers have arranged a shuttle bus from Ideal-B to Ideal-A, but the capacity of the bus is limited and so reservation is absolutely essential. (Pivot p_4 is for shuttle bus reservation.)
   (b) Those who could not get a reservation for the shuttle bus can rent a car to go from Ideal-B to Ideal-A. (Pivot p_5 is for car rental.)
   (c) Public transportation can also be used, but it is time-consuming. A special pass can be purchased to use the public transportation. (Pivot p_6 is for the purchase of a pass.)

An execution of this process program will first try to get the flight tickets (p_1), then try hotel reservation in Ideal-A (p_2), and, if unsuccessful, try reservation in Ideal-B (p_3). If successful in the latter case, it will first try for reservation in the shuttle bus (p_4). If that fails, then a car rental will be tried (p_5). If that also fails, then a pass for the public transportation will be purchased (p_6). Thus different executions may have different outcomes. For example, C_{12} refers to (successful) flight ticket purchase and reservation in Ideal-A, whereas C_{135} refers to flight ticket purchase, reservation in Ideal-B and a car rental. $\Box$

Notice in the previous example that different users may have different requirements and therefore accept different sets of outcomes as s-terminated executions. For instance:

- User$_1$ may not accept anything other than C_{12};
- User$_2$ may accept C_{12}, C_{134}, C_{135}, but not C_{136}; (We ignore preferences in this paper;)
- User$_3$ may accept C_{12}, C_{134}, C_{135}, and C_{136}; and
- User$_4$ may accept successful execution of p_1 (flight tickets purchase) and any outcome of the remaining activities (namely, $C_1, C_{12}, C_{13}, C_{134}, C_{135}, C_{136}$), that is, every non-null execution.

It is reasonable to assume that a given composition $\mathcal{C}$ can be "tailored" to various user requirements. Indeed, consider the users just mentioned: For User$_1$, option p_3 (and the subtree rooted at p_3) should not be provided and $pg(\mathcal{C})$ should contain only $p_\perp$, p_1 and p_2; for User$_2$, option p_6 should not be provided.

The requirement for User$_3$ suggests the following notion for s-termination: Any execution of $\mathcal{C}$ where *all* the pivots in *some* path from the root to a leaf of $pg(\mathcal{C})$ have been executed successfully is an s-termination relative to $\mathcal{U}$. With User$_4$ in mind, we will generalize this notion as follows:

Definition 3.3 (s-termination) An *s-termination* of a composition $\mathcal{C}$ is an execution where, for some path from the root to a leaf, the pivots of some specified prefix of that path have been executed successfully. □

For example, execution C_1 in the previous example is an f-termination for User$_3$, but it is an s-termination for User$_4$. Thus, depending on given user requirements, a non-null execution will be mapped to either an s-termination or an f-termination. The set of executions that are mapped to s-terminations will form the *s-termination set* of $\mathcal{C}$, relative to $\mathcal{U}$.

3.4 Transactional Properties of Composite Activities

We consider the transactional properties next. First we note that the *c, p, r* properties of C relative to U are independent of the properties of the basic activities of C relative to C. We illustrate this with the following examples.

Example 3.3 In the composition of Figure 7, the purchase of the flight tickets p_1 may be a pivot to the travel agency in the sense that the airlines will not refund the money. However, the travel agency may not treat it as a pivot for the customer for whom the ticket is intended, if the agency is able to use the ticket for another customer. (Sometimes travel agencies buy seats in bulk from airlines and then sell them to customers on their own.) That is, C_1 may be compensatable for the customer. □

Example 3.4 Suppose that, in a composition $\mathcal{U}$ like the one shown in Figure 1, C refers to the composite activity *purchase* of an article and has (among others) an activity *payment* denoted as a. Then C may be compensatable relative to U, $c[U]$, (with the compensating composite activity being the *refund*) if the purchased item is returnable; otherwise (for example, if the store policy is "no exchange, no return") it will be pivotal, $p[U]$. Also, even if the refund policy dictates some penalty (for example, 10% of the cost), if the penalty is acceptable for the composition $\mathcal{U}$ then, in that case also, the *purchase* activity may be considered to be compensatable relative to U. Note that in the composition level $\mathcal{C}$, the *payment* activity may always be pivotal relative to C, $p[C]$, and similarly the *refund* activity C' may contain a *refund-payment* activity a' which is also pivotal relative to C', $p[C']$. □

We now define the atomicity and the *c, p, r* properties for a composition $\mathcal{C}$, that is, for any execution C of $\mathcal{C}$. Again, all these properties are relative to the composition $\mathcal{U}$. For brevity, we will not state this in the following definitions. The atomicity definition is similar to that for a basic activity:

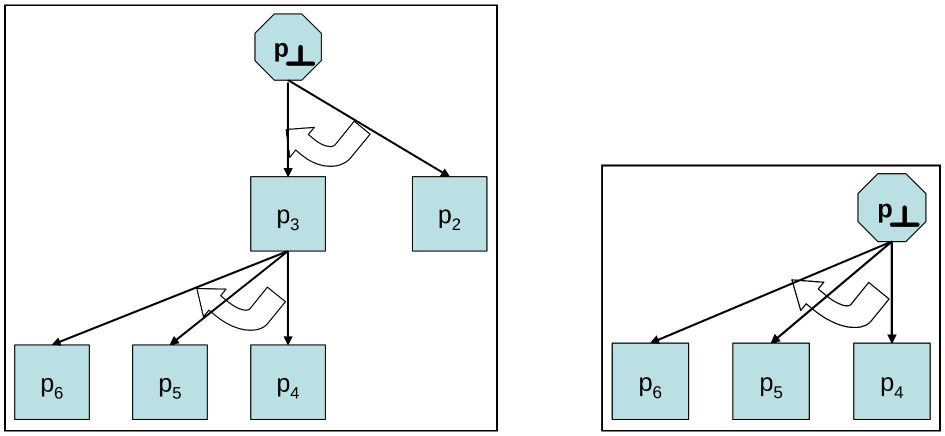

Figure 8: $\mathcal{C}_{[1]}$ (left) and $\mathcal{C}_{[3]}$ (right) suffixes for pivot graph from Figure 7.

Definition 3.4 (Atomicity of a composition) A composition is *atomic* if its execution is guaranteed to result in either the null termination or an s-termination. □

The c, p, r properties can be extended to atomic compositions in a straight-forward manner.

Definition 3.5 (i) A composite activity C is *compensatable* if there exists a compensating activity (relative to $\mathcal{U}$) which semantically undoes the effects of C. An atomic composition $\mathcal{C}$ is *compensatable* if each of its s-terminations is compensatable.

(ii) An atomic composition $\mathcal{C}$ is *retriable* if one of its s-terminations can be guaranteed perhaps after a few attempts.

(iii) A composite activity C is a *pivot* if it is not compensatable. An atomic composition $\mathcal{C}$ is *pivot* if some of its s-terminations are pivots. □

The underlying assumption is that we would like the composition $\mathcal{U}$ to consist of (basic or composite) atomic activities. The above definitions state the requirements for the c, p, r properties in addition to atomicity. Atomicity itself can be described in terms of compensatability and retriability. We first introduce some terminology needed below. For a pivot p_i in $\mathcal{C}$, we define *the suffix of* $\mathcal{C}$ *from* p_i, denoted $\mathcal{C}_{[i]}$, as the subtree of $\mathcal{C}$ rooted at p_i, with p_i replaced by $p_\perp$. Clearly, $\mathcal{C}_{[i]}$ is a (sub) process program. Note that $\mathcal{C}_{[\perp]}$ is the same as $\mathcal{C}$. For example, for the pivot graph $pg(\mathcal{C})$ of Figure 7, $pg(\mathcal{C}_{[1]})$ and $pg(\mathcal{C}_{[3]})$ are given in Figure 8. For various reasons, a suffix $\mathcal{C}_{[i]}$ of $\mathcal{C}$ may not be executable (independent of $\mathcal{C}$). In the following, any property stated for $\mathcal{C}_{[i]}$ is applicable only when $\mathcal{C}_{[i]}$ is executable.

Definition 3.6 (Recoverability) An f-termination $C_{[\perp,i]}$ of $\mathcal{C}$ is:

- *backward-recoverable* if $C_{[\perp,i]}$ is compensatable;
- *forward-recoverable* if $\mathcal{C}_{[i]}$ or a (sub)composition $\mathcal{C}_{[i]'}$ semantically equivalent to $\mathcal{C}_{[i]}$ is retriable; and

- *recoverable* if it is either backward-recoverable or forward-recoverable. □

We are now able to state a sufficient condition for the atomicity of a composition:

Theorem 3.1 *A composition $\mathcal{C}$ is atomic if each of its f-terminations is recoverable.*

Proof: Suppose an execution of $\mathcal{C}$ results in an f-termination $C_{[\perp,i]}$. If $C_{[\perp,i]}$ is backward-recoverable, the execution can be compensated to get the null termination; if it is forward-recoverable, then $\mathcal{C}_{[i]}$ or an equivalent $\mathcal{C}_{[i]'}$ can be retried to get an s-termination. Thus an atomic execution of $\mathcal{C}$ can be guaranteed. □

We can now derive the requirements for the c, p, r properties for an arbitrary composition, incorporating those required for atomicity explicitly. By doing so, we get additional flexibility in obtaining these properties in an execution.

Corollary 3.2 Let $\mathcal{C}$ be an arbitrary composition.

1. If $\mathcal{C}$ has only one non-root pivot, then
   - $\mathcal{C}$ is compensatable if its s-termination can be compensated;
   - $\mathcal{C}$ is retriable if its s-termination can be guaranteed (possibly after several attempts); and
   - $\mathcal{C}$ is pivot if it is not compensatable.
2. If $\mathcal{C}$ has more than one non-root pivot, then
   - $\mathcal{C}$ is compensatable if all its non-null (f- and s-) terminations are compensatable;
   - $\mathcal{C}$ is retriable if one of its s-terminations can be guaranteed (possibly after several attempts) and each of its f-terminations is recoverable; and
   - $\mathcal{C}$ is pivot if some of its s-terminations are not compensatable and each of its f-terminations is recoverable. □

Compensatability is straight-forward. Retriability allows for an f-termination to be compensated and the entire composite activity to be restarted, or the suffix following the f-termination having the retriable property. It is possible that some f-terminations have one option, some others have the other option, and some have both options. The pivot definition implies that if the execution has proceeded far enough that it cannot be compensated any more, then the execution can be carried further towards an s-termination perhaps after a few attempts.

We also note that the atomicity and the c, p, r properties do not require *any* suffix of $\mathcal{C}$ to be executable. The executability of the suffixes simply adds flexibility to the execution of the entire composite activity.

Example 3.5 Let us consider these properties for the composition $\mathcal{C}$ in our running Example 3.1 for User$_3$. Recall that the s-terminations are C_{12}, C_{134}, C_{135}, and C_{136}.

(i) As mentioned, the compensatability notion is straight-forward.

(ii) For retriability, first we need the property that an s-termination of $\mathcal{C}$ can be guaranteed in a finite number of attempts. Next, C_1 and C_{13} are two f-terminations. For C_1, we need the property that C_1 is compensatable, or $\mathcal{C}_{[1]}$ is retriable, or both. Similarly, for C_{13} we need the property that C_{13} is compensatable, or $\mathcal{C}_{[3]}$ is retriable, or both. As stated earlier, it is possible that different options are available for the two f-terminations. For example, (a) C_1 may be compensatable and $\mathcal{C}_{[1]}$ may not be retriable (or even executable), and (b) C_{13} may not be compensatable but $\mathcal{C}_{[3]}$ is retriable. This would mean that every execution of $\mathcal{C}$ resulting in C_1 must be compensated and retried, and if C_{13} is obtained in some attempt then $\mathcal{C}_{[3]}$ is executed a few times until an s-termination is obtained.

(iii) Similarly, for a pivot, if C_1 is not compensatable, then $\mathcal{C}_{[1]}$ must be retriable. If C_1 is compensatable and $\mathcal{C}_{[1]}$ is retriable, then $\mathcal{C}_{[1]}$ can be tried. If C_1 is compensatable and $\mathcal{C}_{[1]}$ is not retriable, then C_1 will be compensated. The options with C_{13} are similar. □

3.5 Higher Level Compositions

After these preparations, it is possible to compose $\mathcal{U}$ as a process program of [23] where basic activities are replaced by any (basic or composite) activities. Each basic activity will be executed by its transaction program and each composite activity will be executed by its own process program; these programs are independent of the process program $\mathcal{U}$. With the atomicity and the c, p, r properties established for each of the constituent activities, null and non-null terminations can be established for $\mathcal{U}$. Now $\mathcal{U}$ can be an end-user level composition or can be used in a higher level composition. In either case, denoting the composition as $\mathcal{G}$, depending on the application semantics, the terminations of $\mathcal{U}$ can be mapped into f-terminations and s-terminations relative to $\mathcal{G}$, and the atomicity and the c, p, r properties can be defined for $\mathcal{U}$ relative to $\mathcal{G}$, exactly as they were defined for $\mathcal{C}$ relative to $\mathcal{U}$. Thus, atomicity and the c, p, r properties can be carried to any activity in any level of the composition. As will be seen below, this allows for an adequate description of a variety of service issues that has not been possible before.

4 Service Issues

In this section, we consider various issues in connection with Web services that can be made precise in our framework. To this end, we first look at different ways of achieving the atomicity and the c, p, r properties for composite activities, in the context of Web services. Then we consider the "added value" aspect in service composition.

As before, we consider a composition $\mathcal{U}$ consisting of (basic or composite) atomic activities. Let $\mathcal{U}$ contain a composition $\mathcal{C}$ whose execution yields a composite activity C. As stated earlier, f-termination, s-termination, atomicity and the c, p, r properties of $\mathcal{C}$ are relative to $\mathcal{U}$. Since we have assumed the process program model of [23] for $\mathcal{U}$, the intended (c, p or r) property of $\mathcal{C}$ is known to $\mathcal{U}$.

4.1 Sharing of Responsibilities

We now assume that each (basic or composite) activity will be executed by a Web service. (We will simply use the term "service" to also mean "service provider.") Thus, let a service SU execute process program $\mathcal{U}$, and let a service SC execute composition $\mathcal{C}$. (We do not exclude the possibility that service SC is using some other (sub) services to execute some of its activities, nor the possibility that SC is SU itself. Also, SC may execute some other compositions of $\mathcal{U}$ in addition to $\mathcal{C}$.)

Our premise is the following:

- Basic activities correspond to atomic transactions, and their atomicity is guaranteed by the database management systems executing them.
- For composite activities, we have distinguished two properties, namely, guaranteed termination and atomicity.
- We expect that a service provider executing a composite activity assures at least its guaranteed termination.
- Atomicity of composite activities is assumed in higher level compositions. Here, atomicity of $\mathcal{C}$ is assumed in $\mathcal{U}$.
- If the provider does not assure atomicity of the composite activity, then the service requestor must be responsible for its atomicity. Thus, if SC does not assure atomic execution of $\mathcal{C}$, then SU takes the responsibility.
- Whether backward- or forward-recovery is done to achieve atomicity of $\mathcal{C}$ may depend upon the c, p, r properties of $\mathcal{C}$ (relative to $\mathcal{U}$).
- We assume that compensation of both f-terminations and s-terminations of $\mathcal{C}$ is the responsibility of SU. In some cases, SC may also do these. We allow for this possibility in the following.

In the following, we look into different ways of SU and SC sharing the responsibilities for achieving the atomicity and the c, p, r properties of $\mathcal{C}$. By *executability* of a composition, we mean the availability of a service provider to execute that composition.

I. SC **guarantees atomicity of $\mathcal{C}$.** In this case, for compensatability, any s-termination of $\mathcal{C}$ must be compensatable by SC or SU (perhaps by delegating the compensation to some other service provider). For retriability, if SC returns the null termination, then SU must delegate $\mathcal{C}$ to another service provider, and keep doing so until an s-termination is obtained. For pivot, SU may simply accept the outcome of SC and proceed appropriately.

II. SC **does not guarantee atomicity of $\mathcal{C}$.** In this case, SC may return non-null f-terminations. With such an f-termination $C_{[\bot,i]}$, (i) for obtaining the null termination SU must execute an appropriate compensating activity, and (ii) for getting an s-termination, when it is possible and desirable, either $C_{[\bot,i]}$ can be compensated and $\mathcal{C}$ retried, or $\mathcal{C}_{[i]}$ or an equivalent $\mathcal{C}_{[i]'}$ retried by SU, perhaps by delegating the task to another service provider.

We illustrate some options with our running example (trip planning from Newfoundland to England) from the previous section.

Example 4.1 Consider an *f*-termination C_1, i.e., only the flight tickets have been purchased, but hotel accommodation has not been reserved.

1. C_1 is compensatable and $\mathcal{C}_{[1]}$ is not retriable. The travel agency is willing to treat the flight tickets purchase as compensatable for a customer. In addition, that travel agency might only be allowed to sell the entire package, not a part of it. There may be several travel agencies delegated to this conference each given a quota of reservations. If one does not succeed, another may succeed. Then C_1 may be compensated and $\mathcal{C}$ tried with another travel agent (another service provider).

2. C_1 is not compensatable and $\mathcal{C}_{[1]}$ is retriable. That particular travel agency might not succeed in hotel reservation (due to a limited quota it has been given), but the conference organizers (another service provider) may step in and guarantee the reservation to the customer directly. □

There may exist other sophisticated ways too, for achieving an atomic execution of $\mathcal{C}$. We illustrate two possibilities next.

(a) Partial forward-recovery: $\mathcal{C}_{[i]}$ may be retried by SC or another provider even if its *s*-termination cannot be guaranteed, but the effective execution can be 'extended' from $C_{[\perp,i]}$ to $C_{[\perp,j]}$, for a node p_j which is a descendant of p_i in $pg(\mathcal{C})$, in case another service provider can take over from $C_{[\perp,j]}$ but not from $C_{[\perp,i]}$.

Example 4.2 After C_1, SC may try and guarantee up to C_{13}. The customer may decide to buy a public transportation pass by himself. □

(b) Partial backward-recovery and retry: It may be possible to do partial compensation in some cases (irrespective of whether full compensation is possible or not). In other words, with a termination $C_{[\perp,i]}$, a compensating activity may yield effectively $C_{[\perp,i']}$ for some node $p_{i'}$ which is an ancestor of p_i in the path $pg(C_{[\perp,i]})$. Then $\mathcal{C}_{[i']}$ may be retried. This will help in the situation where $\mathcal{C}_{[i']}$ is retriable but $\mathcal{C}_{[i]}$ is not, for example, if a service provider is available for the first but not the second, etc. Partial compensation may also result when a compensating activity is also a composite activity and its execution results in an *f*-termination.

Example 4.3 Suppose $\mathcal{C}_{[1]}$ is retriable, but $\mathcal{C}_{[3]}$ is not. Then, after the *f*-termination C_{13}, the hotel reservation part p_3 might be compensated and $\mathcal{C}_{[1]}$ tried again. □

We can summarize the characteristics as follows.

1. The atomicity and the c, p, r properties are those of the activity $\mathcal{C}$, and not necessarily of a service provider of $\mathcal{C}$. SU is ultimately responsible for achieving these properties.

2. The c, p, r properties of $\mathcal{C}$ need not be known to SC. Of course, the retriability requirement of $\mathcal{C}$ should be known to SC when it is capable, and is required, to guarantee retriability. Also, SC needs to know which terminations are *s*-terminations relative to $\mathcal{U}$, whenever it is expected to yield an *s*-termination.

3. When SC does not guarantee atomicity, SU has to perform the forward or backward recovery of *f*-terminations, perhaps using other service providers. Thus SC may not know which *f*-terminations are recoverable. Therefore, it can only specify the *f*-terminations it can provide, and it is up to SU to figure out whether they all are recoverable.

There are two issues which are related to the above. The first is that there may exist distinct *views* in a service composition. While a service provider SC needs to have the complete process program $\mathcal{C}$, the "view" of $\mathcal{C}$ known to SU may be limited to the pivot graph $pg(\mathcal{C})$. In fact, depending on the guarantee provided by SC, some of the pivots may be combined into 'higher level' pivots and a more abstract view may be given to SU. In our running example, if atomicity of $\mathcal{C}_{[3]}$ is guaranteed by the travel agent, then the subtree rooted at p_3 may be represented as a single pivot p'_3 to SU.

The second related issue is the role of subservices. Indeed, service provider SC may employ subservices to execute some of the activities of $\mathcal{C}$. As mentioned before, SC is expected, at the very least, to provide (to SU) a guaranteed termination of $\mathcal{C}$. SC may delegate part of this responsibility to its subservices. For example, the execution of activities related to the atomicity of one or more pivots of $\mathcal{C}$ can be delegated to a subservice.

4.2 Framework for Sharing Responsibilities

In this subsection, we propose a framework for SU and SC to share responsibilities for achieving the transactional properties for $\mathcal{C}$. Our framework is different from the mechanisms proposed in BPEL4WS for the transactional properties. We first describe our framework below and then compare with that of BPEL4WS. We take $\mathcal{C}$ as a composite activity consisting of some basic or composite activities.

1. Fault handlers. We have assumed so far that guaranteed termination of $\mathcal{C}$ is the responsibility of SC. In this section, we allow for SU taking that responsibility, if SC does not provide guaranteed termination. To achieve guaranteed termination, some backward- or forward-recovery may be needed, as per our process program model. We recall the recovery procedure below for the simple case where $\mathcal{C}$ has only one pivot. Note that, in this case, guaranteed termination property is the same as atomicity.

An execution of $\mathcal{C}$ can be denoted as $x_1, x_2, \ldots, x_l, y, z_1, z_2, \ldots, z_m$, where each x_i is compensatable, y is pivot, and each z_j is retriable.

- When some x_i fails, then the backward-recovery, namely, the compensation of the part $x_1 \ldots x_{i-1}$ will be done. The recovery may consist of compensating x_j, for each j between 1 and $i-1$, starting from x_{i-1} in the reverse order, or by some other means, for example, compensating some x_j's together. The important point is that the recovery may depend on the extent of the compensatable activities that have been executed before the failure occurred.

- When the pivot y fails, the compensation has to be performed for $x_1, x_2, \ldots, x_l$.

- When some z_i fails, then the forward-recovery will be done. This might typically involve retrying z_i and then continuing the execution of the rest of the retriable activities of $\mathcal{C}$.

To coordinate such recovery and obtain a guaranteed termination, we assign a *fault handler* $fh_{\mathcal{C}}(\mathcal{C})$ to $\mathcal{C}$. We also assign a *fault handler for* $\mathcal{C}$ *in* $\mathcal{U}$, $fh_{\mathcal{U}}(\mathcal{C})$. If $fh_{\mathcal{C}}(\mathcal{C})$ is unable to get a guaranteed termination of $\mathcal{C}$, then $fh_{\mathcal{U}}(\mathcal{C})$ will try. If that also fails, then it is taken as an unguaranteed termination of $\mathcal{U}$, and $fh_{\mathcal{U}}(\mathcal{U})$ tries for guaranteed termination of $\mathcal{U}$. If that also fails, then the responsibility falls on the fault handlers associated with the parent $\mathcal{G}$ of $\mathcal{U}$, and so on.

2. Recovery handlers. Next, we consider achieving atomicity from guaranteed termination. This amounts to getting the null termination or an s-termination from an f-termination. As we argued above, this can be done by SC or SU. For this, we associate two *recovery handlers*: $rh_{\mathcal{C}}(\mathcal{C})$ associated with SC and $rh_{\mathcal{U}}(\mathcal{C})$ associated with SU. On a (guaranteed) f-termination of $\mathcal{C}$, $rh_{\mathcal{C}}(\mathcal{C})$ will do backward-recovery consisting of compensating the activities executed thus far to get the null termination, or attempt forward-recovery trying to execute the appropriate suffix. Both backward- and forward-recovery may even be partial, as illustrated in the last subsection. Either SC completes the recovery, or it forwards the resulting f-termination to SU and then $rh_{\mathcal{U}}(\mathcal{C})$ will take over the recovery. By the assumption in our model that $\mathcal{C}$ is atomic relative to $\mathcal{U}$, if $rh_{\mathcal{C}}(\mathcal{C})$ does not succeed, then $rh_{\mathcal{U}}(\mathcal{C})$ will definitely succeed in getting an atomic execution of $\mathcal{C}$.

3. Compensating activity. An s-termination of $\mathcal{C}$ may have to be compensated due to an f-termination of an activity subsequent to $\mathcal{C}$ in $\mathcal{U}$. The compensation might be done by SC or SU. Compensation will be triggered by SU. Since compensation can be considered as a backward recovery, we delegate it to $rh_{\mathcal{C}}(\mathcal{C})$, and if it fails then to $rh_{\mathcal{U}}(\mathcal{C})$. The compensation might involve executing an activity $\mathcal{C}'$. Then SU will execute this, perhaps by delegating it to a service provider SC' (which could be the same as SC). Again, SC' may assure atomicity or just guaranteed termination. The fault handlers $fh_{\mathcal{C}'}(\mathcal{C}')$ and $fh_{\mathcal{U}}(\mathcal{C}')$ will be responsible for the guaranteed termination. Any non-null (guaranteed) f-termination will be handled by $rh_{\mathcal{C}'}(\mathcal{C}')$ and then, if needed, by $rh_{\mathcal{U}}(\mathcal{C}')$.

To summarize:

- the fault handlers $fh_{\mathcal{C}}(\mathcal{C})$ and $fh_{\mathcal{U}}(\mathcal{C})$ are responsible for achieving a guaranteed termination of $\mathcal{C}$;
- the recovery handler $rh_{\mathcal{C}}(\mathcal{C})$ in $\mathcal{C}$ tries to achieve the atomicity of $\mathcal{C}$; and
- the recovery handler $rh_{\mathcal{U}}(\mathcal{C})$ in $\mathcal{U}$ achieves the atomicity of $\mathcal{C}$ in case $rh_{\mathcal{C}}(\mathcal{C})$ does not.

We note that $fh_{\mathcal{C}}(\mathcal{C})$ and $fh_{\mathcal{U}}(\mathcal{C})$ deal with compensation at the "lower" level, that is, compensation of the constituent activities of $\mathcal{C}$, whereas $rh_{\mathcal{C}}(\mathcal{C})$ and $rh_{\mathcal{U}}(\mathcal{C})$ deal with compensation of the "pivotal components" of $\mathcal{C}$.

In the next higher level, assuming $\mathcal{G}$ to be the parent composition of $\mathcal{U}$, the fault handlers $fh_{\mathcal{U}}(\mathcal{U})$ and $fh_{\mathcal{G}}(\mathcal{U})$ will be responsible for obtaining a guaranteed termination of $\mathcal{U}$, and the recovery handlers $rh_{\mathcal{U}}(\mathcal{U})$ and $rh_{\mathcal{G}}(\mathcal{U})$ will be responsible for obtaining the atomicity of $\mathcal{U}$ relative to $\mathcal{G}$.

We now compare our proposal with the BPEL4WS proposal. The BPEL4WS mechanisms are described below briefly.

- Two kinds of activities, *basic* and *structured*, are defined. A structured activity is a partially ordered set of activities. It corresponds to a composite activity in our model.
- Each activity implicitly defines a *scope*.
- The activities of a structured activity in a scope either all complete or are all compensated. An execution of the structured activity that does not accomplish this, that is, a non-null *f*-termination, in our terminology, is taken as a *fault*.
- Scopes can be nested.
- *Fault handlers* and *compensation handlers* are associated with a scope.
- Fault handlers "catch" the faults, that is, *f*-terminations of the structured activity, and take care of their compensation, either within that scope or by "throwing" them to the enclosing scope.
- Compensation handlers undo already completed activities, that is, *s*-terminations. Compensation handlers are defined within the scope.
- A compensating activity may also fail, in which case the fault handler will compensate this compensating activity. When compensation is not possible within a scope, the fault is thrown to the enclosing scope.

Thus, identifying a scope for $\mathcal{C}$ and treating $\mathcal{U}$ as its enclosing scope, fault handlers and compensation handlers can be used to define the responsibility for atomicity of $\mathcal{C}$.

We can observe the following main differences between our approach and the BPEL4WS proposal.

1. Fault handlers are used in BPEL4WS for achieving atomicity. They are used in our model to get guaranteed termination. We use recovery handlers additionally to achieve atomicity.
2. The fault handler associated with a scope is expected to handle *any* fault: (i) that may occur in the execution of the normal activities in that scope; (ii) that may be thrown from the compensation handler of that scope; or (iii) that may be thrown from the fault handlers of the enclosed (children) scopes. In our model, fault handlers are used only for the first category above.
3. A fault in a scope can be thrown to any ancestral scope, one scope at a time, in BPEL4WS. In our model, unsuccessful recovery (to atomicity) in one level is thrown to its parent level only where the recovery is expected to be completed.

We note that our framework is simple, modular, and applicable to compositions of any number of levels.

4.3 Added Value

As observed above, we have defined a composition $\mathcal{U}$ as consisting of atomic activities. Consider a composition $\mathcal{C}$ in $\mathcal{U}$ having several pivots. It may be possible to replace $\mathcal{C}$ by a set of (appropriately ordered) subcompositions each consisting of a subset of those pivots, and each such subcomposition executed by a (perhaps different) service. We will call the resulting composition $\mathcal{U}'$. Then, with respect to functionality, $\mathcal{U}$ and $\mathcal{U}'$ will be equivalent. However, $\mathcal{U}$ may have some added value compared to $\mathcal{U}'$. That is, an atomic execution of $\mathcal{C}$ by a single service may be more desirable than the atomic executions of the individual subcompositions of $\mathcal{C}$ by different services. We explain this in the following.

For simple exposition, we will confine our attention to the case of $\mathcal{C}$ decomposed into a sequence $\mathcal{C}_1; \mathcal{C}_2$.

1. *Reduction in the total cost.* It may be cheaper to execute $\mathcal{C}$ by the same service compared to executing $\mathcal{C}_1$ and $\mathcal{C}_2$ by different services. Some examples are:

   (a) If printing and binding of a document are done in the same place, the cost of transporting the printed material for binding can be avoided.

   (b) With electronic documents, the two activities executed at two different sites may necessitate preparing an XML document from the output of one activity, sending that XML file to the second site, and extracting the information from that document for input to the second activity in that site. This intermediate XML document preparation and transportation can be avoided if both activities are executed at the same site.

   (c) A furniture store might be able to deliver the purchased items cheaply through a company contracted for all its deliveries.

   (d) It may be that certain common resources are needed to both activities, and so it will be cheaper for a service provider to do them together.

2. *Quality of service.* There may be implicit dependencies between the activities affecting the quality of the end product. For example, in an e-learning environment, an intermediate test on the materials of a learning session might be easier and better prepared, and administered, by the same service provider who designs and supervises that session, than a different service provider.

3. *Atomicity guarantee.* It is possible that an s-termination of $\mathcal{C}_1$ cannot be compensated (and $\mathcal{C}_2$ is not retriable), but a service provider (only if executing both $\mathcal{C}_1$ and $\mathcal{C}_2$) can keep $\mathcal{C}_1$ in a prepared-to-commit state until the execution of $\mathcal{C}_2$ reaches the commit stage and then commit both $\mathcal{C}_1$ and $\mathcal{C}_2$ together.

   We note that the facility of keeping an execution of an activity in a prepared-to-commit or “pending” state, and later committing or aborting based on the execution of subsequent activities is called *virtual compensatability* in [18]. We do not distinguish virtual compensatability from real compensatability, where a committed activity can be undone by executing a compensating activity, in our model.

4. *Increased security and autonomy.* For service providers, this may amount to, for example, not letting out trade, contract, or service secrets.

Note that in many such examples, a non-null *f*-termination (which necessitates the execution of a suffix of the composition) might imply "loosing" the added value. This, in practice, may prompt some penalty to the service provider who is expected to deliver an *s*-termination. The penalty may be determined depending on the *f*-terminations.

5 From Path to Graph Composition Models

We will call the process model introduced in section 2 the *path model*, for the obvious reason that completion of a process execution always follows a path through the underlying process model. As will be shown in this section, we can generalize this model, still retaining the properties we have established so far.

In order to clearly state the generalizations, we briefly review the path model and highlight some of its main characteristics in the following. We refer only to pivot graphs in this section, and use $\mathcal{C}$ to refer to $pg(\mathcal{C})$ also, and similarly C to refer to $pg(C)$ also.

5.1 Path Model

A. Composition

- Composition $\mathcal{C}$ is a tree, as described in Section 2. $\mathcal{C}$ is part of a higher level composition $\mathcal{U}$.
- In $\mathcal{C}$, the children of each non-leaf node are totally ordered. Exactly one child needs to be executed in an execution of $\mathcal{C}$. The order indicates execution preference among the children. We will call this *children execution logic*, abbreviated as *ce-logic*, at that node. We take the ce-logic at the leaves of $\mathcal{C}$ as null.

B. Execution

- A composite activity C is a path in $\mathcal{C}$, from the root to a leaf node. $\mathcal{C}$ contains all possible composite activities and only those. That is, any path in $\mathcal{C}$ from the root to a leaf refers to a composite activity. (Note that, in Definition 3.3, we allowed for some of the paths from the root to some non-leaf nodes also to be *s*-terminations and hence to be composite activities. For simplicity, we ignore this generalization in this section.)
- Partial execution is represented by a path from the root to some node p_i in the tree, denoted $C_{[\perp,i]}$. The part that is yet to be executed (for an *s*-termination) is the subcomposition of $\mathcal{C}$ from p_i, called the suffix of $\mathcal{C}$ from p_i, denoted $\mathcal{C}_{[i]}$. The subcomposition will contain the subtree of $\mathcal{C}$ rooted at p_i, all nodes in the subtree will have the same ce-logic as in $\mathcal{C}$, and the node label of p_i will be replaced by $\perp$.

C. Transactional Properties

- First, a guaranteed termination of $\mathcal{C}$ is desired. Then, the entire composition $\mathcal{C}$ is intended to be atomic in $\mathcal{U}$. In addition, $\mathcal{C}$ is compensatable, pivot or retriable relative to $\mathcal{U}$.

- For atomicity, every f-termination of $\mathcal{C}$ should be forward- or backward-recoverable.

- Backward-recovery of an f-termination $C_{[\perp,i]}$ amounts to rolling back the entire execution to a null execution. Partial backward-recovery refers to rolling back some pivots in $C_{[\perp,i]}$, in reverse order.

- All roll backs are logical. To roll back from p_i to p_j, a *compensating* subcomposition, denoted $\mathcal{C}^{-1}_{[j,i]}$, rooted at p_i is to be executed. This will facilitate different compensation options. (Again, the compensation may be delegated to some provider.) After the compensating subcomposition has been executed successfully, normal processing can continue with $\mathcal{C}_{[j]}$. The pivots in the compensating part need not correspond to those in the compensated part.

- For forward-recovery from p_i, the suffix $\mathcal{C}_{[i]}$, or an equivalent $\mathcal{C}_{[i]'}$, is to be executed from p_i.

- Whether forward- and/or backward-recovery is possible depends on the c, p, r properties of $\mathcal{C}$ relative to $\mathcal{U}$.

D. Service Issues

- SC and SU are service providers for $\mathcal{C}$ and $\mathcal{U}$ repectively.

- Two fault handlers $fh_{\mathcal{C}}(\mathcal{C})$ and $fh_{\mathcal{U}}(\mathcal{C})$ are associated with SC and SU respectively, for obtaining a guaranteed termination of $\mathcal{C}$. On an unguaranteed termination of $\mathcal{C}$, first, $fh_{\mathcal{C}}(\mathcal{C})$ tries for guaranteed termination of $\mathcal{C}$, and if it fails, then $fh_{\mathcal{U}}(\mathcal{C})$ tries. If that also fails, it is taken as an unguaranteed termination of $\mathcal{U}$.

- Two recovery handlers $rh_{\mathcal{C}}(\mathcal{C})$ and $rh_{\mathcal{U}}(\mathcal{C})$ are associated with SC and SU respectively, for obtaining an atomic execution of $\mathcal{C}$ relative to $\mathcal{U}$. For atomicity, as stated earlier, every (guaranteed) f-termination of $\mathcal{C}$ should be forward- or backward-recoverable. Such recovery is first attempted by $rh_{\mathcal{C}}(\mathcal{C})$, and if that is not successful, then by $rh_{\mathcal{U}}(\mathcal{C})$.

- Compensation of an s-termination is also delegated to $rh_{\mathcal{C}}(\mathcal{C})$ first, and to $rh_{\mathcal{U}}(\mathcal{C})$ later.

5.2 Tree Model

First, we present an extension, called *tree model*, that allows for getting a tree as a pivot graph of a composite activity. All the features of the path model are applicable here also. We describe the additional features in the following.

A. Composition

- Here also, a composition $\mathcal{C}$ is a tree and it is a part of a higher level composition $\mathcal{U}$.

- Again, a ce-logic is associated with each node, and the ce-logic is null for all leaves of $\mathcal{C}$. However, the ce-logic at non-leaf nodes may be sophisticated:
  - More than one child may be required to be executed.
  - In general, several sets of children may be specified with the requirement that one of those sets be executed.
  - These sets may be prioritized in an arbitrary way.
  - Execution of children within a set may also be prioritized in an arbitrary way.

Example 5.1 We consider an elaborate electronic shopping example, Shopping for Bedroom set, denoted SB. We use this a a running example in this subsection. It consists of the purchase followed by the delivery of a set of furnitures from among the following: bed, dresser (D), night table (N), and armoire (A). For bed, a bed frame and a mattress (M) need to be purchased. Two types of bed frames are available, called F1 and F2. For F1, a box spring (B) is also needed.

Denoting the purchase of item I as PI, the preferred purchase options are described by the following ce-logic:

- for the bed, the preference order is {PF1,PB,PM}, {PF2,PM};
- for the bedroom set, any bed and dresser and night table, or any bed and armoire, in that order, that is, ({PF1,PB,PM},{PF2,PM}), and ({PD,PN},PA).

Each of the purchased items has to be shipped. Some items need to be packed for shipping whereas some others are already in a packed form. We denote the packaging and delivery of an item I as XI and DI, respectively. When there are several options for delivery, they are denoted as DI1, DI2, etc. For shipping, we use a simple ce-logic of packaging where needed and choosing any delivery option. The activities involved in SB are shown in Figure 9. □

B. Execution

- A composite activity C is a subtree of $\mathcal{C}$ such that
  - it includes the root, some leaves of $\mathcal{C}$, and all nodes and edges in the paths from the root to those leaves in $\mathcal{C}$, and
  - the children of each non-leaf node of the subtree satisfy the ce-logic specified in $\mathcal{C}$ for that node.

  Then, $\mathcal{C}$ is the union of trees corresponding to the composite activities, and any composite activity C is a subtree of $\mathcal{C}$. However, not every subtree of $\mathcal{C}$ would correspond to a composite activity.

- A partial execution E of $\mathcal{C}$ will be represented by a subtree of $\mathcal{C}$, called *execution-tree*, consisting of all the nodes of $\mathcal{C}$ that have been executed and edges between them. If L is the set of leaves in this subtree, then the execution is denoted as $C_{[\perp,L]}$. (We use the following notation. For a given C, $C_{[k,L_k]}$ will denote the subtree of C from node p_k to the set of descendents L_k of p_k in C. For a set of nodes X, $C_{[X,Y]}$ will refer to the forest which is the union of $C_{[k,L_k]}$ for p_k in X and Y is the union of L_k for p_k in X.)

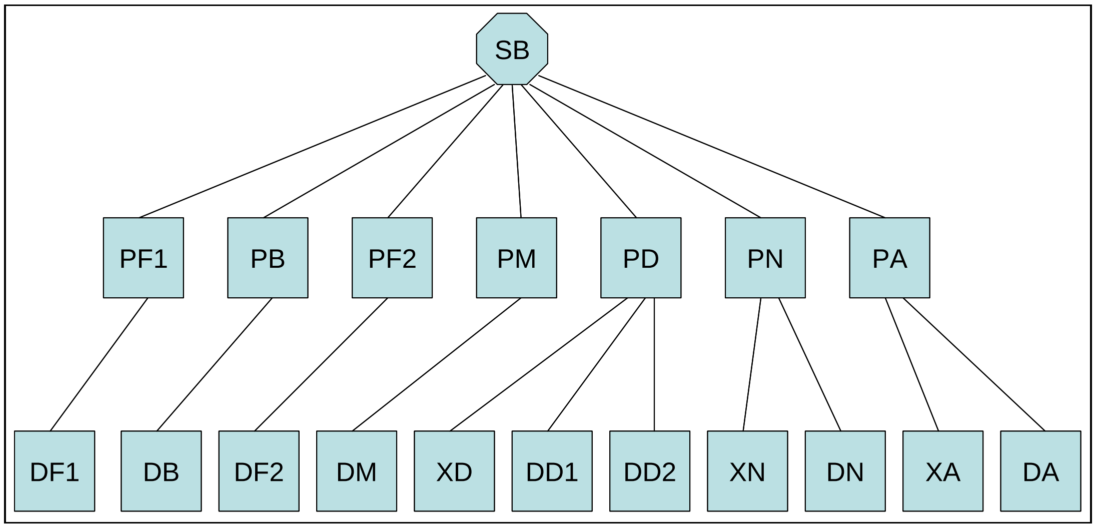

Figure 9: Activities involved in SB shopping.

- In general, the ce-logic would have been completely satisfied for some of the nodes in E. That is, a set of children corresponding to an execution choice of the ce-logic at their nodes would have been executed (successfully). These nodes are called *finished* nodes. Others are called *unfinished* nodes. For some unfinished nodes, *one* execution choice of the ce-logic would have been satisfied partially; we call them *partially unfinished* nodes. Other unfinished nodes are *totally unfinished* ones. 'Finishing' is with respect to the current execution E. We also note that since the ce-logic is null for the leaves of $\mathcal{C}$, all these nodes, if any, in the execution-tree are trivially finished nodes.

- We define the *adjusted ce-logic* for (the nodes in) E as follows:
  - null for the finished nodes;
  - same as in $\mathcal{C}$ for totally unfinished nodes; and
  - for each partially unfinished node, the part of the ce-logic of the set of yet-to-be-executed children in the execution choice chosen for that node in E.

- For p_i in E, the suffix of $\mathcal{C}$ from p_i, denoted again as $\mathcal{C}_{[i]}$, is defined as the sub-composition that contains the subtree of $\mathcal{C}$ with (i) root p_i, (ii) all the children of p_i which have not been executed, and the subtrees rooted at them, and (iii) all nodes in the subtree having the ce-logic adjusted for E.

- The suffix of the execution E is the set of suffixes $\mathcal{C}_{[i]}$ for each unfinished node p_i in E.

Example 5.2 Figure 10 shows a partial execution of the composition in Example 5.1. Here, the root node, PN, XN, DN and XD are finished nodes, PF2 and PM are totally unfinished nodes, and PD is a partially unfinished node. Figure 11 shows another partial execution where all nodes except XD are unfinished. PF2 and PM are totally unfinished.

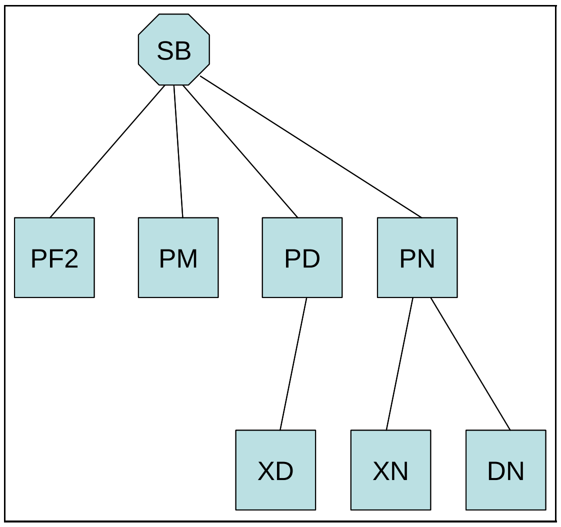

Figure 10: Partial execution.

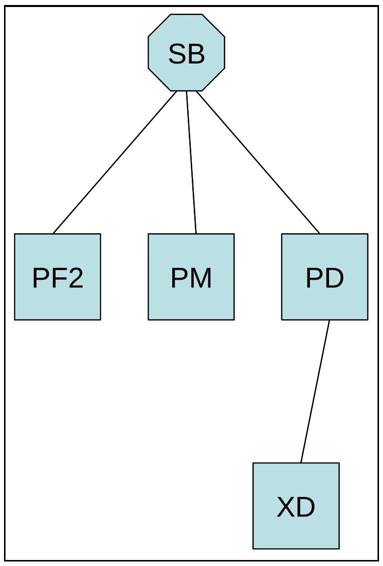

Figure 11: Another partial execution.

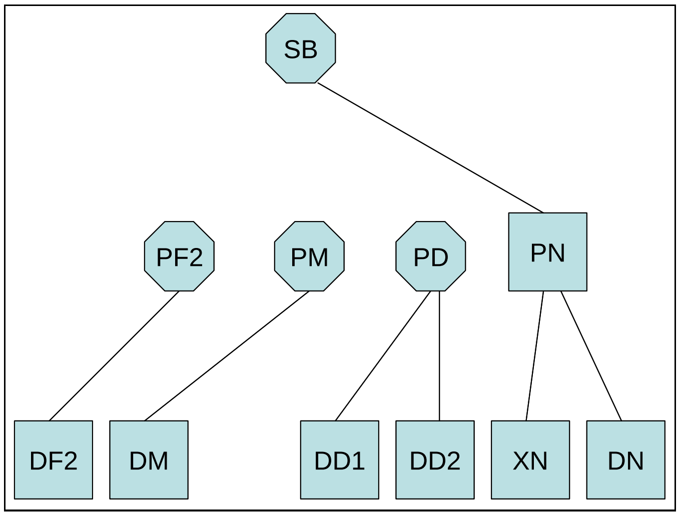

Figure 12: Suffixes of the execution in Figure 11.

The others are partially unfinished. The adjusted ce-logics at SB and PD are {PN} and one of DD1 and DD2, respectively. The adjusted ce-logics at all other nodes are the same as the original ones. Figure 12 shows the suffixes of the execution in Figure 11. $\Box$

C. Transactional Properties

- Forward-recovery of an *f*-termination E will consist of execution of the suffix of E. Again, either SU or some other provider(s) may execute the subcompositions. There could be several subcompositions, each being a tree, and different providers might be delegated for execution of different subcompositions. The subcompositions used in a forward-recovery may even be different from, but equivalent to, those in the original composition.

- Partial backward-recovery of E will consist of (logically) rolling back some of the pivots of the execution-tree. Let L' denote the set of leaves of the tree obtained after a partial backward-recovery. Clearly, L' will contain nodes in L or their ancestors. Then the *recovered part* can be expressed as $C^{-1}_{[L',L]}$, meaning that the part between L' and L has been rolled back. The compensating subcomposition that does this roll back will be denoted as $\mathcal{C}^{-1}_{[L',L]}$. Full backward-recovery should roll back all the pivots in the execution-tree and yield the null execution. Thus the recovered part will be $C^{-1}_{[\perp,L]}$.

- Backward-recovery can also be done as follows. For a given *f*-termination E, the part intended to be recovered, in terms of the set L' can be determined first. Again, the nodes in L' are the ancestors of those in L. (We use the convention that a node is an ancestor of itself.) Then the recovery $C^{-1}_{[L',L]}$ can be carried out by means of executing compensating subcompositions $\mathcal{C}^{-1}_{[j,L_j]}$ at nodes p_j in L'. Here, L_j is the subset of L which are descendents of p_j. This will roll back the descendents

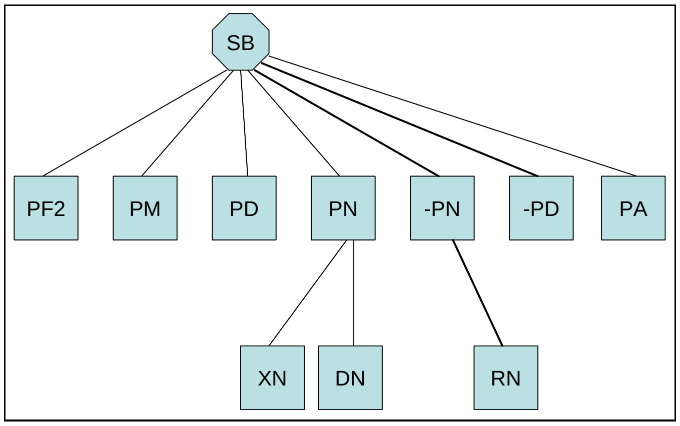

Figure 13: SB shopping execution along alternate routes.

of p_j in E. (Note that the execution of the compensating subcomposition is also according to the tree model.)

Example 5.3 In Example 5.1, starting with the partial execution in Figure 10, suppose that none of the delivery options {DD1, DD2} are feasible. Then, forward recovery would consist of finding some other option for delivering dresser D. A backward recovery of E would essentially involve compensating all the activities in the subtrees of PD and PN. Then, the next choice in the ce-logic of the root node (purchasing F2, M, and A) can be tried. If this is successful, we will obtain the tree shown in Fig. 13. Note that a compensating subtree, consisting of edges shown in thick lines, has been added to the root node. Compensating the delivery DN is implemented by "return" RN, and XD and XN are compensated by the null activities, meaning that the packagings are untouched. The purchases PD and PN are compensated by -PD and -PN. □

Example 5.4 As another example, consider the electronic shopping scenario from Figure 1 once more, where we assume that a tree root called *buying process* has been added. In the resulting tree, the non-leaf nodes include *price comparison*, *purchase*, *payment* and *delivery*, and for each we may assume that more than one child need to be executed. For example, the buyer might decide to buy an expensive piece, and the money needed for that may have to come from several sources (e.g., a bank account, an investment fund, stocks, etc.) in an order specified by the buyer. Thus, the ce-logic for the payment activity may consist of (a) collecting the money from various sources and (b) making the payment. Alternatively, it might consist of getting a loan first and then have the seller agree to a number of, say, monthly payments. □

D. Service Issues

- Fault handlers $fh_{\mathcal{C}}(\mathcal{C})$ and $fh_{\mathcal{U}}(\mathcal{C})$, and recovery handlers $rh_{\mathcal{C}}(\mathcal{C})$ and $rh_{\mathcal{U}}(\mathcal{C})$ are assigned, and have the same role, as in the path model. The fault handlers will be responsible for obtaining a guaranteed termination of an *f*-termination (exactly as in the path model), and the recovery handlers will do the forward- and backward-recoveries, and also compensation of *s*-terminations.

5.3 Multi-Level Model

So far, we have dealt with compositions at a single level. We described a composition $\mathcal{C}$ in terms of a graph $pg(\mathcal{C})$ containing the pivots of $\mathcal{C}$. (Again, each node in $pg(\mathcal{C})$ represents the corresponding (real or dummy) pivot in $\mathcal{C}$ together with the compensatable activities preceding that pivot in $\mathcal{C}$. We continue keeping this distinction implicit.) An execution of $\mathcal{C}$ yields a composite activity C which is also described by means of a pivot graph $pg(C)$. This has the pivots of $\mathcal{C}$ which have been executed. It is a path in the path model, and a tree in the tree model: we call this a *composite activity sequence* (*c-seq* in short), and *composite activity tree* (*c-tree* in short), respectively, in the following. We note that a c-seq is a c-tree also.

So far, for ease of exposition, a node in $pg(C)$ was represented the same way as in $pg(\mathcal{C})$. To describe the multi-level model unambiguously, in the following, we will use different representations in these graphs. Nodes in $pg(C)$ will be represented as p_1, p_2, p_3, etc. as before. However, nodes in $pg(\mathcal{C})$ will be represented as p_1, p_2, p_3, etc. As we have mentioned, each node p_i in the $pg(C)$ is a basic or composite activity. For a basic activity, p_i refers to p_i itself. However, for a composite activity, p_i can be taken as the composition whose execution yields p_i.

Now, our multi-level model is the following:

A. Composition

- A composition $\mathcal{C}$ is a tree as in the tree model where activities p_i are replaced by compositions p_i.
- p_i is the same as p_i for a basic activity.
- For a composite activity, p_i is a composition $\mathcal{C}_i$ which is, again, a tree in the tree model.

We now describe a composite activity. As observed in the previous section, a composition in the tree model yields a composite activity which is a tree, that is, a c-tree. Thus, a node p_i in $\mathcal{C}$ that represents a composition $\mathcal{C}_i$ yields a tree. In $\mathcal{C}$, after p_i, some other node(s) may have to be executed. They may also yield trees. To be able to put these trees together, we use the following notation.

A c-tree is converted to a one source one sink acyclic graph, by adding edges from the leaves of the tree to a single (dummy) node. This is illustrated in Figure 14. (Labelling notations are explained below.) We call this a *closed* c-tree. We consider a c-seq also as a closed c-tree; the dummy sink node is not needed.

In an execution of a multi-level composition $\mathcal{C}$, at the top level we will get a closed c-tree with nodes p_i corresponding to compositions p_i in $\mathcal{C}$. Each p_i can be replaced by a closed c-tree resulting in an execution of p_i. This can be done at every level, until all c-trees are single nodes corresponding to basic activities. We call the resulting graph a *component activity graph*, or simply a *c-graph*.

We illustrate, in the following example, a composite activity. We also illustrate how the transactional properties can be carried over to the multi-level model.

Example 5.5 Figure 15 illustrates the c-graph of a composite activity:

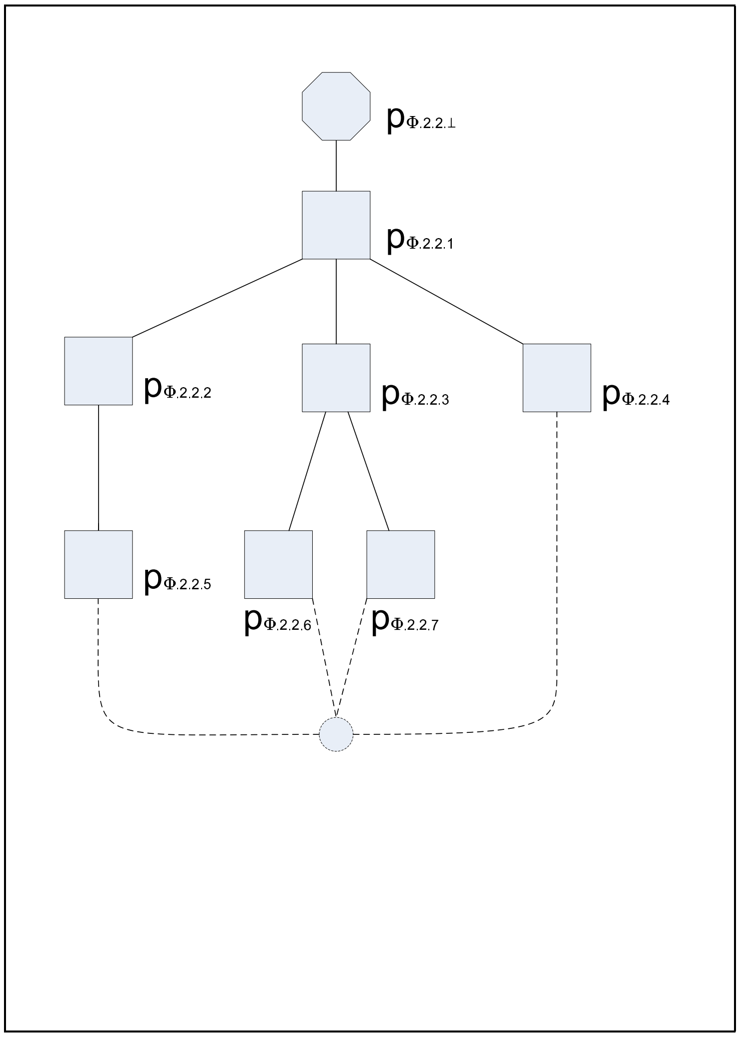

Figure 14: A c-tree for $C_{\phi.2.2}$ of Figure 15.

- Each activity is, again, either a basic activity or a composite activity. An activity is represented, as before, as p, with appropriate subscript.
- We use C_α, where α is a string, to denote a (closed) c-tree of activities, $p_{\alpha.1}, p_{\alpha.2}$, etc. In $p_{\alpha.i}$, α is a *c-tree id*, and i is the *id* of a node in that c-tree. At the outermost level, we represent the composite activity as a c-tree C_ϕ or simply C, where ϕ is the empty string. Therefore, the activities will be $p_{\phi.1}, p_{\phi.2}$, etc., or simply, p_1, p_2, etc.
- A composite activity $p_{\alpha.i}$ will consist of a set of one or more closed c-trees, denoted $C_{\alpha.i.1}, C_{\alpha.i.2}$, etc. In the following, we consider c-seq's in detail, for simplicity. Treatment of c-trees is similar.
- A particular c-seq $C_{\alpha.i.q}$ will have activities $p_{\alpha.i.q.1}, p_{\alpha.i.q.2}$, etc. If we denote $\alpha.i.q$ as α', then the c-seq is $C_{\alpha'}$, and the activities in the sequence are $p_{\alpha'.1}, p_{\alpha'.2}$, etc.

  In Figure 15, we have the following.

  - The nodes of C_ϕ are $p_{\phi.\perp}, p_{\phi.1}, p_{\phi.2}$ and $p_{\phi.3}$.
  - $p_{\phi.2}$ consists of two c-seq's, $C_{\phi.2.1}$ and $C_{\phi.2.3}$, and a closed c-tree $C_{\phi.2.2}$ shown in Figure 14.
  - $C_{\phi.2.1}$ consists of activities $p_{\phi.2.1.1}$ and $p_{\phi.2.1.2}$.
  - $p_{\phi.2.1.2}$ consists of two c-seq's $C_{\phi.2.1.2.1}$ and $C_{\phi.2.1.2.2}$.
  - An example where a node consists of just one c-seq is $p_{\phi.2.3.2}$.
- The composition for a c-seq C_α will be denoted $\mathcal{C}_\alpha$. The composition will be described as in the path model, that is, the process model in Section 2. We use the same notation as in Section 3 to denote an execution of a c-seq and its suffix composition. We assume, as before, that each c-seq starts with a dummy root pivot. An execution of the c-seq C_α, from the root to some pivot $p_{\alpha.m}$ will be denoted as $C_{\alpha.[\perp,m]}$, and its suffix will be the composition $\mathcal{C}_{\alpha.[m]}$.
- We specify *multi-level* atomicity of $\mathcal{C}$: Each activity p, at any level, must be executed atomically, and each c-seq, again at any level, must be executed atomically. For atomicity of $\mathcal{C}_{\alpha'}$, any f-termination $C_{\alpha'.[\perp,m]}$ must be either forward-recoverable or backward-recoverable. For forward-recoverability, $\mathcal{C}_{\alpha'.[m]}$ must be executed, to achieve an s-termination of $\mathcal{C}_{\alpha'}$. For backward-recoverability, $\mathcal{C}^{-1}_{\alpha'.[\perp,m]}$ is to be executed, at $p_{\alpha'.m}$, to achieve the null termination of $\mathcal{C}_{\alpha'}$. Partial forward- and backward-recovery executions can also be specified as in Section 3.
- Suppose α' is $\alpha.i.q$. Then, on s-termination of $\mathcal{C}_{\alpha.i.q}$, and on s-terminations of other $\mathcal{C}_{\alpha.i.r}$'s that constitute $p_{\alpha.i}$, we get an s-termination of $p_{\alpha.i}$. Then, further forward-recovery would consist of execution of $\mathcal{C}_{\alpha.[i]}$, to get an s-termination of $\mathcal{C}_\alpha$. This has to be continued at every level higher up.

  - In Figure 15, if the execution of $p_{\phi.2.1.2.1.2}$ fails resulting in f-termination of $C_{[\phi.2.1.2.1.\perp,\phi.2.1.2.1.1]}$, abbreviated as $C_{\phi.2.1.2.1.[\perp,1]}$, forward-recovery would consist of executing $\mathcal{C}_{\phi.2.1.2.1.[1]}$ to get an s-termination of $\mathcal{C}_{\phi.2.1.2.1}$;

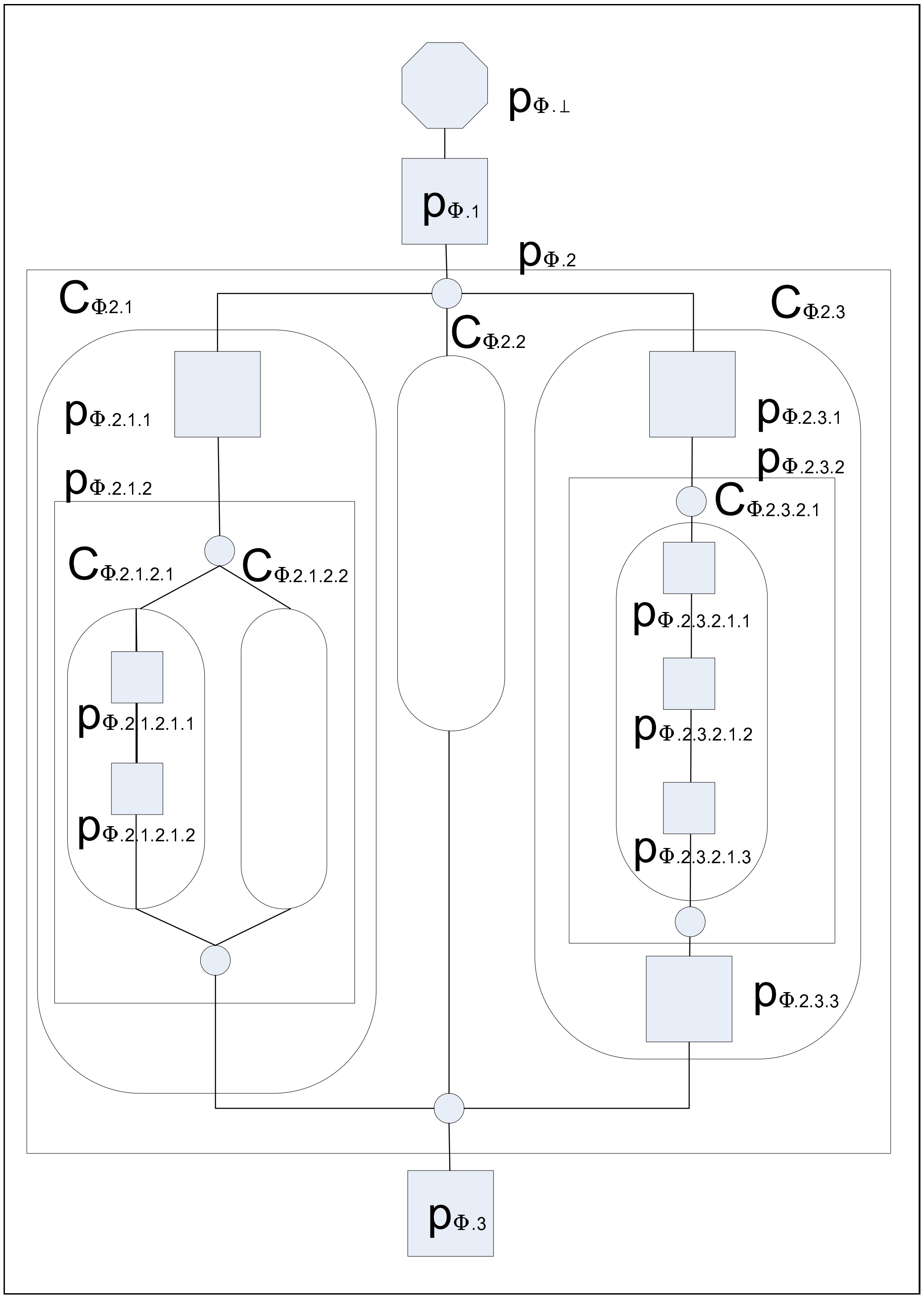

Figure 15: A composite activity in the multi-level model.

  - An s-termination of $C_{\phi.2.1.2.2}$, in forward-recovery or normal execution, will result in an s-termination of $p_{\phi.2.1.2}$; and so on.

- Again, suppose α' is $\alpha.i.q$. On backward-recovery of $C_{\alpha.i.q.[\perp,m]}$ getting the null execution of $\mathcal{C}_{\alpha.i.q}$, backward-recovery of other $C_{\alpha.i.r}$'s that constitute $p_{\alpha.i}$ can be carried out, to achieve backward-recovery of $p_{\alpha.i}$. Then, backward-recovery of $C_{\alpha.[\perp,i']}$, where we assume that the node preceding $p_{\alpha.i}$ is $p_{\alpha.i'}$, will result in the null termination of $\mathcal{C}_{\alpha}$. This can be carried out recursively at every level higher up, to eventually achieve the null termination of $\mathcal{C}$.

  - Referring to Figure 15 again, backward-recovery of $C_{\phi.2.1.2.1.[\perp,1]}$ will result in the null termination of $\mathcal{C}_{\phi.2.1.2.1}$. Then, backward-recovery of $\mathcal{C}_{\phi.2.1.2.2}$ will result in the null termination of $p_{\phi.2.1.2}$, and so on.

□

B. Execution

- A composite activity C of a multi-level composition $\mathcal{C}$ is a c-graph such that
  - at the outermost level, it is a closed c-tree, with nodes p_i corresponding to compositions p_i in $\mathcal{C}$, and
  - each composite activity p_i is replaced by a closed c-tree resulting in an execution of p_i, and
  - this process carried out until all activities are basic.
- Partial execution is considered as in the tree model, level by level, in nested fashion (as illustrated in the Example 5.5).

C. Transactional Properties

- As stated in Section 3.5, the transactional properties can be carried over from one level to another.
- At any individual level, for each p_i, the transactional properties (s-termination, f-termination, compensation of s-termination, forward- and backward-recovery of f-termination, etc.) discussed in Section 5.2 are applicable to the execution-tree of p_i.
- Then, as illustrated in Example 5.5, after the recovery of p_i, the recovery efforts at the parent level execution will continue.

D. Service Issues

- Again, fault and recovery handlers are employed, exactly as before, for every parent-child pair.

5.4 Top-Down Composition

Clearly, a service user would not be interested in composing complex services by starting bottom-up from elementary ones, as we have done so far. Instead, a user would be interested in obtaining a high-level description of each service he or she may need, and then start composing at that level. We imagine that, typically, graphical interfaces will be used to that end, for example an interface where the individual service can be described as a *Petri net* [20]. What would then be needed is a way to map each task represented by an activity or a process in a Petri net to a service appropriately, taking availability, user preferences, timing, costs, etc. into account.

In this section, we show that our model facilitates top-down compositions also. Recall that in the tree model we have:

- A composition $\mathcal{C}$ is a tree;
- At each node p_i, several children may need to be executed, and the execution preferences are described by ce-logic at p_i; and
- After the execution of a set of children satisfying the ce-logic, execution continues with the children of those children.

We now define *descendent execution logic*, abbreviated as *de-logic*, at p_i, as the union of the ce-logic of p_i and all its descendents. Note that the de-logic describes not just the individual children nodes but also (transitively) their subtrees which need to be executed. The execution preferences in the ce-logics at various nodes become collectively the execution preferences in the de-logic.

Then, the execution preferences at each node can be described by the more general de-logic, instead of ce-logic. In fact, we can carry this idea further. If we take a choice of children in the ce-logic at p_i, for each child in that choice, select a choice in the ce-logic of that child, and continue this recursively, we will get a c-graph that reflects the choices made at every level. Different combinations will give rise to different c-graphs. Then, execution preferences at p_i can be stated, in a higher level, in terms of such c-graphs.

Though we defined de-logic from ce-logic, we can also start with de-logic or even (perhaps an abstract description of) the desired c-graphs, and then derive the ce-logic at various nodes. This would be a top-down approach.

6 Discussion

In this paper, we have extended the model originally proposed in [23] to a multi-level model for Web service composition that enables description of desirable transactional properties at each level of the composition. It has been widely accepted that the traditional ACID properties need to be relaxed for transactions in the Web service environment. A few relaxations have appeared in the literature. We discuss some of them in the following and show that the relaxations can be explained neatly in our model.

1. The requirement of atomicity of a composition (with multiple pivots) has been stated in the literature, for example, in [16, 18, 6].

(a) In [16], Mikalsen, et al. introduce *transactional attitudes* "to explicitly describe the otherwise implicit transactional semantic, capabilities, and requirements of individual applications". They consider *Client Transactional Attitudes* (CTAs) and *Provider Transactional Attitudes* (PTAs). One CTA, called *flexible atom* (FA), is given. Here, "a set of client actions (provider transactions) are grouped into an atomic group that can have one out of a set of defined group outcomes; that is to say, some actions are declared *critical* to the success of the transaction, whereas others are part of the transaction though not pivotal to its success. The client specifies the acceptable outcomes as an *outcome condition*, described in terms of the success or failure of the individual actions, and when ready (i.e., after executing the forward operations of these actions), requests the completion of the flexible atom according to that condition". We note that this CTA resembles the specification in our model, by SU to SC, of the s-terminations of $\mathcal{C}$ relative to $\mathcal{U}$ and the requirement of atomicity. We can specify, in addition, the retriability requirement also as a CTA. Three PTAs, *pending-commit, group-pending-commit,* and *commit-compensate*, are described in [16]. The first two relate to providing the prepared-to-commit states for single activity or a group of activities, resp., and the last describes the facility for compensation after the commitment of an activity. Compensatability of f-terminations and s-terminations, atomicity and retriability are some possible additional PTAs. In fact, even the guaranteed termination property is a PTA.

(b) The s-termination set concept appears in [18] as follows. Here also a composite task consists of several tasks each of which could be atomic or composite. Different successful executions of a composite task are specified in terms of successful executions of a set of (component) *mandatory* tasks and a set of *desirable* tasks.

(c) In [6], a set of activities that need to be executed atomically is grouped into a *transactional region.*

2. The OASIS Business Transaction Protocol[3] (BTP) allows a type of composite activity called *cohesion.* It contains a set of activities that can be performed autonomously by different service providers. An s-termination of the composite activity is determined, eventually, by the outcomes of the individual activities. As a result, some of the activities done successfully may have to be undone. It is also possible that some participants "leave", that is, some activities are eliminated from the cohesion. Thus the composition is very dynamic. A coordinated termination, involving commit of certain activities and abort of some activities, is facilitated.

The multi-activity node in our model can depict cohesion effectively. Potential concurrent execution can be described by weak order among the activities. The relaxed atomicity of the cohesion can be translated to s-terminations and the atomicity of the multi-activity composite node.

As illustrated above, our model accommodates many proposals in the literature. Furthermore, our model can explain the context, for example, the purpose of compensation across levels, for the transactional activities.

We note also that whereas compensatability and compensation have been considered at some length in the literature, the concept of retriability has not been discussed, at least explicitly. In our model, both compensatability and retriability are complementary

[3] `http://www.oasis-open.org/committees/business-transactions/documents/primer/Primerhtml/`

towards achieving the atomicity of a composite activity. A related issue, namely, suffix executability has also not been discussed in the literature.

We conclude by mentioning that a number of issues are related to what has been discussed above, and that these issues can now be made precise in the framework of our model:

- Atomicity of an activity will serve as a non-functional trait of a service provider. Atomicity and suffix executability may be taken into account while dealing with compatability and substitutability of services [10].
- In the design of business processes, responsibilities for the execution of business activities (roles) must be specified [19]. Responsibility for atomicity or guaranteed termination will also be a part of the specification.
- It is possible that a service provider offers different levels of atomicity to different customers, and at different costs.
- As stated earlier, SU does not need to know $\mathcal{C}$, but does need $pg(\mathcal{C})$ (especially when SU takes responsibility for executing suffixes of f-terminations). Here, $pg(\mathcal{C})$ can be considered as containing information about *what* are done in $\mathcal{C}$, without exposing *how* they are done. Thus, $pg(\mathcal{C})$ represents a *glass box* view of $\mathcal{C}$, according to the distinction suggested by [4].

Future works along the lines established in this paper may stem from the fact that we have here decided to associate compensatability and retriability with composite activities instead of just individual transactions; what new consequences can be derived from this? Guaranteed termination is implied by our model, but what about termination within predefined bounds (e.g., meeting a deadline, not exceeding a given budget, etc.)? Another question is whether it is possible to quantify the "added value" that is supposed to be brought along by a service composition.

References

[1] Abiteboul, S., V. Vianu, B.S. Fordham, Y. Yesha: Relational Transducers for Electronic Commerce. JCSS 61, 2000, 236-269.

[2] Alonso, G., Casati, F., Kuno, H., Machiraju, V.: *Web Services - Concepts, Architectures and Applications*, Springer 2004.

[3] Alonso, G., R. Vingralek, D. Agrawal, Y. Breitbart, A. El Abbadi, H.-J. Schek, G. Weikum: Unifying Concurrency Control and Recovery of Transactions. *Information Systems* **19**, 1994, 101–115.

[4] Battle, S.: Boxes: black, white, grey and glass box view of web-services, HP Lab Report HPL-2003-30.

[5] Bultan, T., Fu, X., Hull, R, Su, J.: Conversation Specification: A New Approach to Design and Analysis of E-Service Composition. *Proc. WWW 2003.*

[6] Casati, F., Ilnicki, S., Jin, L., Krishnamoorthy, V., Shan, M-C.: Adaptive and Dynamic Service Composition in eFlow, HP Labs Report HPL-2000-39.

[7] Casati, F., Dayal, U., eds.: *Special Issue on Web Services.* IEEE Bulletin of the Technical Committee on Data Engineering, 25 (4), December 2002.

[8] Christophides, V., Hull, R., Karvounarakis, G., Kumar, A., Tong, G., Xiong, M.: Beyond Discrete E-Services: Composing Session-Oriented Services in Telecommunications. *Proc. 2nd Int. Workshop on Technologies for E-Services* (TES) 2001, Springer LNCS 2193, 58–73.

[9] Colombo, E., Francalanci, C., Pernici, B.: Modeling Coordination and Control in Cross-Organizational Workflows. *Proc. CoopIS-DOA-ODBASE* 2002, Springer LNCS 2519, 91–106.

[10] de Antonellis, V., Melchiori, M., Pernici, B., Plebani, P.: A Methodology for e-Service Substitutability in a Virtual District Environment, CAiSE '03, Klagenfurt, Austria.

[11] Fauvet, M.-C., Dumas, M., Benatallah, B., Paik, H.-Y.: Peer-to-Peer Traced Execution of Composite Services. *Proc. 2nd Int. Workshop on Technologies for E-Services* (TES) 2001, Springer LNCS 2193, 103–117.

[12] Fu, X., Bultan, T., Su, J.: Formal Verification of e-Services and Workflows. *Proc. Int. Workshop on Web Services, E-Business, and the Semantic Web* (WES) 2002, Springer LNCS 2512, 188–202.

[13] Huhns, M.N., Singh, M.P.: Service-Oriented Computing: Key Concepts and Principles; IEEE Internet Computing Jan./Feb. 2005, pp. 75–81.

[14] Hull, R., Benedikt, M., Christophides, V., Su, J.: E-Services: A Look Behind the Curtain. *Proc. 22nd ACM Symposium on Priciples of Database Systems* (PODS) 2003, San Diego, CA.

[15] Mecella, M., Presicce, F.P., Pernici, B.: Modeling E-Service Orchestration through Petri Nets. *Proc. 3rd Int. Workshop on Technologies for E-Services* (TES) 2002, Springer LNCS 2444, 38–47.

[16] Mikalsen, T., Tai, S., Rouvellou, I.: Transactional Attitudes: Reliable Composition of Autonomous Web Services. *Workshop on Dependable Middleware-based Systems* (WDMS '02), Washington DC (2002).

[17] Newcomer, E.: *Understanding Web Services: XML, WSDL, SOAP, and UDDI.* Addison-Wesley 2002.

[18] Pires, P.F., Benevides, M.R.F., Mattoso, M.: Building Reliable Web Services Compositions. *Proc. Workshop on the Web, Web-Services, and Database Systems* 2002, Springer LNCS 2593, 59–72.

[19] Papazoglou, P., Yang, J.: Design Methodology for Web Services and Business Processes. *Proc. 3rdInt. Workshop on Technologies for E-Services* (TES) 2002, Springer LNCS 2444, 54–64.

[20] Reisig, W.: *Petri Nets, An Introduction.* EATCS, Monographs on Theoretical Computer Science, W. Brauer, G. Rozenberg, A. Salomaa (Eds.), Springer Verlag, Berlin, 1985.

[21] Schek, H.-J., G. Weikum, H. Ye: Towards a Unified Theory of Concurrency Control and Recovery. In *Proc. 12th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* 1993, 300–311.

[22] Singh, M.P., Huhns, M.N.: *Service-Oriented Computing — Semantics, Processes, Agents.* John Wiley & Sons, Ltd., England, 2005

[23] Schuldt, H., Alonso, G., Beeri, C., Schek, H.J.: Atomicity and Isolation for Transactional Processes. *ACM Transactions on Database Systems* 27, 2002, 63–116.

[24] Schuler, C., Schuldt, H., Schek, H.J.: Supporting Reliable Transactional Business Processes by Publish-Subscribe Techniques. *Proc. 2nd Int. Workshop on Technologies for E-Services* (TES) 2001, Springer LNCS 2193, 118–131.

[25] Vossen, G.: Have Service-Oriented Architectures Taken a Wrong Turn Already?; in: Tjoa, A. Min; Xu, Li; Chaudhry, Sohail (Eds.) *Research and Practical Issues of Enterprise Information Systems*; Proc. of The IFIP TC 8 International Conference on Research and Practical Issues of Enterprise Information Systems (CONFENIS) 2006, Vienna, Austria; Series: IFIP International Federation for Information Processing , Vol. 205

[26] Vidyasankar, K., Vossen, G.: A Multi-Level Model for Web Service Composition. *Proc. 3rd IEEE International Conference on Web Services* 2004, San Diego, USA, 462–469.

[27] Weikum, G., Vossen, G.: *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*; Morgan-Kaufmann Publishers, San Francisco, CA, 2002.