The Assurance Point Model for Consistency and Recovery in Service Composition

Susan D. Urban[1], Le Gao[1], Rajiv Shrestha[1], Yang Xiao[2], Zev Friedman[1], Jonathan Rodriguez[1]

[1]Texas Tech University
Department of Computer Science
Lubbock, TX 79424
susan.urban@ttu.edu le.gao@ttu.edu rajiv.shrestha@ttu.edu z.friedman@ttu.edu
jonathan.rodriguez@ttu.edu

[2]Arizona State University
School of Computing and Informatics
Department of Computer Science and Engineering
Tempe, AZ 85287-8809
yang.xiao@asu.edu

ABSTRACT:

This research has defined an abstract execution model for establishing user-defined correctness and recovery in a service composition environment. The service composition model defines a hierarchical service composition structure, where a service is composed of atomic and/or composite groups. The model provides multi-level protection against service execution failure by using compensation and contingency at different composition granularity levels. The model is enhanced with the concept of assurance points (APS) and integration rules, where APs serve as logical and physical checkpoints for user-defined consistency checking, invoking integration rules that check pre and post conditions at different points in the execution process. The unique aspect of APs is that they provide intermediate rollback points when failures occur, thus allowing a process to be compensated to a specific AP for the purpose of rechecking pre-conditions before retry attempts. APs also support a dynamic backward recovery process, known as cascaded contingency, for hierarchically nested processes in an attempt to recover to a previous AP that can be used to invoke contingent procedures or alternate execution paths for failure of a nested process. As a result, the assurance point approach provides flexibility with respect to the combined use of backward and forward recovery options. Petri Nets have been used to define the semantics of the assurance point approach to service composition and recovery. A comparison to the BPEL fault handler is also provided.

KEY WORDS:
service composition, compensation, contingency, rollback, retry, backward recovery, forward recovery, user-defined correctness constraints, pre-conditions, post-conditions, Petri Net formalization

INTRODUCTION

In a service-based architecture, a process is composed of a series of calls to distributed Web services and Grid services that collectively provide some specific functionality of interest to an

application (Singh & Huhns, 2005). In a traditional, data-oriented, distributed computing environment, a distributed transaction is used to provide certain correctness guarantees about the execution of a transaction over distributed data sources. In particular, a traditional, distributed transaction provides all-or-nothing behavior by using the two-phase commit protocol to support atomicity, consistency, isolation, and durability (ACID) properties (Kifer, Bernstein, & Lewis, 2006). A process in a service-oriented architecture, however, is not a traditional ACID transaction due to the loosely-coupled, autonomous, and heterogeneous nature of the execution environment. When a process invokes a service, the service performs its function and then terminates, without regard for the successful termination of the global process that invoked the service. If the process fails, reliable techniques are needed to either 1) restore the process to a consistent state or 2) correct critical data values and continue running.

Techniques such as compensation and contingency have been used as a form of recovery in past work with transactional workflows (e.g., Worah & Sheth, 1997) and have also been introduced into recent languages for service composition (e.g., Lin & Chang, 2005). In the absence of a global log file, compensation provides a form of backward recovery, executing a procedure that will "logically undo" the affects of completed and/or partially executed operations. Contingency is a form of forward recovery, providing an alternate execution path that will allow a process to continue execution. Some form of compensation may be needed, however, before the execution of contingency plans. Furthermore, nested service composition specifications can complicate the use of compensating and contingent procedures. To provide a reliable service composition mechanism, it is important to fully understand the semantics and complementary usage of compensation and contingency, as well as how they can be used together with local and global database recovery techniques and nested service composition specifications. Service composition models also need to be enhanced with features that allow processes to assess their execution status to support more dynamic ways of responding to failures, while at the same time validating correctness conditions for process execution.

This research has defined an abstract execution model for establishing user-defined correctness and recovery in a service composition environment. The research was originally conducted in the context of the DeltaGrid project, which focused on building a semantically-robust execution environment for processes that execute over Grid Services (Xiao, 2006; Xiao, Urban, & Dietrich, 2006; Xiao, Urban, & Liao, 2006; Xiao & Urban, 2008). The service composition model defines a hierarchical service composition structure, where a service is composed of atomic and/or composite groups. An atomic group is a service execution with optional compensation and contingency procedures. A composite group is composed of two or more atomic and/or composite groups and can also have optional compensation and contingency procedures. A unique aspect of the model is the provision for multi-level protection against service execution failure by using compensation and contingency at different composition granularity levels, thus maximizing the potential for forward recovery of a process when failure occurs. The work in (Xiao and Urban, 2009) presents the full specification of the model using state diagrams and algorithms to define the semantics of compensation and contingency in the recovery process.

Our more recent work has extended the DeltaGrid service composition and recovery model in (Xiao et al., 2009) with the concept of *Assurance Points* (APs) and *integration rules* to provide a more flexible way of checking constraints and responding to execution failures. An AP is a combined logical and physical checkpoint. As a physical checkpoint, an AP provides a way to store data at critical points in the execution of a process. Unlike past work with checkpointing, such as that of (Dialini, Miles, Moreau, Roure & Luck, 2002; Luo, 2000) where checkpoints are used to port an execution to a different platform as part of fault tolerant architectures, our work focuses on the use of APs for user-defined consistency checking and rollback points that can be used to maximize forward recovery options when failures occur. In particular, an AP provides an execution milestone that interacts with integration rules. The data stored at an AP is passed as

parameters to integration rules that are used to check pre-conditions, post-conditions, and other application conditions. Failure of a pre or post-condition or the failure of a service execution can invoke several different forms of recovery, including backward recovery of the entire process, retry attempts, or execution of contingent procedures. The unique aspect of APs is that they provide intermediate rollback points when failures occur that allow a process to be compensated to a specific AP for the purpose of rechecking pre-conditions before retry attempts. APs also support a dynamic backward recovery process, known as cascaded contingency, for hierarchically nested processes in an attempt to recover to a previous AP that can be used to invoke contingent procedures or alternate execution paths for failure of a nested process.

After presenting related work, this chapter first reviews the basic features of the hierarchical service composition model presented in (Xiao et al., 2009), together with an on-line shopping case study and a summary of an evaluation framework that was developed to demonstrate the functionality of the recovery algorithms. An understanding of these basic features is a precursor to a description of the extended model. The AP and integration rule extensions to the model are then presented, with a focus on the different forms of recovery actions as defined in (Shrestha, 2010). Petri Nets are then used to formalize the semantics of the extended model, including atomic and composite groups with shallow and deep compensation integrated with assurance points and rollback, retry, and cascaded contingency recovery activities. After discussing a prototype implementation of an execution engine for the model, a comparison of the approach to the fault handling and recovery procedures of BPEL is presented, demonstrating that the approach presented in this chapter provides a cleaner, hierarchical approach to compensation order rather than the "zigzag" behavior of BPEL as described in (Khalaf, Roller, & Leymann, 2009). The primary contribution of this research is found in the enhancements that assurance points and integration rules lend to the service composition and recovery process. In particular, the assurance point approach provides explicit support for user-defined constraints with rule-driven recovery actions for compensation, retry, and contingency procedures that support flexibility with respect to the combined use of backward and forward recovery options.

The paper concludes with a summary and discussion of future research.

RELATED WORK

The traditional notion of transactions with ACID properties is too restrictive for the types of complex transactional activities that occur in distributed applications, primarily because locking resources during the entire execution period is not applicable for Long Running Transactions (LRTs) that require relaxed atomicity and isolation (Cichocki, 1998). Advanced transaction models (ATMs) have been proposed to better support LRTs in a distributed environment (deBy, Klas, & Veijalainen, 1998; Elmagarmid, 1992), including the Nested Transaction Model, the Open Nested Transaction Model, Sagas, the Multi-level Transaction Model and the Flexible Transaction Model. These advanced transaction models relax the ACID properties of traditional transaction models to better support LRTs and to provide a theoretical basis for further study of complex distributed transaction issues, such as failure atomicity, consistency, and concurrency control. These models have primarily been studied from a research perspective and have not adequately addressed recovery issues for transaction failure dependencies in loosely-coupled distributed applications.

Transactional workflows contain the coordinated execution of multiple related tasks that support access to heterogeneous, autonomous, and distributed data through the use of selected transactional properties (Worah et al., 1997). Transactional workflows require externalizing intermediate results, while at the same time providing concurrency control, consistency guarantees, and a failure recovery mechanism for a multi-user, multi-workflow environment. Concepts such as rollback, compensation, forward recovery, and logging have been used to

achieve workflow failure recovery in projects such as the ConTract Model (Wachter & Reuter, 1992), the Workflow Activity Model (Eder & Liebhart, 1995), the CREW Project (Kamath & Ramamritham, 1998), the METEOR Project (Worah et al., 1997), and Units of Work (Bennett et al., 2000). These projects expose the weaknesses of using ATM techniques alone to support reliable transactional workflow execution, mainly due to the complexity of workflows. Previous work also shows the weakness of ATMs in support of the isolation, failure atomicity, timed constraints, and liveness requirements of distributed transactional workflows (Kuo, Fekete, Greenfield & Jang, 2002). Similar concerns are voiced in papers addressing transactional issues for traditional workflow systems (Alonso, Hagen, Schek, & Tresh, 1997; Kamath and Ramamritham 1996; Kamath et al.,1998) as well as workflow for loosely-coupled distributed sources such as Web Services (Fekete, Greenfield, Kuo, & Jang, 2002; Kuo et al. 2002). More comprehensive solutions are needed to meet the requirements of transactional workflows (Worah et al. 1997).

In the Web Services platform, WS-Coordination (2005) and WS-Transaction (2005) are two specifications that enable the transaction semantics and coordination of Web Service composition using Atomic Transactions for ACID transactions and Business Activity for long running business processes. The Web Services Transaction Framework (WSTx) (Mikalsen, Tai, & Rouvellou, 2002) introduces *Transactional Attitudes*, where service providers and clients declare their individual transaction capabilities and semantics. Web Service Composition Action (WSCA) (Tartanoglu, 2003) allows a participant to specify actions to be performed when other Web Services in the WSCA signal an exception. An agent based transaction model (Jin & Goshnick, 2003) integrates agent technologies in coordinating Web Services to form a transaction. Tentative holding is used in (Limthanmaphon & Zhang, 2004) to achieve a tentative commit state for transactions over Web Services. Acceptable Termination States (Bhiri, Perrin, & Godart, 2005) are used to ensure user-defined failure atomicity of composite services, where application designers specify the global composition structure of a composite service and the acceptable termination states.

More recently, events and rules have been used to dynamically specify control flow and data flow in a process by using Event Condition Action (ECA) rules (Paton & Diaz, 1999). ECA rules have also been successfully implemented for exception handling in work such as (Brambilla, Ceri, Comai, & Tziviskou, 2005; Liu, Li, Huang, & Xiao, 2007). The work in Liu et al. (2007) uses ECA rules to generate reliable and fault-tolerant BPEL processes to overcome the limited fault handling capability of BPEL. Our work with assurance points also supports the use of rules that separate fault handling from normal business logic. Combined with assurance points, integration rules are used to integrate user-defined consistency constraints with the recovery process.

Several efforts have been made to enhance the BPEL fault and exception handling capabilities. BPEL4Job (Tan, Fong, & Bobroff, 2007) addresses fault-handling design for job flow management with the ability to migrate flow instances. The work in (Modafferi & Conforti, 2006) proposes mechanisms like external variable setting, future alternative behavior, rollback and conditional re-execution of the Flow, timeout, and redo mechanisms for enabling recovery actions using BPEL. The work in (Modafferi, Mussi, & Pernici, 2006) presents the architecture of the SH-BPEL engine, a Self-Healing plug-in for WS-BPEL engines that augments the fault recovery capabilities in WS-BPEL with mechanisms such as annotation, pre-processing, and extended recovery. The Dynamo (Baresi, Guiea, & Pasquale, 2007) framework for the dynamic monitoring of WS-BPEL processes weaves rules such as pre/post conditions and invariants into the BPEL process. Most of these projects do not fully integrate constraint checking with a variety of recovery actions as in our work to support more dynamic and flexible ways of reacting to failures. Our research demonstrates the viability of variegated recovery approaches within a BPEL-like execution environment.

In checkpointing systems, consistent execution states are saved during the process flow. During failures and exceptions, the activity can be rolled back to the closest consistent checkpoint to move the execution to an alternative platform (Dialini et al. 2002; Luo, 2000]. The AP concept presented in this paper also stores critical execution data, but uses the data as parameters to rules that perform constraint checking and invoke different types of recovery actions.

Aspect-oriented programming (AOP) is another way of modularizing and adding flexibility to service composition through dynamic and autonomic composition and runtime recovery. In AOP, aspects are weaved into the execution of a program using join points to provide alternative execution paths (Charfi & Mezini, 2007). The work in (Charfi & Mezini, 2006) illustrates the application of aspect-oriented software development concepts to workflow languages to provide flexible and adaptable workflows. AO4BPEL (Charfi et al., 2007) is an aspect-oriented extension to BPEL that uses AspectJ to provide control flow adaptations (Kiczales et al., 2001). Business rules can also be used to provide more flexibility during service composition. APs as described in this paper are similar to join points, with a novel focus on using APs to access process history data in support of constraint checking as well as flexible and dynamic recovery techniques.

Due to the distributed nature of services, service composition is often inflexible and highly vulnerable to errors. Even BPEL, the de-facto standard for composing Web services, still lacks sophistication with respect to handling faults and events. Our research is different than related work by providing a hierarchical composition structure with support for user-defined constraints with the use of rules for pre and post conditions. In addition, the AP model integrates the rules with different recovery actions as well as user-defined compensation and contingency. Thus, our AP model attempts to provide more flexible recovery process semantics with a focus on user-defined constraints, which is a combination of features that are not available in current or past research.

OVERVIEW OF THE DELTAGRID SERVICE COMPOSITION AND RECOVERY MODEL

Before describing the use of APs and integration rules, this section first outlines the basic features of the model in the context of the DeltaGrid project. This section first elaborates on atomic and composite group recovery issues and then presents a case study to illustrate the basic concepts of the model.

Hierarchical Service Composition and Recovery

In the DeltaGrid environment, a process is hierarchically composed of different types of execution entities. Table 1 shows seven execution entities defined in the service composition model. Figure 1 uses a UML class diagram to graphically illustrate the composition relationship among these execution entities. A process is a top-level execution entity that contains other execution entities. A process is denoted as p_i, where p represents a process and the subscript i represents a unique identifier of the process. An Operation represents a service invocation, denoted as $op_{i,j}$, such that op is an operation, i identifies the enclosing process p_i, and j represents the unique identifier of the operation within p_i. Compensation (denoted as $cop_{i,j}$) is an operation intended for backward recovery, while contingency (denoted as $top_{i,j}$) is an operation used for forward recovery.

An atomic group and a composite group are logical execution units that enable the specification of processes with complex control structure, facilitating service execution failure recovery by adding scopes within the context of a process execution. An atomic group contains an operation, an optional compensation, and an optional contingency. A composite group may contain multiple atomic groups, and/or multiple composite groups that execute sequentially or in parallel. A composite group

can have its own compensation and contingency as optional elements. A process is essentially a top level composite group.

An atomic group is denoted as $ag_{i,j}$, while a composite group is denoted as $cg_{i,k}$. The subscripts in the atomic group and composite group notation indicate the nesting levels of an atomic group or composite group within the context of a process. For example, a process p_i is a top-level composite group denoted as cg_1. Assume cg_1 contains two composite groups and an atomic group. The enclosed composite groups are denoted as $cg_{1,1}$ and $cg_{1,2}$, and the atomic group is denoted as $ag_{1,3}$. Assume $cg_{1,1}$ contains two atomic groups. These atomic groups are denoted as $ag_{1,1,1}$ and $ag_{1,1,2}$, respectively.

Table 1. Execution Entities

Entity Name	**Definition**
Operation	A service invocation, denoted as $op_{i,j}$
Compensation	An operation that is used to undo the effect of a committed operation, denoted as $cop_{i,j}$
Contingency	An operation that is used as an alternative of a failed operation ($op_{i,j}$), denoted as $top_{i,j}$
Atomic Group	An execution entity that is composed of a primary operation ($op_{i,j}$), an optional compensation ($cop_{i,j}$), and an optional contingency operation ($top_{i,j}$), denoted as $ag_{i,j} = <op_{i,j} [,cop_{i,j}] [,top_{i,j}]>$
Composite Group	An execution entity that is composed of multiple atomic groups or other composite groups. A composite group can also have an optional compensation and an optional contingency, denoted as $cg_{i,k} = <(ag_{i,k,m} \mid cg_{i,k,n})^+ [,cop_{i,k}] [,top_{i,k}])>$
Process	A top level composite group, denoted as p_i
DE-rollback	An action of undoing the effect of an operation by reversing the data values that have been changed by the operation to their before images, denoted as $dop_{i,j}$

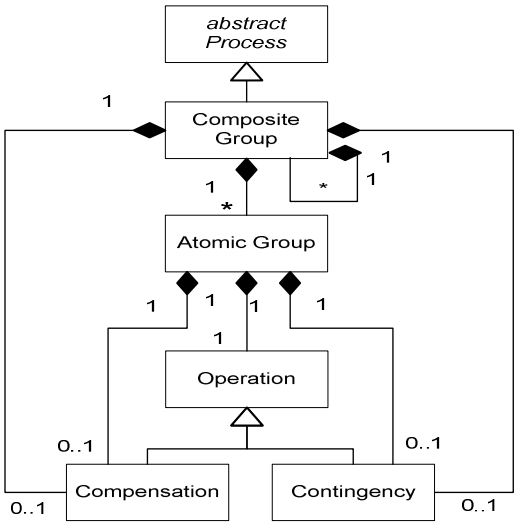

Figure 1. Service Composition Structure

The only execution entity not shown in Figure 1 is the *DE-rollback* entity. DE-rollback is a system-initiated operation that is unique to the DeltaGrid environment. Services in the DeltaGrid environment, referred to as *Delta-Enabled Grid Services (DEGS)*, are extended with the capability of recording incremental data changes, known as *deltas* (Blake, 2005; Urban, Xiao, Blake, & Dietrich, 2009). Deltas are extracted from service executions and externalized by streaming data changes out of the database to a Process History Capture System (PHCS) (Xiao et al., 2006). The PHCS merges deltas from distributed sources into a time-ordered schedule of the data changes associated with concurrently executing processes. Deltas can then be used to backward recover an operation through a process known as *Delta-Enabled Rollback (DE-Rollback)* (Xiao, 2006). DE-rollback can only be used, however, if certain recoverability conditions are satisfied, with the PHCS. The merged schedule of deltas providing the basis for determining the applicability of DE-rollback based on data dependencies among concurrently executing processes. A recoverable schedule requires that, at the time when each transaction t_i commits, every other transaction t_j that wrote values read by t_i has already committed (Kifer et al., 2006). Thus a recoverable schedule does not allow dirty writes to occur. In a recoverable schedule, a transaction t_1 cannot be rolled back if another transaction t_2 reads or writes data items that have been written by t_1, since this may cause lost updates. When interleaved access to the same data item disables the applicability of DE-rollback on an operation, compensation can be used to semantically undo the effect of the operation.

Figure 2 shows an abstract view of a sample process definition based on the DeltaGrid service composition structure. A process p_1 is the top level composite group cg_1. The process p_1 is composed of two composite groups $cg_{1,1}$ and $cg_{1,2}$, and an atomic group $ag_{1,3}$. Similarly, $cg_{1,1}$ and $cg_{1,2}$ are composite groups that contain atomic groups. Each atomic/composite group can have an optional compensation plan and/or contingency plan. Operation execution failure can occur on an operation at any level of nesting. The purpose of the DeltaGrid service composition model is to automatically resolve operation execution failure using compensation, contingency, and DE-rollback at different composition levels.

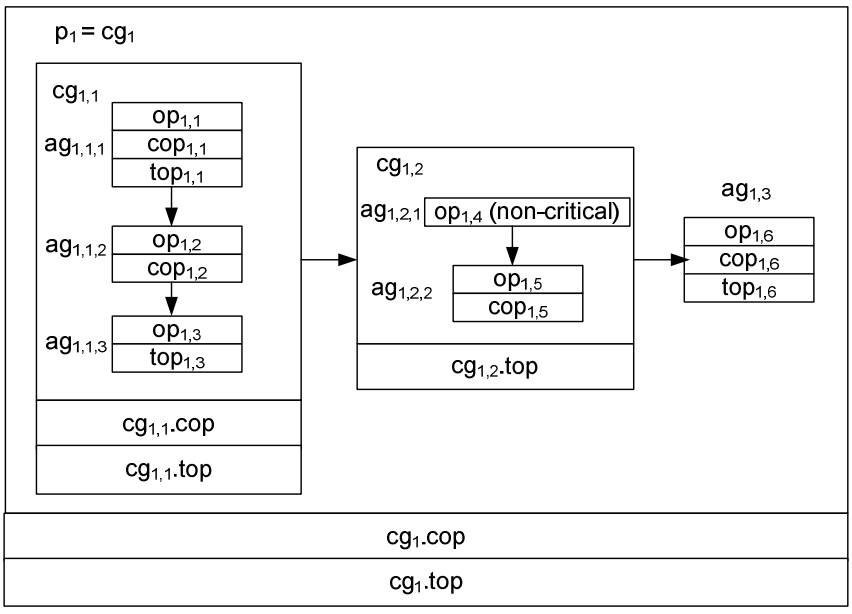

Figure 2. An Abstract View of a Sample Process

Atomic Group Execution and Recovery

When the execution of an atomic group fails, pre-commit recovery activities are applied locally to clean up the failed operation execution before the operation terminates and communicates its terminated status to the process execution environment.

Definition 1 (Pre-commit Recoverability): Pre-commit recoverability specifies how an atomic group should be locally recovered when an execution failure occurs before the operation as an execution unit commits.

Table 2 presents pre-commit recovery options for an atomic group. Ideally, an ag operation's pre-commit recoverability is *automatic rollback* for an ACID operation, or *pre-commit-compensation* for a non-ACID operation. With the delta capture capability of the DeltaGrid environment, an ag can also reverse the effect of the original operation through *DE-rollback* if the recoverability conditions are satisfied. If DE-rollback cannot be applied due to the violation of the semantic conditions for DE-rollback, the service composition model requires the use of a *service reset* function. The service reset function cleans up the effect of a failed operation and prepares the execution environment for the next service invocation. A service reset typically requires a special program or a human agent to resolve the failed operation execution.

Table 2. Atomic Group Pre-commit Recoverability Options

Option	**Meaning**
Automatic rollback	The failed service execution can be automatically rolled back by a service provider
Pre-Commit-Compensation	A pre-commit-compensation is invoked by a service provider to backward recover a failed operation.
DE-rollback	A failed operation can be reversed by executing DE-rollback
Service Reset	The service provider offers a service reset function to clean up the service execution environment.

After an atomic group has been locally recovered, the failed execution transmits its terminated status to the process execution environment. In the context of the global process, an ag maximizes the success of an operation execution by providing an optional contingency plan that is executed as an alternative path if the original service execution of the ag fails.

In contrast to pre-commit recoverability, which defines how to locally clean up a failed operation execution, *post-commit recoverability* specifies how the process execution environment can semantically undo the effect of a successfully terminated atomic group due to another operation's execution failure.

Definition 2 (Post-commit Recoverability): Post-commit recoverability specifies how an operation's effect can be semantically undone after the operation has successfully terminated.

Post-commit recoverability is considered when a completed operation inside of a composite group needs to be undone due to runtime failure of another operation. Table 3 defines three post-commit recoverability options: *reversible* (through DE-rollback), *compensatable*, or *dismissible*. Post-commit recovery is only applicable in the context of composite group execution. Furthermore, the dismissable option indicates that a process execution can be application-dependent and might not require every operation to be successfully executed. The DeltaGrid service composition model offers the flexibility of marking execution entities with a *criticality*

decorator when failure does not affect the execution of the enclosing composite group. By default, an operation's post-commit recoverability is compensatable.

Table 3. DEGS Post-Commit Recoverability Options

Option	**Meaning**
Reversible (DE-rollback)	A completed operation can be undone by reversing the data values that have been modified by the operation execution.
Compensatable	A completed operation can be semantically undone by executing another operation, referred to as post-execution compensation.
Dismissible	A completed operation does not need any cleanup activities.

Definition 3 (Criticality): An atomic group is *critical* if its successful execution is mandatory for the enclosing composite group. A *non-critical* group indicates that the failure of this group will not impact the state of the enclosing composite group, and the composite group can continue execution. When runtime execution failure occurs, contingency must be executed for critical groups, while contingency is not necessary for a non-critical group. By default, a group is critical.

As an example, in Figure 2, if $ag_{1,2,1}$ fails, $cg_{1,2}$ will continue executing since $ag_{1,2,1}$ is non-critical. Thus in the specification, there is no need to define a compensation and contingency plan for $ag_{1,2,1.}$

Composite Group Execution and Recovery

The recoverability of a composite group can be defined using the concepts of *shallow compensation* and *deep compensation.* The terms shallow and deep compensation were originally defined in (Leymann, 1995). Our research extends these concepts for use with nested service composition.

Definition 4 (Shallow Compensation): Assume a composite group $cg_{i,k}$ is defined as $cg_{i,k} = <(ag_{i,k,m} \mid cg_{i,k,n})^{+}, cop_{i,k} [,top_{i,k}])>$. Shallow compensation of $cg_{i,k}$ is the invocation of the compensation operation defined for the composite group $cg_{i,k,}$ which is $cop_{i,k}$.

Definition 5 (Deep Compensation): Assume a composite group $cg_{i,k}$ is defined as $cg_{i,k} = <(ag_{i,k,m} \mid cg_{i,k,n})^{+}, cop_{i,k} [,top_{i,k}])>$. Within the context of a composite group $cg_{i,k}$, a subgroup is either an atomic group defined as $ag_{i,k,m} = <op_{i,j}, cop_{i,j} [,top_{i,j}]>$, or a composite group defined as $cg_{i,k,n} = <(ag_{i,k,n,x} \mid cg_{i,k,n,y})^{+}, cop_{i,k,n} [,top_{i,k,n}])>$. Deep compensation of $cg_{i,k}$ is the invocation of post-commit recovery activity (compensation or DE-rollback) for each executed subgroup within the composite group, such as $cop_{i,j}$ for an atomic group, and $cop_{i,k,n}$ for a nested composite group.

Shallow compensation is invoked when a composite group successfully terminates but needs a semantic undo due to the failure of another operation execution. A deep compensation is invoked if: 1) a composite group fails due to a subgroup execution failure, and needs to trigger the post-commit recovery of executed subgroups, or 2) a composite group successfully terminates, but no shallow compensation is defined for the composite group.

As a backward recovery mechanism for a successfully executed composite group, shallow compensation has higher priority than deep compensation. For example, in Figure 2, the failure of a critical subgroup $ag_{1,3}$ (both $op_{1,6}$ and $top_{1,6}$ fail) within the enclosing composite group cg_1 causes the two executed composite groups $cg_{1,1}$ and $cg_{1,2}$ to be compensated. Since $cg_{1,1}$ has a pre-defined

shallow compensation, the shallow compensation $cg_{1,1}$.cop will be executed. $cg_{1,2}$'s deep compensation will be invoked since $cg_{1,2}$ does not have shallow compensation.

An Online Shopping Case Study

This section introduces an online shopping case study to illustrate the use of the service composition and recovery model. The online shopping application contains typical business processes that describe the activities conducted by shoppers, the store and vendors. For example, the process placeClientOrder is responsible for invoking services that place client orders and decrease the inventory quantity. The process placeVendorOrder checks the inventory, calculates restocking need, and generates vendor orders. The process replenishInventory invokes services that increase the inventory quantity when vendor orders are received.

Figure 3 presents a graphical view of the placeClientOrder process using the same notation as the abstract process example presented in Figure 2. As shown in Figure 3, the process placeClientOrder is hierarchically composed of composite groups and atomic groups. An atomic group has an operation, an optional post-commit compensation (cop) and contingency (top).

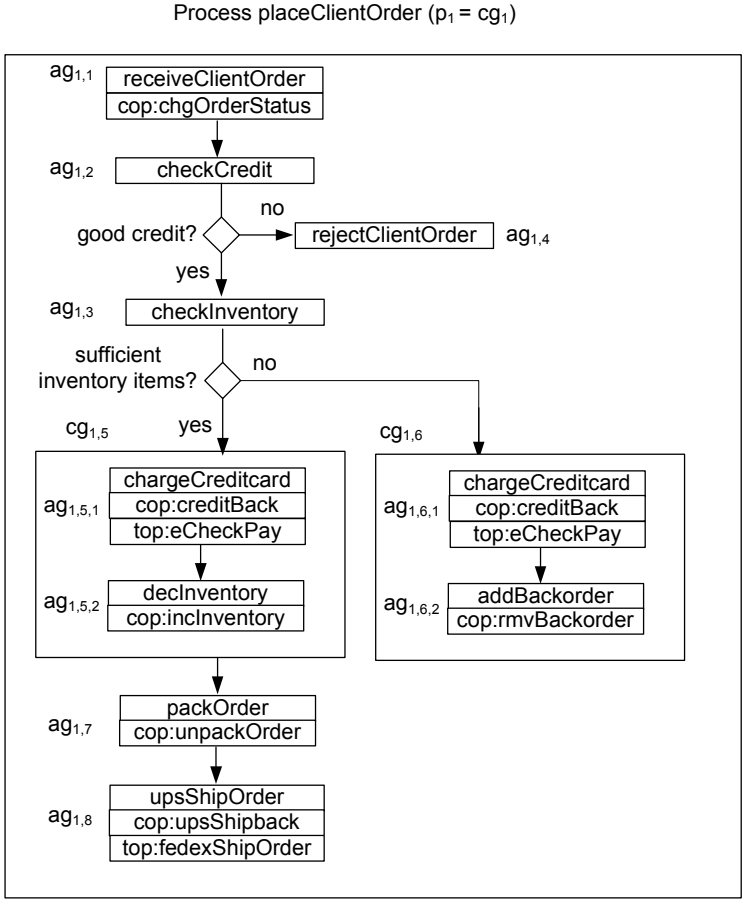

Figure 3. placeClientOrder Process Definition

The placeClientOrder process starts when a client submits a client order by invoking a DEGS operation receiveClientOrder. The next operation creditCheck verifies if the client has a good credit standing to pay for the order. If the client passes the creditCheck, the inventory will be checked to

see if there are sufficient inventory items to fill the order by executing checkInventory. If the client does not pass the credit check, the order will be rejected. If there are sufficient inventory items, the operation chargeCreditCard is to be executed to charge the client's credit card, and the operation decInventory is executed to decrease inventory. These two operations are grouped into a composite group indicating that both operations should be successfully executed as a unit. Then the order will be packed through operation packorder and shipped through operation upsShipOrder. If the inventory is not sufficient to fill the order, the order will be marked as a backorder through operation addBackorder, and the client will be charged the full amount.

When there is a service execution failure during process execution, the process will be recovered based on the recovery specification embedded in the process definition, such as compensation and contingency, as well as the recovery semantics of the service composition and recovery model. For example, if operation upsShipOrder fails, the contingency fedexShipOrder will be invoked, sending the order package through Fedex instead of UPS. If a client requests to cancel the order after the operation packOrder but before upsShipOrder, each executed operation will be backward recovered in the reverse execution order using the following list of recovery commands: [cop:unpackOrder, cop:incInventory, cop:creditBack, DE-rollback:checkInventory, DE-rollback:checkcredit, cop:chgOrderStatus]. DE-rollback is to be performed on operations checkInventory and checkCredit since these two operations do not have pre-defined compensation and no other concurrently executing processes are write dependent on these two operations. Furthermore, since these two operations do not modify any data, no recovery actions will be performed for these two operations. Thus the final recovery commands for cancellation of an order is: [cop:unpackOrder, cop:incInventory, cop:creditBack, cop:chgOrderStatus].

Figure 4 gives a graphical view of the process placeVendorOrder. The process first invokes the operation getLowInventoryItems which goes through all the inventory items to create an entry for each inventory item whose quantity falls below a specified threshold. The operation getBackOrderItems goes through backorderList, adding items in the backorder list to the items to be ordered from the operation getLowInventoryItems. The process proceeds with the operation confirmPrice, which confirms the unit price of a product with each vendor. Then the operation genVendorOrder will generate vendor orders for different suppliers. After reviewVendorOrder which performs a final check on the vendor orders, these vendor orders are sent to suppliers by executing the operation sendVendorOrder.

If the operation reviewVendorOrder fails, the process placeVendorOrder will be backward recovered by executing post-commit recovery activity for each executed operation in reverse execution order: [cop:chgVOStatus, DE-rollback:confirmPrice, DE-rollback:getBackOrderItems, DE-rollback:getLowInventoryItems]. DE-rollback will be invoked on operations confirmPrice, getBackOrderItems and getLowInventoryItems since these operations do not have pre-defined compensation.

Figure 5 presents the replenishInventory process which is invoked when a vendor order package is received from a supplier. The process first verifies if there is any missing item by performing operation verifyVOItem. If there is any missing item, the relevant vendor will be contacted through operation contactVendor. Otherwise, received items are entered into the inventory and the operation incInventory is executed to update quantity for each received inventory item. The operation packBackorder iterates through the backorder list and pack backorders for shipment. After packBackorder, inventory will be decreased through operation decInventory to deduct the backorder quantity from the inventory. At last, operation sendBackorder dispatches backorders to customers.

As in the processes placeClientOrder and placeVendorOrder, the recovery procedure of process replenishInventory also conforms to the semantics defined in the service composition and recovery model. For example, if the vendor recalls deficient items when the process finishes the execution of the operation decInventory, the process replenishInventory needs a backward recovery followed by sending the deficient items back to the vendor for a replacement. The backward recovery of

the process will execute the compensation of every executed operation in reverse execution order: [cop:incInventory, cop:unpackBorder, cop:decInventory]. The operation verifyVOItems will not be recovered since verifyVOItems does not modify any data.

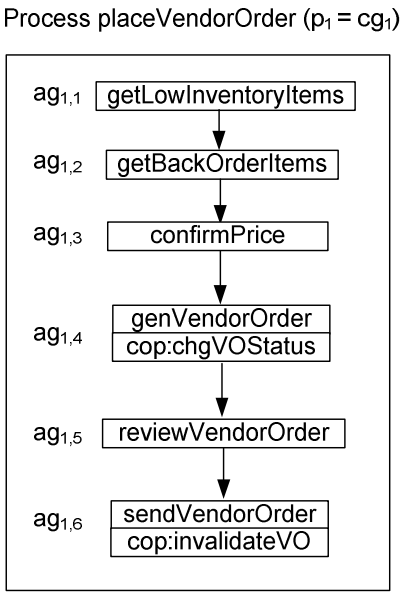

Figure 4. placeVendorOrder Process Definition

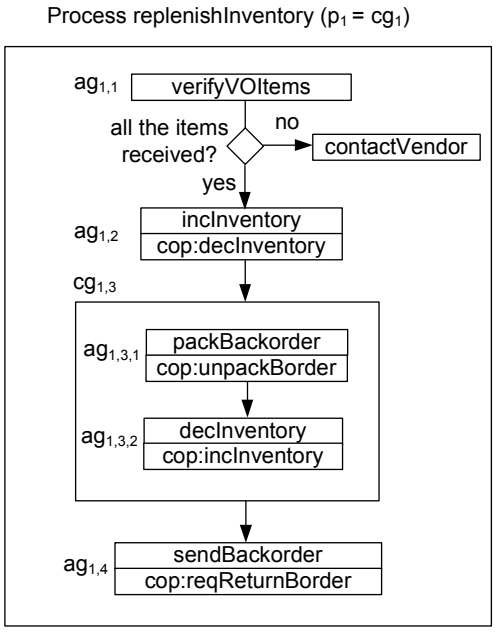

Figure 5. replenishInventory Process Definition

Simulation and Evaluation Framework

The original version of the DeltaGrid service composition and recovery model as described in this section has been formally presented in (Xiao et al., 2009; Xiao, 2006). The presentation includes state diagrams and algorithms that define the semantics of applying compensation and contingency when failure occurs. The work in (Xiao et al., 2009) also includes the description of a DeltaGrid simulation framework using the DEVSJAVA discrete event simulation tool (Zeigler & Sarjoughian 2004), as well as a performance evaluation of some of the implemented components of the simulation environment. Interested readers should refer to (Xiao et al., 2009) for further details on the formalization, simulation, and evaluation of the original model. In the remainder of this paper, we describe an extension of the model to more completely address data consistency issues during execution and to also provide a means for partial rollback together with increased options for forward recovery. The concepts presented in this section are formalized together with the extended features using Petri Nets in the following sections of this chapter.

ASSURANCE POINTS AND INTEGRATION RULES FOR ENHANCING CONSISTENCY AND RECOVERY

The model described in the previous section has been extended with the concept of *Assurance Points (APs)* (Shrestha, 2010; Urban, Gao, Shrestha, & Courter, 2010a; Urban, Gao, Shrestha, and Courter, 2010b). An AP is a process execution correctness guard as well as a potential rollback point during the recovery process. Given that concurrent processes do not execute as traditional transactions in a service-oriented environment, inserting APs at critical points in a process is important for checking consistency constraints and potentially reducing the risk of failure or inconsistent data. An AP also serves as a milestone for backward and forward recovery activities. When failures occur, APs can be used as rollback points for backward recovery, rechecking pre-conditions relevant to forward recovery. In the current version of the model, it is assumed that APs are placed at points in a process where they are only executed once, and not embedded in iterative control structures. The version described in this chapter also does not address the use of APs in parallel execution structures, such as the <flowgroup> activity of BPEL, although a prototype execution engine supports this capability. An elaboration of these issues is beyond the scope of the current paper and is addressed at the end of the chapter as part of future research.

An AP is defined as: AP = <apId, apParameters*, IR_{pre}?, IR_{post}?, IR_{cond}*>, where:

- apID is the unique identifier of the AP
- apParameters is a list of critical data items to be stored as part of the AP,
- IR_{pre} is an integration rule defining a pre-condition,
- IR_{post} is an integration rule defining a post-condition,
- IR_{cond} is an integration rule defining additional application rules.

In the above notation, * indicates 0 or more occurrences, while ? indicates zero or one optional occurrences.

IR_{pre}, IR_{post}, and IR_{cond} are expressed as Event-Condition-Action (ECA) rules using the format shown in Figure 6, which is based on previous work with using integration rules to interconnect software components (Jin, 2004; Urban, Dietrich, Na, Jin, Sundermier, & Saxena, 2001). An IR is triggered by a process that reaches a specific AP during execution. Upon reaching an AP, the condition of an IR is evaluated. The action specification is executed if the condition evaluates to true. For IR_{pre} and IR_{post}, a constraint C is always expressed in a negative form (not(C)). The action (action 1) is invoked if the pre or post condition is not true, invoking a recovery action or an

alternative execution path. If the specified action is a retry activity, then there is a possibility for the process to execute through the same pre or post condition a second time, where action 2 is invoked rather than action 1. In general, any number of actions can be specified.

```
CREATE RULE     ruleName::{pre | post | cond}
EVENT           apId(apParameters)
CONDITION       rule condition specification
ACTION          action 1
[ON RETRY       action 2]
```

Figure 6. Integration Rule Structure

When pre and post conditions fail (not(C) = True), recovery actions are invoked. In its most basic form, a recovery action simply invokes an alternative process. Recovery actions can also be one of the following actions:

- **APRollback**: APRollback is used when the entire process needs to compensate its way back to the start of the process according to the semantics of the service compensation model.
- **APRetry**: APRetry is used when the running process needs to be backward recovered using compensation to a specific AP. By default, the backward recovery process will go to the first AP reached as part of the shallow or deep compensation process within the same scope. The pre-condition defined in the AP is re-checked. If the pre-condition is satisfied, the process execution is resumed from that AP by re-trying the recovered operations. Otherwise, the action of the pre-condition rule is executed. The APRetry command can optionally specify a parameter indicating the AP that is the target of the backward recovery process.
- **APCascadedContingency (APCC)**: APCC is a backward recovery process that searches backwards through the hierarchical nesting of composite groups to find a possible contingent procedure for a failed composite group. During the APCC backward recovery process, when an AP is reached, the pre-condition defined in the AP will be re-checked before invoking any contingent procedures for forward recovery.

The most basic use of an AP together with integration rules is shown in Figure 7, which shows a process with three composite groups and an AP between each composite group. The shaded box shows the functionality of an AP using AP2 as an example. Each AP serves as a checkpoint facility, storing execution status data in a checkpoint database (denoted as AP data in Figure 7). When the execution reaches AP2, IRs associated with the AP are invoked. The condition of an IR_{post} is evaluated first to validate the execution of cg_2. If the post-condition is violated, the action invoked can be one of the pre-defined recovery actions as described above. If the post-condition is not violated, then an IR_{pre} rule is evaluated to check the pre-condition for the next service execution. If the pre-condition is violated, one of the pre-defined recovery actions will be invoked. If the pre-condition is satisfied, the AP will check for any additional, conditional rules (IR_{cond}) that may have been expressed. IR_{cond} rules do not affect the normal flow of execution but provide a way to invoke additional parallel activity based on application requirements. Note that the expression of a pre-condition, post-condition or any additional condition is optional.

Assurance Point and Integration Rule Example

This section provides an example of assurance points, integration rules, and conditional rules in Figure 8 using a revised version of the online shopping application. All atomic and composite groups are shown in the solid line rectangles, while optional compensations and contingencies are shown in dashed line rectangles, denoted as cop and top, respectively. APs are shown as ovals

between composite and/or atomic groups, where the AP identifiers and parameters are OrderPlaced(orderId), CreditCardCharged(orderId, cardnumber, amount), UPSShipped(orderId, UPSShippingDate), USPSShipped (orderId), Delivered(orderId, shippingMethod, deliveryDate).

Table 4 shows the integration rules and conditional rules associated with the APs in Figure 8. The components of an assurance point are explained below using the APs in Figure 8 and the rules in Table 4.

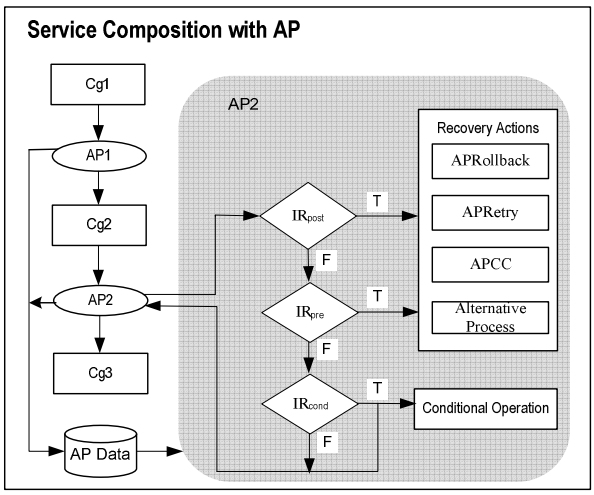

Figure 7. Basic Use of AP and Integration Rules

Component 1 (AP Identifiers and Parameters): An AP identifier defines the current execution status of a process instance. Each AP may optionally specify parameters that store data when the process execution reaches the AP. The data can then be examined in the conditions of rules associated with the AP. For example, the first AP is orderPlaced, which reflects that the customer has finished placing the shopping order. The parameter orderId is used in the rules associated with the AP.

Component 2 (Integration Rules): An integration rule is optionally used as a transition between logical components of a process to check pre and post conditions. In Table 4, the orderPlaced AP has a pre-condition that guarantees that the store must have enough goods in stock. Otherwise, the process invokes the backOrderPurchase process. The CreditCardCharged AP has a post-condition that further guarantees the in-stock quantity must be in a reasonable status after the decInventory operation.

Component 3 (Conditional Rule): In Table 4, the CreditCardCharged AP has a conditional rule that sends a message notification for large charges. Since no pre or post condition is specified for the Delivered AP, only the conditional rule shippingRefund is evaluated. Assume the delivery method was overnight through UPS with an extra shipping fee. If UPS has delivered the item on time, then the Delivered AP is complete and execution continues. Otherwise, refundUPSShippingCharge is invoked to refund the extra fee while the main process execution continues. If backward recovery with retry takes place, it is possible that the process will execute the same conditional rule a second time. The action of the rule will only be executed during the retry process if the action was not executed the first time through.

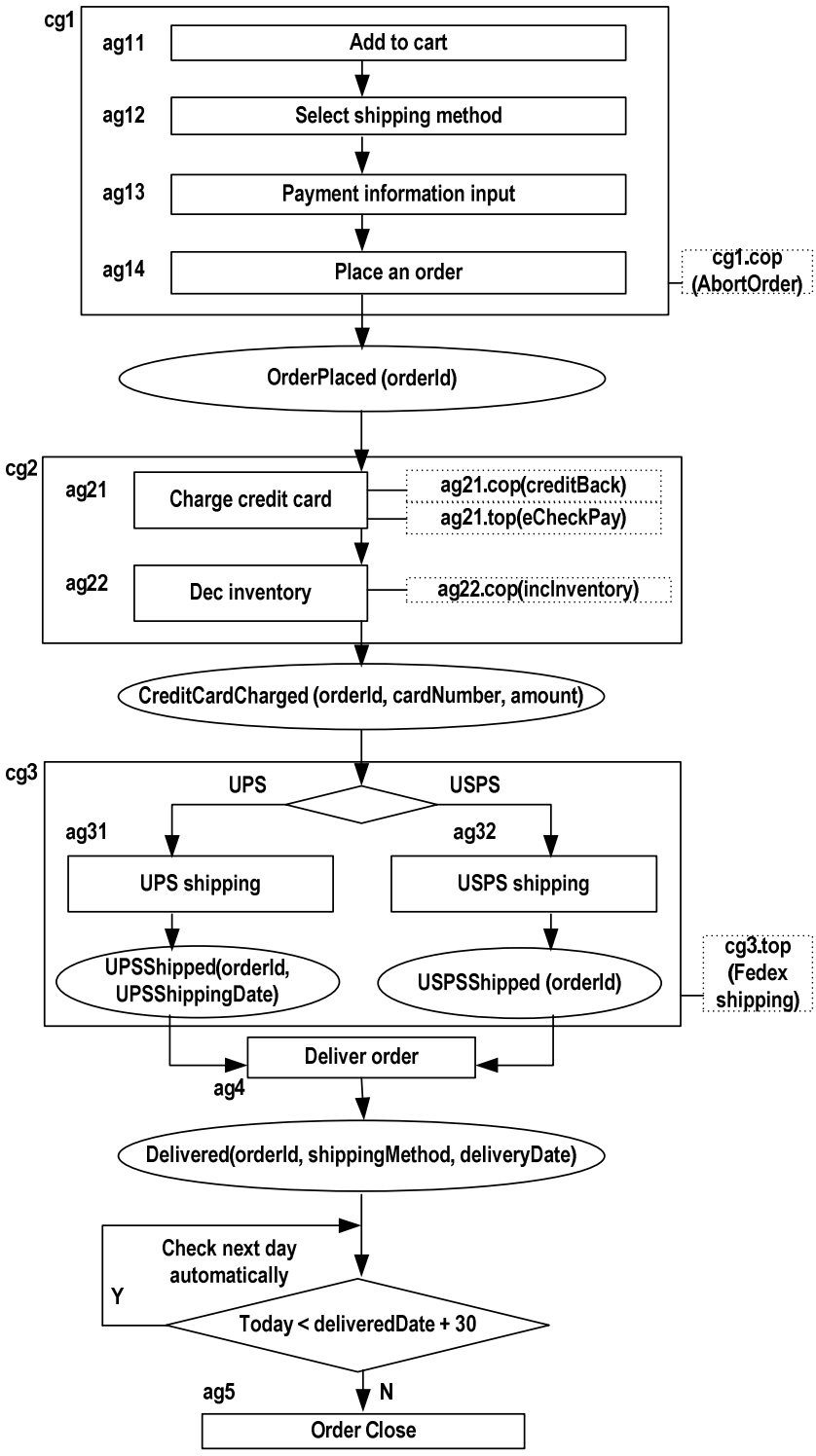

Figure 8. Online Shopping Process with APs

Table 4. AP Rules in the Online Shopping Process

Integration Rule	Conditional Rule
create rule QuantityCheck::pre **event:** ***OrderPlaced*** (orderId) **condition:** exists(select L.itemId from Inventory I, LineItem L where L.orderId=orderId and L.itemId=I.itemId and L.quantity>I.quantity) **action:** backOrderPurchase(orderId)	**create rule** Notice::cond **event:** ***CreditCardCharged*** (orderId, cardNumber , amount) **condition:** amount > $1000 **action:** highExpenseNotice(cardNumber)
create rule QuantityCheck::post **event:** ***CreditCardCharged*** (orderId, cardNumber, amount) **condition:** exists(select L.itemId from Inventory I, LineItem L where L.orderId=orderId and L.itemId=I.itemId and I.quantity<0) **action1**: APRetry **action2:** APRollback	**create rule** ShippingRefund::cond **event**: ***Delivered*** (orderId, shippingMethod, deliveryDate) **condition:** shippingMethod = UPS && deliveryDate != UPSShipped.UPSShippingDate+1 **action:** refundUPSShippingCharge(orderId)

A Closer Look at Recovery Actions

This section provides an informal illustration of the semantics of the APRollback, APRetry, and APCC recovery actions using the generic sample process in Figure 9 as well as the Online Shopping example in Figure 8. The remainder of this chapter does not elaborate on the use of conditional rules. Further details on conditional rules can be found in (Jin, 2004).

The process (cg_0) in Figure 9 is successively composed of composite groups cg_{01}, cg_{02} and cg_{03}, as well as atomic groups ag_{04} and ag_{05}. The assurance points AP1, AP2 and AP4 are inserted in the cg_0 scope following cg_{01}, cg_{02} and ag_{04}, respectively. AP3 is inserted in the cg_{03} scope after ag_{031}. As a result, AP3 is at a more deeply nested level than the other assurance points. In the following, assume that each AP in Figure 9 has an IR_{pre} and an IR_{post} rule. Recovery actions for failed pre and post conditions are considered first, followed by recovery actions for execution errors.

APRollback. Recall that APRollback is used to logically reverse the current state of the entire process using shallow and deep compensation.

Scenario 1 (APRollback): Assume that the post-condition fails at AP4 in Figure 9 and that the action of IR_{post} is APRollback. Since APRollback is invoked, the process compensates all completed atomic and/or composite groups. The APRollback execution sequence is numbered in Figure 9. First the process invokes ag_{04}.cop to compensate ag_{04}. Second, the APRollback process will deep compensate ag_{031} by invoking ag_{031}.cop since 1) there is no shallow compensation for cg_{03} and 2) ag_{032} is non-critical and therefore has no compensating procedure. Finally, APRollback invokes shallow compensation cg_{02}.cop and cg01.cop.

The APRollback procedure is a standard way of using compensation in past work. The originality of the rollback process in this work is the way in which it is used together with APs in the retry and cascaded contingency process to support partial rollback together with forward recovery options.

APRetry. APRetry is used to recover to a specific AP and then retry the recovered atomic and/or composite groups. If the AP has an IR_{pre}, then the pre-condition will be re-examined. If the pre-condition fails, the action of the rule is executed, which either invokes an alternate execution path

for forward recovery or a recovery procedure for backward recovery. By default APRetry will go to the most recent AP. APRetry can also include a parameter to indicate the AP that is the target of the recovery process.

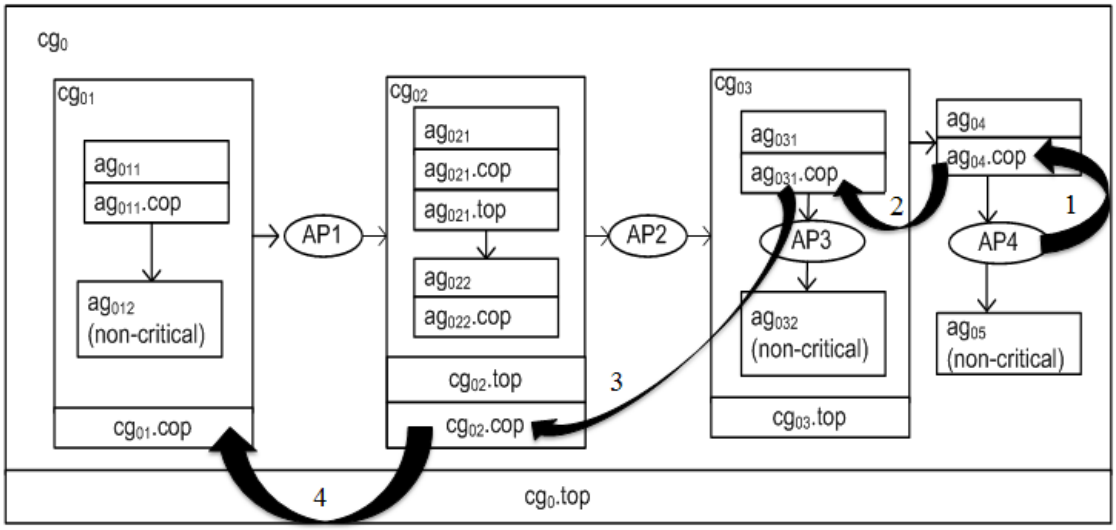

Figure 9. Generic Process: Scenario 1 (*APRollback*)

Scenario 2 (APRetry-default): Assume that the post-condition fails at AP4 in Figure 10 and that the action of IR_{post} is APRetry. This action compensates to the most recent AP within the same scope by default. In Figure 10, APRetry first invokes ag_{04}.cop to compensate ag_{04} at step 1. The process then deep compensates cg_{03} by executing ag_{031}.cop at step 2. At this point, AP2 is reached and the pre-condition of IR_{pre} is re-evaluated shown as step 3. If the pre-condition fails, the process executes the recovery action of IR_{pre}. If the pre-condition is satisfied or if there is no IR_{pre}, then execution will resume again from cg_{03}. In this case, the process will reach AP4 a second time through steps 4, 5 and 6, where the post-condition is checked once more. If failure occurs for the second time, the second action defined on the rule is executed rather than the first action. If a second action is not specified, the default action will be APRollback as steps 7 through 10.

Scenario 3 (APRetry-parameterized): As shown in Figure 11, now assume that the action of the pre-condition for AP4 is parameterized as APRetry(AP1), indicating that the retry activity should rollback to AP1. The process will first compensate the procedure back to the point of AP1 through steps 1, 2, 3 and 4, ignoring all APs in between. The process then resumes execution from AP1 at step 5.

APCascadedContingency (APCC). The APCC process provides a way of searching for contingent procedures in a nested composition structure, searching backwards through the hierarchical process structure. When a pre or post condition fails in a nested composite group, APCC will compensate its way to the next outer layer of the nested structure. If the compensated composite group has a contingent procedure, it will be executed. Furthermore, if there is an AP with a pre-condition before the composite group, the pre-condition will be evaluated before executing the contingency. If the pre-condition fails, the recovery action of IR_{pre} will be executed instead of executing the contingency. If there is no contingency or if the contingency fails, APCC continues by compensating the current composite group back to the next outer layer of the nested structure and repeating the process described above.

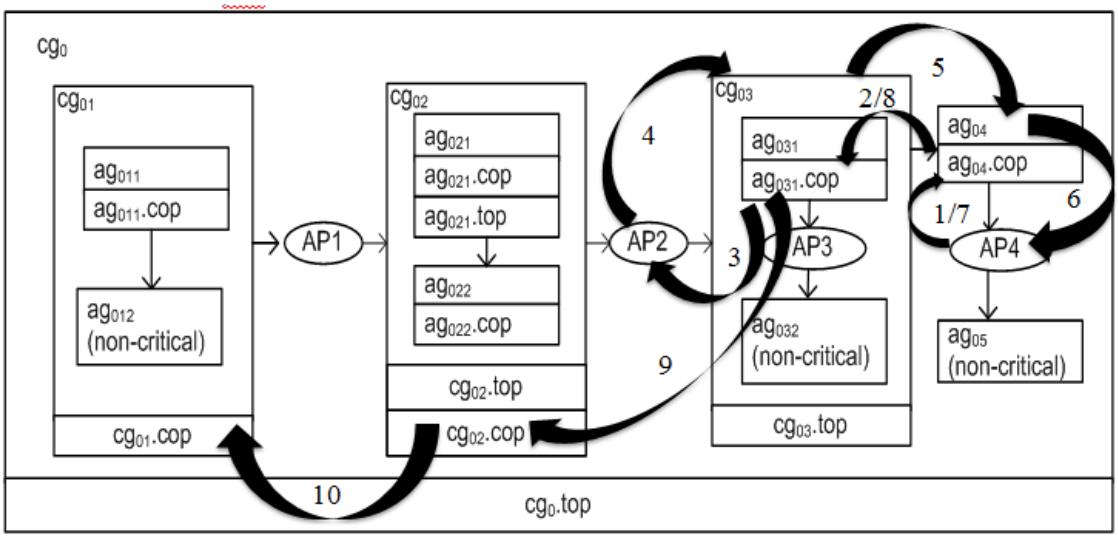

Figure 10. Scenario 2 (APRetry-default)

Scenario 4 (APCC): Assume that the post-condition fails at AP4 in Figure 9 and that the IR_{post} action is APCC. The process starts compensating until it reaches the parent layer. In this case, the process will reach the beginning of cg_0 after compensating the entire process through deep or shallow compensation through the same steps as shown in Figure 9. Since there is no AP before cg_0, cg_0.top is invoked.

Scenario 5 (APCC): Assume that the post-condition fails at AP3 in Figure 12 and that the IR_{post} action is APCC. Since AP3 is in cg_{03}, which is nested in cg_0, the APCC process will compensate back to the beginning of cg_{03}, executing ag_{031}.cop at step 1. The APCC process finds AP2 with an IR_{pre} rule for cg_{03} at step 2. As a result, the pre-condition will be evaluated before trying the contingency for cg_{03}. If there is no pre-condition or if the pre-condition is satisfied, then cg_{03}.top is executed at step 3 and the process continues, shown as step 4. Otherwise, the recovery action of IR_{pre} for AP2 will be executed and the process quits APCC mode. If cg_{03}.top fails at step 3 then the process will still be under APCC mode, where the process will keep compensating through steps 5 and 6 until it reaches the cg_0 layer, where cg_0.top is executed at step 7.

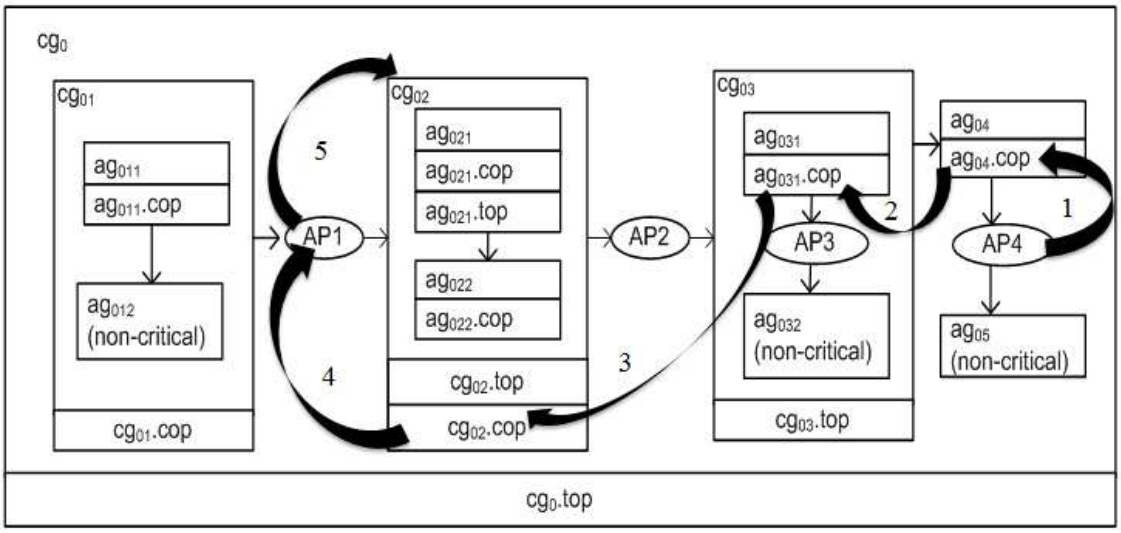

Figure 11. Scenario 3 (APRetry-parameterized)

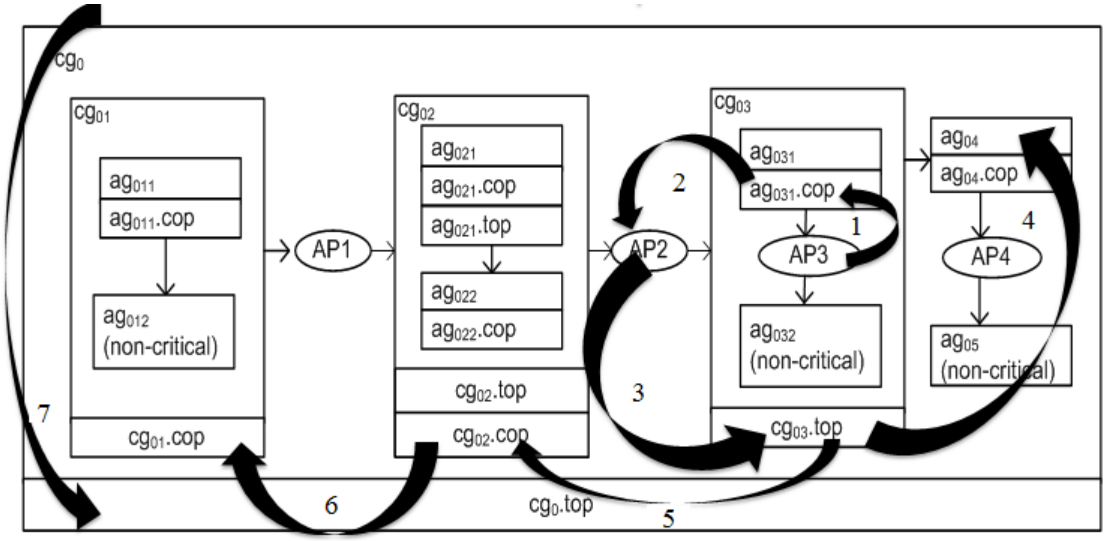

Figure 12. Scenario 5 (APCC)

When process execution encounters an internal error, the running operation first tries the most immediate contingency. If the contingency succeeds, the recovery is complete and the execution continues. If the contingency fails or if there is no immediate contingency, then APCC mode is invoked.

Scenario 6 (Online Shopping Example - Failure at ChargeCreditCard): Returning to the Online Shopping Example of Figure 8, assume the process fails while executing chargeCreditCard. The process then executes the contingency ag_{21}.top (eCheckPay). If ag_{21}.top fails, then APCC process begins, during which the process reaches the orderPlaced AP, where the pre-condition of the AP is re-checked (rule QuantityCheck in Table 1). If the pre-condition is violated, the action backOrder is invoked, which means there are not enough goods in stock.

Scenario 7 (Online Shopping Example – Failure at UPShipping): From Figure 8, assume the process fails on the operation UPSShipping. Since there is no immediate contingency, the process invokes the APCC process, rolling back to the CreditCardCharged AP at the outer level. Since there is no pre-condition defined at the CreditCardCharged AP, the contingency cg_3.top (FedexShipping) will be executed. If cg_3.top fails, the process will be still under APCC mode, compensating its way back to the beginning of the transaction.

PETRI NET FORMALIZATION OF SERVICE COMPOSITION WITH ASSURANCE POINTS AND RECOVERY ACTIONS

In this section, the formal execution semantics of the web service composition and recovery model with assurance points and integration rules is presented using Petri nets. Petri nets have been useful for modeling systems that demonstrate control flow behavior (Peterson, 1981). Van der Aalst (1998) was one of the first to use Petri nets to represent workflow management systems. Desel (2005) discusses process modeling with Petri nets. Stahl (2005) also gives the complete Petri net semantics for the Business Process Execution Language for Web Services (BPEL).

General Approach

A Petri Net (Murata, 2002) is a directed, connected, and bipartite graph in which nodes represent places and transitions, and tokens occupy places. A directed arc in a Petri Net connects a place to a transition or a transition to a place. The places that have arcs running to a transition are called input places of the transition. The places that have arcs coming from a transition are called output places of the transition. A transition is enabled when each of its input places has at least one token. After firing a transition, exact one token at each of its input places has been consumed, while one token at each of its output places has been generated.

In the Petri Net formalization of the service composition and recovery model presented in this chapter, a transition represents a basic task, such as invoking an operation of a process. A place represents an execution status, a condition, or a resource. A token at the place of an execution status corresponds to the thread of control in the flow. A token at the place of a condition indicates that some condition regarding the current status of a process instance is true. A token at the place of a resource indicates that the resource is (or in some cases is not) available. For example, in the service composition model, compensation is a resource associated with an atomic or composite group within a process, so resource places are used to indicate whether compensation is or is not available for a given group.

Before discussing the details of the Petri Net formalization, the notation used in the Petri Net diagrams is introduced. All transitions are labeled as Tn inside a transition node. Each place in a Petri Net has a short phrase beside the place node. Short phrases are used to label places due to limited room in the Petri Net graph. The complete set of all places that appear in the graphs that follow for atomic and composite execution groups are shown in Table 5, while Table 6 indicates the places that are associated with graphs for APs. The left column of each table contains the short phrase of each place. The middle column contains the actual meaning of places. The right-most column indicates the type of the place, which is specified as status, condition, or resource.

Atomic Group

An Atomic group is the most basic executable unit in the model. An atomic group contains an operation, an optional compensation, and an optional contingency. Figure 13 depicts the execution semantics for an atomic group as a Petri net. All places standing on the lines of the box around a Petri Net represent the execution status, conditions, and resources of the atomic group.

An atomic group is activated when a token appears at place A. By firing transition T1, the operation of the atomic group is invoked, indicated by the place labeled Running. If the operation succeeds, the atomic group is finished successfully by marking place S through transition T2. Otherwise, the operation execution fails and must be aborted to place Aborted by transition T3. If an execution error occurs, the process will first try the immediate contingency if it is available. Places T and N_T are two resource places that represent the availability or non-availability, respectively, of an immediate contingency. If place T and aborted are marked, transition T4 is enabled, which means the immediate contingency is available. By firing T4, the immediate contingency is running. Note that place T is a resource place, therefore after firing T4, a token must be returned to place T. If the immediate contingency succeeds, transition T7 fires and then place S (successful) is marked. If the immediate contingency fails or does not exist, the APCC mode is triggered to cascade the search for contingencies to outer levels of the process. Transition T5 depicts contingency failure by marking places AP_CC (cascaded contingency) and US (unsuccessful). Similarly, if places N_T and aborted are marked, transition T6 is enabled which represents the case that the immediate contingency does not exist.

Table 5. Places in an Execution Group Petri Net

SHORT PHRASE	MEANING	TYPE
A	Activate	Status
S	Group executes successfully	Status
US	Group executes unsuccessfully	Status
AP_CC	AP_Cascaded Contingency	Status
AP_RB	AP_Rollback	Status
AP_RT	AP_Retry	Status
C_A	Compensation activates	Status
Running	Operation executing	Status
Aborted	Operation aborted	Status
T_Running	Contingency executing	Status
C_Running	Shallow compensation executing	Status
C_Error	Shallow compensation failed	Status
C_S	Compensation succeeds	Status
Critical	Critical atomic group	Resource
N_ Critical	Non-critical atomic group	Resource
T	Contingency exists	Resource
N_T	Contingency does not exist	Resource
Shallow_C	Shallow compensation exists	Resource
N_ Shallow_C	Shallow compensation does not exist	Resource

Table 6. Places in an AP Petri Net

SHORT PHRASE	MEANING	TYPE
A	Activate	Status
P	AP Passed	Status
ALT	Alternative Process	Status
AP_CC	AP_Cascaded Contingency	Status
AP_RB	AP_Rollback	Status
AP_RT	AP_Retry	Status
APCC_PRE	Pre-condition re-check (AP-CC)	Status
APCC_P	Pre-condition re-check passed (AP-CC)	Status
APRT_PRE	Pre-condition re-check (AP-Retry)	Status
APRT_P	Pre-condition re-check passed (AP-Retry)	Status
POST_VIO_F	First time post-condition violation	Condition
PRE_VIO_F	First time pre-condition violation	Condition
POST_VIO_S	Second time post-condition violation	Condition
PRE_VIO_S	Second time pre-condition violation	Condition
POST	Post condition exists	Resource
N_POST	Post condition does not exist	Resource
PRE	Pre condition exists	Resource
N_PRE	Pre condition does not exist	Resource
POST-Checking	Checking post condition	Status
PRE-Checking	Checking pre condition	Status
POST-Passed	Post condition passed	Status
Pre-Passed	Pre condition passed	Status
POST-Violated	Post condition violated	Status
Pre-Violated	Pre condition violated	Status

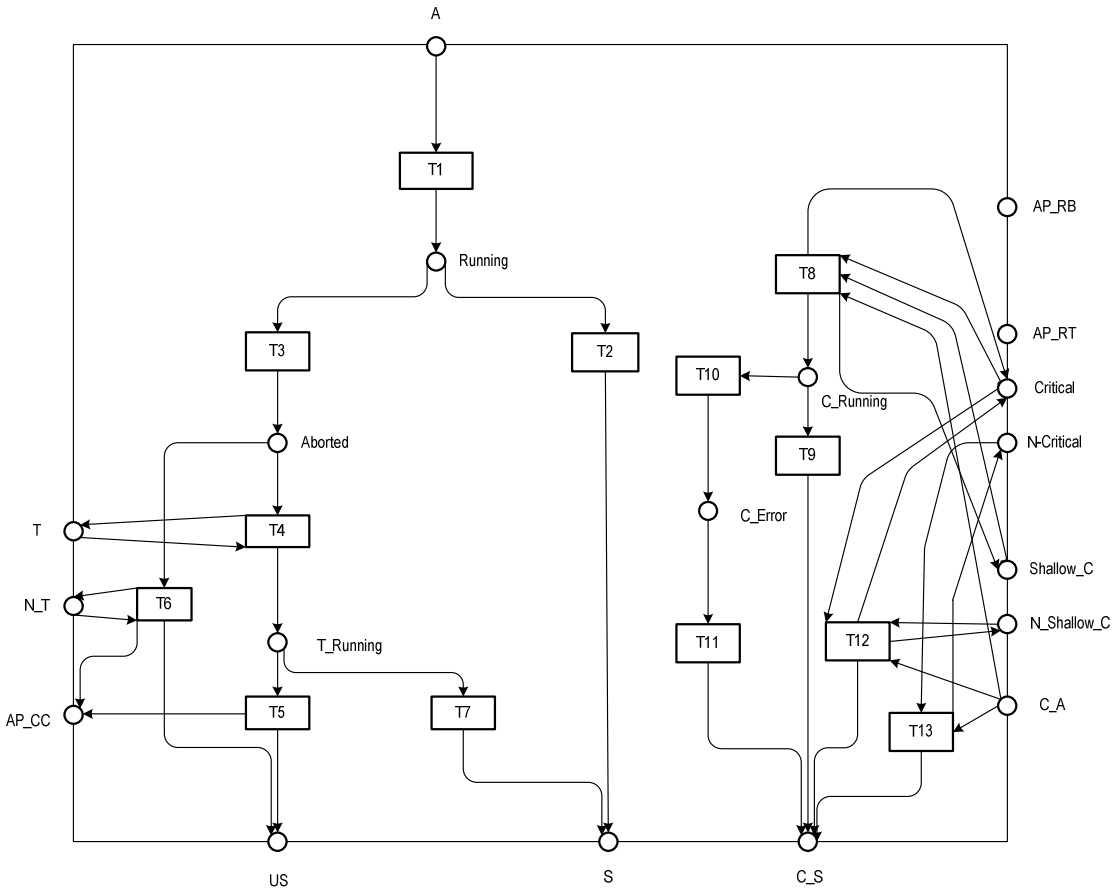

Figure 13. Petri Net of Atomic Group

The discussion above represents the normal atomic group invocation semantics. The normal atomic group invocation starts from place A and ends at either place S or places US and AP_CC. Now consider the compensation semantics of an atomic group. In Figure 13, if place C_A (compensation activity) is marked, the atomic group needs to be compensated. Here, four resource places are introduced. Place Critical represents that the atomic group is critical, whereas place N-Critical indicates that the atomic group is not critical. Place Shallow-C represents that the pre-defined compensation procedure is available, whereas place N-Shallow-C indicates that compensation is not available. There are four different atomic compensation cases in Figure 13:

- **Compensation is available and the atomic group is critical**: Transition T8 fires. After invoking compensation, two different situations may exist:
  - **Compensation succeeds**: Transition T9 fires and then place C_S is marked, indicating that compensation is successful.
  - **Compensation fails**: Transitions T10 fires marking the C_Error status, indicating that compensation has failed. Then transition T11 fires which represents invoking DE-Rollback or service reset (involving human activity) to reset the error. Finally, place C_S is marked.
- **Compensation is unavailable and the atomic group is critical:** Transition T12 fires, which represents invoking DE-Rollback or service reset. Then place C_S is marked.
- **The atomic group is non-critical:** If the atomic group is non-critical, the process just ignores the compensation request by firing transition T13 and marking place C_S.

To preserve the group condition consistency, if a token at a resource place is consumed after firing a transition, a new token must be returned to the resource place immediately after the transition. For example, in Figure 13, firing transition T8 consumes three tokens at places C_A, Shallow_C and Critical respectively. After T8, two new tokens are generated back to places Shallow_C and Critical, respectively, as they are resource places. One might question the situation that a token appears at place C_A before the atomic group finishes successfully. Such a situation will never happen, however, since the compensation of a completed group can only be caused by an error that occurs in the remainder of the process. Therefore, the place C_A can only be marked by transitions after the completion of the current group. The section below on Deep and Shallow Compensation will discuss compensation issues for multiple groups.

Assurance Points

Before describing the semantics of a composite group, this section first describes the semantics of APs. Figure 14 gives the Petri Net for AP execution semantics. There are four resource places. Places POST and N_POST represent the presence and absence of a post-condition respectively. Similarly, places PRE and N_PRE represent the presence and absence of a pre-condition, respectively. In addition, places POST_VIO_F and PRE_VIO_F are condition places indicating that the post and pre conditions have never been violated before. Thus, places POST_VIO_F and PRE_VIO_F each must have a default token before execution. In the same manner, places POST_VIO_S and PRE_VIO_S are the conditions indicating that the post and pre conditions have been violated once, respectively.

A token at place A activates the AP. Depending on the status of the condition and resource places, different execution cases exist:

- **Post and pre conditions both exist**:
  - **Post and pre conditions are both satisfied:** Firing transition T1 and T2 represents that the post-condition is satisfied. Firing transition T4 and T6 similarly indicates that the pre-condition is satisfied. Finally, transition T8 fires and place P is marked, indicating that the AP was successfully executed (passed) with all relevant conditions satisfied.
  - **Post condition violated:** Transition T1 fires to check the post-condition. If the post-condition is violated, transition T3 is fired to mark the status place Post-Violated. If place POST_VIO_F is marked, indicating that this is the first time to execute the post condition, then either transition T11, T12, T13 or T14 will be fired to invoke the first action of the rule, depending on the rule action specification. POST_VIO_S is then marked. If place POST_VIO_S is already marked, indicating that this is the second time to execute the post condition, then either transition T15 or T16 will be fired to execute the second action defined in the rule.
  - **Post condition passed and pre condition violated:** Firing transition T1 and T2 that the post-condition is satisfied. Then transition T4 fires to check the pre-condition. If the pre-condition is violated, transition T7 is fired to mark the status place Pre-Violated. If place PRE_VIO_F is marked, indicating that this is the first violation of the pre-condition, then either transition T17, T18, T19 or T20 will be fired to invoke the first action of the rule and PRE_VIO_S is marked. If place PRE_VIO_S is already marked, indicating that this is the second violation of the pre-condition, then either transition T21 or T22 will be fired depending on the second action defined in the rule.
- **Only post condition exists:** Firing transition T1 checks the post-condition.

- **Post condition is satisfied:** If the post-condition is satisfied, transition T2 fires and status place Post-Passed is marked. Because of the absence of a pre-condition, transition T9 fires and then place P is marked.
- **Post condition is violated:** Transition T1 fires to check the post-condition. If the post-condition is violated, transition T3 is fired to mark the status place Post-Violated. If place POST_VIO_F is marked, indicating that this is the first time to execute the post condition, then either transition T11, T12, T13 or T14 will be fired to invoke the first action of the rule, depending on the rule action specification. POST_VIO_S is then marked. If place POST_VIO_S is already marked, indicating that this is the second time to execute the post condition, then either transition T15 or T16 will be fired to execute the second action defined in the rule.

- **Only pre condition exists:** Because of the absence of the post-condition, when the AP is activated, transition T5 will be fired to check the pre-condition directly.
  - **Pre condition is satisfied**: If the pre-condition is satisfied, transitions T6 and T8 will be fired successively. Finally, place P is marked.
  - **Pre condition is violated:** If the pre-condition is violated, transition T7 is fired to mark the status place Pre-Violated. If place PRE_VIO_F is marked, indicating that this is the first violation of the pre-condition, then either transition T17, T18, T19 or T20 will be fired to invoke the first action of the rule and PRE_VIO_S is marked. If place PRE_VIO_S is already marked, indicating that this is the second violation of the pre-condition, then either transition T21 or T22 will be fired depending on the second action defined in the rule.
- **Post and pre condition do not exist:** After place A is marked, transition T10 fires and then place P is marked.

Note that there are four unlinked status places in Figure 14: APCC_PRE, APRT_PRE, APCC_P and APRT_P. These status places are relevant to the semantics of cascaded contingency and retry actions, which will be addressed in following sections.

Hierarchical Process Composition

In the service composition and recovery model, a composite group is composed of two or more atomic and/or composite groups and can also have optional compensation and contingency procedures. Clearly, a process under this model may contain multiple groups that are embedded at different levels. To represent the hierarchical model, a hierarchical approach is taken to the use of Petri Nets. Specifically, a dashed-line quadrilateral represents either an atomic or a composite group. A dashed-arc connecting a transition and a place represents repeating the same token movement pattern described at the current level. Furthermore, all dashed-line atomic and composite groups have the same places standing on the lines as introduced in Figure 13 and in Figure 15. However, to make the graphs concise, the unlinked places are omitted in hierarchical representations. APs that appear in hierarchical representations also omit unlinked places. Finally, to easily explain the semantics of the hierarchical organization of the diagrams, two group levels are defined. L_n is the outer level defined by a solid-line. L_{n+1} is the level of the inner dashed-line groups. For example, in Figure 15, the outer solid-line group is at level L_n and all inner dashed-line groups, as well as the inner AP, are at level L_{n+1}.

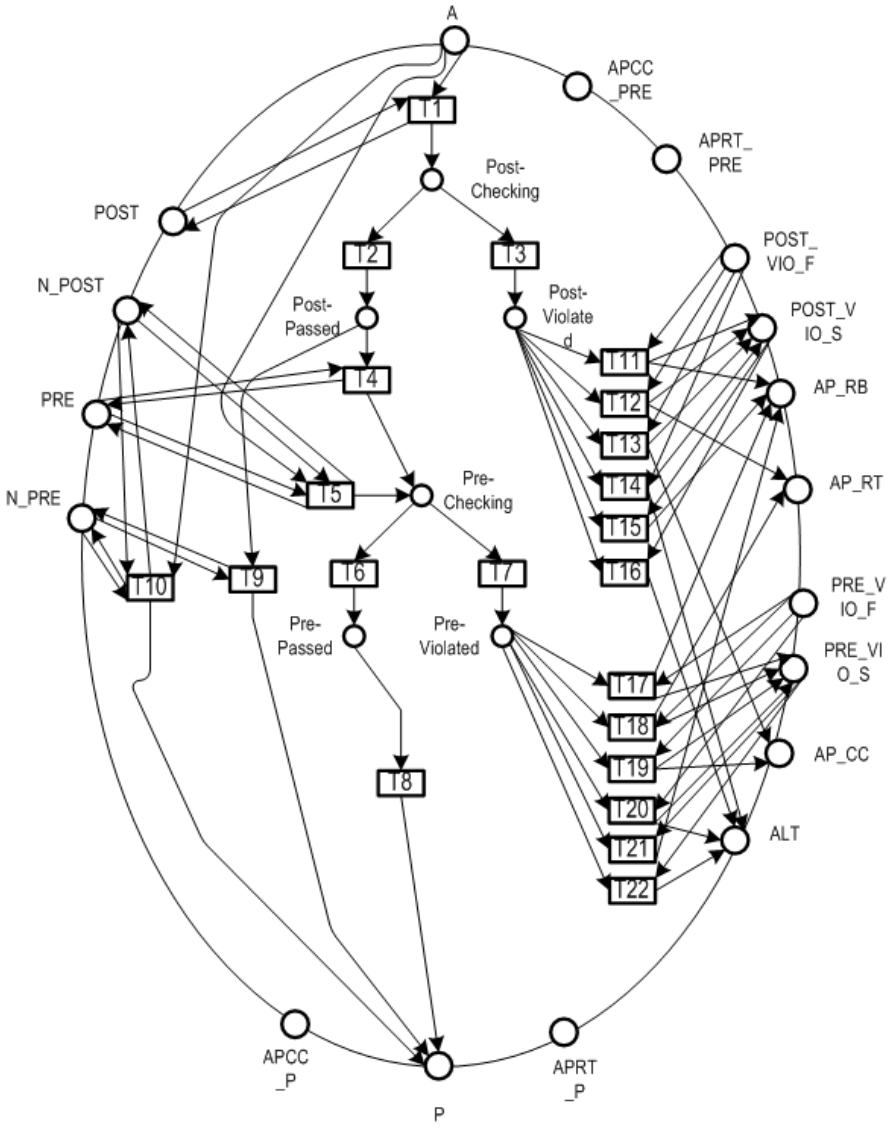

Figure 14. Petri Net of Assurance Point

Composite Group with Assurance Points

The Petri Net for normal execution semantics of a composite group with APs is shown in Figure 15. A composite group may contain multiple groups and APs. All activities in a composite group are executed sequentially. Therefore, the normal execution semantics expressed in Figure 15 are straightforward. When a token appears at place A at level L_n, transition T1 fires to activate the first activity at the inner level L_{n+1}. Upon completion of the first inner activity, a transition T2 fires and the next activity is activated. In Figure 15, an AP is invoked after one of the atomic (or composite) groups. When place P at the AP at level L_{n+1} is marked, the AP is passed and the inner execution continues. Finally, after the last inner activity finishes, a transition fires and then place S at level L_n is marked. It is important to emphasize that in Figure 15, the Petri Net only shows the token movement pattern of the normal execution semantics of a composite group. The first activity in a composite group can be either an AP, an atomic group, or a composite group. However, no matter what activities a composite group contains, the activities are executed sequentially.

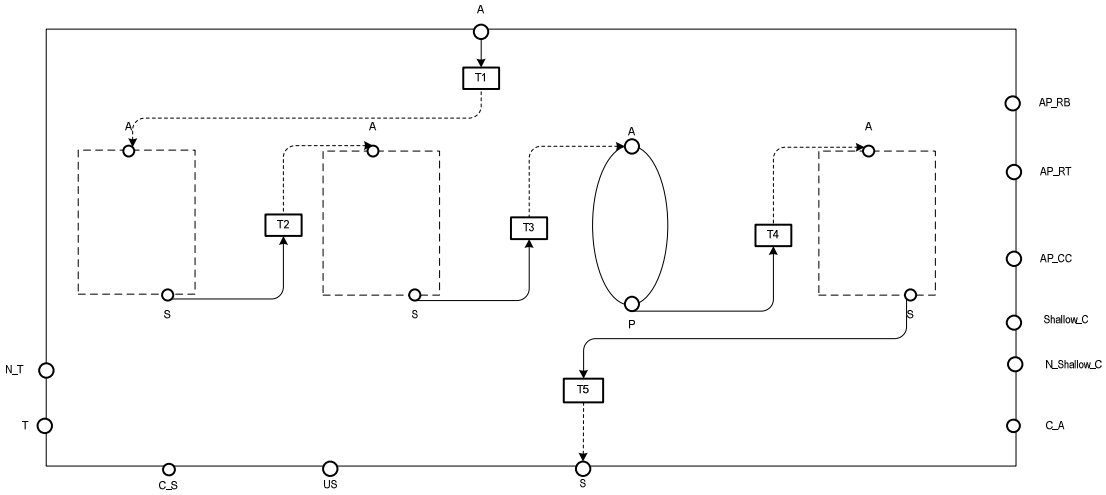

Figure 15. Petri Net of Composite Group with APs

Shallow and Deep Compensation

The semantics of deep and shallow compensation are shown in Figure 16 and Figure 17, respectively. The special place AP_ACTION is also introduced as a short hand notation for recovery actions. Because compensation is invoked under either AP-RB (rollback), AP-RT (retry), or AP-CC (cascaded contingency) mode, we introduce place AP_ACTION in Figures 16 and 17, representing either place AP_RB, AP_RT or AP_CC, since a process instance can only be under one of these recovery modes at any given time. So in Figures 16 and 17, places AP_ACTION at level L_n and L_{n+1} must represent the same mode status. For example, if in one scenario, place AP_ACTION represents AP_RB in Figure 16 at level L_n, all places AP_ACTION at level L_{n+1} must also represent AP_RB.

First consider deep compensation. Deep compensation is invoked directly when a composite group has no shallow compensation. The invocation of deep compensation is indicated by firing transition T1 in Figure 16. After firing T1, the token at the resource place N_Shallow_C at level L_n is consumed but also immediately returned. Also, the token at place AP_ACTION at level L_n is consumed and the places AP_ACTION and C_A at last inner group at level L_{n+1} are marked. Afterward, all groups at level L_{n+1} are backward compensated one by one through transitions T2 and T3. After place C_S at the first inner group at level L_{n+1} is marked, transition T4 is enabled to finish the compensation of the current level. The deep compensation ends when places C_S and AP_ACTION at level L_n are marked.

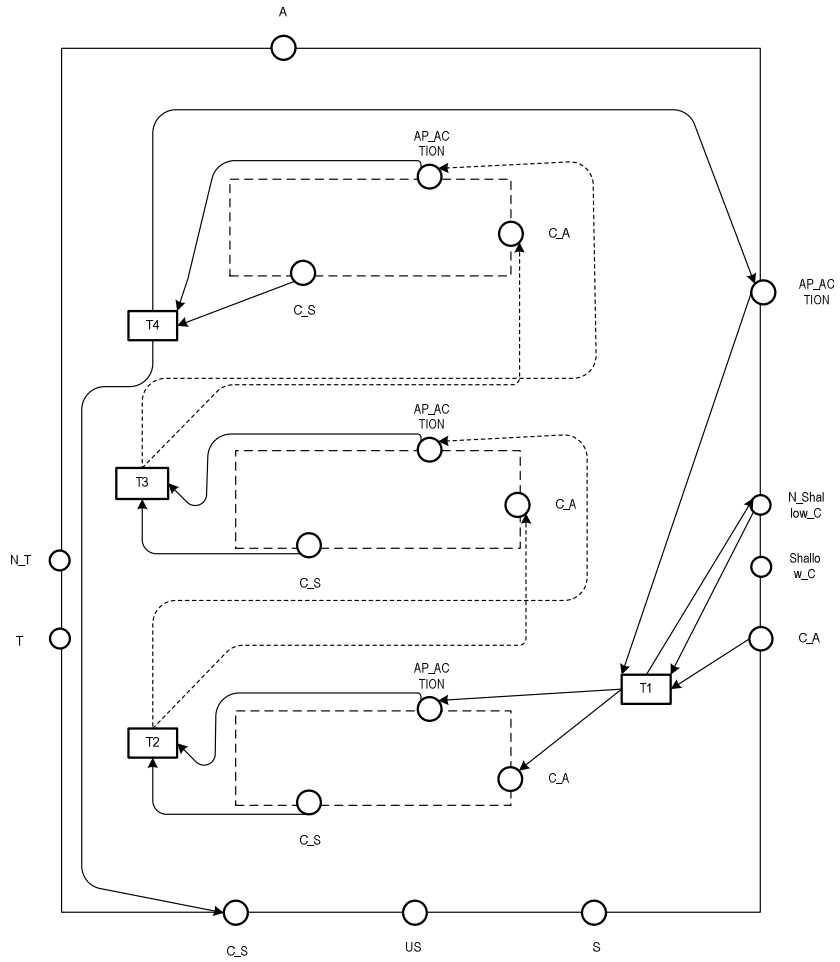

Figure 16. Petri Net of Deep Compensation

In Figure 17, the semantics of shallow compensation is presented. Shallow compensation invokes a pre-defined procedure to compensate the entire composite group rather than executing compensation for each group within the composite group. However, if shallow compensation fails, deep compensation is initiated. Firing transition T1 indicates the invocation of the shallow compensation procedure. If the execution of shallow compensation succeeds, place C_S at level L_n is marked by firing transition T2. Then the shallow compensation ends. Otherwise, transition T3 fires and the status place Shallow_C_Error is marked. To complete the compensation, deep compensation takes place by firing transition T4. Through transitions T5, T6 and T7, the deep compensation semantics is performed. Finally, places C_S and AP_ACTION at level L_n are marked.

Note that during either shallow or deep compensation, APs are ignored. Also, if the dashed-line quadrilateral represents an atomic group, compensation semantics defined in Figure 13 takes effect. If the dashed-line quadrilateral represents a composite group, either shallow or deep compensation invokes depending on the availability of the shallow compensation procedure.

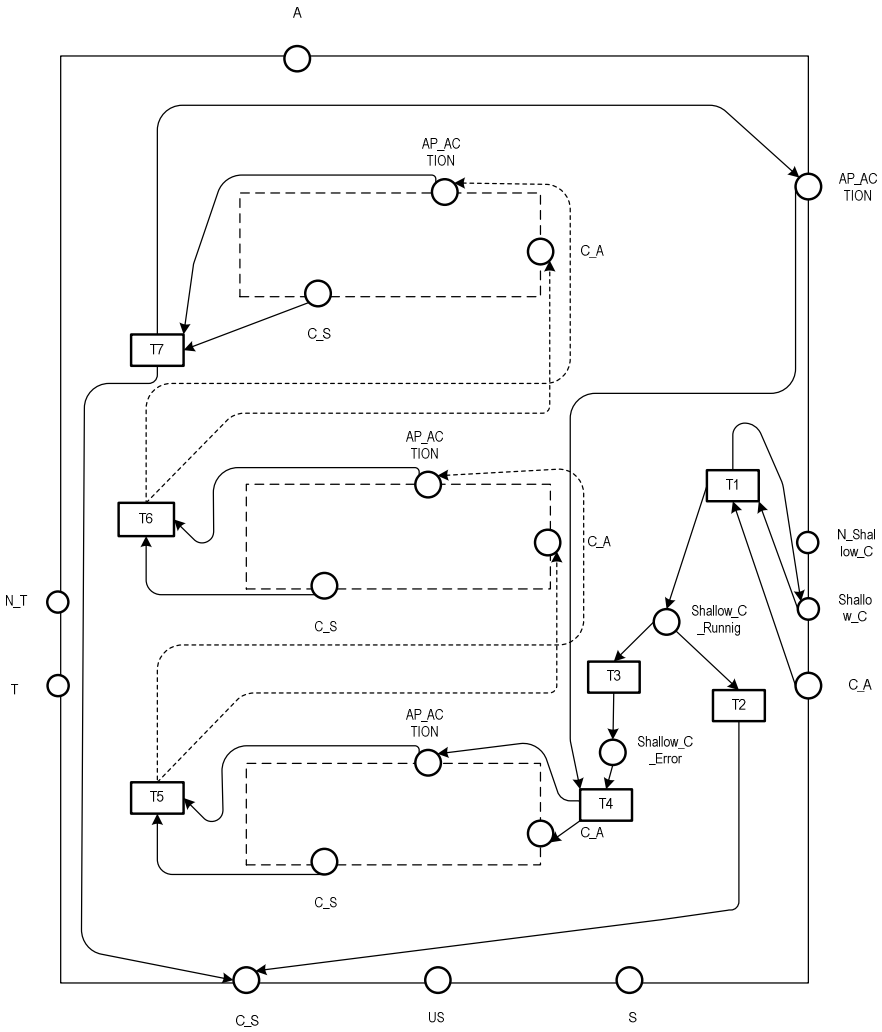

Figure 17 Petri Net of Shallow Compensation

AP-Rollback

AP-Rollback mode is triggered when a status place AP_RB at an AP is marked. As shown in Figure 18, transition T1 fires to begin the AP-Rollback mode at level L_{n+1}. The purpose of AP-Rollback is to recover the overall process. Thus, all completed groups need to be compensated under AP-Rollback mode. Through transitions T2 and T3, all completed groups at level L_{n+1} are compensated. When the first group at level L_{n+1} is compensated, transition T4 fires and the AP-Rollback mode is propagated to level L_n by marking the status places AP_RB and C_S at level L_n. Then, the same AP-Rollback semantics executed at level L_{n+1} will take effect at level L_n to further rollback the overall process. Note that during the backward recovery, the status place AP_RB at a completed group is marked when the group is compensating.

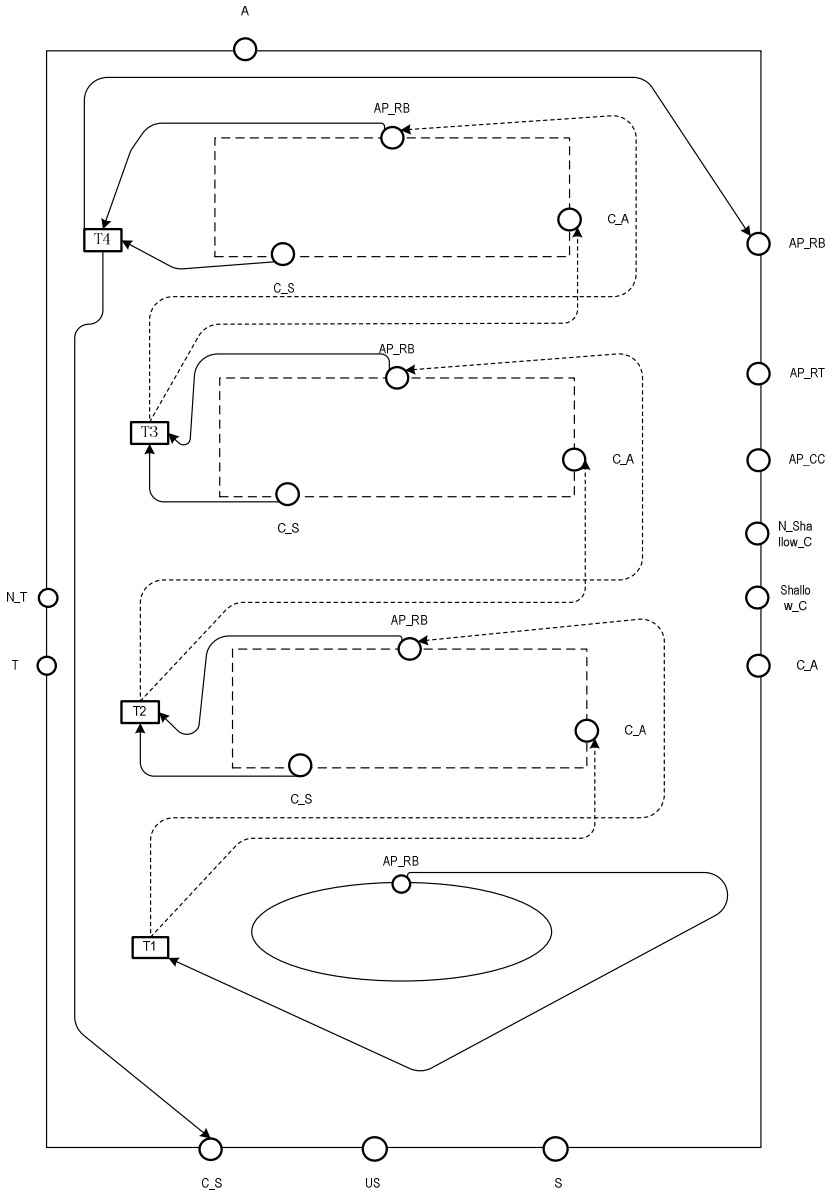

Figure 18. Petri Net of AP-Rollback

AP-Retry

When a status place AP_RT at an AP is marked, AP-Retry mode is triggered. Figure 19 presents the semantics of the default AP-Retry mode, which recovers the process back to the most recent AP and checks the pre-condition before the re-execution. In Figure 19, transition T1 fires to start the recovery. Similar to AP-Rollback, the status place AP_RT at a completed group is marked when the group is compensating. When the group just after the most recent AP is compensated, transition T3 fires and the place APRT_PRE at the most recent AP is marked. Then the pre-condition defined at the most recent AP is re-checked. If the pre-condition is satisfied, the status

place APRT_P is marked and transition T4 is enabled to start the retry process. If the pre-condition fails, another action will take place depending on the action specified in the rule.

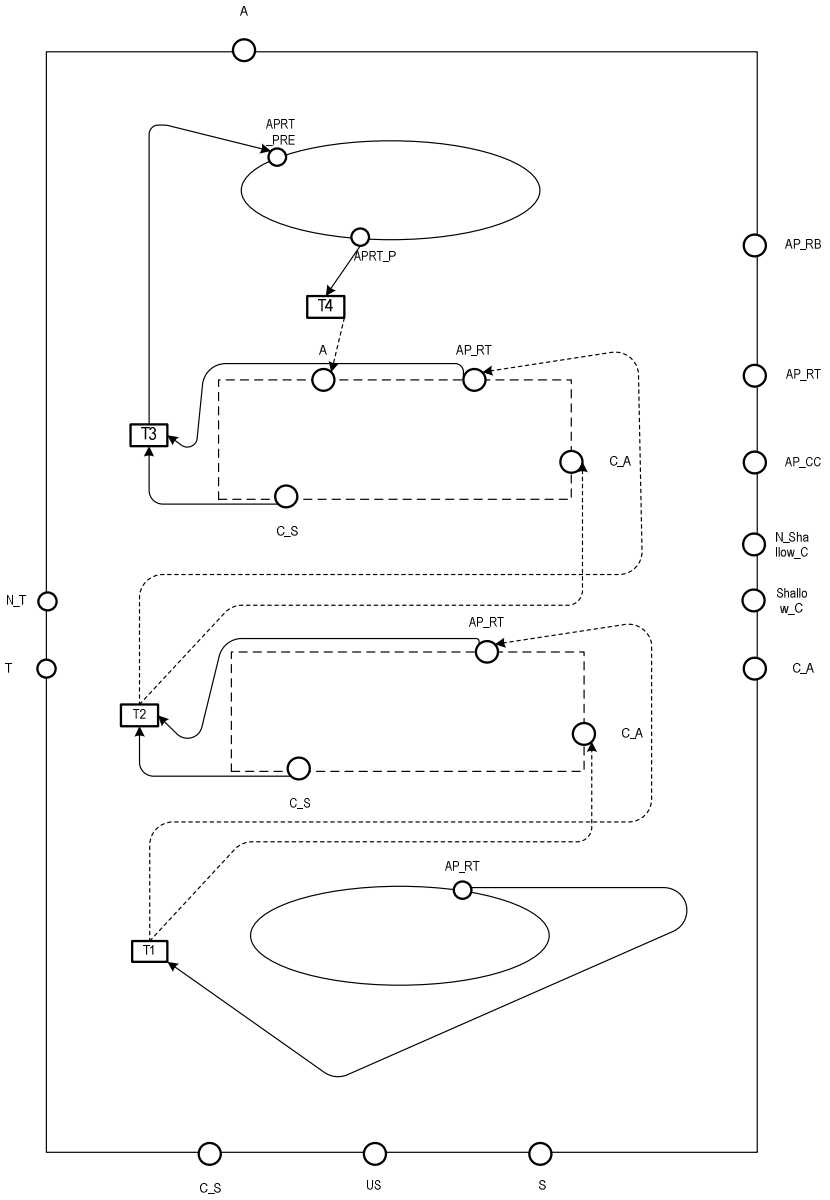

Figure 19. Petri Net of AP-Retry (default)

Figure 20 presents the semantics of re-checking the pre-condition under AP-Retry mode. In Figure 20, if the pre-condition exists, transition T1 fires. If the pre-condition is satisfied, the APRT_P is marked through transitions T2 and T3. If pre-condition is violated, transition T4 fires to mark the status place Pre-Violated. At this point, two different situations can occur. If the place PRE_VIO_F is marked, either transition T5, T6, T7 or T8 is fired to invoke the first action and then PRE_VIO_S is marked. If the place PRE_VIO_S is already marked, either transition T9 or T10 will be fired depending on the second action defined in the rule. In both cases, the process quits AP-Retry mode and enters a new mode that depends on the rule action.

Recall that the recovery process only allows AP-Retry to occur within one composite group. For example, AP-Retry only affects level L_{n+1} in Figure 19 and does not extend to level L_n. For the Petri Net of the parameterized AP-Retry, refer to (Gao, L. & Urban, S., 2010).

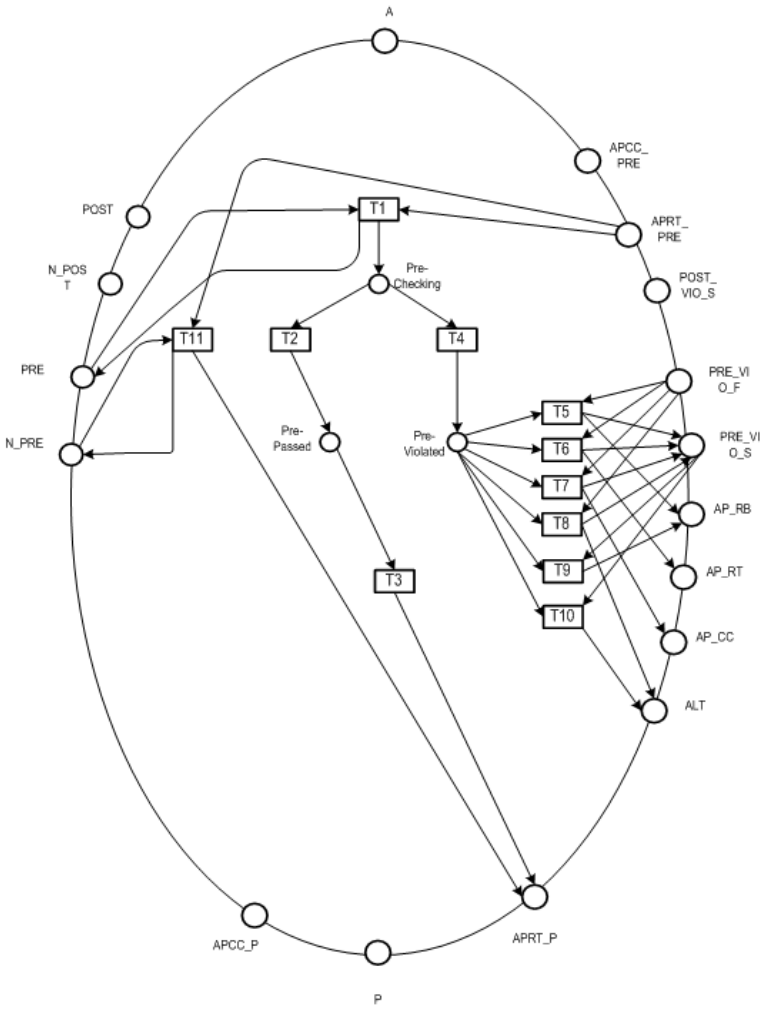

Figure 20. Petri Net of Re-Checking Pre-Condition (AP-Retry)

AP-Cascaded Contingency

Two situations will trigger the AP-CC mode. One is when the process encounters an execution error. The other is when a post or pre condition violation invokes the AP-CC action. Furthermore, if there is an AP just before the failed group, then the pre-condition will be checked before executing the contingency. As a result, there are several different execution scenarios for AP-CC mode. Only one case is shown in Figure 21. All other cases have similar Petri Nets, which can be found in (Gao, L. et al., 2010).

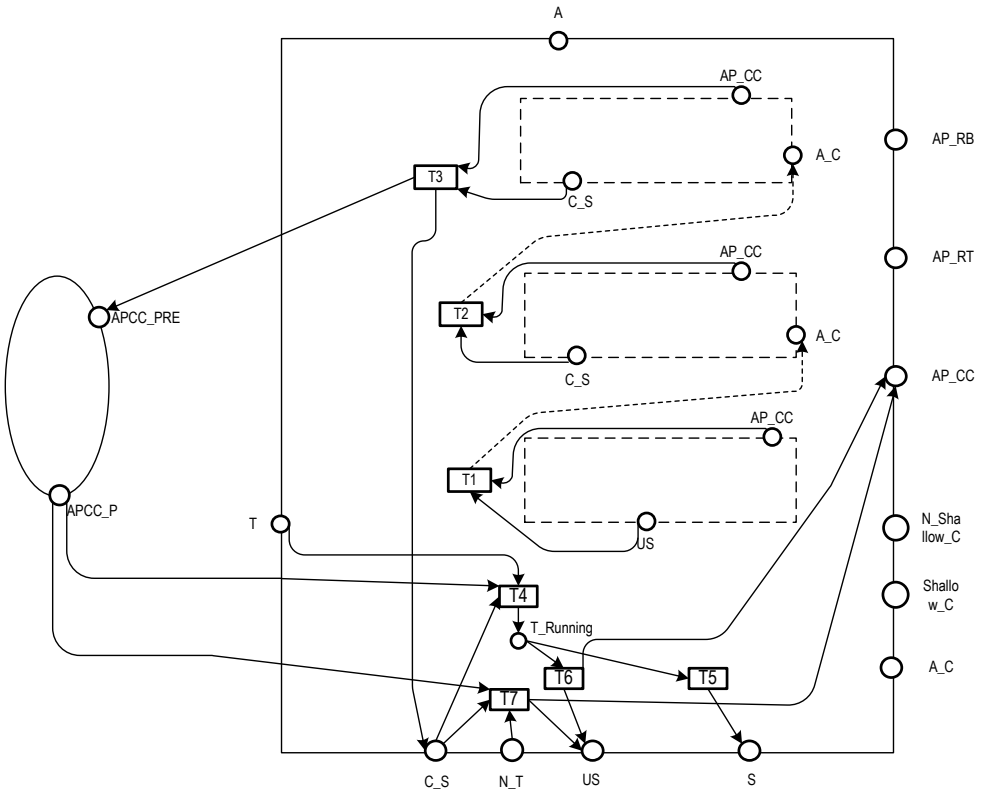

Figure 21. Petri Net of AP-CC (AP Exists Before Group)

Figure 21 presents the semantics of the AP-CC mode triggered by an execution error. An AP exists just before the failed group. If any error happens in a group at level L_{n+1}, the places US and AP_CC at the failed group are marked. This means the failed group has already tried the possible contingency at level L_{n+1}, but failed. To maximize forward recovery, the process attempts to execute the contingency at the outer level. First, transitions T1 and T2 fire to compensate all completed groups before the failed group at level L_{n+1}. After the first completed group at level L_{n+1} is finally compensated, transition T3 fires and the place C_S is marked. Since there is an AP before the compensated group at level L_n, the place APCC_PRE at the AP is marked as well after firing T3 so that the pre-condition is re-evaluated before trying the contingency at level L_n. If the pre-condition is satisfied, either transition T4 or T7 will fire depending on the availability of the contingency at level L_n.

If the contingency exists, there are two possible cases to consider. If the contingent procedure is successful, transition T5 fires. The process quits AP-CC mode by marking the place S at level L_n. If the contingent procedure is unsuccessful, transition T6 fires. The process is still under AP-CC mode by marking the places US and AP_CC at level L_n.

If the contingency at level L_n does not exist, transition T7 fires and the places US and AP_CC at level L_n are marked directly. The unsuccessful result will cause the process to search and execute other contingencies at the outer levels following the same semantics described in Figure 21.

Figure 22 presents the semantics of re-checking the pre-condition under AP-CC mode. This is the same semantics as in Figure 20, except that the AP logic starts at the place APCC_PRE and ends at the place APCC_P if the pre-condition is satisfied. If the pre-condition is violated, the process quits AP-CC mode and enters a new recovery mode depending on the action of the pre-condition rule.

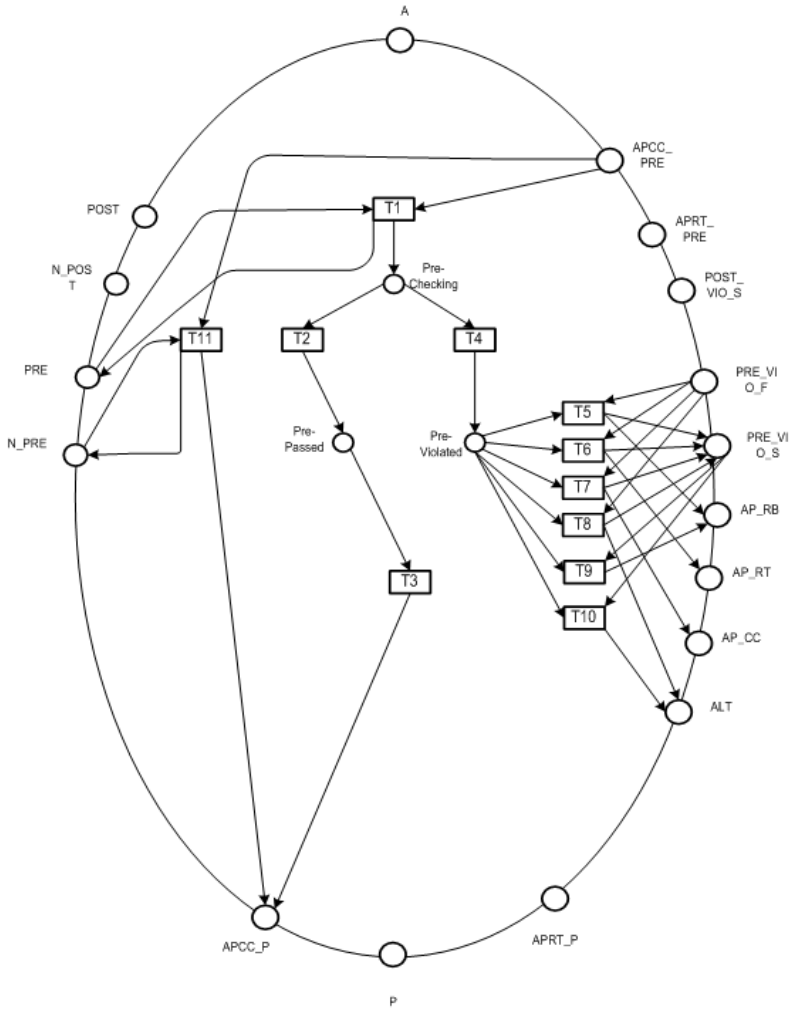

Figure 22. Petri Net of Re-Checking Pre-Condition (AP-CC)

PROTOTYPE IMPLEMENTATION

A prototype execution environment has been developed to demonstrate the extended service composition and recovery model with APs and integration rules. The execution environment does not directly use BPEL since the broader scope of the research is addressing techniques for decentralized data dependency analysis among distributed Process Execution Agents (PEXAs) (Urban, Liu & Gao, 2009). Existing BPEL engines do not provide the flexibility needed to experiment with this form of decentralized communication among process execution engines. BPEL also has limitations with respect to demonstrating the functionality described in this paper as outlined in the following section that addresses a comparison of the assurance point concept to the BPEL fault handler. The process specification framework, however, is based on BPEL using previous work with the Process Modeling Language (PML) described in (Ma, Urban, Xiao, & Dietrich, 2005).

The process specification framework uses a minimal set of activities, such as assign, invoke, and switch to illustrate the functionality of APs and the different forms of recovery. Figure 23 shows a sample process in XML to illustrate the syntax for defining atomic (<ag …>) and composite (<cg …>) groups with compensating (<cop …>) and contingent (<top …>) procedures.

The syntax for APs and their parameters is also illustrated (<ap …>). Integration rules are also specified using an XML format as shown in Figure 24

```
<cg name= “cg0”>
   ⋮
   <ap name= “OrderPlacedAP”>
      <apDataIn variable=“orderId” />
   </ap>
   <cg name=”cg2”>
     <ag name = “ag21”
        <invoke name=“chargeCreditCard” serviceName=“chargeCreditCard”
          portType=“cc:CreditCardPortType” operation=“chargeCreditCard”
          inputVariable = “chargeCardInput”
          outputVariable = “chargeCardtOutput” />
       <top name=“top21”>
           <invoke name=“eCheckPay” serviceName=“eCheckPay”
           portType=“cc:CreditCardPortType” operation=“eCheckPay”
           inputVariable = “makePaymentInput”
           outputVariable = “makePaymentOutput” />
       </top>
       <cop name=“cop21”>
          <invoke name=“creditBack” serviceName=“creditBack”
           portType=“cc:CreditCardPortType” operation=“ creditBack”
           inputVariable = “makePaymentInput”
           outputVariable = “makeRefundOutput” />
       </cop>
     </ag>
     .
     .
     .
   </cg>
   <ap name= “creditCardChargedAP”>
      <apDataIn variable=“orderId” />
      <apDataIn variable=“cardNumber” />
      <apDataIn variable=“amount” />
   </ap>
   ⋮
</cg>
```

Figure 23. PML Activity Syntax

The parser for the XML Java binding process has been implemented in the execution engine using XMLBeans. For each activity defined in a process, a wrapper class has been developed that implements the semantics of the activity. AP data is stored in a db40 object-oriented database. The functionality described in this paper has been fully developed to test and demonstrate all algorithms associated with the creation and use of APs, rules, and recovery procedures. The execution engine has also integrated the use of APs and recovery procedures into the <flowgroup> activity of BPEL to demonstrate how APs are used in the context of parallel execution threads. Discussion of the use of APs with the <flowgroup> activity, howver, is beyond the scope of the current chapter.

```
<rules>
   ⋮
<event ap="orderPlacedAP">
   <pre>
      <ecaRule>
         <condition name=“QuantityCheck”
            <invoke name="checkQuantity" serviceName="ruleConditions"
               portType=“rule:ruleConditionsPortType” operation="checkQuantity1"
               inputVariable="quantity" outputVariable="result" />
         </condition>

         <actions>
            <action name=“backOrderPurchase”>
               <invoke name="backOrderPurchase" serviceName="shopping"
               portType=“sho:ShoppingPortType” operation="BackOrderPurchase"
               inputVariable="orderId" outputVariable="result" />
            </action>
         </actions>
      </ecaRule>
   </pre>
</event>
   ⋮
</rules>
```

Figure 24. Integration Rule Syntax

COMPARISON TO THE BPEL FAULT HANDLER

This research has included a comparison of the AP model with the BPEL fault and compensation handlers. In BPEL, when a fault occurs, the fault handler attached to a scope catches the fault. The aim of the fault handler is to continue the process execution, which might require undoing certain actions already completed in the current scope. Since the compensation handler defines the semantics of undoing such changes, the fault handler may start the compensation handler (Khalaf et al., 2009). Similar to our approach of deep and shallow compensation, the <compensate> activity does the compensation of the completed activities in the nested scopes, whereas, the <compensateScope> activity causes compensation of one single completed scope. If any of the handlers are not specified, then the default handler is assigned to each scope. Default compensation invokes the installed compensation handlers for all the inner scopes. When the default compensation is applied to a scope, the compensation handlers are executed in reverse order of completion of the scopes.

The work in (Khalaf et al., 2009) highlights two main problems with the fault and compensation mechanism in the current BPEL standard. In particular, compensation order can violate control link dependencies if control links cross the scope boundaries. In addition, high complexity of default compensation order can result due to default handler behavior. Like BPEL, the AP model also honors control links between peer-scopes. Unlike BPEL, however, the order of compensation is clear since the AP approach does not support control links between non-peer scopes, making the semantics of compensation in the AP approach unambiguous. In addition, the

AP model supports a hierarchical structure during compensation as promoted in (Khalaf et al., 2009).

In general, the notion of compensation should also be capable of handling constraint violations (Coleman, 2005). Since BPEL's compensation handling mechanism through the <compensate> activity can only be called inside a fault handler, this limits the ability to call compensation outside of a fault handler. Thus, a fault has to occur to invoke a compensation procedure. In the case of the AP model, compensation can be invoked during normal execution (no error has yet occurred) when integration rules are not satisfied. This allows a flexible way to recover the process through compensation in response to constraint violations.

BPEL does not explicitly support a contingency feature other than fault, exception, and termination handlers. The designer is also responsible for complex fault handling logic, which, as pointed out in (Coleman, 2005; Khalaf et al., 2009), has the potential to increase complexity and create unexpected errors. The AP model provides explicit contingency activities so that forward recovery is possible. Compared to BPEL, the AP logic allows designers to have a clearer notion of how recovery actions take place and at the same time provide flexibility through different recovery actions depending upon the status of execution and user-defined integration rule conditions.

SUMMARY AND FUTURE DIRECTIONS

This research has defined a hierarchical service composition model that provides multi-level protection against service execution failure by using compensation and contingency at different composition granularity levels. The model has been enhanced with the concept of assurance points and integration rules to provide a flexible way of checking constraints and responding to execution failures. As a combined logical and physical checkpoint, an AP is used for user-defined consistency checking, invoking integration rules that check pre and post conditions at different points in the execution process. The unique aspect of APs is that they provide intermediate rollback points when failures occur that allow a process to be compensated to a specific AP for the purpose of rechecking pre-conditions before retry attempts. APs also support a dynamic backward recovery process, known as cascaded contingency, for hierarchically nested processes in an attempt to recover to a previous AP that can be used to invoke contingent procedures or alternate execution paths for failure of a nested process. As a result, the assurance point approach provides flexibility with respect to the combined use of backward and forward recovery options. Petri Nets have been used to define the semantics of the assurance point approach to service composition and recovery.

There are several directions for future research. As described in the implementation section, assurance points and the recovery actions described in this chapter have already been extended to support parallel execution threads within a process. We are currently in the process of extending the Petri Net formalization to describe the behavior of APs and recovery actions for parallel execution groups. We are also evaluating other high-level Petri Net theories, such as colored Petri Nets (Jensen, 1987), timed Petri Nets (Ramchandani, 1973), and the Workflow Net approach of Van Del Aalst (2005) to provide a more concise approach to description of the model.

Our research is also extending the concept of integration rules in several ways. One extension involves the use of invariant rules. Invariants provide a way to monitor the status of a condition in between two different APs to provide a more optimistic way for concurrent processes to access the same data. When a condition is violated, a process can be interrupted to invoke recovery procedures. Our initial results with the use of invariants are described in (Courter, 2010). We are also extending integration rules to the concept of application exception rules (AERs). AERs allow a process to be interrupted by an external event and to respond to the event depending on the

execution status of the process as determined by the most recent AP that has been executed. We are also integrating the use of AERs with the data dependency analysis algorithms in (Urban, Liu, and Gao, 2009) so that the recovery process can identify data dependencies among concurrently executing processes and use AERs as a means to communicate with the dependent processes of a recovered process about the need to check consistency constraints and possibly invoke recovery procedures.

ACKNOWLEDGMENTS

This research has been partially supported by NSF Grant CCF-0820152. Opinions, findings, conclusions or recommendations expressed in this paper are those of the author(s) and do not necessarily reflect the views of NSF.

REFERENCES

Alonso, G., Hagen, C., Schek, H.-J., and Tresh, M. (1997). Towards a platform for distributed application development. *Workflow Management Systems and Interoperability*, edited by A. Dogac, L. Kalinichenko, M. Ozsu and A. Sheth: Springer Verlag.

Baresi, L., Guinea, S., Pasquale, L. (2007). Self-Healing BPEL processes with dynamo and the JBoss rule engine. *ACM Int. Workshop on Eng. of Software Services for Pervasive Environments*, pp. 11-20. ACM New York.

Bennett, B., Hahm, B., Leff, A., Mikalsen, T., Rasmus, K., Rayfield, J., and Rouvellou, I. (2000). A distributed object-oriented famework to offer transactional support for long running business processes, *Proc. of Int. Conf. on Distributed Systems Platforms Middleware.*

Bhiri, S., Perrin, O., and Godart, C., (2005). Ensuring required failure atomicity of composite web services, *Proc. of the 14th Int. Conf. on the World Wide Web.*

Blake, L. (2005). *Design and implementation of Delta-Enabled Grid Services*, MS Thesis, Department of Computer Science and Engineering, Arizona State University.

Brambilla, M., Ceri, S., Comai, S., Tziviskou, C. (2005). Exception handling in workflow-driven web applications. *Proc. of the14th Int. Conf. on World Wide Web*, pp. 170-179. ACM New York.

Charfi, A., Mezini, M. (2006). Aspect-oriented workflow languages. *Lecture Notes in Computer Science 4275*, 183.

Charfi, A., Mezini, M. (2007). AO4BPEL: An aspect-oriented extension to BPEL. *WWW Journal: Recent Advances on Web Services*, March, 309-344.

Cichocki, A., Helal, S., Rusinkiewicz, M., and Woelk, D., (1998). *Workflow and Process Automation Concepts and Technology*: Kluwer Academic Publishers.

Coleman, J. (2005). Examining BPEL's compensation construct. *Workshop on Rigorous Eng. of Fault-Tolerant Systems*.

Courter, A. (2010). Supporting Data Consistency in Concurrent Process Execution with Assurance Points and Invariants, M.S. Thesis, Texas Tech University, Department of Computer Science.

Desel, J. (2005). Process modeling using petri nets. *Process-Aware Information Systems: Bridging People and Software through Process Technology*, 147-177.

Dialani, V., Miles, S., Moreau, L., De Roure, D., Luck, M. (2002). Transparent fault tolerance for web services based architectures. *Lecture Notes in Computer Science* 889-898.

de By, R., Klas, W. and Veijalainen, J. (1998). *Transaction Management Support for Cooperative Applications*: Kluwer Academic Publishers.

Eder, J. and Liebhart, W. (1995). The workflow activity model WAMO, Proc. of the *the 3rd Int. Conference on Cooperative Information Systems (CoopIs).*

Elmagarmid, A. (1992). *Database Transaction Models for Advanced Applications*: Morgan Kaufmann.

Fekete, A., Greenfield, P., Kuo, D., and Jang, J. (2002). Transactions in loosely coupled distributed systems. *Proceedings of Australia Database Conference* (ADC2003).

Gao, L. and Urban, S., (2010) A Formal Representation of the Assurance Point Service Composition and Recovery Model Using Petri Nets. Technical Report. Texas Tech University, Department of Computer Science, 2010

Jensen, K. (1987). Coloured petri nets. *Petri nets: central models and their properties*, 248-299.

Jin, T., and Goschnick, S. (2003). Utilizing web services in an agent based transaction model (ABT), *Proc. of the 1st Int. Workshop on Web Services and Agent-based Engineering.*

Jin, Y. (2004). *An Architecture and Execution Environment for Component Integration Rules*. Ph.D. Diss., Arizona State University.

Jin, Y., Urban, S., Dietrich, S., and Sundermier, A., (2006). An integration rule processing algorithm and execution environment for distributed component integration, vol. 30, *Informatica*, pp. 193-212.

Jin, Y., Urban, S., and Dietrich, S. (2007). A concurrent rule scheduling algorithm for active rules, *Data and Knowledge Engineering*, vol. 60, no. 1, pp. 530-546.

Kamath, M., and Ramamritham, K. (1996). Correctness issues in workflow management. *Distributed Systems Engineering* 3(4):213-221.

Kamath, M. and Ramamritham, K. (1998). Failure handling and coordinated execution of concurrent workflows, *Proc. of the IEEE Int. Conference on Data Engineering*.

Khalaf, R., Roller, D., Leymann, F. (2009). Revisiting the behavior of fault and compensation handlers in WS-BPEL. *On the Move to Meaningful Internet Systems: OTM 2009*, 286-303.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G. (2001). An overview of AspectJ. *Lecture Notes in Computer Science* 327-353.

Kifer, M., Bernstein, A., and Lewis, P. (2006). *Database Systems: An Application-Oriented Approach*. 2nd ed: Pearson.

Kuo, D., A. Fekete, P. Greenfield, and J. Jang. (2002). Towards a framework for capturing transactional requirements of real workflows. *Proceedings of the 2nd Int. Workshop on Cooperative Internet Computing*, Hong Kong.

Leymann, F. (1995). Supporting business transactions via partial backward recovery in workflow management, *Proc. of the GI-Fachtagung für Datenbanksysteme in Business, Technologie und Web (BTW'95)*.

Lin, F., Chang, H. (2005). B2B E-commerce and enterprise Integration: the development and evaluation of exception handling mechanisms for order fulfillment process based on BPEL4WS, *Proc. of the 7th IEEE Int. Conference on Electronic Commerce*.

Limthanmaphon, B., and Zhang, Y. (2004). Web Service Composition Transaction Management, *Proc. of the 15th Australasian Database Conf.*

Liu, A., Li, Q., Huang, L., Xiao, M. (2007). A declarative approach to enhancing the reliability of BPEL processes, *Proc. Of the Int. Conf. on Web Services*, pp. 272-279.

Luo, Z. W. (2000). Checkpointing for workflow recovery. *Proc. of the 38th Annual Southeast Regional Conf.*, pp. 79-80. ACM New York.

Ma, H., Urban, S.D., Xiao, Y., Dietrich, S.W. (2005). GridPML: A process modeling language and history capture system for grid service composition. *Proceedings of the International Conference on e-Business Engineering*, Beijing, China. (2005)

Mikalsen, T., Tai, S., and Rouvellou, I. (2002). Transactional attitudes: reliable composition of autonomous web services, *Proc. of the Workshop on Dependable Middleware-based Systems (WDMS)*, Part of the *Int. Conference on Dependable Systems and Networks (DSN)*.

Modafferi, S., Conforti, E. (2006). Methods for enabling recovery actions in WS-BPEL. *Lecture Notes in Computer Science* 4275, 219.

Modafferi, S., Mussi, E., Pernici, B. (2006). SH-BPEL: A self-healing plug-in for WS-BPEL engines. *Proc. Of the 1st Workshop on Middleware for Service-Oriented Computing*, pp. 48-53. ACM New York.

Murata, T. (2002). Petri nets: Properties, analysis and applications. *Proceedings of the IEEE, 77*(4), 541-580.

Paton, N.W., Díaz, O. (1999). Active database systems. *ACM Computing Surveys* 31.

Peterson, J. L. (1981). *Petri net theory and the modeling of systems*: Prentice Hall PTR Upper Saddle River, NJ, USA.

Ramchandani, C. (1973). *Analysis of asynchronous concurrent systems by timed Petri nets.* Massachusetts Institute of Technology.

Shrestha, R. (2010). *Using Assurance Points, Events, and Rules for Recovery in Service Composition.* M.S. Thesis, Texas Tech University.

Singh, M and Huhns, M. (2005). *Service-Oriented Computing: Semantics, Processes, and Agents*, J. Wiley & Sons, 2005.

Stahl, C. (2005). *A Petri net semantics for BPEL*, Technical Report 188, Humboldt-Universitat zu Berlin.

Tan, W., Fong, L., Bobroff, N. (2007). Bpel4job: A fault-handling design for job flow management. *Lecture Notes in Computer Science* 4749, 27.

Tartanoglu, F., Issarny, V., Romanovsky, A., and Levy, N. (2003). Dependability in the web services architecture, *Proc. of Architecting Dependable Systems*, LNCS 2677.

Wachter, H. and Reuter, A. (1992). The conTract Model," *Database Transaction Models for Advanced Applications*, A. Elmagarmid, Editor.

WS-Coordination (2005). Web Services Coordination, http://www-106.ibm.com/developerworks/library/ws-coor/.

WS-Transaction (2005). Web Services Transaction, http://www.ibm.com/developerworks/library/ws-transpec/.

Worah, D., and Sheth, A. (1997). Transactions in Transactional Workflows, *Advanced Transaction Models and Architectures*, edited by S. Jajodia and L. Kershberg: Springer.

Urban, S.D., Dietrich, S.W., Na, Y., Jin, Y., Sundermier, A., Saxena, A. (2001). The IRules project: using active rules for the integration of distributed software components. *Proc. of the 9th IFIP Working Conf. on Database Semantics: Semantic Issues in E-Commerce Systems*, pp. 265-286.

Urban, S. D., Gao, L., Shrestha, R., and Courter, A. (2010a). Achieving recovery in service composition with assurance points and integration rules, *Proceedings of the Cooperative Information Systems Conference* (Crete, Greece) as part of *On the Move (OTM) 2010*, Part 1, Lecture Notes in Computer Science 6426, Springer, Heidelberg, pp. 428-437.

Urban, S. D., Gao, L., Shrestha, R., and Courter, A. (2010b). The dynamics of process modeling: new directions for the use of events and rules in service-oriented computing, in *The Evolution of Conceptual Modeling*, R. Kaschek and L. Delcambre (editors), Lecture Notes in Computer Science 6520, pp. 205-224, Springer, Heidelberg.

Urban, S.D., Liu, Z.A., Gao, L. (2009). Decentralized data dependency analysis for concurrent process execution. *Proceedings of the 13th Enterprise Distributed Object Computing Conference Workshops* (EDOCW 2009) 74-83.

Urban, S. D., Xiao, Y., Blake, L., and Dietrich, S. (2009). Monitoring data dependencies in concurrent process execution through delta-enabled grid services, *International Journal of Web and Grid Services*, vol. 5, no. 1, pp. 35-66.

Van der Aalst, W. (1995). A class of Petri nets for modeling and analyzing business processes, Computing Science Reports 95/26, Eindhoven University of Technology, Eindhoven.

Van der Aalst, W. M. P. (1998). The application of Petri nets to workflow management. *Journal of Circuits Systems and Computers, 8*, 21-66.

Xiao, Y. (2006). *Using Deltas to Support Semantic Correctness of Concurrent Process Execution*, Ph.D Dissertation, Department of Computer Science and Engineering, Arizona State University.

Xiao, Y., Urban, S. D., and Dietrich, S. W. (2006). A process history capture system for analysis of data dependencies in concurrent process execution," *Proc. Second Int. Workshop on Data Engineering in E-Commerce and Services*, San Francisco, California, pp. 152-166.

Xiao, Y., Urban, S. D., and Liao, N. (2006). The DeltaGrid abstract execution model: service composition and process interference handling, *Proc. of the Int. Conf. on Conceptual Modeling (ER 2006)*, pp. 40-53.

Xiao, Y. and Urban, S. D. (2008). Process dependencies and process interference rules for analyzing the impact of failure in a service composition environment, *Journal of Information Science and Technology, vol. 5, no. 2, pp. 21-45.* Xiao, Y. and Urban, S. D. (2009). The DeltaGrid service composition and recovery model, *International Journal of Web Services Research*, vol. 6, no. 3, pp. 35-66.

Zeigler, B. P., and Sarjoughian, H. S. (2004). *DEVSJAVA*. Available from http://acims.eas.asu.edu/SOFTWARE/software.shtml#DEVSJAVA.

ABOUT THE AUTHORS

Susan D. Urban is a professor in the Department of Computer Science at Texas Tech University. She received the Ph.D. degree in computer science from the University of Louisiana at Lafayette in 1987. She is the co-author of *An Advanced Course in Database Systems: Beyond Relational Databases* (Upper Saddle River, NJ: Prentice Hall, 2005). Her research interests include Active/Reactive Behaviour in Data-Centric Distributed Computing Environments; Event, Rule, and Transaction Processing for Grid/Web Service Composition; Integration of Event and Stream Processing. Dr. Urban is a member of the Association for Computing Machinery, the IEEE Computer Society, and the Phi Kappa Phi Honour Society.

Le Gao received his B.S. degree in computer science in 2004 from Nanjing University of Aeronautics and Astronautics, China. He is currently a Ph.D. student in the Department of Computer Science at Texas Tech University. Before coming to Texas Tech University, he was a database engineer at China Mobile corporation from 2006-2007. He was also working at China Telecom Corporation as a database operator from 2004-2006.

Rajiv Shrestha received the M.S. degree in computer science from the Department of Computer Science at Texas Tech University in 2010. He is currently a software developer with Baker Hughes in Houston, Texas.

Yang Xiao received the Ph.D. degree in computer science from Arizona State University in 2006. She is currently a software testing engineer at Microsoft, focusing on integrated development environment testing methodologies and practices. Her research interests include process failure recovery and application-dependent correctness in Grid/Web service composition environment.

Zev Friedman is a student at Texas Tech University in an integrated B.S and M.S. degree program in Computer Science and Mathematics. He is currently participating in an NSF-funded Research Experience for Undergraduates program with research focused on service composition and recovery in service-oriented environments.

Jonathan Rodriguez is a student at Texas Tech University currently working towards a B.S. degree in Computer Science and B.A. in Mathematics. He is currently participating in an NSF-funded Research Experience for Undergraduates program with research focused on service composition and recovery in service-oriented environments. He is a member of the Association for Computing Machinery and Mathematical Association of America.