Using Assurance Points and Integration Rules for Recovery in Service Composition

by

Rajiv Shrestha, B.S.

A Thesis

In

COMPUTER SCIENCE

Submitted to the Graduate Faculty
of Texas Tech University in
Partial Fulfillment of
the Requirements for
the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

Approved

Dr. Susan D. Urban
Chairperson of the Committee

Dr. Michael Shin

Dr. Susan Mengel

Fred Hartmeister
Dean of the Graduate School

May, 2010

Copyright 2010, Rajiv Shrestha

ACKNOWLEDGMENTS

I am deeply grateful to my academic advisor, Dr. Susan Urban for her guidance and support at all times. Her advice and comments have been significant for the completion of this thesis.

I would like to thank Dr. Michael Shin and Dr. Susan Mengel for their helpful advice and assistance. Also, I would like to thank our research group members and all my other friends for being supportive.

This work is dedicated to my parents.

TABLE OF CONTENTS

ABSTRACT

This research defines the concept of Assurance Points (APs) together with the use of integration rules to provide a more flexible way of checking constraints and responding to execution errors in processes formed through service composition. An AP is a combined logical and physical checkpoint, providing an execution milestone that stores critical data and interacts with rules, known as integration rules, to alter program flow and to invoke different forms of recovery depending on the recovery mode and execution status. During normal execution, APs store execution state and invoke integration rules that check pre-conditions, post-conditions, and other application rule conditions. If a condition fails, recovery modes can be invoked that involve retry, rollback, and cascaded contingency. When execution errors occur, APs are also used as rollback points for backward recovery using compensation as well as forward recovery through rechecking preconditions before retry attempts and contingent procedures. This thesis describes the semantics of APs, integration rules, and the different forms of recovery actions, illustrating the functionality of the AP approach through the development of a prototype execution environment. The research is also evaluated through a comparison of the AP functionality for constraints and recovery to the BPEL fault and exception handling capability, as well as other relevant work with checkpointing and aspect-oriented programming. The primary contribution of this research is found in the definition of a service composition and recovery model with explicit support for user-defined constraints, contingency, and compensation that is embedded in well-defined recovery actions that make use of the execution state supported by assurance points to provide flexibility in the recovery process.

List of Tables

LIST OF FIGURES

CHAPTER I

INTRODUCTION

As Internet use is growing in this evolving Web era, more online businesses are emerging, thus increasing the use of Web Services and Service-Oriented Architectures (SOA) to enhance business-to-business and business-to-consumer transactions. Service-Oriented computing and Web Services ease the accessibility, availability, scalability, and reusability of software components. The work in (Papazoglou and Heuvel 2007) describes services as an exposed piece of functionality with three essential properties: (1) self-contained, as it maintains its own state, (2) platform independent with services running on different platforms that are independent and autonomous, and (3) dynamically located, invoked and (re-) combined. Web Services therefore allow customers as well as business partners to easily access the service without disrupting their own business processes. Furthermore, new services and processes can be created from the composition of other services. As a result, service composition is a key to the generation of new internet-based distributed applications.

Since Web Services and service-oriented computing are becoming more widely used for business-to-business integration, there is a need for providing better support for service composition, especially with respect to execution correctness and recovery. In the past several years, prevalent techniques such as the Unified Modeling Language (UML) (Engels et al., 2005), the Business Process Modeling Notation (BPMN) (White 2004), and Event-Driven Process Chains (Scheer, Thomas, and Adam 2005) have been widely adopted for process modeling at the conceptual level, with execution engines based on standards such as the Business Process Execution Language (BPEL) (Jordan et al., 2007) providing a framework for execution of conceptual process designs. Service composition for business integration, however, creates challenges for traditional process modeling techniques, especially considering the increasing use of events and rules to create greater execution flexibility through event-driven applications. Modeling extensions have been introduced to many of these tools to provide support for responding to events, handling exceptional conditions, and using events and rules as a way to control process flow. Most of these extensions, however, are still too rigid to support the type of flexibility that is needed for service-oriented environments.

In a service execution environment, a process must be flexible enough to respond to different types of events that represent errors, exceptions, and interruptions. Backward and forward recovery mechanisms (Lee et al., 1990) can be used to respond to such events. For example, a compensation handler is a backward recovery mechanism that performs a logical undo operation. Contingency is a forward recovery mechanism that provides an alternative execution path to keep a process running. Nevertheless, most service composition techniques do not provide flexibility with respect to the combined use of compensation and contingency. This absence of flexibility hampers the efficiency of exception handling and often does not do enough to keep processes running in a forward manner. Furthermore, most process modeling techniques for service composition do not make adequate use of pre-conditions, post-conditions, and other constraint checking techniques that can be used to validate the correctness of execution, especially considering that most processes execute in an environment that does not support traditional transaction processing. Service composition models need to be enhanced with features that allow processes to assess their execution state to support more dynamic ways of responding to failures, while at the same time validating correctness conditions for process execution.

The research presented in this thesis defines the concept of *Assurance Points* (APs) together with the use of *integration rules* to provide a more flexible way of checking constraints and responding to execution failures. The research is a subcomponent of a larger project addressing decentralized data dependency analysis and concurrently executing processes, where distributed execution units communicate about process failures, identify processes that are dependent on a failed process, and invoke the recovery procedure on dependent processes (Urban, Ziao, and Le 2009).

The research in this thesis is focused in enhancing the constraint checking and recovery procedures for individual processes with the goals of 1) strengthening the specification of user-defined correctness conditions, and 2) increasing the use of forward recovery when failure occurs.

In this thesis, APs are defined as an extension to the service composition and recovery model in (Xiao and Urban 2009). An AP is a combined logical and physical checkpoint, providing an execution milestone that stores critical data and interacts with integration rules to alter program flow and invoke different forms of recovery depending on the execution status. During normal execution, APs invoke integration rules that check pre-conditions, post-conditions, and other application conditions. Failure of a pre or post-condition can invoke several different forms of recovery, including backward recovery of the entire process, backward recovery to a specific AP for retry attempts, or a dynamic backward recovery process, known as cascaded contingency, in an attempt to find a previous AP that can be used to invoke contingent procedures or alternate execution paths. When failures occur, APs are also used as rollback points for rechecking preconditions and determining whether to invoke further forward or backward recovery actions.

This thesis describes the semantics of APs, integration rules, and the different forms of recovery actions. The functionality of the AP concept is illustrated using an online shopping example as well as other generic test cases that illustrate APs, integration rules, and recovery actions in the context of different nested composition scenarios. This thesis also outlines a prototype execution environment that has been developed to test the AP and integration rule concept in a BPEL-like execution environment. Finally, the research is also evaluated through a comparison of the AP functionality for constraints and recovery to the BPEL fault and exception handling capability as well as other relevant work with checkpointing and aspect-oriented programming. The primary contribution of this research is found in the definition of a service composition and recovery model with explicit support for user-defined constraints, contingency, and compensation that is embedded in well-defined recovery actions that make use of the execution state supported by assurance points to provide flexibility in the recovery process.

The remainder of this thesis is organized as follows. Chapter 2 gives an overview of the related work. Chapter 3 provides an overview of the service composition and recovery model with extended functionalities for APs and integration rules. Chapter 4 describes a prototype implementation of assurance points. Chapter 5 presents an evaluation of the assurance point concept. The thesis concludes in Chapter 6 with a summary and discussion of future research directions.

CHAPTER II

RELATED WORK

This chapter presents related work. Section 2.1 provides an overview of BPEL with a focus on fault, compensation, and exception handling capabilities. Section 2.2 outlines additional research related to fault and exception handling in workflows. Section 2.3 presents the related work that uses events and rules in workflow for handling failures and exceptions. Section 2.4 highlights research in aspect-oriented workflows to provide flexible and adaptable workflows. The chapter concludes in Section 2.5 with a comparison of the research in this thesis to related work.

2.1 BPEL

The interoperability of services by using standard protocols is necessary for consistency and advancement in Web services. Web service for Business Process Execution Language (WS-BPEL or BPEL) defines a model and a grammar for describing the behavior of a business process based on interactions between the process and its partners (Jordan et al., 2007). BPEL 2.0 is an Organization for the Advancement of Structured Information Standards (OASIS) standard, a high-level language for composing Web Services for modeling and executing workflows of business processes. Since BPEL is an officially approved standard for workflow language, it is desirable to compare the assurance points concept with BPEL and its advanced fault, compensation, and event handling features. BPEL fulfills the key requirements for a workflow language since it represents the business logic of the process, has the ability to provide asynchronous and synchronous invocations of Web services, supports long running transactions (LRTs), and manages failures, exceptions and recovery (Ezenwoye and Sadjadi 2006a).

BPEL consists of basic and structured activities to describe business process flow steps. Basic activities are primitive constructs for general tasks such as invoke for invoking Web services, receive for receiving a request, assign for operating on variables, wait for waiting for a time period, reply for generating a response, and terminate for terminating the whole process. These basic activities which are used for standard simple tasks can be combined with the help of structured activities to generate more complex processes. Structured activities like flow for allowing activities in parallel, pick to select one of the options, and while for loops are used for the combination of basic constructs and helps in asynchronous execution. A Scope in BPEL defines the collection of activities which has its own variables, partner links, message exchanges, event handlers, fault handlers, and compensation handlers. A scope can successfully or unsuccessfully terminate after execution of a business process, and in case of unsuccessful termination, it can reverse the activities, while at the same time other parts of the process can keep running. A partner represents both a consumer of a service provided by the business process and a provider of a service to the business process. The definition of properties creates a unique name for a WS-BPEL process definition and associates it with an XML Schema type.

2.1.1 Fault, Compensation and Event Handlers of BPEL

For any transaction to be safe and correct there must be some way to guarantee the integrity of the transactions. In traditional database systems, atomicity, consistency, isolation and durability (ACID) properties are used to guarantee the reliability of database transactions. Atomicity guarantees that all the tasks of a transaction are either committed or aborted. Consistency ensures the database remains in a consistent state through checking the changes in data values. The isolation property is required during concurrent transactions where execution of one transaction should not affect the execution of another transaction. Durability guarantees that once committed, a transaction persists and cannot be undone.

Even though these ACID properties are fundamental for traditional databases, they are not suitable for long running transactions during service composition. Each service in a process is autonomous and platform independent. The commit of a service execution is controlled by the residing service instead of the global process. As a result, distributed processes composed of services do not execute as traditional transactions. The concept of serializability is too strong for concurrently executing services to conform to global transaction semantics as one process. As a result, ACID properties and traditional concurrency control mechanisms are not generally suitable for this environment, since a process cannot afford to block individual services to ensure a commit of the global process (Mikalsen, Tai, and Rouvellou 2002). Hence, dirty writes and dirty reads are inevitable since a service can commit before a process completes which can cause data inconsistency problems. Therefore, undoing the transaction or finding an alternative solution is crucial in such situations. To address this issue in a service environment, BPEL has defined the role of fault, event, and compensation handlers.

A fault handler helps to undo the partial job done within a scope when an exception occurs during run-time. Faults can be explicitly generated through the throw activity. A catch construct is used to catch a specific fault and a catchAll constructs to handle all other faults not caught by a catch fault handler. There are three kinds of faults in BPEL (Jordan et al., 2007). The first one is *application/service faults* which are generated by services invoked by the process. Another fault type is *process defined faults* which are generated by the throw activity. The third type of fault is *system faults* which are generated by the process engine. We can add fault handlers to the process (global) or to a scope within the process (local). Once the fault occurs at the process or scope level, the scope is terminated and the corresponding fault handler takes control. In such case, the scope is said to be faulted and is not qualified for compensation. If the faults are not handled in the current scope or the fault handler cannot resolve a fault, then it is re-thrown by rethrow to the parent scope. If any nested scope is already completed without a fault being thrown or re-thrown, then it qualifies for compensation as backward recovery.

The work in (Ezenwoye and Sadjadi 2006b) points out the limitations of fault handling in BPEL, where in case of system faults, which are common in service-oriented architecture (e.g. unavailability of Web services), the catchAll can catch these faults in BPEL, but it cannot recognize the difference between faults. Knowing the different system faults is necessary to take corresponding related action, thus such an approach in BPEL is not desirable for providing good quality of service.

Compensation provides execution semantics with the relaxed notion of undoing a successfully completed activity in the scope or process level of a business process. Every scope of the BPEL process has a compensation handler which can revert its effects in the reverse order of the execution of activities. The compensate and compensateScope activities are used within the fault handler, compensation handler, or termination handler to invoke the compensation handlers where compensate causes execution of compensation of all completed and not compensated child scopes in default order, whereas compensateScope causes execution of compensation of one specified successfully completed child scope.

The event handler in BPEL specifies logic to deal with events. Event handlers are associated with a whole process or scopes where activity is invoked concurrently when the corresponding event occurs. Two kinds of events are available in BPEL: Message Events and Alarm Events. Alarm events are useful if the process needs to wait for certain period of time since Web services are not always available in loosely coupled service oriented architecture. A message event is useful when the business process needs to wait for several multiple messages. Therefore, events can be helpful for activities that cannot be scheduled, like a customer cancelling a flight in the middle of the process. Event handlers are a normal part of the BPEL process, unlike fault and compensation handlers.

2.1.2 BPEL's Shortcomings

Even though faults, compensation, and event handlers give a very basic recovery mechanism, WS-BPEL provides these three as a standard mechanism and leaves the rest up to the designer about any other task specification when the handler is fired (Modafferi and Conforti 2006). Although it gives power to the designer, a lot of effort is required if the designer wants to specify advanced recovery procedures. Also, BPEL does not have any distributed coordination in regard to multiple concurrent services (Gannod, Burge, and Urban 2007). BPEL offers the above standard features to support transactional integrity, but lack of formal semantics makes it hard to implement and guarantee that a BPEL process behaves correctly (Breugel and Koshkina 2006). Moreover, transactions are long running, therefore, it is more difficult to analyze such transactions without formal semantics.

Formalizing BPEL will lead to several benefits. Several formal approaches have been researched and implemented to formalize BPEL, such as Petri-Nets (Desel 2005) and π-calculus (Sangiorgi and Walker 2001). The work in (Kovács, Varró, and Gönczy 2007) presents a formal modeling technique for BPEL workflows with model checking. BPEL_{fct} is a formalization of BPEL focused only on fault, compensation and termination (FCT) handling (Eisentraut and Spieler 2009). In (Rouached, Perrin, and Godart 2006), the authors propose an event-driven approach for formalizing and verifying Web service composition expressed in BPEL. The work offers the consistency checking of a business process in three cases: static verification (before running process), dynamic verification (runtime), and non-functional requirements. The framework is still under development. The work in (Kuhne et al., 2008) presents a prototype of a modeling tool that uses graph-based rules to find problems in business process models. Several ongoing research projects are based on self-healing BPEL to overcome the lack of formal semantics and to provide automatic service composition and adaptation (Breugel and Koshkina 2006), (Kovács, Varró, and Gönczy 2007).

In (Modafferi and Conforti 2006), the authors provide three options to overcome BPEL's limitations for the support of recovery actions. The first option is to define a totally new workflow language and engine. Another option is to define an extended BPEL and the corresponding extended engine, as in (Modafferi, Mussi, and Pernici 2006) and (Dialani et al., 2002), which gives the designer the option of advance recovery mechanisms with very few changes to the current technology. The last option is to use the concepts of annotation and preprocessing for enhancing BPEL at design time without modifying the workflow engine as in (Baresi, Guinea, and Pasquale 2007) and (Wang, Bandara, and Pahl 2009).

2.2 Research on Fault and Exception Handling

A fault is an abnormal condition or defect at the component or sub-system level which may lead to failure (Chan et al., 2006), whereas exceptions are facts or situations that are raised to signal errors, faults, failures, and other deviations which depend on what we want and what we can achieve (Luo et al., 2000). One of the major issues in distributed service-oriented applications is fault management. There is no guarantee that a composition of even good services will always work (Chan et al., 2006). Run-time strategies which check whether the composition behaves correctly and reactive strategies to detect and recover from errors can be used to ensure the correctness of the composition. Several mechanisms are being developed to discover and recover from faults automatically (Modafferi, Mussi, and Pernici 2006) (Friese, Muller, and Freisleben 2005) (Baresi, Guinea, and Pasquale 2007). The standard orchestration language BPEL provides mechanisms like fault handlers, exception handler, termination handlers, and compensation handlers for managing recovery activities as described in Section 2.2. This section highlights the current research in fault tolerance mechanisms in Web Service composition.

The research in (Brambilla et al., 2005) recognizes three types of exceptions to clarify the conditions under which failures occur: *behavioral exceptions*, which are user-generated due to improper execution order of process activities, *semantic or application exceptions*, which are due to unsuccessful logical outcomes of activity execution, and *system exceptions*, which are caused by malfunctioning of the Web application at the client and server side, such as network failure or system breakdown. The work in (Eder and Liebhart 1996) highlights the sources of failures in workflow that can be from i) workflow engine failures, ii) activity failures, or iii) communication failures between scheduler and activities. Common ways of handling these failures are rollback and compensation for backward recovery, contingency for forward recovery, re-try, undo, timeout, safe termination, executing alternative activities, or even human interactions (Greenfield et al., 2003). However, in service-oriented architectures, a single process may be part of multiple applications due to data dependency during concurrent execution; thus, rollback does not always help to recover from the failures (Dialani et al., 2002). Compensation is also not always enough to handle and recover failures during LRTs (Greenfield et al., 2003). Therefore, more strong and dynamic ways of handling errors are required for fault tolerant systems.

Many efforts have been made to enhance the standard BPEL’s fault and exception handling capabilities. BPEL4Job (Tan, Fong, and Bobroff 2007) is an extended BPEL for fault-handling design for job flow management in distributed computing environments which has cleanup, task level re-try, and flow re-submit policies with the novel idea of migrating flow instances between different flow engines for scalable failure recovery. Moreover, it uses a job proxy to facilitate the asynchronous nature of job submission and notification which helps to extend the re-try policy with advance schemes, like for example to alter the input parameters. Since there is an increasing complexity of processes and autonomous agents in workflow, self-healing mechanisms are necessary for automatic recovery during run-time. The work in (Modafferi and 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 the standard BPEL language. These sophisticated recovery strategies can be used for developing a self-healing engine. The work in (Modafferi, Mussi, and Pernici 2006) presents the architecture of SH-BPEL engine, a Self-Healing plug-in for BPEL engines. SH-BPEL augments the fault recovery capabilities in BPEL with mechanisms like annotation, pre-processing, and extended recovery. Moreover to support self-healing execution of business processes, (Friese, Muller, and Freisleben 2005) provides the middleware framework called Robust Execution Layer (REL) that acts as a transparent, configurable add-on to any BPEL engine to support the peer-to-peer communication failure during interaction with business process engines in distributed environment. Dynamo (Baresi, Guinea, and Pasquale 2007) adds recovery capabilities to BPEL processes to create self-healing BPEL compositions using two special languages:

WSCol for specifying constraints and WSRel for executing state recovery strategies when constraints are violated.

Failures are not always easily detectable in Web services, thus methods are required to automatically detect failures in a self-healing environment such as in (Baresi, Ghezzi, and Guinea 2004) which proposes two methods for dynamic detection of failures. One method is Defensive Process Design, in which services are designed to cope with failures. Another method is service run-time monitoring where external monitoring tools are used to check violations of functional and non-functional properties. Other ways of detecting faults are by using monitoring and verifying tools such as the ASTRO toolset (Trainotti et al., 2005), WSAT (Web Service Analysis Tool) (Fu, Bultan, and Su 2004), SPIN (Holzmann 2004), and BPELCheck (Fischer, Majumdar, and Sorrentino 2008) which provides execution monitoring facilities that check the predefined properties like pre-condition and post-condition of the processes and give feedback in the event of a failure, thus it helps to check the consistency of the BPEL processes. However, these monitoring and verifier tools do not suffice since some services may be outside the control of the developer (Ezenwoye and Sadjadi 2006b). The more dynamic approach of monitoring BPEL-processes and embedding the monitored process into a WS-BPEL engine is given in (Baresi and Guinea 2005).

Promises (Jang, Fekete, and Greenfield 2007) is a model for Web service applications that addresses the situation of lack of isolation mechanisms in LRTs by providing assurance that the resources are available and not violated during a certain period of time. A Promise is an agreement between a client and the resource owner which helps to maintain the integrity constraints in the workflow so that operations can be completed successfully. The Promise system has three components: a *Promise Manager* for recording all active promises in the promise table, an *Application* for processing the activities requested by the Promise Manager, and a *Resource Manager* for storing and updating the state of the system. Here, the most important task is guaranteeing the validity of promises, thus the promise system has different ways of checking and validating or updating promises as described in (Jang, Fekete, and

Greenfield 2007). The Promise system helps to handle the concurrent processes in Web-service applications.

Checkpointing techniques are helpful to increase the efficiency of the system in case of failure. In checkpointing, consistent execution states are saved to obtain checkpoints during the process flow. During failures and exceptions, the activity can be rolled back to the closest consistent checkpoint, resuming the execution from that point (Luo 2000) rather than the whole process, which can consume extra resources and reduce the efficiency of the system. The checkpointing method can be used with events and rules, where the state of the execution can be used to determine the responsible action towards application events, exceptions, and faults. The work in (Marzouk et al., 2009) presents the periodic checkpointing-based approach which can be used as a self-healing mechanism to recover from stopped process instances due to failure in the workflow. The work in (Dialani et al., 2002) provides the fault tolerant architecture for Web services to detect the faults and to recover by means of checkpointing and rollback. The AP concept presented in this thesis also stores critical execution data that can be used for constraint checking and passing parameters to rules that invoke different types of recovery actions.

In addition to above techniques to handle faults and exceptions in workflows, several formalization and validating techniques (Desel 2005) (Sangiorgi and Walker 2001) (Kovács, Varró, and Gönczy 2007) as described in Section 2.3 can be used to guarantee the correctness, to avoid ambiguities and inconsistencies, and to also monitor failures. Moreover, semantics can be added to more dynamically recover from failures. A Semantic Web Service is "a means for providing service specifications with rich semantic annotations that facilitate flexible dynamic discovery, invocation and composition of services" (Wiesner et al., 2008). The Ontology Web Language for Services (OWL-S) (Martin et al., 2005) brings semantics to Web services, which can support automation and dynamism during service composition by providing declarative descriptions to Web service. The work in (Vaculín, Wiesner, and Sycara 2008) gives the exception handling and recovery mechanisms in OWL-S by introducing constraint violation handlers (CV-handlers) and combining them with event handlers. Moreover, (Wiesner et al., 2008) adds semantic annotation to these existing methods to generate more dynamic, flexible and adaptive ways of handling and recovering from failures. Methods like *ReplaceByEquivalent* and *Advanced Back and Forward Recovery* actions help to dynamically find alternatives to erroneous state, whereas as *Automatic Compensation* method uses the semantic information to undo the completed processes. Therefore, semantic web services can be a key solution for achieving dynamism with reliable and adaptable service executions.

Earlier work with fault and exceptional handling in transactional workflow can be found in work such as the ConTract model (Wächter and Reuter 1992) and the CREW project (Kamath and Ramamritham 1998). The ConTract Model supports the correct execution of non-atomic, long-lived applications with application-dependent consistency constraints. The model provides a mechanism for grouping transactions into a multi-transaction activity. A ConTract consists of a set of predefined actions (steps) and an explicitly specified execution plan (script). The ConTract Model provides compensation for backward recovery, and user-defined consistency through the specification of pre-conditions or post-conditions for steps. After the execution of each step, the ConTract Model will release locks and if failure occurs, the ConTract Model will semantically undo the effect of completed steps. The pre-/post-condition guarantees the user-defined way of specifying correctness criteria. In the Correct and Reliable Execution of Workflows (CREW) project (Kamath and Ramamritham 1998), the correctness requirements and other constraints are specified for workflow executions based on the earlier work on transactional workflows such as ConTract model. A workflow executes in multiple steps, where a step is triggered by the completion of one or more previous steps, or the occurrence of specific events. The rules, events or conditions predefined will be used to dynamically generate the rule sets to manage the execution of workflows. A mechanism is proposed for the handling of failures to eliminate unnecessary compensations and re-execution of steps. Depending on whether the previous execution of steps is acceptable, complete compensation and re-execution, or partial compensation and incremental re-execution is used to undo the effects. Therefore, CREW makes the execution of workflows more dynamic by the use of dynamic rule sets. The handling of failures and exceptions can be better managed during execution.

Most of these projects do not fully utilize pre and post conditions or other constraint checking mechanisms integrated with a variety of recovery actions to support more dynamic and flexible ways of reacting to failures. The research described in this thesis demonstrates the viability of variegated recovery approaches within a BPEL-like execution environment.

2.3 Events for Handling Failures and Exceptions

An important aspect of business processes is to integrate them with business events and rules, which can enforce business policies and constraints during the execution of a business process. Event driven architectures provide an approach for designing and creating applications where events trigger certain actions in real-time (Michelson 2006). An event is a notable thing that may signify a problem or implement a problem, an opportunity, a threshold, or a deviation (Michelson 2006). A rules-based event processing agent may be used to listen to incoming events. Also, events can be used with rules for failure and recovery of activities. Rules provide a more dynamic way to react to events, providing an alternative to the normal flow of execution and creating more reactive and dynamic systems. Since there is an increasing occurrence of complex events, event-driven applications, and business activity monitoring, the use of rules in business process modeling is an increasing necessity.

Events can be classified into application-oriented events, application exceptions, and system faults associated with service execution (Gannod, Burge, and Urban 2007). There are three general ways of processing the events (Michelson 2006): simple, stream, and complex. Events and rules can be used to control the flow of execution as wells as handling failures and exceptions that affect the normal flow of execution. Active rules have been used to extend traditional database systems which are able to monitor and react to specific circumstances of relevance to an application by using the Event Condition Action (ECA) rules (Paton and Díaz 1999). Rules can be used to automatically execute actions in the case of an event, provided that the condition holds. Details about the benefits, challenges, and limits of using ECA rules for business processes can be found in (Bry et al., 2006). ECA rules are useful for efficient and convenient exception handling, since exceptions can be easily expressed as events. Even though some languages may only use ECA rules for controlling workflow, as in XChange (Bailey et al., 2005), workflow can be easily extended and implemented to handle failures and events using ECA rules. ECA rules have been successfully implemented for exception handling in (Brambilla et al., 2005) (Liu et al., 2007). The work in (Liu et al., 2007) uses ECA rules to handle faults and then integrate ECA rules with normal business logic to generate reliable and fault-tolerant BPEL processes to overcome the limited fault handling capability in BPEL. Thus the use of rules increases the productivity and reusability by separating fault handling logic from normal business logic. Their future work is to use semantics for more efficient fault handling. In (Luo et al., 2000), justified ECA (JECA) rules are used to handle exceptions. In addition, a case-based reasoning (CBR) system is introduced which can provide a method to understand the exceptions and retrieve similar prior exception handling cases. The system can reuse the exception handling experiences captured in case of new circumstances.

2.4 Aspect-Oriented Workflows

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 where join points are specified. Join points are well-defined points in the execution of the program. The behavioral code specified in the join point is known as *advice*. The advice code can be executed *before*, *after*, or *instead* of the join points (Charfi and Mezini 2007). The work in (Charfi and Mezini 2006) illustrates the application of aspect-oriented software development concepts to workflow languages to provide flexible and adaptable workflows. AO4BPEL is presented in (Charfi and Mezini 2007) as an aspect-oriented extension to BPEL, where aspects can be plugged into the composition during runtime. The system uses AspectJ, which provides control flow adaptations such as insertion of a new activity to the process or replacement of an activity by another (Kiczales et al., 2001). AO4BPEL enhances the limited capabilities of BPEL in terms of modularity and dynamic adaptability. Aspects are written in XML in different files, helping to minimize the need for changing the composition during runtime. Business rules can also be used to provide more flexibility during service composition. Currently business rules are not well modularized in BPEL process specifications, thus AO4BPEL successfully addresses such issues. APs as described in this paper are similar to the concept of 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.

2.5 Conclusion

In this related work section, we have described several past and ongoing research projects that support dynamic Web service composition with respect to fault and exception handling. Due to the distributed nature of services, the 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 as described in BPEL shortcomings in Section 2.3. The research in this thesis is different than the related work by providing comprehensive support for user-defined constraints with the use of pre, post, and conditional rules. In addition, the AP model integrates the rules with different recovery actions as well as user-defined compensation and contingency. Thus, our 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.

CHAPTER III

OVERVIEW OF SERVICE COMPOSITION AND RECOVERY MODEL WITH ASSURANCE POINTS

The research described in this thesis is an extension of the service composition and recovery model described in (Xiao and Urban 2009). The model is based on BPEL, with a nested composition structure and support for compensation and contingency. The model was originally defined to support a more flexible environment for research involving data dependency analysis and recovery procedures between concurrent processes (Xiao and Urban 2007), (Xiao and Urban 2008). This chapter gives an overview of the model in Section 3.1. Section 3.2 then presents assurance points, integration rules, and recovery actions supported by APs. An online shopping example is presented in Section 3.3 to illustrate the concepts.

3.1 Overview of Service Composition and Recovery Model

In (Xiao and Urban 2009), a process is defined as a top-level execution entity that is composed of 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 ($cop_{i,j}$) is an operation intended for backward recovery, while contingency ($top_{i,j}$) 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 (denoted $ag_{i,j}$) contains an operation, an optional compensation, and an optional contingency. A composite group (denoted $cg_{i,k}$) 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.

Figure 1 shows an abstract view of a sample process definition. The 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 and composite group can have an optional compensation plan and/or contingency plan. Some operations, such as $op_{1,4}$, can also be marked as non-critical, meaning that the failure of the operation does not invoke any recovery activity and that the process can proceed even if the operation fails.

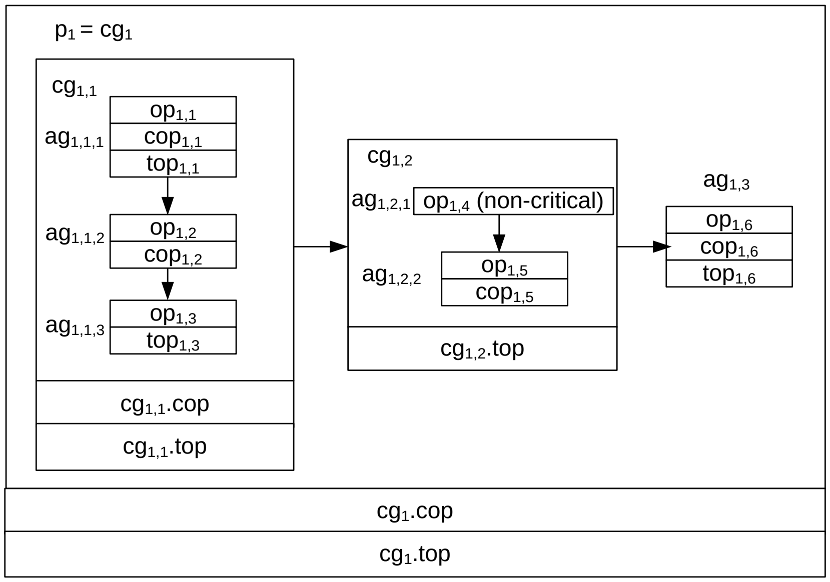

Figure 1. An Abstract View of a Sample Process (Xiao and Urban 2009)

Contingency is always tried first upon the failure of a group. Compensation will only be invoked if there is no contingency or if the contingency fails. For example in Figure 1, if $op_{1,6}$ fails, $top_{1,6}$ will be executed. If $top_{1,6}$ fails, $cg_{1,2}$ and $cg_{1,1}$ will be compensated in that order.

Compensation is a recovery activity that is only applied to completed atomic and composite groups. Shallow compensation involves the execution of a compensating procedure attached to an entire composite group, while deep compensation involves the execution of compensating procedures for each group within a composite group. As an example in Figure 1, if the contingent procedure for $op_{1,6}$ fails, the recovery process will first try to compensate $cg_{1,2}$. Since $cg_{1,2}$ does not have a compensating procedure for the entire group (i.e., no shallow compensation procedure), deep compensation will be invoked by executing $cop_{1,5}$. Note that $op_{1,4}$ is non-critical and does not require compensation. After deep compensation of $cg_{1,2}$, $cg_{1,1}$ will be compensated. In this case, $cg_{1,1}$ provides $cg_{1,1}$.cop as a shallow compensation process. After compensating $cg_{1,1}$, the contingent procedure for the top-most composite group (i.e., cg_1.top) will be executed. The reader should refer to (Xiao and Urban 2009) for a formal presentation of the recovery semantics.

In the service composition and recovery model, more intelligent and automated compensation and contingency procedures are necessary for dynamic service composition. The goal of the AP concept is to create a more dynamic approach to the combined use of compensation and contingency procedures through the use of checkpointing and rules that can examine the execution state. Also in current DeltaGrid recovery procedures, faults are detected only during run-time. This can be significantly improved by adding pre-conditions and post-conditions, thus we can detect faults in advance and avoid failures by executing other likely successful alternatives. The details of the integration of APs and rules with the composition model are illustrated in next section.

3.2 Extending the Model with Assurance Points and Rules

Our work has extended the model described in the previous section with the concept of assurance points. An AP is a process execution correctness guard. 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 our work, we assume that APs are placed at points in a process where they are only executed once, and not embedded in iterative control structures.

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, representing the current status of the process execution.
- IR_{pre} is an integration rule defining a pre-condition to be checked prior to the execution of an atomic or composite group.
- IR_{post} is an integration rule defining a post-condition to be checked after the execution of an atomic or composite group.
- IR_{cond} is an integration rule defining additional application rules that invoke conditional actions that run in parallel with the main flow of execution.

In the above notation, "*" indicates 0 or more occurrences, while "?" indicates an optional occurrence that can be either zero or one.

IR_{pre}, IR_{post}, and IR_{cond} are expressed as Event-Condition-Action (ECA) rules using the format shown in Figure 2, which is based on previous work with using integration rules to interconnect software components (Urban et al., 2001), (Jin 2004). An IR is triggered by a process reaching 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 therefore 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. In such a case, action 2 is invoked rather than action 1, to invoke a different recovery action.

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 be abandoned, which means that the execution engine compensates 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. After backward recovery to the AP, 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 hierarchical backward recovery process that searches for a possible contingent procedure. 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.

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

Figure 2. Integration Rule Structure

The most basic use of an AP together with integration rules is shown in Figure 3, 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 (AP Data in Figure 3). 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.

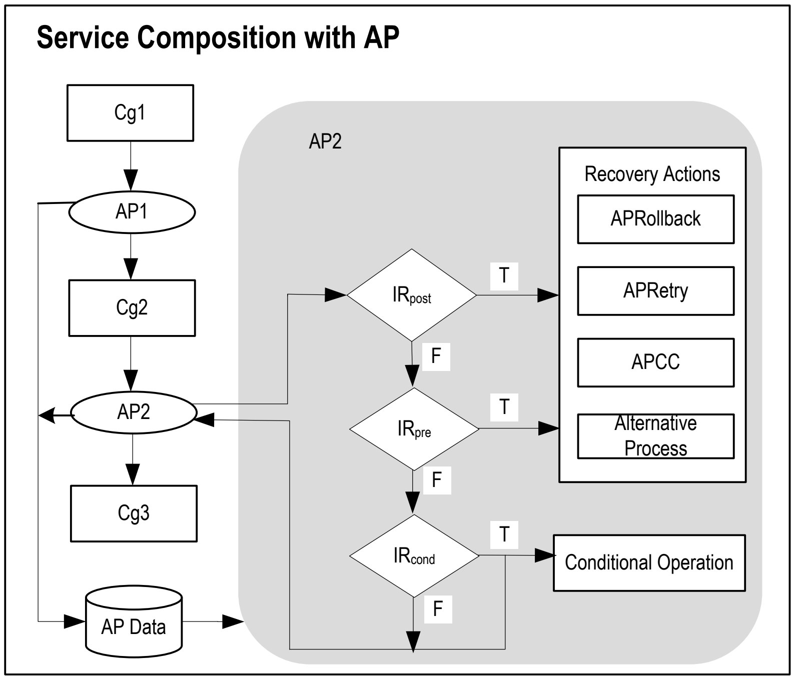

Figure 3. Basic Use of AP and Integration Rules

3.3 Online Shopping Example

This section provides a specific example of assurance points, integration rules, and conditional rules using an online shopping application. A typical online shopping process contains several phases such as selecting goods; paying the bill; shipping goods, and delivering goods. Based on these phases, a process example using APs is presented in Figure 4. All atomic and composite groups are shown in the solid line rectangles, while optional compensations and contingencies are shown in dash line rectangles denoted as cop and top, respectively. APs are shown as ovals between composite and/or atomic groups. To simplify the case for illustration of the concepts, several suppositions are made:

a) If the transaction amount is greater than $1000, the system will automatically send an SMS notice to the customer after the money is charged successfully.

b) The customer pays an extra shipping fee for UPS overnight delivery. If UPS fails to deliver the item overnight, the extra shipping fee will be refunded.

c) The order will be open for cancellation (return) for 30 days after the delivery date. Then the order will be closed.

The available AP identifiers and parameters for the online shopping process are shown in Table 1, which corresponds to Figure 4. Also, Table 1 shows the integration rules and conditional rules associated with the APs in Figure 4. Below, the components of an assurance point are explained using the APs in Figure 4 and the rules in Table 1.

Component 1 (AP Identifiers and Parameters): The AP identifier defines the current execution status of a process instance. Each AP may optionally specify parameters that store critical 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 is orderId (the identifier of the order), which is used in the rules associated with the AP as described in Component 2 below.

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 1, 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.

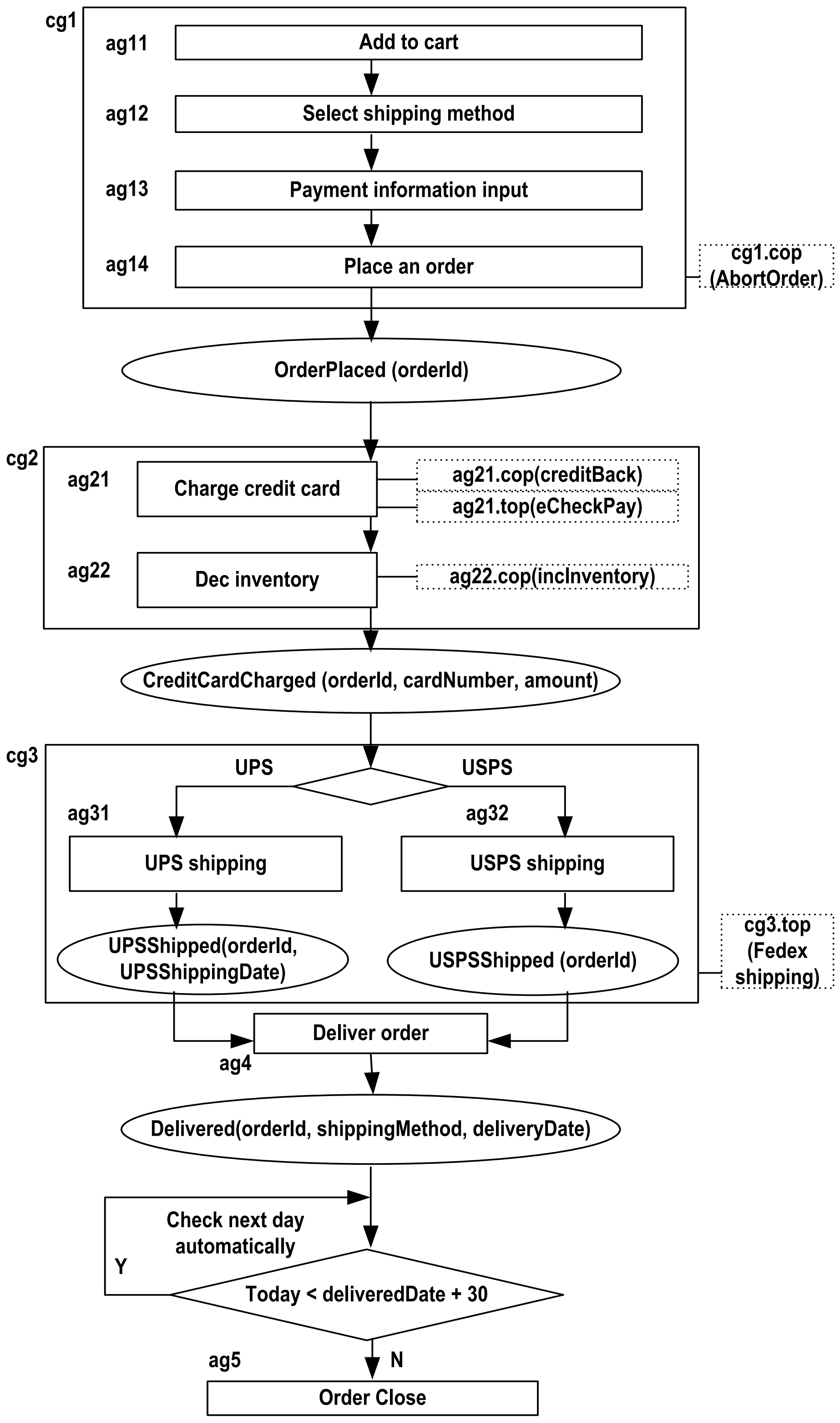

Figure 4. Online Shopping Process with APs

Table 1. AP Structure in the Online Shopping Process

Assurance Point Identifiers and Parameters	Integration Rule	Conditional Rule
OrderPlaced (orderId, itemID, N) orderId is the identifier of the order itemID is the ID number of the goods; N is a number which represents the order quantity.	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)	
CreditCardCharged (orderId, cardNumber, amount) cardNumber is the card to be charged amount is a number which represents the shipping charge.	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 **Notice**::cond event: **CreditCardCharged** (orderId, cardNumber , amount) condition: amount > $1000 action: highExpenseNotice(cardNumbe r)
UPSShipped(orderId, UPSShippingDate) UPSShippingDate is the date on which the UPS gets the item.		
USPSShipped (orderId)		
Delivered(orderId, shippingMethod, deliveryDate) shippingMethod is either UPS or USPS; deliveryDate is the delivery date.		create rule **ShippingRefund**::cond event: **Delivered** (orderId, shippingMethod, deliveryDate) condition: shippingMethod = UPS && deliveryDate != UPSShipped.UPSShippingDate+1 action: refundUPSShippingCharge(orderId)

Component 3 (Conditional Rule): In Table 1, the CreditCardCharged AP has a conditional rule associated with the Delivered AP that sends a text message notification for large charges. After the execution of ag_4, the Delivered AP is reached. Since no pre or post condition is specified, 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, the conditional action refundUPSShippingCharge is invoked to refund the extra fee and the process execution continues. If backward recovery with retry takes place, it is possible that the process will execution 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.

3.4 Summary

As demonstrated through the online shopping process, we have enhanced previous work with a service composition and recovery model with addition of user-defined constraints and different recovery actions to provide more flexible options for recovering a process. In the next chapter, a prototype of the AP model is presented with algorithms that illustrate the semantics of different recovery actions with generic sample scenarios.

CHAPTER IV

A PROTOTYPE OF ASSURANCE POINTS, INTEGRATION RULES, AND RECOVERY ACTIONS

To illustrate the feasibility of the AP model, this research has prototyped an execution environment to demonstrate the extended service composition and recovery model with APs and integration rules. BPEL was not used for the prototype since the broader scope of the research is addressing techniques for decentralized data dependency among distributed Process Execution Agents (PEXAs) (Urban, Ziao, and Le 2009), where PEXAs execute processes, communicate about process failures, identify processes that are dependent on a failed process, and invoke recovery procedures on dependent processes. Existing BPEL engines do not provide the flexibility needed to experiment with this form of decentralized communication among process execution engines. The process specification framework, however, is based on BPEL using the Process Modeling Language (PML) described in (Ma et al., 2005). This work therefore demonstrates the feasibility of extending or modifying BPEL in the future to support assurance points and the recovery capabilities described in this thesis.

The following sections provide details of the implementation of APs and rules. Section 4.1 explains the structure and syntax design of APs and integration rules. Section 4.2 discusses the implementation of the execution engine using XMLBeans. Section 4.3 and Section 4.4 presents algorithms and recovery action semantics with the use of generic examples in Section 4.5. Section 4.6 presents the execution history generation feature. Section 4.7 provides a summary of the prototype.

4.1 Specification of Assurance Points and Integration Rules

This section presents a framework for the specification of assurance points and integration rules. Section 4.1.1 gives an overview of the existing PML. Section 4.1.2 introduces the syntax for the APs and integration rules with supporting examples that were extended to support APs and rules.

4.1.1 PML Overview

The PML described in (Ma et al., 2005) is an XML-based modeling language for defining processes using the basic functionalities from BPEL, such as invoke as well as different forms of control flow specification. An XML format is desirable for the language specification since XML is extensible and platform-independent. The activities supported by the PML are invoke, assign, sequence, flow, switch, and while activities. Support for exception handling and failure recovery was used initially in (Lao 2005) to enhance the PML with the features that support the use of compensation and contingency plans during execution of processes. The enhanced version in (Lao 2005) added elements for the specification of *atomic groups*, *composite groups*, *contingency plan*, and *compensation plans*. The work in (Xiao and Urban 2009) provides state diagrams that define the semantics of the model for the use of compensation and contingency. An initial overview of the semantics of the model was presented in Section 3.1. This section demonstrates how PML has been extended to incorporate the AP concept with integration rules and the recovery actions outlines in Chapter 3.

4.1.2 AP and Integration Rule Syntax

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. We have added the capabilities to define atomic groups and composite groups, with features to express compensation and contingency. Table 2 shows the list of activities supported by our process modeling language to illustrate the AP model. There are two activity categories: atomic activities and complex activities. The atomic activities consist of invoke for service invocation, assign for changing variable values, ag for atomic group, cg for composite group, top for contingency, cop for compensation, and ap for assurance points. The complex activities category includes switch to define alternate control flow. As a simplifying assumption in this initial stage of the research, we have omitted looping and parallel constructs to clearly demonstrate the functionality of APs and recovery actions. Future research directions will address support for these features.

Table 2. Activities

Activity Type	**Activity**	**Description**
Atomic	assign	Changes the value of a property
	invoke	Performs or invokes an operation involving the exchange of input and output messages
	ag	Atomic Group containing a single invoke activity with an optional contingency plan or compensation plan
	cg	Composite Group which containing one or more atomic or composite groups with an optional contingency plan or compensation plan
	top	Contingency plan for executing an alternate action
	cop	Compensation plan for executing a logical rollback
	ap	Assurance points for invoking integration rules and recovery activity
Complex	switch	Executes activities from one of multiple sets, based on a Boolean value

The following notation guidelines will help to read the scripts used in this thesis:

- "?" indicates an optional occurrence that can be either zero or one,
- "*" means 0 or more occurrences, and
- "+" specifies 1 or more occurrences.

An XML Schema is provided as a formal definition of the language features. The XML Schema with support for APs is shown in Appendix I, with the XML schema definition for integration rules in Appendix II.

An AP definition contains an AP name together with zero or more variables as parameters, indicated by apDataIn. The actual variable refers to global process variables defined in the variables section of the process definition. The apDataIn variable should be one of the variables defined in the variable element. The following figure shows the general structure of how an AP and its parameters are defined. The name of the first AP parameter is variable1. For an AP, there can be more than one parameter or no parameters at all.

```
<process>
   ⋮
   <ap name= "APName"> *
       <apDataIn variable="variable1" /> *
   </ap>
   ⋮
<process>
```

Figure 5. AP Specification Syntax

An integration rule definition supports the list of events, conditions, and actions that can be correlated with a named AP in the process specification. Figure 6 shows the general structure of how an integration rule with its events, conditions, and actions are defined. The rules construct can contain zero or more events, where each event is associated with an AP name. The event element can contain zero or one pre-condition rule, zero or one post-condition rule, or zero or more conditional rule definitions. Each of these must have one condition and one or more actions defined. The condition element must have an invoke construct which allows a process to invoke a request-response operation on a Port Type offered by a Web Service. The details of the invoke construct is described in (Ma et al., 2005). The actions elements can have one or more actions, where each action can be APRetry, APRollback, APCC, or the name of a Web Service that can be used to invoke an alternate execution path.

Figure 7 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 are also illustrated (<ap …>). This syntax is used to illustrate the XML grammar of the language structure using the Online Shopping Process as described in Section 3.1.

```
<rules>
  <event ap=”APName”> *
     <pre> ?
        <ecaRule>
           <condition name=“conditionName”>
               <invoke serviceName="ncname" portType="qname”
                    operation="ncname" inputVariable="ncname"?
                    outputVariable="ncname"?>
               </invoke>
           </condition>
           <actions>
               <action name=“actionName”> +
                    <invoke serviceName="ncname" portType="qname”
                       operation="ncname"  inputVariable="ncname"?
                       outputVariable="ncname"?>?
                    </invoke>
               </action>
           </actions>

        </ecaRule>

     </pre>
     .
     .
     .
  </event>
</rules>
```

Figure 6. Rule Specification Syntax

Figure 8 shows the XML rule specification associated with the orderPlacedAP in Figure 7. Each rule indicates the event (i.e., assurance point) that triggers the rule (<event ap = …>), whether the rule is a pre (<pre>) or post (<post>) condition or a conditional (<cond>) rule, as well as the condition (<condition …>) and action (<action …>) of the rule, where rule conditions are implemented in web services. In Figure 7, composite group cg_0 has two APs defined: orderPlacedAP and creditCardChargedAP. When the execution reaches orderPlacedAP during normal execution, it checks the corresponding pre-condition, i.e., QuantityCheck as described in Figure 8. If the condition is not satisfied, the corresponding action is invoked, i.e., backOrderPurchase procedure is called. Otherwise, the AP and process execution continues (as defined in this chapter). The complete process definitions for the online shopping process are presented in Appendix III with the corresponding rule definitions in Appendix IV.

```
<cg name= “cg0”>
   ⋮
   <ap name= “OrderPlacedAP”>
       <apDataIn variable=“orderId” />
   </ap>

   <ag name = “ag02”
      <invoke name=“makePayment” serviceName=“creditCard1”
              portType=“cc:CreditCardPortType” operation=“makePayment”
              inputVariable = “makePaymentInput”
              outputVariable = “makePaymentOutput” />

      <top name=“top02”>
             <invoke name=“makePayment” serviceName=“creditCard2”
              portType=“cc:CreditCardPortType” operation=“makePayment”
              inputVariable = “makePaymentInput”
              outputVariable=“makePaymentOutput” />
      </top>

      <cop name=“cop02”>
             <invoke name=“makeRefund” serviceName=“creditCard1”
              portType=“cc:CreditCardPortType” operation=“ makeRefund”
              inputVariable = “makePaymentInput”
              outputVariable= “makeRefundOutput” />
      </cop>
   </ag>

   <ap name= “creditCardChargedAP”>
       <apDataIn variable=“orderId” />
       <apDataIn variable=“cardNumber” />
       <apDataIn variable=“amount” />
   </ap>

   ⋮
<cg>
```

Figure 7. Process Sample

```
<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 8. Rules Sample

4.2 Process Execution Architecture

The parser in charge of the XML Java binding process has been implemented in the execution engine using XMLBeans. The XML Java binding process fully utilizes the XML Schema definition for unmarshalling and validating XML input documents. XML schema defines the language syntax and is used for document validation. The execution engine uses NetBeans IDE. After parsing a process defined in XML, XMLBeans creates the Java types that represent schema types, which makes it easier to access the instances of the schema through get and set methods. The processor initializes variables and begins executing the activities defined in the input XML script. For each activity defined, a wrapper class has been developed that implements the semantics of the activity. The processor keeps track of nested execution layers for supporting the different recovery options. AP data is also stored in a db40 object-oriented database (db4objects 2006).

Figure 9 shows a revised version of the execution engine originally presented in (Ma et al., 2005). The execution engine consists of three components: the XML parser, the XML processor, and the History Manager.

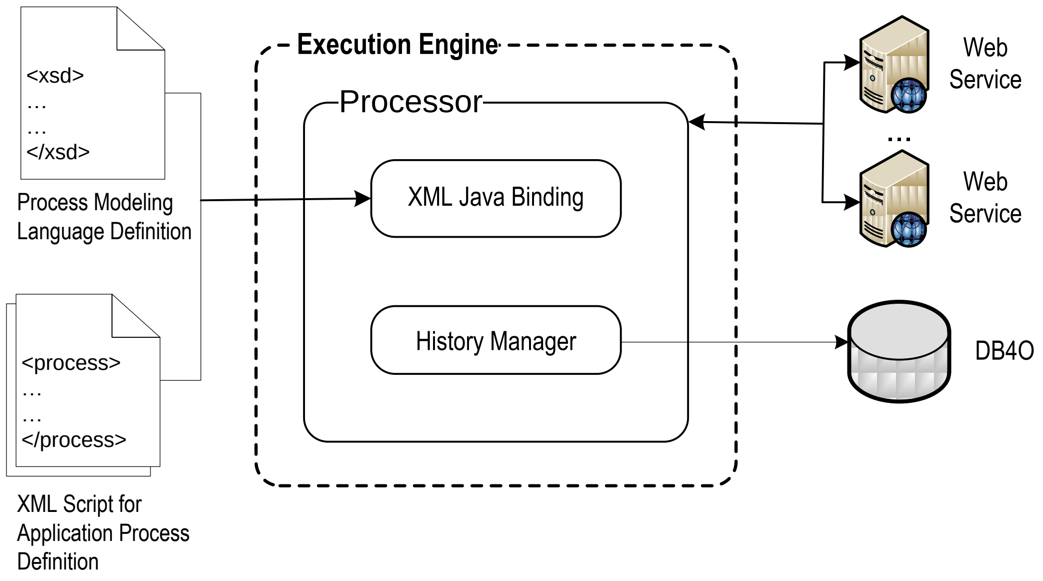

Figure 9. The Execution Engine Architecture

The XML parser converts an XML document to a Java representation. The process is also called *XML Java data binding*, which allows a simple and direct way to use XML in applications. With data binding, an application can largely ignore the actual structure of XML documents and work directly with the data content of each document. In the XML Java binding process, *marshalling* is the process of generating an XML representation for a Java object in memory. *Unmarshalling* is the reverse process, building a Java object (and dependent objects) in memory from an XML representation. XML Java binding can be achieved by code generation, which builds classes that reflect the XML document structure and provides a convenient approach to start working with documents quickly.

After the parsing process, the XML processor initializes process parameters, service provider information, and variable information and starts executing activities. The XML processor is the core component in the execution engine and is also in charge of Web service invocation. The processor utilizes and integrates other components to provide all functionalities supported by the execution engine.

The history of the process execution includes metadata and runtime execution information. Process metadata can be extracted by querying a process specification described in an XML format. At runtime, the execution history, including the information of the process, and Web service invocations are created during the process execution and written to an object oriented database.

The execution engine has been implemented using NetBeans, an open source extensible integrated development environment. The APProject NetBeans project was created which contains all the java binding process files, implementation files, and execution history generation files. The file structure is shown in Table 3.

Table 3. File Structure for the AP Project

File or Folder	**Content**
build.xml	The build file for the xmlbeans project that contains all tasks for the binding process.
/src └─/impl └─/org.ap.pml └─/org.ap.eca └─/org.ap.db4o	Contains Java source file folder: -Folder impl includes the source code for the execution engine, - pml and eca folders contains the Java class files generated by XMLbeans during Java binding process, -db4o contains the database files and other execution history generating implementation Java files.
/file └─/test	Contains input files Test input files, including XML scripts
/lib	The libraries used in this project, including the xmlbeans.jar.
/build	The class file folder containing all Java sources.
/schemas	XML Schema folder containing: pml.xsd, rule.xsd

The XML Java binding process has been implemented in the AP project using XMLBeans (XMLBeans 2005). The XML Java binding process can be done by running the default task defined in the build file. Before running the build task, the XML Schema, named pml.xsd and rules.xsd has to exist in the /schemas directory. A Java representation of the schema (a jar file) is created for use by the execution engine. The actual binding process consists of two steps: 1) the XMLBeans compiler generates a Java representation of the XML Schema. This representation is a set of generic Java classes and interfaces that represent the structure and constraints of the schema, and 2) an actual XML instance document, the XML process script that conforms to the above schema, is bound to the instances of the Java classes and interfaces generated in Step 1. The binding process involves using the XMLBeans API to access the data in the actual XML instance document in an object-oriented manner.

XMLBeans performs the code generation for each of the elements of the XML schema into the source tree of the project using the Ant task named xmlbean defined in the build.xml as shown in Figure 10. The standard build process of the project compiles those sources. In this way, we can make sure that any schema changes are reflected in the generated code on the next compile. More details of the Ant Task with XMLBeans can be found in (Xmlbean Ant Task 2010).

```
<taskdef name="xmlbean"
        classname="org.apache.xmlbeans.impl.tool.XMLBean"
        classpath=" lib/xbean.jar" />
<target name="-pre-compile">
        <antcall target="gen-schema"/>
</target>
<target name="gen-schema">
        <xmlbean srconly="true"
            verbose="true"  srcgendir="src"
            failonerror="true" download="true"
            classgendir="${build.dir}"
            classpath="${classes.path}"
            destfile="APSchema.jar">
            <fileset dir="schema/" includes="**/*.xsd"/>
        </xmlbean>
</target>
```

Figure 10. Configuration of XMLBeans in NetBeans

Figure 11 shows the class diagram of the actual Java interfaces generated for the process XML Schema file. The XMLBeans generates an interface for each type defined in the schema, such as the process type (ProcessType) and the activity type (ActivityType). Each interface contains get and set methods to retrieve and modify attribute information. The get and set methods are shown for the ActivityType interface. All interfaces of the activity (invoke, assign, cop, top, ag, cg, ap) extend a generic interface ActivityType. Similarly, Figure 12 shows the class diagram for the rule XML Schema file. The interface for each type defined in the schema such as the event type (EventType), condition type (ConditionType), and the actions type (ActionsType) are generated by XMLBeans with get and set methods to retrieve and modify attribute information. This generic activity interface abstracts all activities which can be executed by the execution engine to provide an object-oriented design hierarchy.

Figure 13 shows how to bind an incoming XML document instance to the ProcessType interface described above in Step 2. This code creates a method that receives a file representing the XML process instance. The XML document containing the root element and its children is bound to the ProcessDocument interface generated in Step 1 by calling the ProcessDocument.Factory.parse method. The ProcessDocument interface provides a factory class with which to create a new process document instance. The factory class provides various versions of the parse method, each receiving XML source as a different Java type (file, input stream, or URL). Once the ProcessDocument object is created, the process definition is easily obtained by calling the getProcess method on the document object.

The code generated by XMLBeans only contains the static information defined in an XML file. The actual behavior of each activity has been implemented by the activity wrappers in the AP project. The AP project files are integrated with the XMLBeans-generated files as shown in Table 3. There are nine activity wrappers implemented in the execution engine corresponding to the activities listed in Table 2. The high level wrapper class hierarchy is shown in Figure 14.

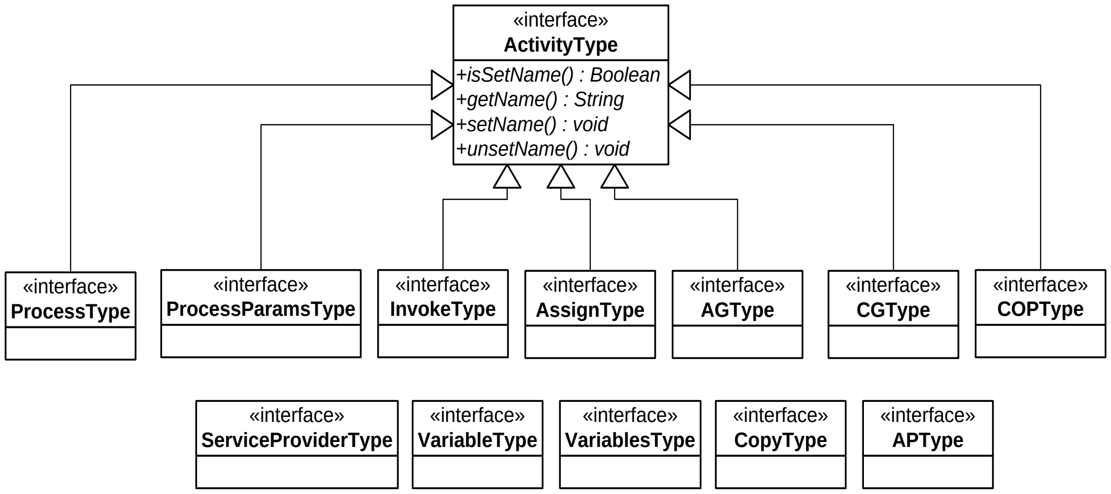

Figure 11. Class diagram of the Process XML Schema

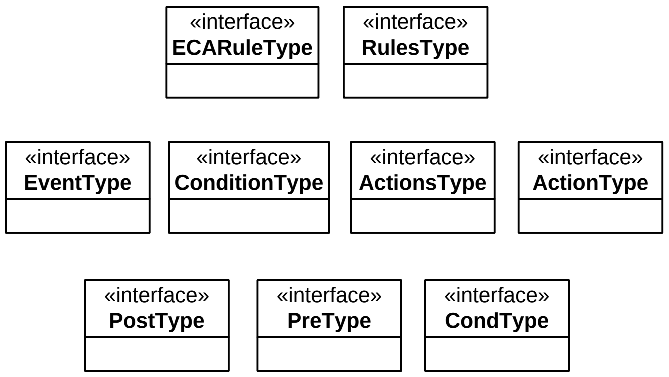

Figure 12. Class diagram of the Rule XML Schema

```
public static org.ap.pml.ProcessType createIpmlProcess(File file) {
        org.ap.pml.ProcessDocument pDoc = null;
        try {
           pDoc = ProcessDocument.Factory.parse(file);
        } catch (XmlException e) {
           e.printStackTrace();
        } catch (IOException e) {
           e.printStackTrace();
        }
        return pDoc.getProcess();
     }
```

Figure 13. Code for binding a Process XML document

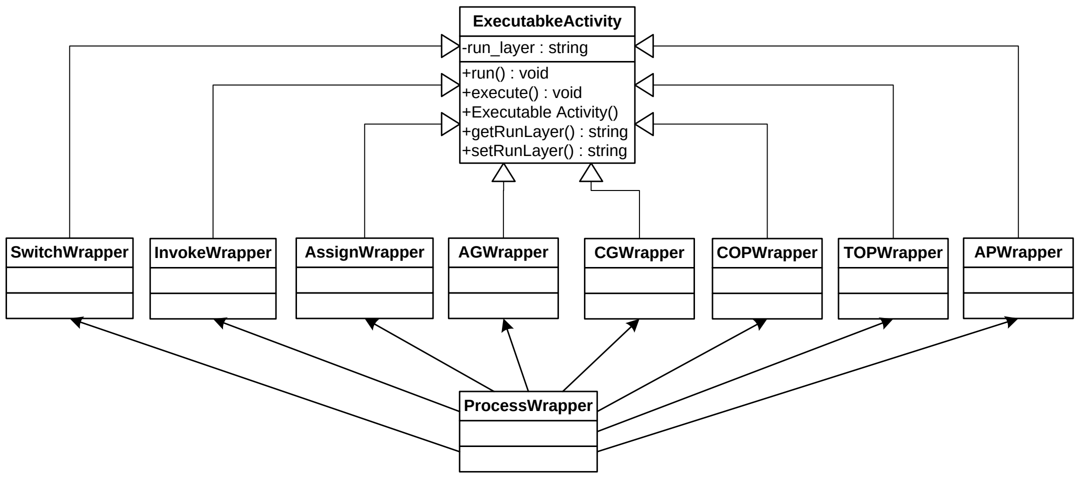

Figure 14. Class diagram of activity wrappers

As shown in Figure 14, an abstract class ExecutableActivity is defined as the super class for the eight activity wrappers. This class provides the basic common functions that are supported by the activity wrappers. Two important methods are defined in the ExecutableActivity class. The first method is called execute. The second method is an abstract method, named run, called by the execute method. The abstract run method is the place where the execution uniqueness of each activity is defined. Each concrete subclass has to implement the run method. In addition, all activities will have private member variables called run_layer. This variable helps to differentiate the layers that each activity is running on so that the semantics of AP recovery actions can be executed successfully. The explanation of the process wrapper and AP wrapper with recovery algorithm details are presented in the following two sub-sections.

4.3 Process Wrapper

The ProcessWrapper is the wrapper class for the top-level process. The class maintains all of the defined variables and service provider information. Variables and service instances are managed in a hash table in the process wrapper. The key to the variable hash table is the variable name and the value is the variable object. The key to the service hash table is the instance name defined in XML instance file.

The process starts by executing a composite group, since the top layer process is itself a composite group with no higher enclosing construct. Therefore, the logical flow of the activities starts with the execution of the CompositeWrapper class, which implements how each activity inside the composite group should be handled. Figures 15(a) and 15(b) show the logic of executeCG. Each composite group can have several other activities and each of these activities can be executed or skipped based on the semantics of the recovery actions mode. There are three process boolean variables, which are used to indicate the invoked recovery actions as defined in Section 3.2: APROLLBACK, APRETRY, and APCC. These variables are saved in the variables hash table with other variables.

When the process begins, the recovery mode variables are set to false as a default value to indicate that no recovery action has been set. However, when an internal error occurs or when certain rule conditions are not satisfied, then any one mode may be turned on at one time by setting the variable to true instead of false. Also, after executing or skipping an activity, the state of the process is checked. If there is an error in any one operation, the process immediately tries to find the contingency for the corresponding atomic group. If the contingency succeeds, the process continues, but if the contingency fails or does not exist, then the process goes into APCC mode. Even if there is no error, the recovery mode may have been set to true due to failure to satisfy a rule condition. In this case, if the mode is APRETRY or APROLLBACK, then either the composite or atomic group is compensated. For the APCC mode, the semantics are more complex. If an activity is a composite group where the APCC mode is set, then the compensation begins to recover inside the scope of the composite group. When the process reaches the outside layer of the composite group after running compensation the process checks if there is an AP immediately preceding the activity in the same scope. If there is, then the process goes back to the previous AP to check the pre-condition before trying to execute the contingency of atomic or contingency group. If there is no AP specified, the process assumes that the pre-condition is satisfied and tries to continue the process by executing the contingent

```
void executeCG(org.ap.pml.CGType cgTypeObj) {
     // For Each Activity in composite group
     for(int i=0; cgTypeObj.getActivityArray().length > i; i++) {
        // if the process is the top most layer and No more Previous Activity Exists during backward recovery
        if ((i<0) && (this.getRunLayer().equalsIgnoreCase("0")))  {
             if ((Boolean) ProcessWrapper.variables.get("APCC")) { //check if the process is in APCC
Mode
                success = findContingency(this.cgTypeObj); //invoke contingency if available
             }
             break; //break for For Loop and end of recovery
        }

        Cases if Activity is
            AG: //AG is of type org.ap.pml.AGType
                If (!APCC && !APRollBack && !APRetry) //None of the Recovery mode is on-Default Mode
                        error = ExecuteActivity (AGWrapper);
                break; //break for Case
            CG: //CG is of type org.ap.pml.CGType
                If (!APCC && !APRollBack && !APRetry) //None of the Recovery mode is on-Default Mode
                        ExecuteActivity (CGWrapper); //CG is of type org.ap.pml.CGType
                If (APCC && (Reached Outside Layer)) { // reaches parent scope during backward
            recovery
                        boolean checkAPPrev = checkAPPrevious(); //Check if Previous Activity is AP
                        if (!checkAPPrev)  { //No AP is found
                            findTOP (CG)  //find contingency assuming precondition is satisfied
                                Succeeds
                                    setAPCC(false); //continue forward execution
                                Fails
                                    //still under APCC mode
                        }
                }
                break; //break for Case
            AP: //AP is of type org.ap.pml.APType
                If (!APCC && !APRetry && !APRollback) //None of the Recovery mode is on -Default Mode
                        ExecuteActivity (APWrapper, getAPRules(APName));
                else If (!APRollBack && (APCC || APRetry)) {
                        If (APRetry && ((APDefined = APName) || (APDefined = null) ))
                                ExecuteActivity (APWrapper, getAPRules(APName));
                        else if (APCC && Reached Outside Layer)
                                ExecuteActivity (APWrapper, getAPRules(APName));
                }
                break; //break for Case
            Assign: //Assign is of type org.ap.pml.AssignType
                If (!APCC && !APRollBack && !APRetry) //None of the Recovery mode is on-Default Mode
                        ExecuteActivity (AssignWrapper);
                break; //break for Case
            Switch: //Switch is of type org.ap.pml.SwitchType
                ExecuteActivity (SwitchWrapper);
                break; //break for Case
        End Case //end of executing activity
```

Figure 15(a). Execution of a Composite Group

```
        Cases after Executing/Skipping Activity
            error:      //Internal Error
                        findTOP (AG)
                              Succeeds
                                   setAPCC(false); //continue forward execution
                              Fails
                                   setAPCC(true);
                                   i = i - 2; //go to previous activity
                        break;
            APCC:
                        If (!CheckAPPrev) {
                             If (Activity = AP) && (Reached Outside Layer while Recovering) {
                                   i = i + 1; //Go to Next Activity, i.e. AG/CG
                                   findTOP (AG/CG)
                                        Succeeds
                                             //Continue to Next Activity
                                        Fails
                                             i = i - 2; //go to previous activity to continue APCC mode
                             }
                             else If (Activity = AG/CG) && (Not Reached Outside Layer while Recovering) {
                                   findCOP (AG/CG); // Deep or Shallow compensation for CG
                                   i = i - 2; //go to previous activity to continue APCC mode
                             }
                             else
                                   i = i - 2; //go to previous activity to continue APCC mode

                        }
                        else {
                             i = i - 2; //go to previous activity to continue APCC mode
                             checkAPPrev =  false; //reset the variable
                        }
                        break;
            APRollBack || APRetry:
                        If (Activity = AG/CG)
                             findCOP (AG/CG); //can be Deep or Shallow for CG
                        i = i - 2; //go to previous activity to continue APRollBack || APRetry mode
                        break;
            Default:
                        //continue to forward execution
                        break; //running on normal mode, i.e. No Recovery mode is on or No error has
                        occurred
        End Case
    }
}
```

Figure 15(b). Execution of a Composite Group (Continued) procedure.

The contingency and compensation procedures follows the same semantics for the atomic and composite groups as described in (Xiao and Urban 2007). In this thesis, we've simplified the compensation and contingency procedure as shown in Figure 16 and Figure 17, respectively. The findCompensation procedure will execute the compensation activity if it is available for a completed atomic or composite group. If the compensation is not available for the atomic group, it is assumed to be non-critical and execution continues with compensation of other activities. But if the compensation is not available for the composite group, then the procedure will look for nested atomic or composite groups and execute the available compensation activity in a recursive way. Sometimes the compensation activity might not be executed successfully for a composite group. In such a case, the recursive procedure will look for nested atomic or composite group and compensates accordingly. The findContingency procedure is straightforward, where the contingency activity is executed if it is available and returns true if successfully completed, otherwise it will return false. Also, in case of unavailability of a contingency procedure, it will return false.

```
void  findCompensation(org.ap.pml.ActivityType activityType) {

        Cases if activityType is
            AG: //AG is of type org.ap.pml.AGType
                If AG has cop
                        Execute AG.cop;
            CG: //CG is of type org.ap.pml.CGType
                If CG has cop
                        Execute CG.cop; //shallow compensation
                                Suceeds
                                        Continue
                                Failure
                                        For Each subGroup //in reverse order
                                                findCompensation(AG/CG)
                else
                        For Each subGroup //in reverse order
                                findCompensation(AG/CG)
}
```

Figure 16. Compensation Implementation

```
boolean findContingency(org.ap.pml.ActivityType activityType) {

        Cases if activityType is
            AG: //AG is of type org.ap.pml.AGType
                If AG has top
                        return Execute AG.top;
                else
                        return False;
            CG: //CG is of type org.ap.pml.CGType
                If CG has top
                        return Execute CG.top;
                else
                        return False;
}
```

Figure 17. Contingency Implementation

4.4 AP Wrapper

The APWrapper is the wrapper class that contains the AP logic. The AP wrapper implements the functionalities of an ap activity with the use of integration rules. Before executing the ap activity, the rules are initialized based on the AP name defined in the rule and XML scripts. Figure 18 shows the code for binding an XML rule file which is similar to the description of Figure 12 for process rule documents.

```
public org.ap.eca.RulesType setRules(File file) {
    org.ap.eca.RulesDocument rulesDoc = null;
    try {
      rulesDoc = org.ap.eca.RulesDocument.Factory.parse(file);
    } catch (XmlException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }
    return rulesDoc.getRules();
  }
```

Figure 18. Code for binding an XML Rule Document

The ap activity can be executed in two ways:

1) *Normal*: when the particular AP is reached for the first time, there has been no error during execution, and none of the recovery actions have been specified.
2) *Revisit*: when the particular AP has been reached again during the recovery process, since an error might have occurred or one of the recovery actions has been specified.

The pseudo code in Figures 19(a) and 19(b) presents the logic of AP execution. During normal AP execution, the post condition is checked first. If the post-condition is not violated then the pre-condition is evaluated. Similarly, if the pre-condition is satisfied, the conditional rules are evaluated. During the course of condition validation, the AP calls the integrationRule function, where the condition is evaluated through the evaluateCondition method. This method invokes the operation by calling a Web service for the corresponding condition as specified in the rule file. If the condition is not violated then the method returns false, else the method looks for the action. For the sake of simplicity, we have assumed that the rule file can specify at most two actions, even though it can be manipulated with more actions as necessary. The global integer variable countHerebefore, keeps track of whether the process has reached a certain AP before. If the process execution reaches an AP for the first time, then the first action is executed. If during RETRY mode the process reaches the same AP again, the second action is executed. Moreover, there might be chances of calling APRETRY again for three or more times. In this case, the default APROLLBACK mode is turned on. The executeAction function is invoked to execute the action to turn on one of the recovery modes. The action can invoke a procedure through a Web service, which is accomplished by invokeAction function.

During the revisit of an AP, the revisitAP procedure is called which has semantics similar to normalAP. The main difference is that it does not have a post-condition check since we are retrying the process from a certain AP point by looking through contingencies. Thus we want to check the pre-condition again. Also, when revisiting the AP, a conditional rule is evaluated only if the action has not been executed the first time through.

```
void execute(org.ap.pml.APType apObjectType, org.ap.eca.RulesType rulesType) {

            If (APRetry || APCC)
                        revisitAP();
            else
                        normalAP();
}

void normalAP() {

            if (PostConditionRule Exists)
                        integrationRule (PostConditionRuleType) ;
            if (PostCondition does not Exists) || (PostCondition is Satisfied)) {
                        if (PreConditionRule Exists)
                                    integrationRule (PreConditionRuleType)
                        if (PreConditionRule does not Exists) || (PreCondition is Satisfied)) {
                                    if (ConditionalRule Exists)
                                        for (int i=0; i< eventTypeObj.getCondArray().length; i++) {
                                           if (condRule(eventTypeObj.getCondArray(i)))
                                               // Conditional Rule Violated
                                        }
                        }
            }
}

void revisitAP() {

            if (PreConditionRule Exists)
                        integrationRule (PreConditionRuleType)

            if (PreConditionRule does not Exists) || (PreCondition is Satisfied)) {

                        if (ConditionalRule Exists) && (Action for Conditional Rule Not Executed Before) {
                            for (int i=0; i< eventTypeObj.getCondArray().length; i++) {
                               if (condRule(eventTypeObj.getCondArray(i)))
                                   // Conditional Rule Violated
                            }
                        }
            }
}
```

Figure 19(a). AP Wrapper Implementation

```
boolean integrationRule(ecaRuleType) {
        boolean executeAction = evaluateCondition(ecaRuleType. getCondition())
        if (!executeAction)
                return false; //condition is not violated
        else {
                if (countHereBefore(APName) = 1) //check if the execution flow has been in this
        AP before {
                        If (isActionTypeNormal(getActionArray(0)))
                                executeAction(getActionArray(0).getName(),
                                                getActionArray(0).getTargetAP());
                        else
                                invokeAction(getActionArray(0));
                }
                else if (countHereBefore(APName) = 2) {
                         if (second Action Exists) {
                                If (isActionTypeNormal(getActionArray(1)))
                                        executeAction(getActionArray(1).getName(),
                                                        getActionArray(1).getTargetAP();
                                else
                                        invokeAction(getActionArray(1));
                         }
                         else
                                executeAction("APRollback", null);
                }
                else
                        executeAction("APRollback", null);

               return true; //condition is violated
        }
}

void executeAction(String action, String targetAP) {
    if (action = APRetry) {
      if ((targetAP = null) || (targetAP.length() <= 0))
        setAPRetry(true);
      else
        setAPRetry(true, targetAP);
    }
    else if (action = APCC)
        setAPCC(true);
    else if (action = APRollback)
      setAPRollback(true);
  }
```

Figure 19(b). AP Wrapper Implementation (Continued)

4.5 Sample Scenarios

This section illustrates the semantics of the APRollback, APRetry, and APCC recovery actions using the generic sample process in Figure 20 as well as the Online Shopping example in Figure 4. In the following, assume that each AP in Figure 20 has an IR_{pre} and an IR_{post} rule. These sample scenarios follow the algorithms presented in the previous section.

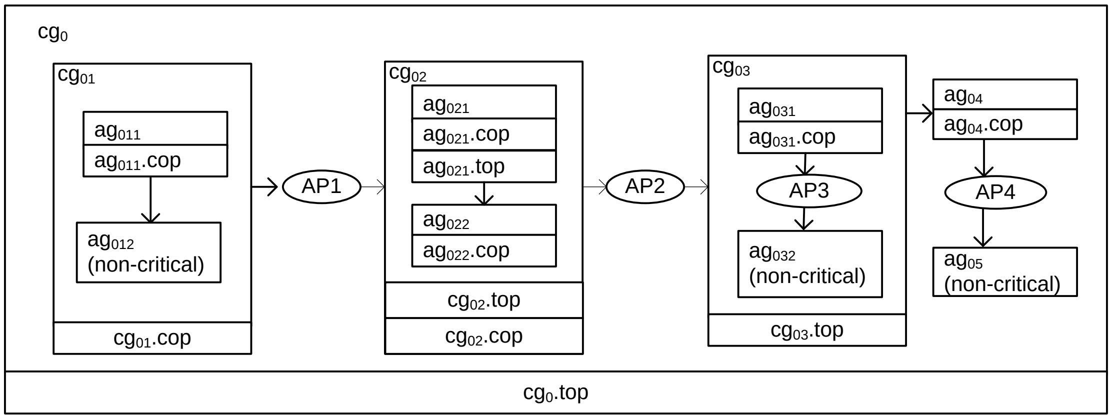

Figure 20. Generic Process for Recovery Actions

4.5.1 Recovery Actions for Pre and Post Conditions

Recall that APRollback is used logically to reverse the current state of the entire process using shallow and deep compensation as described in Section 3.1.

Scenario 1 (APRollback):

Assumption:	IR_{post} condition fails at AP4
Action1 of IR_{post} at AP4:	APRollback
Execution trace:	ag_{04}.cop
	ag_{031}.cop
	cg_{02}.cop
	cg_{01}.cop

Since the post-condition fails at AP4 in Figure 20 and the action of IR_{post} is APRollback, the process compensates all completed atomic and/or composite groups as describe in Section 3.1. Here, the process invokes ag_{04}.cop to compensate ag_{04}. The

APRollback process will then invoke deep compensation ag_{031} by invoking ag_{031}.cop since

1) no shallow compensation for cg_{03} exists and

2) ag_{032} is non-critical and therefore has no compensating procedure

APRollback then invokes shallow compensation cg_{02}.cop, with no specific action to ag_{01} since it is non-critical.

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. Otherwise, the relevant section of code is re-executed. 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.

Scenario 2 (APRetry-default):

Assumption:	IR_{post} condition fails at AP4
Action1 of IR_{post} at AP4:	APRetry
Execution trace:	ag_{04}.cop
	ag_{031}.cop
	IR_{pre} Condition succeeds at AP2
	cg_{03}
	IR_{post} Condition fails at AP4
	ag_{04}.cop
	ag_{031}.cop
	cg_{02}.cop
	cg_{01}.cop

Since the post-condition fails at AP4 in Figure 20 and the action of IR_{post} is APRetry, this action compensates to the most recent AP within the same scope by default. APRetry first invokes ag_{04}.cop to compensate ag_{04}. The process then deep compensates cg_{03} by executing ag_{031}.cop. At this point, AP2 is reached and the pre-condition of IR_{pre} is re-evaluated. 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, 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. Therefore, the execution process will now be the same as in Scenario 1.

Scenario 3 (APRetry-parameterized):

Assumption:	IR_{post} condition fails at AP4
Action1 of IR_{post} at AP4:	APRetry(AP1)
Execution trace:	ag_{04}.cop
	ag_{031}.cop
	cg_{02}.cop
	IR_{pre} condition succeeds at AP1
	cg_{02}
	cg_{03}
	ag_{04}
	ag_{05}

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 then compensate the procedure back to the point of AP1 for the retry process, ignoring all APs in between. Then the execution continues until the end if no error occurs or no condition is violated.

Scenario 4 (APRetry-Default):

Assumption:	IR_{post} condition fails at AP3
Action1 of IR_{post} at AP3:	APRetry
Execution trace:	ag_{031}.cop
	IR_{pre} condition succeeds at AP2
	cg_{03}
	ag_{04}
	ag_{05}

Since the post-condition fails at AP3 in Figure 20 and the action of IR_{post} is APRetry without parameter, this action compensates to the most recent AP within the same scope by default. APRetry first invokes ag_{031}.cop to compensate ag_{031}. Now, the process exits the cg_{03} group to reach AP2 activity in the parent group cg_0 without discovering any previous AP within the same group. Therefore, the process will reach retry from AP2 where the pre-condition of IR_{pre} is re-evaluated. 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 AP3 for a second time, 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.

The APCascadedContingency process, or APCC, 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, the APCC process 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, the APCC process continues by compensating the current composite group back to the next outer layer of the nested structure and repeating the process described above.

Scenario 5 (APCC):

Assumption:	IR_{post} condition fails at AP4
Action1 of IR_{post} at AP4:	APCC
Execution trace:	ag_{04}.cop
	ag_{031}.cop
	cg_{02}.cop
	cg_{01}.cop
	cg_{0}.top

Assume that the post-condition fails at AP4 in Figure 20 and that the IR_{post} action is APCC. As soon as APCC is invoked, 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. Since there is no AP before cg_0, then cg_0.top is invoked.

Scenario 6 (APCC):

Assumption:	IR_{post} condition fails at AP3
Action1 of IR_{post} at AP3:	APCC
Execution trace:	ag_{031}.cop
	IR_{pre} condition succeeds at AP2
	cg_{03}.top

Here the post-condition fails at AP3 in Figure 20 and 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. The APCC process finds AP2 with an IR_{pre} rule for cg_{03}. 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 and the process continues. Otherwise, the recovery action of IR_{pre} for AP2 will be executed. If cg_{03}.top fails then the process will still be under APCC mode, where the process will keep compensating until it reaches the cg_0 layer, where cg_0.top is executed.

4.5.2 Recovery Actions for Execution Errors

When process execution encounters an internal error, the running operation first tries the most immediate contingency, as defined in Section 3.1. If the contingency succeeds, the recovery is complete and the execution continues. If the contingency fails or if there is no immediate contingency, then the execution goes into APCC mode as described in Section 4.3.

Scenario 6 (Failure at ag_{031}):

Assumption:	Internal error at ag_{031}
Execution trace:	IR_{pre} condition succeeds at AP2
	cg_{03}.top

In Figure 20, as soon as an internal error occurs at ag_{031}, the process looks for the contingency for this group. Since there is no contingency specified, the process goes into APCC mode where it backward recovers to AP2. The process will evaluate IR_{pre} before executing the cg_{03}.top (as in Scenario 5).

Scenario 7 (Online Shopping Example - Failure at ChargeCreditCard):

Returning to the Online Shopping Example of Figure 4, 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 8 (Online Shopping Example – Failure at UPShipping):

From Figure 4, 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.

4.6 Execution History Generation

The AP data storage layer contains a process runtime information repository. This section illustrates the storage structure of AP data and parameters along with other process information, and how this information is entered into the data storage. In our implementation, we use the db4o object-oriented database (db4objects 2006) to store the process runtime information as well as AP data. Since process runtime and AP data information are represented by object relationships, it is better to store them in an object-oriented way. Moreover, db4o provides easy and efficient access to store objects. The process execution history consists of the metadata information and the run-time execution information as shown in Figure 21.

The process runtime information repository stores process execution context. Figure 21 presents the metadata that can be retrieved from existing process definitions. Also, the figure shows the runtime process instance information. At runtime, a process

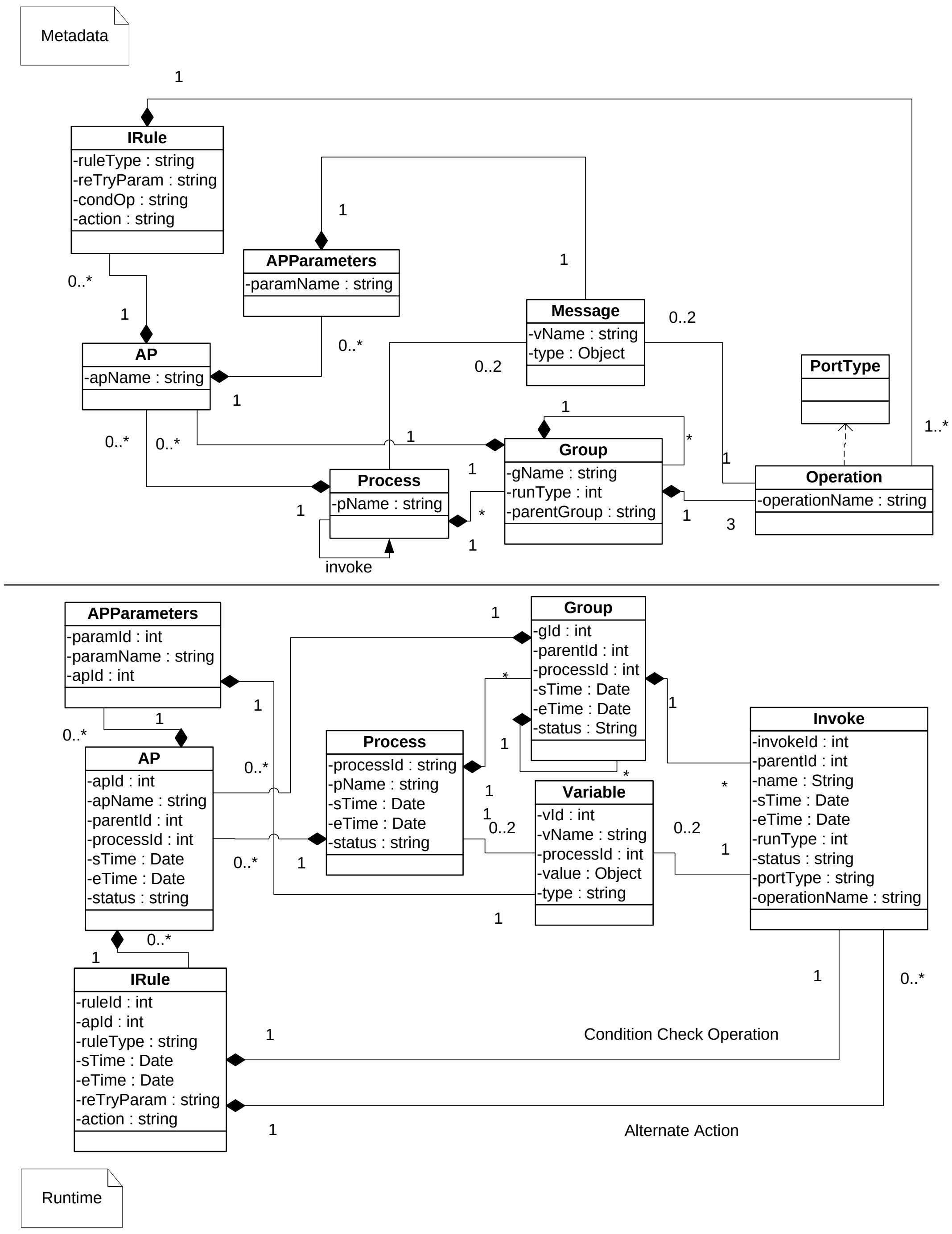

Figure 21: Process Metadata and Runtime Information

is instantiated and its execution information such as instances of groups (atomic or composite), invoke status, variables, APs, and rules are recorded as the runtime information associated with each process instance. The metadata also describes that a Web service can provide multiple operations packaged in different Port Types. A process may invoke multiple operations on different services. A process or an operation can have an input and/or an output parameter defined through a unique message type. The AP also has the capability of invoking an operation to evaluate the rule condition or to invoke the action specified in the rule. The details associated with each message type can be extracted by querying the WSDL file associated with a Web service. In addition to these information stored, the history manager also identifies the process to which an invoke operation or a group belongs and also, the nested group relationship can be established through their corresponding parent identifier.

The following classes are added to define the history of execution in the src/org/ap/db4o folder as shown in Table 3:

1. Process(processed, pName, sTime, eTime, status): defines the process history with a unique indenfier (processId) for each process instance with the name of the process (pName), the process start time (sTime), end time (eTime), and the process execution status (status), which can be success or failure.
2. Group(gId, parented, processId, sTime, eTime, status): defines the execution information for an atomic or a composite group. A group has a unique identifier (gId), and an identifier to its immediate parent group (parentId). The processId identifies which process instance it belongs to. Also, the GroupInfo class has the group start time (sTime), end time (eTime), and its execution status (status).
3. Invoke(invokeId, parented, sTime, eTime, status, operationName, portType, runType): defines the invoke activity history. Each invoke activity has a unique attribute called invokeId with name of invoke (name), start time (sTime), end time (eTime), and execution status. A group identifier (parentId) is used to identify the atomic group that calls the operation. Moreover, the operationName and portType of the

service on which the invocation is called are also stored and these attributes map to an Operation and PortType classes in the metadata. For any operation, the runType attribute can be original for the original primary operation, or compensation for primary operation's compensation plan, or contingency for primary operation's contingency plan, or conditionCheck for condition evaluation of an integration rule, or alternateAction for invoking an alternate action which can be defined in an integration rule

4. Variable(vId, vName, processId, value, type): defines the history of process variables that consists of process parameters, invocation, or AP input/output parameters. A process, an invoke activity, and an AP can contain variables. Thus the variable information is included in the execution history. In the VariableInfo class definition, its instance can have a primary key (vId) and a variable name (vName), its value which is stored as a Java object and the type (input or output) are included. This variable can be associated with an invoke activity as either an input variable or an output variable. Also the variable can be associated with AP parameters. The message type associated with this Variable class instance can be found through the metadata associated with the corresponding APParameters or Invoke instance.

5. AP(apId, apName, parentId, sTime, eTime, status): stores the runtime AP information which is uniquely identified by apId. Also the name of the AP (apName), start time (sTime) and end time (eTime) are stored for each AP. The parent group identifier (parentId) is also stored. The status of AP indicates whether the AP is run as normal or as revisit as described in Section 4.4. Each APInfo instance will have a APParameters instance which stores the variable information used by AP.

6. APParameters(paramId, paramName, apId): stores the AP parameters for each AP. The paramId unique identifies each AP parameter with its parameter name (paramName) and its APs identifier (apId). Each APInfo instance can be associated multiple number of APParameters class since each AP can have several parameters where each parameter are associated with message type.

7. IRule(ruleId, apId, ruleType, sTime, eTime, retryParam, action): stores the integration rules information for each AP if available. Each rule is uniquely identified by ruleId and the ruleType can be one of the integration rule IR_{pre}, IR_{post}, or IR_{cond} which corresponds to AP by apId. Also the rule information contains the start time (sTime) and end time (eTIme), and the action specified by the rule which can be one of the recovery actions: APRollback, APRetry, or APCC. These rules also have capability to invoke a Web service for the condition evaluation as well as to invoke an action other than recovery actions.

In addition to above classes, the following classes were added for the ease of accessibility to db4o database:

- Util: stores the static variables such as db4o file name and location, execution status. Also it has the operations that can get the list of results from the database.
- DB4oAccess: used for setting up database and accessing the database with operations such as openDB(), getDB(), commitDB(), and accessDb4o().

4.7 Summary

This chapter has provided a comprehensive view of a prototype design and architecture of an AP Model with integration rules to extend an existing service composition and recovery model. This extended model has provided a way to specify APs in the workflow and incorporate these APs with integration rules defined in a different XML file. Also, in addition to compensation and contingency plan utilization, we have presented three recovery modes during process recovery: APRollback, APRetry, and APCC. Incorporating these new features, the service composition and recovery model is much more flexible by invoking different recovery actions during situations such as internal errors or during violation of pre, post, or conditional rules. This research has prototyped the initial version of the AP model.

CHAPTER V

EVALUATION OF ASSURANCE POINTS

This section presents an evaluation of the AP concept developed as part of this research. The evaluation compares APs, integration rules, and recovery algorithms with other recovery mechanisms and semantics. Our goal is to demonstrate that recovery approach of the AP model can be used to improve existing workflow languages such as in BPEL with better flexibility and modularity.

This chapter is organized as follows. Section 5.1 gives the comparison analysis with the recovery semantics provided in BPEL. Section 5.2 compares APs with fault-tolerant capabilities in Aspect-Oriented Workflows. Section 5.3 compares the AP logic with workflows that implement checkpointing concepts for recovery. The chapter concludes in Section 5.4 with a summary of the evaluation.

5.1 Comparison to BPEL

WS-BPEL 2.0 is the standard for orchestrating Web service composition. As described in Section 2.1.1, BPEL uses fault, compensation, and termination handlers to guarantee transactional integrity during LRTs. Moreover, Section 2.1.2 highlighted several shortcomings of BPEL with respect to recovery issues. Unlike BPEL, the AP logic allows designers to have a clear notion of how the recovery actions can take place and at the same time provide flexibility with the option of different recovery actions depending upon the status of execution and integration rules. Additional considerations arise during concurrent execution of activities. The flow construct in BPEL allows the process to execute activities in parallel. In BPEL, if one of the branches is faulted, then the compensation policy for concurrent processes forces all branches to compensate at the same time. If a control link is specified, then all available compensation handlers are run for immediately enclosed scopes in the reverse order. Recall that a scope in BPEL is a set of activities that is grouped together, which is comparable to a composite group in the AP model. However, for parallel scopes the compensation takes place in arbitrary order. Moreover, the termination handler is run before the compensation handler to terminate all the activities in the running scope.

Even though the initial AP model implementation does not support concurrent execution within a process, this section outlines AP compatibility with concurrent execution with a comparison to BPEL's recovery semantics. Section 5.1.1 summarizes relevant issues for BPEL's recovery semantics. Section 5.1.2 provides a comparison of our recovery model concept with the recovery semantics used in BPEL. Section 5.1.3 discusses how the AP model can support faults during concurrent execution and also compares our approach with the way BPEL handles such concurrent issues.

5.1.1 Problems in BPEL

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, Roller, and Leymann 2009). Similar to our approach of deep or shallow compensation in service composition and recovery model (see Section 3.1), 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, Roller, and Leymann 2009) highlights the two main problems with the fault and compensation mechanism in the current BPEL standard: 1) compensation order can violate control link dependencies if control links cross the scope boundaries, 2) high complexity of default compensation order due to default handler behavior. Figure 22 shows a modified version of a figure from (Khalaf, Roller, and Leymann 2009) which illustrates the fault and compensation mechanisms of BPEL. This figure is used below to illustrate BPEL anomalies.

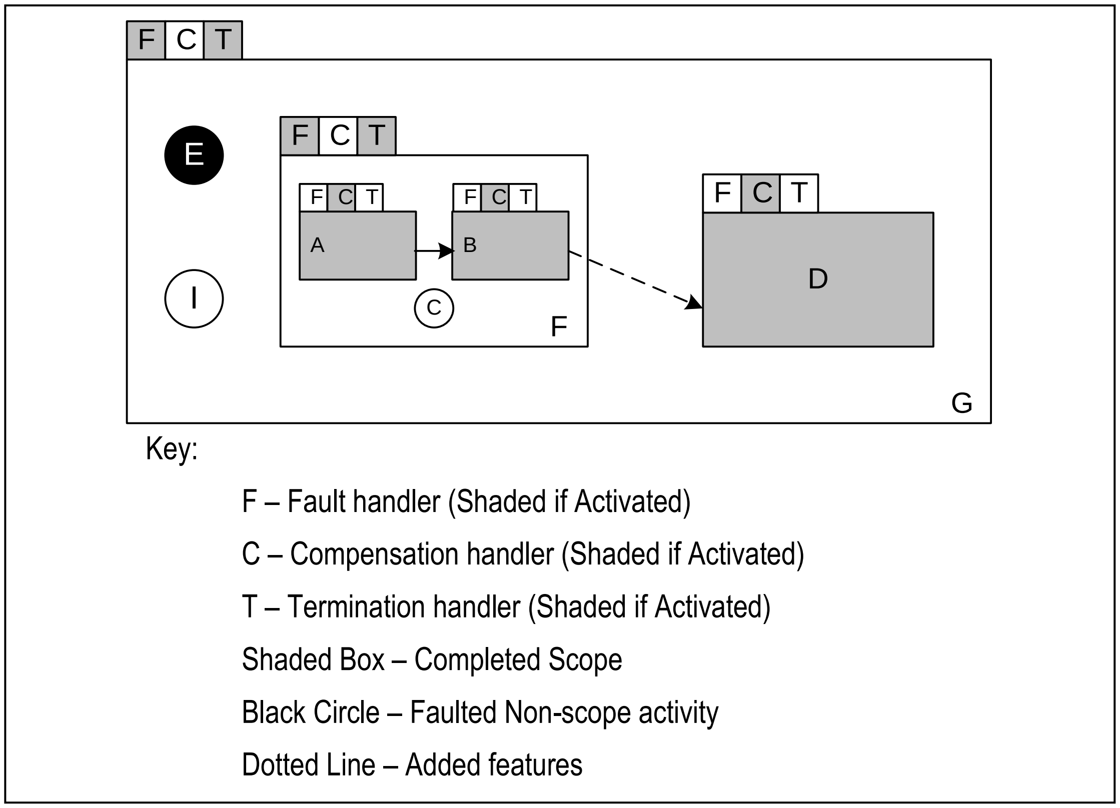
F C T
E
I
F C T
F C T
A
F C T
B
C
F
F C T
D
G
Key:
F – Fault handler (Shaded if Activated)
C – Compensation handler (Shaded if Activated)
T – Termination handler (Shaded if Activated)
Shaded Box – Completed Scope
Black Circle – Faulted Non-scope activity
Dotted Line – Added features

Figure 22: Compensation Sequence Demonstration
(Khalaf, Roller, and Leymann 2009)

The outer scope G in Figure 22 is still running with its fault and termination handlers active. Inside scope G are scopes F and D, where F is still running with fault and termination handlers active. Scope D is already completed and therefore, the compensation handler is active. Scope F has two scopes, A and B, which are both completed with its compensation handlers active. Scope F also has a non-scope activity C which is still running. The arrows (solid and dashed) in Figure 22 show control links between activities. In BPEL, a control link specifies that a target activity must not start until the source activity is completed. In Figure 22, the solid arrow represents control links between peer scopes within the same scope F. The dashed line represents a control link among non-peer scopes.

Assume a fault occurs at activity E. The termination handler of scope G aborts all the running non-scope activities, such as activity I. BPEL will deactivate the fault handler of scope F and activate the termination handler which tries to terminate the running activities inside the scope F. Therefore, non-scope activity C is terminated. Since there is control link between the completed scope A and B, the compensation order honors the control link and B is compensated before A. If scope A and B were still running, the termination handler of either A or B can be invoked first.

Consider the control link that exists between scope B and D (shown by the dashed arrow), where scope D is outside of the parent scope of scope B. In this case, BPEL compensation takes place from scope B to A to D. This order violates the reverse control dependencies where the completed scope D should be compensated before the completed scope B does. Therefore, a wrong compensation sequence is followed. This simple example demonstrates the confusion that can be caused by lack of consistent and clear semantics regarding BPEL fault handling capability. Moreover, the research in (Khalaf, Roller, and Leymann 2009) explains how the default compensation order leads to complication in the compensation process due to what is called the zigzag behavior of compensation. Zigzag behavior is caused by the way scopes get compensated at different nesting levels with no clear direction of compensation order. To support the solution to these problems, (Khalaf, Roller, and Leymann 2009) offers way of calculating the compensation order graph by eliminating the default handlers and still honoring the control links.

Interested readers are advised to go through the (Khalaf, Roller, and Leymann 2009), (Coleman 2005), (Koenig 2006) for details of BPEL's limitations and issues in fault, compensation, and termination handler.

5.1.2 Comparison Criteria

This section provides a comparison between the AP model and BPEL's recovery semantics. Table 4 shows the comparison between BPEL and AP's recovery semantics with respect to the criteria below:

Criteria 1 (Control Link Support): BPEL honors the control link during compensation of peer scopes as shown in Figure 22 with the control link between scope A and scope B. But sometimes the notion of control link reversal during compensation procedure is violated when the control link is present between non-peer scopes (Khalaf, Roller, and Leymann 2009); i.e., when an activity in one flow is dependent on an activity in another flow, such as in Figure 22 (control link between scope B and scope D). Like BPEL, the AP recovery model allows both deep and shallow compensation, where deep compensation is executed by default if shallow compensation is not present (see Section 3.1). The AP model also honors the control links between peer-scopes. Unlike BPEL, 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, without any surprises in the compensation order as in BPEL. In addition, the AP model supports a hierarchical structure during compensation as promoted in (Khalaf, Roller, and Leymann 2009).

Criteria 2 (Compensation for Constraint Violations): 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 a fault handling. Thus, a fault has to occur to invoke a compensation procedure. In the case of 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.

Criteria 3 (Behavior of Recovery Procedures): In BPEL, the designer is responsible for handling complex fault handling logic and, in addition, BPEL's default compensation procedure increases the potential for unexpected errors as described in Criteria 1, causing confusion about the flow of BPEL compensation order. The AP concept provides well-defined recovery actions with hierarchical structure. When an execution error occurs, a process goes into APCC mode as described in Section 4.3. Also when a process reaches an AP, an integration rule can trigger any of the recovery actions where the semantics of the recovery actions are clearly defined and the designer is involved in defining how and when to apply different recovery actions.

Criteria 4 (Contingency Procedure): Currently, BPEL currently does not explicitly support contingency other than through fault, exception, and termination handlers. A fault handler can be used for forward recovery. In contrast, our recovery strategy encourages the use of contingency activities so that forward recovery is possible rather than always rolling back. The contingency procedure increases the possibilities of recovering the process through alternate procedures and thus, saves time and resources that may be wasted through compensation as a rollback procedure.

Criteria 5 (Rules Support): The integration of rules with workflow languages helps to modularize the process execution engine. BPEL does not provide any such rule mechanisms for the specification of constraints, thus every time the business logic changes, the designer has to change the logic of the workflow. In the AP approach, rules that are associated with execution correctness are specified separate from the main logic and integrated into the workflow engine through AP's. Thus, when business logic changes, we can change the rule definition files without significant changes to the main process flow. APs therefore provide a more modular approach to the specification of execution constraints.

Table 4. Comparison of Recovery Semantics (BPEL VS. AP)

No.	Comparison Criteria	BPEL	AP
1	Support for Control Links between Non-Peer Scopes	Yes	No
2	Compensation for Constraint violations	No	Yes
3	Behavior of Recovery procedures	Zigzag	Hierarchical
4	Explicit contingency procedure	No	Yes
5	Rules support	No	Yes

In addition to the comparison in Table 4, the ability to access the process execution state is an important feature required for triggering appropriate recovery procedures. The compensation handler in the older version of BPEL did not have access directly to the current status of the process (Coleman 2005), BPEL 2.0 version does provide such a feature (Khalaf, Roller, and Leymann 2009). The AP model also provides access to the state of the current process. In BPEL, however, compensation is based on the notion of a scope, which allows the scope to compensate as a whole. The compensation notion in the AP model is similar to BPEL during APCC mode, where it rolls back the scope as a whole. But during APRetry mode, the process may reach an AP within same scope during compensation. As a result, compensation in the AP model can be based on AP markers in addition to scope boundaries.

5.1.3 Concurrent Issues for Assurance Points

The AP model can be extended to support parallel execution with a fork construct, such as in the flow activity of BPEL. In this section, we will illustrate different situations for the integration of APs with the parallel activities using the diagram in Figure 23. Extending the AP model with parallel activities is a direction for future work.

Figure 23 shows a process that starts with cg_0 as the top-most process. The process starts with atomic activity ag_{01} and then reaches AP1. After successfully checking the constraints through AP1, the flow activity is reached where cg_{02} and cg_{03} are executed simultaneously. As the execution goes through these parallel activities, there are chances that an error can occur or that integration rule conditions can be violated. In this case, it is desirable to handle the error in the faulted group through forward recovery, while the other parallel groups continue to execute. But there will be situations where the faulted group cannot be recovered and in such a case we might have to rollback through terminating the running parallel activity and compensating the completed activities.

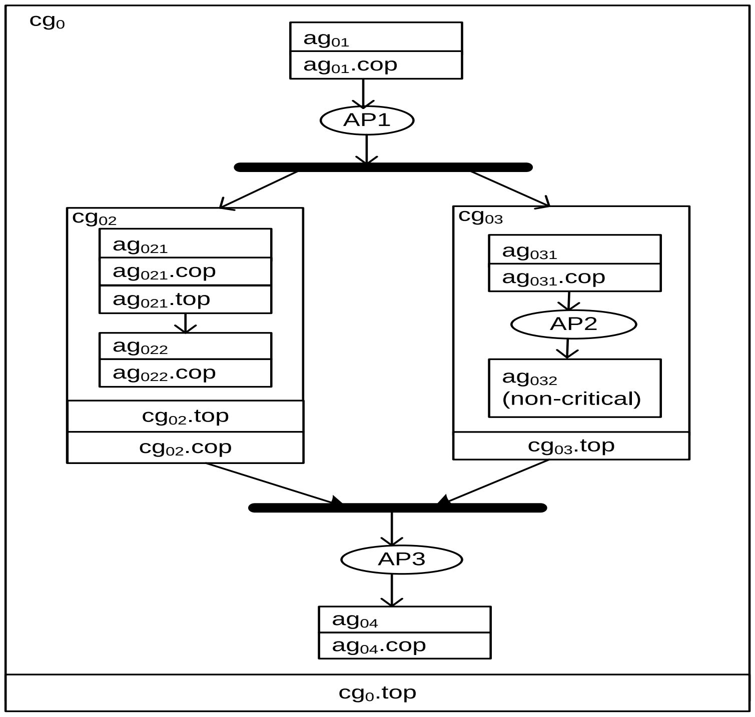

Figure 23. Sample Process

Below are the list of situations described for how AP recovery action can be extended to concurrent execution:

Internal error: As soon as an internal error occurs in one of the parallel activities, the group goes under APCC recovery mode while the other activity might be still running or completed. The faulted group initially tries to recover through the contingency procedures. If the forward recovery is unsuccessful, then the process has to recover back to a previous activity which might be beyond the flow construct. At that time, the other parallel activity is also rolled back to the same previous activity. For example assume that an internal error occurs at ag_{031} in Figure 23. As soon as the error occurs at ag_{031}, the process goes under APCC mode since there is no contingency. The APCC semantics is followed so that the forward recovery is possible for cg_{03} through cg_{03}.top. If cg_{03}.top succeeds, then the process continues as if nothing has happened. Otherwise, the other parallel activity running inside cg_{02} is terminated and cg_{02} rolls back through compensation until it reaches AP1. Now, IR_{pre} condition at AP1 is checked and the process either continues or recovers as defined in the AP.

Violation of Rule Condition: There are chances of violating the rule conditions at an AP during the execution of parallel activities, such as in AP2 of Figure 23. In such a case, the recovery depends upon the recovery actions. If the action defined is:

- APRollback: The process stops executing all parallel activities and each composite group backward recovers all the way up to the top-most group through compensation.
- APRetry: The group cg_{03} looks for a previous or specified AP in the same scope. If one is found, the retry activity proceeds without any affect on the parallel group. Since no other AP defined before AP2 within its scope, the process keeps rolling back to its parent scope until an AP in the parent scope is encountered. The semantics are similar to the case described in Section 4.5. But if during recovery the process reaches an AP outside the flow activity, parallel activities are terminated and compensated back to the same AP. The whole process is now under APRetry mode. One way to make APRetry semantics unambiguous for designer is by providing a list of options to choose APRetry parameter as AP identifier.
- APCC: the semantic is similar as described above for internal error where the faulted group looks for contingency until it reaches the activity before flow construct.

5.2 Comparison to Aspect-Oriented Workflows

Section 2.4 in related work has briefly described the aspect-oriented programming (AOP) concept. The aspect notion can be used by workflows languages in order to modularize the process specification with respect to crosscutting concerns which are the functions that affect the entire workflow and thus should be centralized into one location. The examples of crosscutting concerns are exception management, business rules, logging, authentication, and persistence. Typically, the code for these concerns is scattered all over the system and requires expensive code management when changes are necessary. Therefore AOP provides a more modularized implementation with clear separation of the core concerns and the cross cutting concerns (Domokos and Majzik 2007). In (Charfi and Mezini 2007), the idea of AOP was implemented in BPEL to create an aspect-oriented extension to BPEL (AO4BPEL). A separate XML file defines the aspect activities with pointcut and advice declarations.

Similar to the idea of integrating aspects into BPEL in AO4BPEL, the AP model also has a separate XML rule file for checking constraints. Our approach is more focused on the integration of the checkpointing concept with integration rules to support recovery actions. AOP has not yet focused on recovery issues. Also, even with the advantages of AOP, most programming languages haven’t really evolved in the aspect-oriented direction (Manolescu 2002). Future research should investigate the integration of the AP approach with AOP concepts to support recovery activity.

5.3 Comparison to Workflows using the Checkpointing Concept

Section 2.2 of the related work section discussed the checkpointing concept. Most checkpointing techniques are concepts used for storing states, rolling back to a previous state as in (Luo 2000), and mobilizing orchestrated services for portability in the system as in (Marzouk et al., 2009). The AP model supports the traditional checkpointing concept through storing states of a process and rolling back, but enhances checkpointing capabilities by providing more flexible recovery mechanisms through constraint checking using integration rules and implementing different types of recovery actions. Therefore, our work differs from previous research work done in checkpointing mainly due to the fact that the APs and integration rules defined will also have access to the information collected during processing, thus enabling the system to make more prudent decisions for the next processing action. Moreover, the main goal of this work is to improve fault handling and constraint checking through APs, integration rules, and recovery actions rather than to improve the portability of the system.

CHAPTER VI

SUMMARY AND FUTURE RESEARCH

The research in this thesis has defined the concept of assurance points and illustrated how assurance points can be used together with integration rules and recovery actions to 1) provide a way of expressing user-defined constraints for process execution and 2) provide greater flexibility for use of forward and backward recovery options when constraints are not satisfied or execution fails. This is especially important considering that concurrent processes often execute with relaxed isolation assumptions between the service executions of a process. Assurance points enhance traditional work with checkpointing, providing logical points for backward recovery with semantics that increase the potential for forward recovery by rechecking pre-conditions, retrying services, and looking for contingencies. Planning for failure and recovery should be an important part of every process specification. The assurance point, integration rule, and recovery option functionality demonstrated in this thesis provides a more flexible way to address failure and recovery issues.

There are several directions for future work. One direction involves the integration of the AP concept into a BPEL processor, with performance studies to address the overhead associated with the AP functionality. Another direction involves formalization of the assurance point concept with Petri net and model checking. Methodological issues for the specification of APs, integration rules, and recovery procedures should also be addressed in the context of more extensive application scenarios. In addition, external events that are interruptions from the external application environment can be integrated into the AP model to provide more dynamic event handling capabilities. While the research mainly focused on the AP and integration rule language syntax design and incorporation of recovery actions, there are several other situations that need to be considered to fully integrate the AP model with the DeltaGrid environment, such as concurrent execution within a group and Grid Services support.

Another research direction involves the integration of invariant conditions with the use of assurance points. In the context of the AP model, an invariant is a data condition that must be true from one AP to another when data cannot be locked over across several service invocations. Techniques can be developed to monitor data changes and to inform a process with invariant conditions are violated. The strength of the invariant technique is that it provides a way to monitor data consistency in an environment where the coordinated locking of data items across multiple service executions is not possible, thus providing better support for reliability and maintenance of user-defined correctness conditions among concurrent processes. Future research should also investigate the integration of the assurance point concept with current work on decentralized data dependency analysis (Urban, Ziao, and Le 2009) in Process Execution Agents (PEXAs), where PEXAs communicate about data dependencies so that when one process fails and recovers, other data dependent processes can be notified of potential data inconsistencies. The AP concept can be used to enhance decentralized PEXAs with greater flexibility for process recovery options.

REFERENCES

Bailey, J., F. Bry, M. Eckert, and P. L. Patranjan. (2005). Flavours of XChange, A Rule-based Reactive Language for the (Semantic) Web. *In Proc. Intl. Conf. on Rules and Rule Markup Languages for the Semantic Web,* Springer.

Baresi, L., Ghezzi, C., and Guinea, S. (2004). Towards Self-Healing Service Compositions. *Proceedings of the First Conference on the Principles of Software Engineering*, 11-20.

Baresi, L. and S. Guinea. (2005). Towards Dynamic Monitoring of WS-BPEL processes. *Proceedings of the 3rd International Conference on Service Oriented Computing (ICSOC2005), Springer, Amsterdam.*

Baresi, L., Guinea, S., and Pasquale, L. (2007). Self-Healing BPEL Processes with Dynamo and the JBoss Rule Engine. *International workshop on Engineering of software services for pervasive environments: in conjunction with the 6th ESEC/FSE joint meeting*, 11-20.

Breugel, F. V., and Koshkina, M. (2006). Models and Verication of BPEL. http://www.cse.yorku.ca/~franck/research/drafts/tutorial.pdf (September 2006).

Brambilla, M., Ceri, S., Comai, S., and Tziviskou, C. (2005). Exception handling in Workflow-Driven Web Applications. *Proceedings of the 14th international conference on World Wide Web,* 170-179.

Bry, F., Eckert, M., Patranjan, P., and Romanenko, I. (2006). Realizing Business Processes with ECA Rules: Benefits, Challenges, Limits. *In Proc. Int. Workshop on Principles and Practice of Semantic Web*, Springer.

Chan, K. S. M., Bishop, J., Steyn, J., Baresi, L., and Guinea, S. (2006). A Fault Taxonomy for Web Service Composition. *In Proceedings of The Workshop on Engineering Service-Oriented Applications: Analysis, Design and Composition (WESOA' 2007), Vienna, Austria*.

Charfi, A. and Mezini, M. (2006). Aspect-Oriented Workflow Languages. *In CoopIS 2006 Proceedings.*

Charfi, A. and Mezini, M. (2007). AO4BPEL: An Aspect-Oriented Extension to BPEL. *World Wide Web Journal*, 10, no. 3: 309-344.

Coleman, J. (2005). Examining BPEL's Compensation Construct. *In: Workshop on Rigorous Engineering of Fault-Tolerant Systems, REFT.*

db4objects, Inc. (2006). Db4objects. Website: http://www.db4o.com.

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., and Luck, M. 2002. Transparent Fault Tolerance for Web Services based Architectures. *Proceedings of the 8th International Euro-Par Conference on Parallel Processing (August 27-30, 2002)*,889-898.

Domokos, P. and Majzik, I. 2007. Aspect-Oriented Modelling and Analysis of Information Systems. *Periodica Polytechnica, Electrical Engineering* 51, no. 1-2: 21-31.

Eder, J. and Liebhart, W. 1996. Workflow recovery. *Proceedings of the First IFCIS International Conference on Cooperative Information Systems*. 124-134.

Eisentraut, C. and Spieler, D. 2008. Fault, Compensation and Termination in WS-BPEL 2.0 - A Comparative Analysis. *Web Services and Formal Methods: 5th International Workshop (September 4-5, 2008), Milan, Italy*, 107-126.

Engels, G., Förster, A., Heckel, R., and Thöne, S. 2005. Process Modeling using UML. *Process-aware Information Systems: Bridging People and Software through Process Technology. Hoboken, New Jersey, Wiley*, 85-117.

Ezenwoye, O. and Sadjadi, S. 2006(a). Composing Aggregate Web Services in BPEL. *Proceedings of the 44th annual Southeast regional conference*. 458-463.

Ezenwoye, O. and Sadjadi, S. 2006(b). Enabling robustness in existing bpel processes. *In Proceedings of the 8th International Conference on Enterprise Information Systems*., 95-102.

Fischer, J., Majumdar, R., and Sorrentino, F. 2008. The consistency of web conversations, *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, 415-418.

Friese, T., Muller, J., and Freisleben, B. 2005. Self-Healing Execution of Business Processes based on a Peer-to-Peer Service Architecture. *In Proceedings of the 18th Int. Conf. on Architecture of Computing Systems (ARCS '05)*, 108–123.

Fu, X., Bultan, T., and Su, J. 2004. WSAT: A Tool for Formal Analysis of Web Services. *In Proc. 16th Int. Conf. on Computer Aided Verification*, 3114: 510-514.

Gannod, G. C., Burge, J., and Urban, S. 2007. Issues in the Design of Flexible and Dynamic Service-Oriented Systems, *Proceedings of the International*

Workshop on Systems Development in SOA Environments, Minneapolis, Minnesota.

Greenfield, P., Fekete, A., Jang, J., and Kuo, D. 2003. Compensation is not Enough. In *7th Int. Conf. on Enterprise Distributed Object Computing*, 232.

Holzmann, G. J. 2004. *The Spin Model Checker: Primer and reference manual*: Addison-Wesley Professional.

Jang, J., Fekete, A., and Greenfield, P. 2007. Delivering Promises for Web Services Applications. *IEEE International Conference on Web Services*, 599-606.

Jordan, D., Evdemon, J., Alves, A., Arkin, A., Askary, S., Barreto, C., et al. (2007). Web services Business Process Execution Language version 2.0. *OASIS Standard, 11*.

Jin, Y. 2004. *An architecture and execution environment for component integration rules*: Ph.d Dissertation, Arizona State University, Department of Computer Science and Engineering.

Kamath, M. and Ramamritham, K. 1998. Failure handling and coordinated execution of concurrent workflows, *Proceedings of the Fourteenth International Conference on Data Engineering*, 334-341.

Khalaf, R., Roller, D., and Leymann, F. 2009. Revisiting the behavior of Fault and Compensation handlers in WS-BPEL, *In Proceedings of the Confederated international Conferences, Coopis, Doa, Is, and ODBASE 2009 on on the Move To Meaningful internet Systems: Part I*, 286-303.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. 2001. An Overview of AspectJ. *In Proceedings of the 15th European Conference on Object-Oriented Programming*, 327-353.

Koenig, Dieter. 2006. R26: Default Compensation Order Conflict. http://www.oasis-open.org/committees/download.php/21303/WS_BPEL_review_issues_list.htm l.

Kovács, M., D. Varró, and L. Gönczy. 2007. Formal modeling of BPEL workflows including fault and compensation handling, *In Proceedings of the 2007 Workshop on Engineering Fault Tolerant Systems*, 1.

Kuhne, S., Kern, H., Gruhn, V., and Laue, R. 2008. Business process modelling with continuous validatio, 1st International Workshop on Model-Driven Engineering, 37.

Lao, Ning. (2005). The extended GRIDPML design and implementation, Masters Report, Arizona State University, Department of Computer Science and Engineering.

Lee, P. A., Anderson, T., Laprie, J. C., Avizienis, A., and Kopetz, H. (1990). *Fault tolerance: Principles and practice*: Springer-Verlag New York, Inc. Secaucus, NJ, USA.

Liu, A., Li, Q., Huang, L., and Xiao, M. (2007). A declarative approach to enhancing the reliability of bpel processes, *In Proceedings of International Conference on Web Services*, 272-279.

Luo, Z., Sheth, A., Kochut, K., and Miller, J. (2000). Exception handling in workflow systems. *Applied Intelligence* 13, no. 2: 125-147.

Luo, Z. W. 2000. Checkpointing for workflow recovery, *In Proceedings of the 38th Annual on Southeast Regional Conference*, 79-80.

Ma, H., Urban, S. D., Xiao, Y., and 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*, 433-440.

Manolescu, D. A. 2002. Workflow enactment with continuation and future objects, *In Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, Seattle, Washington*, 40-51.

Martin, D., Paolucci, M., McIlraith, S., Burstein, M., McDermott, D., McGuinness, D., Parsia, B., Payne, T., Sabou, M., and Solanki, M. 2005. Bringing Semantics to Web Services: The OWL-S approach. *In Proceedings of the 1st International Workshop on Semantic Web Services and Web Process Composition (SWSWPC)*, 3387: 26-42.

Marzouk, S., Maalej, A. J., Rodriguez, I. B., and Jmaiel, M. 2009. Periodic checkpointing for strong mobility of orchestrated web services, *In Proceedings of the 2009 Congress on Services - I (July 06 - 10, 2009)*, 203-210.

Michelson, B. M. 2006. Event-driven architecture overview. *Patricia Seybold Group*.

Mikalsen, T., Tai, S., and Rouvellou, I.. 2002. Transactional attitudes: Reliable composition of autonomous web services, *In Proceedings of the Workshop on Dependable Middleware-based Systems.*

Modafferi, S. and Conforti, E. 2006. Methods for enabling recovery actions in ws-bpel*, In Proc. of Int. Conf. on Cooperative Information Systems (CoopIS), Montpellier, France,* 4275: 219.

Modafferi, S., Mussi, E., and Pernici, B. 2006. SH-BPEL: A self-healing plug-in for ws-bpel engine, *In Proceedings of the 1st Workshop on Middleware For Service Oriented Computing (MW4SOC 2006),*48-53.

Pagen, F. G. 1981. *Formal specifications of programming language: A panoramic primer*: Prentice Hall PTR Upper Saddle River, NJ.

Papazoglou, M. P. and Heuvel, W. J. van den. 2007. Service oriented architectures: Approaches, technologies and research issues. *The International Journal on Very Large Data Bases* 16, no. 3: 389-415.

Paton, N. W. and Díaz, O. 1999. Active database systems. *ACM Computing Surveys* 31, no. 1.

Rouached, M., Perrin, O., and Godart, C. 2006. Towards formal verification of web service composition. *Lecture Notes in Computer Science, In Forth International Conference on Business Process Management*, 4102: 257.

Sangiorgi, D. and Walker, D. 2001. The Pi-calculus: A theory of mobile processes. Cambridge University Press.

Scheer, A. W., Thomas, O., and Adam, O. 2005. Process modeling using event-driven process chains. *Process-aware Information Systems: Bridging People and Software through Process Technology. Hoboken, New Jersey: Wiley*: 119-145.

Tan, W., Fong, L., and Bobroff, N. 2007. BPEL4JOB: A fault-handling design for job flow management, *Proceedings of the 5th international conference on Service-Oriented Computing*, 4749: 27-42.

Trainotti, M., Pistore, M., Calabrese, G., Zacco, G., Lucchese, G., Barbon, F., Bertoli, P., and Traverso, P. 2005. Astro: Supporting composition and execution of web services. *Lecture notes in computer science, In Service Oriented Computing -- ICSOC 2005, Third International Conference, Amsterdam, The Netherlands* (December 12-15, 2005), Springer, 3826: 495.

Urban, S. D., Dietrich, S. W., Na, Y., Jin, Y., Sundermier A.,, and Saxena, A. 2001. The irules project: Using active rules for the integration of distributed software components. *Proceedings of the 9th IFIP Working Conference on Database Semantics: Semantic Issues in E-Commerce Systems*:265-286.

Urban, S. D., Liu, Z., Gao, L. (2009). Decentralized data dependency analysis for concurrent process execution. Proceedings of the *IEEE EDOC Workshops: MIddleware for Web Services*, Auckland, New Zealand, 74-83.

Vaculín, R., Wiesner, K., and Sycara, K. (2008). Exception handling and recovery of semantic web services, *In Proceedings of the Fourth international Conference on Networking and Services (March 16 - 21, 2008)*, 217-222.

Wang, M. X., Bandara, K. Y., and Pahl, C. (2009). Constraint integration and violation handling for bpel processes, *In Proceedings of the 2009 Fourth international Conference on internet and Web Applications and Services (May 24-28, 2009)*, 337-342.

White, S. A. (2004). Business process modeling notation (BPMN). *URL:* http://www. *bpmi. org/bpmi-downloads/BPMN-V1. 0. pdf.*

Wiesner, K., Vaculin, R., Kollingbaum, M., and Sycara, K. (2008). Recovery mechanisms for semantic web services, In 8th IFIP WG 6.1 *International Conference, DAIS 2008 Distributed Applications and Interoperable Systems, Oslo, Norway*, 5053: 100-105.

Wächter, H., and Reuter, A. (1992). The contract model. *Database transaction models for advanced applications* 7,. 4: 219-263.

Xiao, Y., and S. D. Urban. (2008). Process dependencies and process interference rules for analyzing the impact of failure in a service composition environment, *Journal of Information Science and Tec* Appendix I2), 21-45.

Xiao, Y., & Urban, S. D. (2009). The DeltaGrid Service Composition and Recovery Model. *International Journal of Web Services Research*, 6(3), 35-66.

Xiao, Y., & Urban, S. D. (2008a). Using Data Dependencies to Support the Recovery of Concurrent Processes in a Service Composition Environment, *Proceedings of the International Conference on Cooperative Informatio Systems, Monterrey, Mexico*, 139-156.

Xmlbean Ant Task. Website: http://xmlbeans.apache.org/docs/2.0.0/guide/antXmlbean.html.

XMLBeans. Apache XMLBeans. 2005. Website: http://xmlbeans.apache.org.

APPENDIX I

XML SCHEMA DEFINITION FOR PML WITH ASSURANCE POINTS

```
<?xml version="1.0" encoding="utf-8" ?>
<!--
  *****************************************************************
  **          AP XML Schema Definition                **
  *****************************************************************
  -->
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:ipml="http://ap.org/pml/" targetNamespace="http://ap.org/pml/" elementFormDefault="qualified">
  <annotation>
    <documentation>This is the XML Schema definition for the AP</documentation>
  </annotation>
  <import namespace="http://schemas.xmlsoap.org/wsdl/" schemaLocation="http://schemas.xmlsoap.org/wsdl/"/>
  <element name="process" type="ipml:ProcessType"/>
  <!--
    ==============================
    =       Process
    ==============================
  -->
  <complexType name="ProcessType">
    <sequence>
      <!--<element name="processParams" type="ipml:processParamsType" minOccurs="0" maxOccurs="1"/>
      <element name="serviceProvider" type="ipml:ServiceProviderType"
        minOccurs="1" maxOccurs="unbounded"/>-->
      <element name="variables" type="ipml:VariablesType" minOccurs="0" maxOccurs="1"/>
      <element name="cg" type="ipml:CGType" minOccurs="0" maxOccurs="1"/>
    </sequence>
    <attribute name="name" type="NCName" use="required"/>
    <attribute name="targetNamespace" type="anyURI" use="required"/>
  </complexType>
  <!--
    ==============================
    =   Process Parameter
    ==============================

  <complexType name="processParamsType">
    <sequence>
      <element name="processParam" type="ipml:processParamType" maxOccurs="unbounded"/>
    </sequence>
  </complexType>
  <complexType name="processParamType">
    <attribute name="name" type="NCName" use="required"/>
    <attribute name="type" type="ipml:ParamIOType" use="required"/>
  </complexType>
 -->

  <!--
    ==============================
    =   Service Provider
    ==============================

  <complexType name="ServiceProviderType">
    <annotation>
      <documentation>Service provider type definition</documentation>
    </annotation>
    <sequence>
      <element name="locator" minOccurs="1" maxOccurs="1">
        <complexType>
          <attribute name="type" type="ipml:ServiceInstanceType" use="required"/>
          <attribute name="handle" type="anyURI" use="required"/>
        </complexType>
      </element>
    </sequence>
    <attribute name="name" type="NCName" use="required"/>
    <attribute name="type" type="NCName" use="required"/>
  </complexType>
  -->

  <!--
    ==============================
    =        Variables
    ==============================
  -->
  <complexType name="VariablesType">
    <sequence>
      <element name="variable" type="ipml:VariableType" minOccurs="0" maxOccurs="unbounded"/>
    </sequence>
  </complexType>

  <!--
    ==============================
    =        Variable
    ==============================
  -->
  <complexType name="VariableType">
    <sequence>
      <element name="variableValue" type="ipml:ValueType" minOccurs="0"/>
    </sequence>
    <attribute name="name" type="NCName" use="required"/>
    <attribute name="messageType" type="QName" use="optional"/>
    <attribute name="type" type="QName" use="optional"/>
  </complexType>

  <!--
    ==============================
    =         Activity
    ==============================
  -->
  <element name="activity" type="ipml:ActivityType"/>
  <complexType name="ActivityType" abstract="true">
    <attribute name="name" type="NCName"/>
  </complexType>

  <!--
    ==============================
    =         Invoke
    ==============================
  -->
  <element name="invoke" type="ipml:InvokeType" substitutionGroup="ipml:activity"/>
  <complexType name="InvokeType">
    <complexContent>
      <extension base="ipml:ActivityType">
        <attribute name="serviceName" type="NCName" use="required"/>
        <!--<attribute name="serviceId" type="int" use="required"/>-->
        <attribute name="portType" type="QName" use="optional"/>
        <attribute name="operation" type="NCName" use="required"/>
        <attribute name="inputVariable" type="NCName" use="optional"/>
        <attribute name="outputVariable" type="NCName" use="optional"/>
        <attribute name="instance" type="NCName" use="optional"/>
      </extension>
    </complexContent>
  </complexType>

  <!--
    ==============================
    =         Assign
    ==============================
  -->
  <element name="assign" type="ipml:AssignType" substitutionGroup="ipml:activity"/>
  <complexType name="AssignType">
    <complexContent>
      <extension base="ipml:ActivityType">
        <sequence>
          <element name="copy" type="ipml:CopyType" minOccurs="1" maxOccurs="unbounded"/>
        </sequence>
      </extension>
    </complexContent>
  </complexType>
  <complexType name="CopyType">
    <sequence>
      <element ref="ipml:from"/>
      <element ref="ipml:to"/>
    </sequence>
  </complexType>
  <element name="from" type="ipml:FromType"/>
  <complexType name="FromType">
    <attribute name="variable" type="NCName"/>
    <attribute name="expression" type="string"/>
    <attribute name="part" type="NCName"/>
  </complexType>
  <element name="to">
    <complexType>
      <complexContent>
        <restriction base="ipml:FromType">
          <attribute name="expression" type="string" use="prohibited"/>
        </restriction>
      </complexContent>
    </complexType>
  </element>

  <!--
    ==============================
    =   Derived Simple Type
    ==============================
  -->
  <simpleType name="ParamIOType">
    <restriction base="token">
      <enumeration value="input"/>
      <enumeration value="output"/>
    </restriction>
  </simpleType>
  <simpleType name="ServiceInstanceType">
    <restriction base="token">
      <enumeration value="factory"/>
      <enumeration value="persistence"/>
    </restriction>
  </simpleType>

   <!--
    ========================
    =         Contingency
    ==============================
  -->
  <element name="top" type="ipml:TopType"/>
    <complexType name="TopType">
        <complexContent>
            <extension base="ipml:ActivityType">
                <sequence>
                  <!--<element ref="ipml:activity" minOccurs="1" maxOccurs="unbounded"/>-->
                  <element name="invoke" type="ipml:InvokeType"/>
                </sequence>
            </extension>
        </complexContent>
    </complexType>

  <!--
    ==============================
    =         Compensation
    ==============================
  -->
  <element name="cop" type="ipml:CopType"/>
    <complexType name="CopType">
        <complexContent>
            <extension base="ipml:ActivityType">
                <sequence>
                  <!--<element ref="ipml:activity" minOccurs="1" maxOccurs="unbounded"/>-->
                  <element name="invoke" type="ipml:InvokeType"/>
                </sequence>
            </extension>
        </complexContent>
    </complexType>

  <!--
    ==============================
    =    Data Input for Ap (Variables as Parameters)
    ==============================
  -->
  <complexType name="apDataInType">
    <attribute name="variable" type="string" use="required"/>
  </complexType>

  <!--
    ==============================
    =         Assurance Points
    ==============================
  -->
  <element name="ap" type="ipml:APType" substitutionGroup="ipml:activity"/>
    <complexType name="APType">
        <complexContent>
            <extension base="ipml:ActivityType">
                <sequence>
                  <!--<element ref="ipml:activity" minOccurs="0" maxOccurs="unbounded"/>-->
                  <!--<element name="invoke" type="ipml:InvokeType"/>-->
                  <element name="apDataIn" type="ipml:apDataInType" minOccurs="0" maxOccurs="unbounded"/>
                </sequence>
            </extension>
        </complexContent>
    </complexType>

  <!--
    ==============================
    =         Atomic Group
    ==============================
  -->
  <element name="ag" type="ipml:AGType" substitutionGroup="ipml:activity"/>
    <complexType name="AGType">
        <complexContent>
            <extension base="ipml:ActivityType">
                <sequence>
                  <element name="invoke" type="ipml:InvokeType"/>
                  <element name="cop" type="ipml:CopType" minOccurs="0" maxOccurs="1"/>
                  <element name="top" type="ipml:TopType" minOccurs="0" maxOccurs="1"/>

                </sequence>

            </extension>
        </complexContent>
    </complexType>
```

```
<!--
  ==============================
  =        Composite Group
  ==============================
-->
<element name="cg" type="ipml:CGType" substitutionGroup="ipml:activity"/>
  <complexType name="CGType">
      <complexContent>
          <extension base="ipml:ActivityType">
              <sequence>
                <element ref="ipml:activity" minOccurs="1" maxOccurs="unbounded"/>
                <element name="cop" type="ipml:CopType" minOccurs="0" maxOccurs="1"/>
                <element name="top" type="ipml:TopType" minOccurs="0" maxOccurs="1"/>
              </sequence>

          </extension>
      </complexContent>
  </complexType>

<!--
  ==============================
  =        Switch
  ==============================
-->
<element name="switch" type="ipml:SwitchType" substitutionGroup="ipml:activity"/>
<complexType name="SwitchType">
  <complexContent>
    <extension base="ipml:ActivityType">
      <sequence>
        <element name="case" maxOccurs="unbounded">
          <complexType>
            <sequence>
              <element name="cg" type="ipml:CGType" minOccurs="0" maxOccurs="1"/>
            </sequence>
            <attribute name="condition" type="string" use="required"/>
          </complexType>
        </element>
        <element name="otherwise" minOccurs="0" maxOccurs="1">
          <complexType>
            <sequence>
              <!--<element ref="ipml:activity" minOccurs="0" maxOccurs="unbounded"/>-->
              <element name="cg" type="ipml:CGType" minOccurs="0" maxOccurs="1"/>
            </sequence>
          </complexType>
        </element>
      </sequence>
    </extension>
  </complexContent>
</complexType>

</schema>
```

APPENDIX II

XML SCHEMA DEFINITIONS FOR RULES

```
<?xml version="1.0" encoding="utf-8" ?>
<!--
  ***************************************************************************
  **AP ECA rules (Integration Rules) Schema Definition **
  ***************************************************************************
  -->
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:eca="http://ap.org/eca/" targetNamespace="http://ap.org/eca/" elementFormDefault="qualified">

  <import namespace="http://schemas.xmlsoap.org/wsdl/" schemaLocation="http://schemas.xmlsoap.org/wsdl/"/>

  <element name="rules" type="eca:rulesType"/>

  <complexType name="rulesType">
    <sequence>
      <element name="event" type="eca:eventType" minOccurs="0" maxOccurs="unbounded"/>
    </sequence>
  </complexType>

  <complexType name="eventType">
    <sequence>
      <element name="pre" type="eca:preType" minOccurs="0" maxOccurs="1"/>
      <element name="post" type="eca:postType" minOccurs="0" maxOccurs="1"/>
      <element name="cond" type="eca:condType" minOccurs="0" maxOccurs="unbounded"/>
    </sequence>
    <attribute name="ap" type="string" use="required"/>
  </complexType>
  <!--
    ==============================
    =         Pre-Condition
    ==============================
  -->
  <complexType name="preType">
    <sequence>
      <element name="ecaRule" type="eca:ecaRuleType" minOccurs="1" maxOccurs="1"/>
    </sequence>
  </complexType>
  <!--
    ==============================
    =         Post-Condition
    ==============================
  -->
  <complexType name="postType">
    <sequence>
      <element name="ecaRule" type="eca:ecaRuleType" minOccurs="1" maxOccurs="1"/>
    </sequence>
  </complexType>
  <!--
    ==============================
    =         Conditional Rule
    ==============================
  -->
  <complexType name="condType">
    <sequence>
      <element name="ecaRule" type="eca:ecaRuleType" minOccurs="1" maxOccurs="1"/>
    </sequence>
  </complexType>

  <complexType name="ecaRuleType">
    <sequence>
```

```
      <element name="condition" type="eca:conditionType" minOccurs="1" maxOccurs="1"/>
      <element name="actions" type="eca:actionsType" minOccurs="0" maxOccurs="1"/>
   </sequence>
 </complexType>

 <complexType name="conditionType">
   <sequence>
      <element name="invoke"  type="eca:invoke" minOccurs="1" maxOccurs="1"/>
   </sequence>
   <attribute name="name" type="string" use="required"/>
 </complexType>

 <complexType name="actionsType">
   <sequence>
     <element name="action" type="eca:actionType" minOccurs="1" maxOccurs="2"/>
   </sequence>
 </complexType>

 <complexType name="actionType">
   <sequence>
      <element name="invoke"  type="eca:invoke" minOccurs="0" maxOccurs="1"/>
   </sequence>
   <attribute name="name" type="string" use="required"/>
   <attribute name="targetAP" type="string" use="optional"/>
 </complexType>
 <!--
   ==============================
   =          Invoke Condition as Service
   ==============================
 -->
    <complexType name = "invoke">
          <attribute name="name" type="NCName" use="required"/>
          <attribute name="serviceName" type="NCName" use="required"/>
          <!--<attribute name="portType" type="QName" use="required"/>-->
          <attribute name="operation" type="NCName" use="required"/>
          <attribute name="inputVariable" type="NCName" use="optional"/>
          <attribute name="outputVariable" type="NCName" use="optional"/>
          <attribute name="instance" type="NCName" use="optional"/>
    </complexType>

</schema>
```

APPENDIX III

PROCESS DEFINITION FOR THE ONLINE SHOPPING EXAMPLE

```
<?xml version="1.0" encoding="utf-8"?>

<!--
	*****************************************************************
	**	Online Shopping Checkout Process Definition	**
	*****************************************************************
-->
<process xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xsi:schemaLocation="http://ap.org/pml/
file:/C:/Documents%20and%20Settings/rshresth/My%20Documents/NetBeansProjects/APProject/src/org/ap/schema/APSchema.
xsd"
  xmlns:tns="urn:checkoutService"
  xmlns:sho="http://www.ap.org/service/Shopping"
  xmlns:cc="http://www.ap.org/service/CreditCard"
  xmlns:echk="http://www.ap.org/service/ECheck"
  xmlns:inv="http://www.ap.org/service/Inventory"
  xmlns:shi="http://www.ap.org/service/Shipping"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns="http://ap.org/pml/" name="checkout" targetNamespace="urn:checkoutService">

  <variables>
    <!--input -->
    <variable name="orderId" type="xsd:string"/>
    <variable name="loginName" type="xsd:string"/>
    <!--shopping variables -->
    <variable name="getCustomerInfoInput" messageType="sho:getCustomerInfo"/>
    <variable name="getCustomerInfoOutput" messageType="sho:getCustomerInfoResponse"/>
    <variable name="getProductListInput" messageType="sho:getProductList"/>
    <variable name="getProductListOutput" messageType="sho:getProductListResponse"/>
  </variables>

  <cg name="cg0">
     <assign name="cg0_assign1">
        <copy>
          <from variable="loginName"/>
          <to variable="getCustomerInfoInput" part="loginName"/>
        </copy>
        <copy>
          <from variable="loginName"/>
          <to variable="getProductListInput" part="loginName"/>
        </copy>
        <copy>
          <from variable="orderId"/>
          <to variable="getProductListInput" part="orderID"/>
        </copy>
     </assign>

     <cg name="cg01">
        <ag name="ag011">
          <invoke name="getCustomerInfo" serviceName="shopping"
              portType="sho:ShoppingPortType" operation="getCustomerInfo"
              inputVariable="getCustomerInfoInput" outputVariable="getCustomerInfoOutput"/>
        </ag>

        <ag name="ag012">
            <invoke name="selectShipper" serviceName="shopping"
              portType="sho:shoppingPortType" operation="selectShipper"
              inputVariable="selectShipperInput" outputVariable="selectShipperOutput"/>
            <top name = "top_ag012">
               <invoke name="selectShipper" serviceName="shopping"
            portType="sho:shoppingPortType" operation="selectShipper"
            inputVariable="selectShipperInput" outputVariable="selectShipperOutput"/>
        </top>
    </ag>

    <cop name = "cop_cg01">
        <invoke name="selectShipper" serviceName="shopping"
          portType="sho:shoppingPortType" operation="selectShipper"
          inputVariable="selectShipperInput" outputVariable="selectShipperOutput"/>
    </cop>
    <top name = "top_cg01">
        <invoke name="selectShipper" serviceName="shopping"
          portType="sho:shoppingPortType" operation="selectShipper"
          inputVariable="selectShipperInput" outputVariable="selectShipperOutput"/>
    </top>
</cg>

<ap name ="orderPlacedAP">
    <apDataIn variable= "orderId"/>
</ap>

<cg name="cg02">
    <ag name="ag021">
          <invoke name="chargeCreditCard" serviceName="creditCard"
              portType="cc:CreditCardPortType" operation="chargeCreditCard"
              inputVariable="chargeCreditCardInput" outputVariable="chargeCreditCardOutput"/>
          <cop name="cop_ag021">
            <invoke name="creditBack" serviceName="CreditCard"
                portType="cc:CreditCardPortType" operation="creditBack"
                inputVariable="creditBackInput" outputVariable="creditBackOutput"/>
          </cop>
          <top name="top_ag021">
            <invoke name="eCheckPay" serviceName="eCheckPay"
                portType="echk:ECheckPortType" operation="chargeECheckPay"
                inputVariable="eCheckPayInput" outputVariable="eCheckPayOutput"/>
          </top>
    </ag>
    <ag name="ag022">
        <invoke name="decInventory" serviceName="inventory"
            portType="inv:InventoryPortType" operation="decInventory"
            inputVariable="decInventoryInput" outputVariable="decInventoryOutput"/>
        <cop name="cop_ag022">
          <invoke name="incInventory" serviceName="inventory"
              portType="inv:InventoryPortType" operation="incInventory"
              inputVariable="incInventoryInput" outputVariable="incInventoryOutput"/>
        </cop>
    </ag>
</cg>

<ap name ="creditCardChargedAP">
      <apDataIn variable= "orderId"/>
      <apDataIn variable= "amount"/>
</ap>

<cg name="cg03">
    <assign name="assigncg03">
          <copy>
            <from variable="orderId"/>
            <to variable="loginName"/>
          </copy>
    </assign>

    <switch name="switch">
        <case condition="${shippingMethod} == USPS">
            <cg name="cg031_if">
                <ag name="ag0311_if">
```

```
                    <invoke name="sendShippingRequest" serviceName="Shipping"
                          portType="shi:ShippingPortType" operation="sendShippingRequest"
                          inputVariable="sendShippingRequestInput" outputVariable="sendShippingRequestInput"/>
                  </ag>
                  <ap name="UPSShippedAP">
                       <apDataIn variable= "orderId"/>
                       <apDataIn variable= "UPSShippingDate"/>
                  </ap>
               </cg>
            </case>
            <otherwise>
               <cg name="cg031_else">
                  <ag name="ag0311_else">
                     <invoke name="sendShippingRequest" serviceName="Shipping"
                           portType="shi:ShippingPortType" operation="sendShippingRequest"
                           inputVariable="sendShippingRequestInput" outputVariable="sendShippingRequestInput"/>
                  </ag>

                  <ap name="USPSShippedAP">
                     <apDataIn variable= "orderId"/>
                     <apDataIn variable= "USPSShippingDate"/>
                  </ap>
               </cg>

            </otherwise>
         </switch>
         <assign name="assignShippingTop">
            <copy>
               <from variable="getCustomerInfoOutput" part="altShippingMethod"/>
               <to variable="sendShippingRequestInput" part="shippingMethod"/>
            </copy>
         </assign>
         <top name="shippingMethodTop">
            <invoke name="sendShippingRequest" serviceName="Shipping"
               portType="shi:ShippingPortType" operation="sendShippingRequest"
               inputVariable="sendShippingRequestInput" outputVariable="sendShippingRequestInput"/>
         </top>
      </cg>
      <ag name="ag04">
         <invoke name="deliverOrder" serviceName="shipping"
            portType="shi:ShippingPortType" operation="deliverOrder"
            inputVariable="sendShippingRequestInput" outputVariable="sendShippingRequestInput"/>
      </ag>
      <ap name ="deliveredAP">
         <apDataIn variable= "orderId"/>
         <apDataIn variable= "shippingMethod"/>
         <apDataIn variable= "deliveryDate"/>
      </ap>
      <ag name="ag05">
        <invoke name="OrderClose" serviceName="shopping"
           portType="sho:ShoppingPortType" operation="orderClose"
           inputVariable="OrderCloseInput" outputVariable="OrderCloseOutput"/>
      </ag>
   </cg>

</process>
```

APPENDIX IV

RULE DEFINITIONS FOR THE ONLINE SHOPPING EXAMPLE

```
<?xml version="1.0" encoding="utf-8"?>

<!--
        *****************************************************************
        **  Integration Rule for Online Shopping **
        *****************************************************************
-->
<rules xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xsi:schemaLocation="http://ap.org/eca/
file:/C:/Documents%20and%20Settings/rshresth/My%20Documents/NetBeansProjects/APProject/src/org/ap/schema/rules.xsd"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns="http://ap.org/eca/">

   <event ap="orderPlacedAP">
      <pre>
         <ecaRule>
            <condition name="quantityCheck">
                <invoke name="checkQuantity" serviceName="ruleConditions"
                  operation="checkQuantity1"
                  inputVariable="quantity"
                  outputVariable="result"/>
            </condition>
            <actions>
               <action name="backOrderPurchase">
                  <invoke name="backOrderPurchase" serviceName="shopping"
                    operation="backOrderPurchase" inputVariable="orderId"
                    outputVariable="result" />
               </action>
            </actions>
         </ecaRule>
      </pre>
   </event>
   <event ap="creditCardChargedAP">
      <post>
         <ecaRule>
            <condition name= "quantityCheck">
                <invoke name="checkInStockQuantity" serviceName="ruleConditions"
                  operation="checkQuantity2"
                  inputVariable="quantity"
                  outputVariable="result"/>
            </condition>
            <actions>
               <action name="APRetry" targetAP= "orderPlacedAP"/>
               <action name="APRollback"/>
            </actions>
         </ecaRule>
      </post>
      <cond>
         <ecaRule>
            <condition name="notification">
                <invoke name="checkAmount" serviceName="ruleConditions"
                  operation=" checkAmount"
                  inputVariable="amount"
                  outputVariable="result"/>
            </condition>
            <actions>
               <action name="highExpenseNotice">
                  <invoke name="highExpenseNotice" serviceName="shopping"
                    operation="highExpenseNotice" inputVariable="cardNumber"
                    outputVariable="result" />
                </action>
            </actions>
        </ecaRule>
    </cond>
</event>

<event ap="deliveredAP">
    <post>
        <ecaRule>
            <condition name= "testSample">
                 <invoke name="checkInStockQuantity" serviceName="ruleConditions"
                    operation="testPostatDelivered"
                    inputVariable="quantity"
                    outputVariable="result"/>
            </condition>
            <actions>
                <action name="APRetry"/>
            </actions>
        </ecaRule>
    </post>
    <cond>
        <ecaRule>
            <condition name="ShippingRefund">
                 <invoke name="ShippingRefund" serviceName="ruleConditions"
                    operation="ShippingRefund"
                    inputVariable="orderId"
                    outputVariable="result"/>
            </condition>
            <actions>
                <action name="refundUPSShippingCharge">
                    <invoke name="refundUPSShippingCharge" serviceName="shipping"
                      operation="refundUPSShippingCharge" inputVariable="orderId"
                      outputVariable="result" />
                </action>
            </actions>
        </ecaRule>
    </cond>
</event>

<event ap="USPSShippedAP">
    <post>
        <ecaRule>
            <condition name= "testSample">
                 <invoke name="checkInStockQuantity" serviceName="ruleConditions"
                    operation="testPostatDelivered"
                    inputVariable="quantity"
                    outputVariable="result"/>
            </condition>
            <actions>
                <action name="APCC"/>
            </actions>
        </ecaRule>
    </post>
</event>

</rules>
```