A Robust Web Service Composition Model with Decentralized Data Dependency Analysis and Rule-based Failure Recovery Capability*

by

Le Gao, Bachelor of Science

A Proposal

In

Computer Science

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

Doctor of Philosophy

July, 2011

*This research is supported by the National Science Foundation under Grant No. CCF-0820152. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

TABLE OF CONTENTS

ABSTRACT

This research defines a hierarchical web service composition and recovery model based on the use of assurance points (APs), integration rules, and application exception rules. Assurance points are correctness guards in a process to check pre and post conditions before and after critical operations and to provide dynamic recovery actions in case of execution errors or constraint violations. The pre and post conditions are represented by integration rules that are structured as event-condition-action. Application exception rules provide a case-based structure of responding to external events that interrupt the execution of a process. The AP service composition and recovery model will be fully developed in the context of programming language control structures, with specific emphasis on flow groups for parallel control activity. The complete execution and recovery semantics will be formally specified and analyzed using the YAWL workflow tool, which is based on Petri Nets. To provide a more robust model, the rule-based failure recovery approach will be integrated with a decentralized data dependency analysis algorithm so that decentralized Process Execution Agents (PEXAs) can communicate about process dependencies associated with partial recovery. A simulation environment will be developed to evaluate the performance and functionality processes that execute within PEXAs using the AP model and data dependency analysis. By integrating rule-based and event-driven techniques into web service composition, process failures and exceptions can more effectively combine backward and forward recovery techniques. Furthermore, past work with transactional workflows is inadequate for service composition since most techniques that support relaxed isolation do not actively address the impact that the failure and recovery process can have on other data dependent processes. The integration of decentralized data dependency analysis with the AP model will not only provide a way to detect data dependency, but also use rule-based techniques to minimize the impact caused by data dependencies between a failed process and other concurrently executing processes.

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1
INTRODUCTION

With the development of the internet, Web Services and service-oriented computing are becoming more widely used for business-to-business integration, providing better support to enterprise business processes and data exchange. These processes are heterogeneous, autonomous, and long-lived. In a traditional, data-oriented, distributed computing environment, a distributed transaction is used to provide certain correctness guarantees about the execution of a transaction over distributed data. In particular, a traditional, distributed transaction provides all-or-nothing behavior by using the two-phase commit protocol to support atomicity, consistency, isolation, and durability (ACID) properties. A process in a service-oriented architecture (SOA), however, is not a traditional ACID transaction due to the loosely-coupled, autonomous, and heterogeneous nature of the execution environment.

1.1 Research Challenge

SOA (Singh & Huhns, 2005) is an architecture in which a number of services communicate in a loosely-coupled manner. A service in SOA is a unit of work executed by a service provider to achieve desired results for a service consumer. In SOA, a service is highly independent of the context and the state of other services. Since a service is autonomous and platform-independent, the commit of a service execution is controlled by the residing service instead of the global process. Therefore, processes composed of web services do not generally execute as transactions that conform to the concept of serializability. In such an environment, concurrently running processes may access or modify the same data through independent services without the isolation property in between service execution. As a result, dirty reads or dirty writes may occur from the global process perspective. If a process fails, data recovery may affect the data consistency in other concurrent processes. Therefore, the recovery of a failed process is not enough to maintain data consistency due to potential dirty reads or writes.

As shown in Figure 1.1, three processes are running concurrently in an SOA. Process$_1$ is initiated by agent$_1$, while process$_2$ and process$_3$ are both controlled by

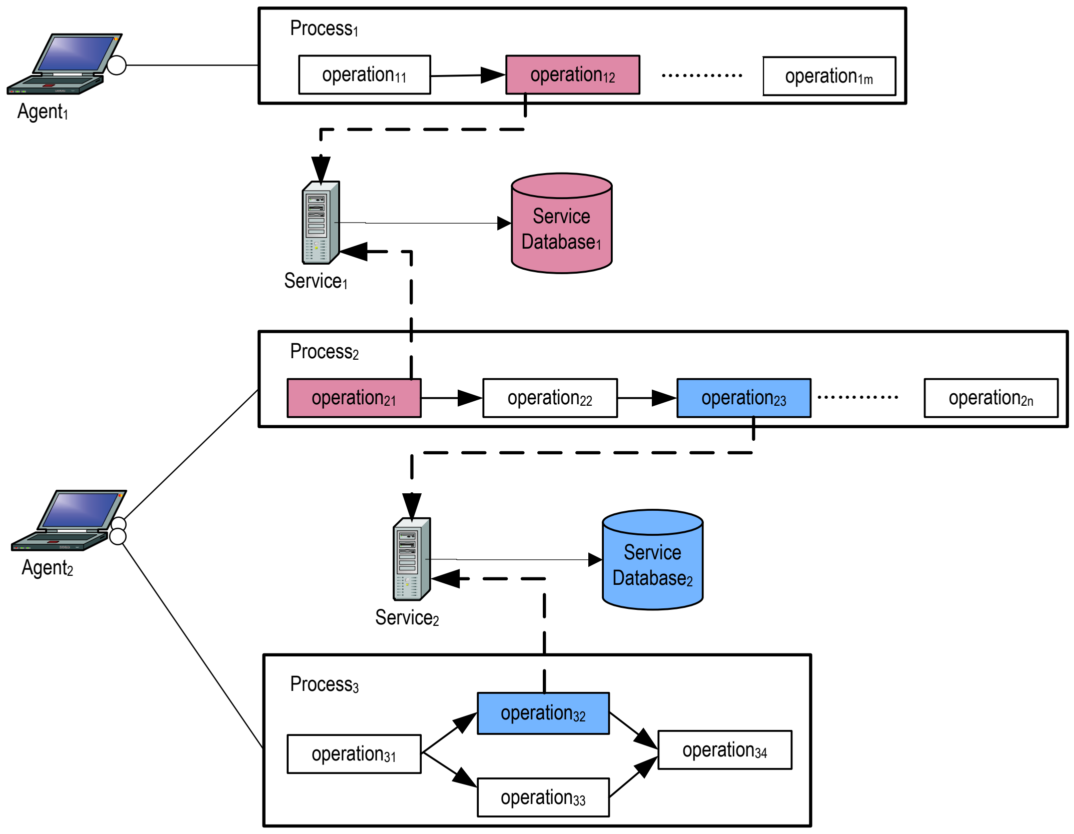


Figure 1.1. Process Execution in an SOA

agent$_2$. In Figure 1.1, operation$_{32}$ and operation$_{23}$ each invoke service$_2$. If process$_3$ fails at operation$_{34}$ and recovers operation$_{32}$, then process$_2$ might be affected due to the potential dirty read/write problem. The dirty read/write issue might also happen between processes that are controlled by different agents. For example, the recovery of process$_2$ controlled by agent$_2$ might affect the correctness of process$_1$ controlled by agent$_1$, since operation$_{21}$ and operation$_{12}$ both execute at service$_1$ and potentially access the same data. Therefore, it is a challenge to analyze data dependencies between concurrent running processes to determine how the data changes caused by the recovery of one process can possibly affect other processes that have either read or written data modified by the services of the failed process.

On the other hand, a web service composition must be flexible enough to respond to individual service errors, exceptions, and interruptions. To respond such events, backward and forward recovery techniques (Worah & Sheth, 1997; Anderson & Lee, 1981) can be adopted. For example, compensation 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. However, adequate combined use of compensation and contingency to keep a process continuously executing as much as possible is still a challenging problem. To achieve this goal, the use of rule-based techniques has also been introduced into web service composition to validate the correctness of execution, especially considering that most processes executing in an SOA do not support traditional transaction processing with guarantees for correctness and consistency of data.

In addtion, from the software engineering point of view, web service composition is also an architecture of software development. So how to verify the web service composition is another research interest. These interesting issues include whether the web service composition can successfully terminate and whether each process can perform correctly. Ideally a composition should be simulated and verified at design time to detect and correct errors before implementation. In the past decade, prevalent techniques such as the Unified Modeling Language (UML) (Booch, Rumbaugh, & Jacobson, 2005), the Business Process Modeling Notation (White et al., 2004), and Event-Driven Process Chains (EPC) (White et al., 2004) have been widely adopted for process modeling, with execution engines based on standards such as the Business Process Execution Language (BPEL) (Andrews et al., 2003) providing a framework for execution of conceptual process designs. Service composition for business integration, however, creates challenges for traditional process modeling techniques.

1.2 Data Dependency Analysis

Xiao (2006) provides a formal definition of a process dependency model, defining read and write dependencies at the operation and process level. Due to the nature of SOA, a failed process may affect the correctness of all other processes that are dependent on the failed process. By using the process dependency model, a set of processes that are dependent on the failed process can be formed. Therefore, necessary recovery actions for the processes in the dependent set must be performed. However, efficient techniques to evaluate the recovery necessity and to dynamically perform the recovery actions is still challenging work. Xiao and Urban (2007) proposed the use of process interference rules (PIRs) to test user-defined conditions that determine if the dependent process should continue running or invoke its own recovery procedures.

The decentralized data dependency analysis project is based on the investigation of Delta-Enabled Grid Services (DEGS) (Blake, 2005; Urban, Xiao, Blake, & Dietrich, 2009). A DEGS is a Grid Service that has been enhanced with an interface that provides access to the incremental data changes, or deltas, that are associated with service execution in the context of globally executing processes. Deltas captured over the source database are stored in a delta repository that is local to the service. Deltas are then generated as a stream of XML data from the delta repository and communicated to the delta event processor of the DeltaGrid environment. A centralized Process History Capture System (PHCS) (Xiao, Urban, & Dietrich, 2006) has been developed to receive deltas from different DEGSs. After receiving deltas, the PHCS then forms a complete execution history for distributed, concurrent processes that are composed of Delta-Enabled Grid Services. The PHCS stores the deltas from distributed DEGSs and the process runtime context. A complete global delta object schedule can be formed according to the timestamps of the deltas. Therefore, the global delta object schedule can be used to form the process dependent set as discussed in the previous paragraph.

Based on the DEGS and the PHCS, Z. Liu (2009) developed an approach that performs decentralized data dependency analysis among concurrently executing processes by introducing Process Execution Agents (PEXAs). PEXAs are responsible for controlling the execution of processes that are composed of web services. PEXAs are associated with specific distributed sites and are also responsible for capturing and exchanging information with other PEXAs about the data changes that occur at those sites in the context of service executions. However, the existing investigation of data dependency analysis has not been fully integrated into a service composition recovery model.

1.3 The Assurance Point Model

A hierarchical service composition and recovery model was initially developed in (Xiao & Urban, 2009). The service composition and recovery model introduced the combined use of compensation and contingency operations to maximize the forward recovery of the process when failure occurs. To enhance the flexibility in process execution, Urban, Gao, Shrestha, and Courter (2010) extended the model by introducing Assurance Points (APs) in the execution of a process, providing integration rules that are capable of checking pre/post conditions. An AP can also be used as a rollback point for backward recovery. Three different forms of backward recovery are described in (Urban, Gao, Shrestha, & Courter, 2010) as actions triggered by integration rules, where rule actions are capable of either full backward recovery or a combination of backward and forward recovery.

Invariant rules have also been defined in (Courter, 2010). Invariant rules provide a stronger way of monitoring constraints and guaranteeing that a condition holds for a specific duration of execution as defined by starting and ending Assurance Points, using the change notification capabilities of Delta-Enabled Grid Services. In (Ramachandran, 2011), application exception rules (AERs) have a case structure, defining recovery actions based on the AP status of a process.

The current AP recovery model, however, has not yet been formally specified and has not fully defined recovery actions for processes that contain parallel as well as iteration and looping control structures.

1.4 Proposed Research

The proposed research represents an integration of the current components of the decentralized data dependency analysis project to create a more robust web service composition model with decentralized data dependency analysis and rule-based failure recovery capability. In the current state of the AP model, only sequential execution has been addressed for recovery actions. In this research, more complex execution control structures, including, if-else, loop, and parallel structures, will be investigated with the use of recovery actions provided by APs. In addition, advanced workflow modeling techniques, such as YAWL (Van Der Aalst & Ter Hofstede, 2005), will be adopted to formalize and verify the semantics of recovery in fully-defined web service composition with APs. Furthermore, a study of decentralized data dependency analysis with the context of rule-based failure recovery feature will be performed based on (Xiao & Urban, 2007; Urban, Xiao, et al., 2009; Urban, Liu, & Gao, 2009).

The existing data dependency analysis method does not provide a dynamic recovery capability based on the execution status of a process. In this research, decentralized data dependency analysis will be integrated into the use of event-driven and rule-based recovery techniques. In sum, the proposed research will develop an intelligent process execution environment supporting use-defined constraint checking, dynamic event-based recovery techniques and decentralized data dependency analysis.

This research has three unique strengths. First, the concept of APs provides better flexibility for process recovery. With the recovery actions provided by APs, a process can be maximally forward recovered even if an execution error or an event interruption occurs. Second, the correctness of the execution semantics in the composition model will be demonstrated and validated by formal modeling techniques. Third, with the help of event-driven and rule-based techniques, a more efficient and effective decentralized data dependency analysis technique can minimize the process interferences caused by a failure in the SOA, and at the same time, help to maintain a higher degree of data consistency among concurrently executing processes.

In the remainder of this proposal, Chapter 2 presents related work. After outlining the motivation and objectives of this research in Chapter 3, the research methodology plan is presented in Chapter 4. Chapter 5 then provides a tentative outline for the dissertation. The proposal concludes with a summary and a discussion of expected contributions in Section 6.

CHAPTER 2
RELATED WORK

This chapter summarizes the related work that provides the foundation and motivation for this research. Section 2.1 presents an overview of advanced transaction models. Section 2.2 discusses the concept of transactional workflow. Recent approaches supporting data consistency in web service composition are presented in Section 2.3. Dynamic modeling techniques and failure recovery strategies for processes composed of web services are discussed in Section 2.4 and 2.5, respectively.

2.1 Advanced Transaction Models

The traditional notion of transactions with ACID properties is too restrictive for the types of complex transactional activities that occur in distributed applications, primarily because locking resources during the entire execution period is not applicable for Long Running Transactions (LRTs) that require relaxed atomicity and isolation (Cichocki, 1998). Advanced transaction models (ATMs), such as Sagas, the Multilevel Transaction Model and the Flexible Transaction Model, have been proposed to better support LRTs in a distributed environment (Rolf, Klas, & Veijalainen, 1997; Elmagarmid, 1992).

The notion of a Saga (Garcia-Molina & Salem, 1987) was proposed in 1987 as a base model for long-running activities. A Saga consists of many ordered smaller tasks that conform to ACID properties and these tasks can execute as interleaved operations. Therefore, the Saga relaxes the requirement of the entire transaction as an atomic action by releasing resources before the transaction completes without sacrificing the consistency of the database. A compensator is created with each task in a Saga. The compensator is an execution that can logically undo the results of the task. When a Saga needs to be aborted, the system aborts the current active task and executes the compensators for each task in reverse order to backward recovery the entire Saga process.

A more relaxed concurrent execution model of independent transactions was introduced in (Weikum, 1991), where a transaction is decomposed into a nested set of sub-transactions at different levels and then each sub-transaction can commit on its own before the whole transaction commits. A sub-transaction can create its sub-transactions at the next level as child sub-transactions. The commit of parent transaction must wait until the child sub-transactions commit. In case of abort at the parent level, the committed subtransactions will run their own compensators to perform "undo"actions.

A flexible transaction model which is suitable for a multidatabase environment was presented in (Elmagarmid, Leu, Litwin, & Rusinkiewicz, 1990). A flexible transaction defines a set of equivalent alternative subtransactions. The flexible transaction model relaxes global atomicity by allowing the transaction designer to define a set of acceptable termination states. Therefore the successful execution of a transaction will be the successful execution of a set of subtransactions or its alternatives. In the flexible transaction model, the acceptable state is also used to decide whether to commit, abort, or compensate a subtransaction.

These advanced transaction models relax the ACID properties of traditional transaction models to better support LRTs and to provide a theoretical basis for further study of complex distributed transaction issues, such as failure atomicity, consistency, and concurrency control. These models have primarily been studied from a research perspective and have not adequately addressed recovery issues for transaction failure dependencies in loosely-coupled distributed applications.

2.2 Transactional Workflow

The term transactional workflow was introduced to recognize the relevance of transactions to workflow activity that does not fully support ACID properties. Transactional workflows contain the coordinated execution of multiple related tasks that support access to heterogeneous, autonomous, and distributed data through the use of selected transactional properties for individual tasks or entire workflows (Worah & Sheth, 1997). Transactional workflows are usually non-atomic and long-lived processes, containing a set of tasks executed at different sites. Transactional workflows require externalizing intermediate results, while at the same time providing concurrency control, consistency guarantees, and a failure recovery mechanism for a multi-user, multi-workflow environment. Concepts such as rollback, compensation, forward recovery, and logging have been used to achieve workflow failure recovery in several projects.

The ConTract Model provides a classic example of work with transactional workflows (Wächter & Reuter, 1991). A ConTract model consists of a set of predefined actions which conform to ACID properties called steps and an explicitly specified execution plan called a script. The ConTract Model provides compensation for backward recovery, and basic constraint checking 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 logically recover completed steps. However, the execution of compensation in the ConTract model is not flexible enough.

Eder and Liebhart (1995) introduced the workflow activity model (WAMO). WAMO supports modeling complex business processes in a simple and reliable way. In WAMO, a complex business process is composed by a set of smaller work units, known as activities. In this model, a workflow consists of three basic units: activity, form and agent. An activity represents the abstract description of a work unit in the business process. A data repository or container to store relevant data is called a form. An agent is a processing entity to perform the execution of activities. An activity may consist of multiple other activities as its steps. Furthermore, activities are reusable by other activities. So new processes are allowed to be composed of existing activities. Five control structures are provided to flexibly compose a workflow: Sequence, Ranked Choice, Free Choice, Parallel and Nesting. In WAMO, the state after each execution is used to control activities of the workflow.

The Correct and Reliable Execution of Workflows (CREW) project (Karnath & Ramamritham, 1998) introduced correctness requirements and other defined constraints into transactional workflows. A workflow in CREW includes multiple steps. The completion of previous steps will trigger the execution of the next steps. The occurrence of specific events can also trigger the execution of specified steps. The rules, events or conditions are used to manage the execution of workflows. Handling of failures to eliminate unnecessary compensations and re-execution of steps are also supported in CREW. If the execution of a step fails, complete compensation and re-execution, or partial compensation and incremental re-execution will be invoked to recover the error. Therefore, CREW provides a more dynamic workflow by the use of rules and the mechanisms for handling failures and exceptions.

The METEOR model (Wodtke, Weißenfels, Weikum, & Dittrich, 1996) combines many features such as two-phase commit (2PC) coordination, error handling, and failure recovery from other transactional workflows models. A METEOR model includes four components: processing entities and their interfaces, tasks, task managers and workflow schedulers. A processing entity is responsible for executing a task. A task is a basic execution agent that performs some operations. The task manager takes control of each task. The workflow scheduler is responsible for coordinating the execution of tasks. METEOR uses a three-layer error model to handle workflow errors. The error model categorizes runtime errors into three classes: task error, task manager error and workflow engine error. These errors can be handled automatically or by human agents by different methods introduced in METEOR.

Workflow management systems have been studied in the context of transactional workflows. Workflow management systems typically provide exception handlers to support backward and forward recovery (Kiepuszewski, Muhlberger, & Orlowska, 1998; Hagen & Alonso, 2002; Chiu, Li, & Karlapalem, 2000). However, they do not fully support constraint checking and do not adopt event-driven techniques for execution interruption.

2.3 Data Consistency Approaches for Web Service Composition

Due to the loosely-coupled, autonomous, and heterogeneous natures of SOA environment, services are independent execution units. Therefore, additional techniques must be used to ensure data consistency. In this section, web service standards are first summarized. Then some recent techniques that ensure the data consistency in web service composition are reviewed.

2.3.1 Web Service Standards

WS-Coordination (F. Cabrera, Copeland, Freund, et al., 2002) describes an extensible framework for providing protocols that coordinate the actions of distributed applications. Such coordination protocols are used to support a number of applications, including those that need to reach consistent agreement on the outcome of distributed activities. WS-Coordination describes a framework for a coordination service (or coordinator) which consists of these component services. The first component is an activation service with an operation that enables an application to create a coordination instance or context. The second component is a registration service with an operation that enables an application to register for coordination protocols. The third component is a coordination type-specific set of coordination protocols. The coordination protocols that can be defined in this framework can accommodate a wide variety of activities, including protocols for simple short lived operations and protocols for complex long-lived business activities.

Atomic Transactions defined in WS-Transaction (F. Cabrera, Copeland, Cox, et al., 2002) build on WS-Coordination, which defines an activation and a registration service. The WS-Transaction has two characteristics. One is the all or nothing property, where the actions taken prior to commit are only tentative. The other is atomic transactions that require a high level of trust between participants and are short in duration. WS-Transaction usually uses Two-Phase commit (2PC) to guarantee the ACID properties. The 2PC protocol coordinates registered participants to reach a commit or abort decision, and ensures that all participants are informed of the final result. It has two types. Volatile 2PC involves participants managing volatile resources such as a cache. Durable 2PC involves participants managing durable resources such as a database. Based on each protocol's registered participants, the coordinator begins with Volatile 2PC, then proceeds through Durable 2PC.

The WS-Business Activity (L. Cabrera et al., 2005) specification defines protocols that enable existing business processes and workflow systems to wrap their proprietary mechanisms and interoperate across trust boundaries and different vendor implementations. Usually WS-Business Activity provides long-running, compensation-based transaction protocols, where a business activity may consume many resources over a long duration. There may be a significant number of atomic transactions involved. Individual tasks within a business activity can be seen prior to the completion of the business activity since their results may have an impact outside of the computer system. Responding to a request may take a very long time. Human approval, assembly, manufacturing, or delivery may have to take place before a response can be sent. In the case where a business exception requires an activity to be logically undone, abort is typically not sufficient. Exception handling mechanisms may require business logic, for example in the form of a compensation task, to reverse the effects of a previously completed task. Participants in a business activity may be in different domains of trust, where all trust relationships are established explicitly.

In contrast to WS-Transaction, the model of WS-Business activity has several distinct differences. The participant list is dynamic and a participant may exit the protocol at any time without waiting for the outcome of the protocol. WS-Business activity allows a participant task within a business activity to specify its outcome directly without waiting for solicitation. It allows participants in a coordinated business activity to perform "tentative" operations as a normal part of the activity. There are two coordination protocols for business activities. One is BusinessAgreementWithParticipantCompletion protocol. A participant registers for this protocol with its coordinator, so that its coordinator can manage it. A participant must know when it has completed all work for a business activity. The other is BusinessAgreementWithCoordinatorCompletion protocol, which means a participant registers for this protocol with its coordinator, so that its coordinator can manage it. A participant relies on its coordinator to tell it when it has received all requests to perform work within the business activity. The main difference between the two protocols is that one executes by itself and the other one executes by a coordinator.

Web Services Business Process Execution Language (WS-BPEL 2.0) (Jordan et al., 2007) provides the language to specify business processes that are composed of Web services as well as exposed as Web services. The main concepts in the BPEL 2.0 standard include process, partner links, properties and correlation, as well as basic and structured activities scopes. BPEL processes are exposed as WSDL services. Message exchanges map to WSDL operations and WSDL can be derived from partner definitions. A partner link is an instance of a typed connector. A partner link type specifies required and/or provided portTypes. Properties and Correlations mean messages in long-running conversations are correlated to the correct process instance. Typed properties defined in WSDL are named and mapped to parts of several WSDL messages used by the process. Activities are modeled as composite patterns, which means that the top level activity most likely is a structured activity. Structured activities contain other activities. Basic activities are just operations. The activities of BPEL are similar to control structures in traditional programming languages. A scope is a set of (basic or structured) activities. Variables in a scope are visible only within the scope. Local correlation sets, compensation handlers, fault handlers, termination handlers and event handlers can also be defined in a scope. Event handlers are message events or timer events (deadline or duration). Fault handlers deal with different exceptional situations (internal faults). Compensation handlers undo persisted effects of already completed activities. Termination handlers support forced scope termination (external faults).

2.3.2 Recent Data Consistency Approaches

Mikalsen, Tai, and Rouvellou (2002) introduced a new Web Service Transaction (WSTx) framework, called transactional attitudes, to support the issue of transactional reliability in web service composition. In the WSTx framework, transactional attitudes are used to allow web service providers to declare their individual transactional capabilities and semantics and to allow web service clients to declare their transactional requirements. There are two types of attitudes defined in WSTx framework: Provider Transactional Attitudes (PTAs) and Client Transactional Attitudes (CTAs). PTAs are used for web service providers to explicitly describe their specific transactional behavior, while CTAs allow the clients to describe their expectations and outcome acceptance criteria explicitly. Each client executes one or more actions within the scope of a web transaction, where each action represents a provider transaction that executes within the context of the larger web transaction. The WSTx framework provides reliability during execution by using both PTAs and CTAs to define attitudes for web services transaction compositions. Also, middleware which acts as an intermediary between a client and multiple web service providers has been developed.

Another similar method to temporarily lock data in a concurrent environment, is the reservation-based approach (Zhao, Moser, & Melliar-Smith, 2005). This approach reserves resources that meet the criteria of what the web service has requested. In this protocol, each task within a business activity is divided into two steps. The first step is to reserve resources based on business logic. Basically, the reservation is a contract between the client and the resource provider. To maximize the execution concurrency in the system, a ‘fee’ is associated with each reservation proportional to the duration of the reservation, which discourages the application to reserve the same resources for an extended period of time. In the second step, the reservation is either confirmed or cancelled according to the business rules. Because the resource that the application requests is reserved in the first step, the application has the choice and freedom to decide about either continued execution or backtracking. A two-phase protocol is used to coordinate the different tasks within a business activity. In the first phase, the client coordinator sends reservation requests to all the participants. The confirmation or cancellation of the reservations is decided by the coordinator at the end of the first phase. In the second phase, the confirmation or cancellation requests are sent to the corresponding participants. If a participant has accepted a reservation, it must be committed to the reserved resource unless the coordinator cancels the reservation. In traditional transactions, any of the participants have the right to rollback or abort the entire transaction. In the reservation-based coordination protocol, however, only the coordinator can determine this.

A Promise approach was proposed in (Jang, Fekete, & Greenfield, 2007) to support the isolation property in web service composition. The goal of the Promises approach is to ensure that certain values are not overwritten or changed by concurrently executing web services. A promise is an agreement between a client application and a service or promise maker. A promise assures the client that some set of conditions (predicates) will be maintained over a set of resources for a specific duration of time, as requested by the client. Instead of locking data, the approach defines the promise maker to be a promise manager that records promises. The main functionality of the promise manager is to address promise making, check on resource availability, and also ensure that promises are not violated during the specific time period. Client applications send the promise manager information in the form of predicates about the resources they want in order to complete successfully. These predicates are Boolean expressions over the resources. The request for a promise will be examined by the promise manager, which will either grant or reject the request. Once a promise request is granted, the client application is isolated from the effects of concurrent execution and can complete successfully. One method that has been used to implement promises is the concept of soft locks. This method uses a field in the database record to indicate whether an item has been allocated already for a client or not. The record is not a real lock. When an application requests the same recourse, this field is read to determine availability of the resources. Promises are a weaker form of locking, but do allow other web services to access the data so that any wait is avoided.

2.4 Dynamic Modeling of Business Processes

As described by Lu and Sadiq (Lu & Sadiq, 2007), most modeling techniques can be categorized as either graph-based techniques or rule-based techniques. The following two subsections summarize graph and rule-based techniques, with a focus on support for dynamic capabilities.

2.4.1 Graph-Based Modeling Techniques

In graph-based modeling techniques such as BPMN (White et al., 2004), UML (Engels, Förster, Heckel, & Thöne, 2005), and EPC (Scheer, Thomas, & Adam, 2005), a business process is described by a graph notation in which activities are represented as nodes, and control flow and data dependencies between activities as arcs or arrows

BPMN. The Business Process Modeling Notation (BPMN V1.0) was introduced by the Business Process Management Initiative in 2004 (White et al., 2004). The objective of BPMN is to provide a graphical model that can depict business processes and can be understood by both users and developers. Flow objects include symbols to represent events, activities, and gateways (i.e., decision points). Flow objects are connected to each other via connecting objects that represent sequence flow, message flow, and association. A process always starts from an event and ends in an event. All other events inside the process are called intermediate events and can be part of the normal flow or attached to the boundary of an activity. An attached event indicates that the activity to which the event is attached should be interrupted when the event is triggered. The attached event can trigger either another activity or sub-process. Typically, error handling, exception handling, and compensation are triggered by the attached event.

To detail a business process, swim lanes and artifacts can be used. Swim lanes are used to either horizontally or vertically group a process into subgroups by rules, such as grouping processes by departments in a company business process. Artifacts provide additional information in a business process to make a model more readable, such as text descriptions attached to an activity.

BPMN (V2.0 beta 1) (Bpmn, 2009) was released in 2009. In BPMN 2.0, the most important update is standardized execution semantics which provide execution semantics for all BPMN elements based on token flows. A choreography model is also supported in BPMN 2.0. Other significant changes include 1) a data object supporting assignments for activity; 2) updated gateways supporting exclusive/parallel event-based flow; 3) event-subprocesses used to handle event ocurrences in the bounding subprocess; 4) a call activity type that can call another process or a global task; and 5) escalation events for transferring control to the next higher level of responsibility.

Mapping tools that can convert BPMN to executable languages, where the translation is enhanced with the execution semantics of BPMN 2.0. For example, the Business Process Execution Language (BPEL) (Alves et al., 2007) is used for executing business processes that are composed of Web Services. A well-known, open-source mapping tool is BPMN2BPEL (Ouyang, Dumas, Aalst, Hofstede, & Mendling, 2009).

UML. The Unified Modeling Language (UML) is a general-purpose modeling language with widespread use in software engineering. UML provides a set of graphical modeling notations to model a system. An activity diagram describes a business process in terms of control flow. A state diagram represents a business process using a finite number of states. UML also provides sequence diagrams that emphasize interactions between objects.

In UML 2.0, new notations have been added to activity diagrams to provide support for the specification of pre and post conditions, events and actions, time triggers, time events, and exceptions. These notations provide more dynamic support to process modeling in UML. Researchers have also proposed process modeling enhancements to UML. For example, the work in (Pintér & Majzik, 2005) proposes a framework that supports exception handling using UML state charts. In (Halvorsen & Haugen, 2006), the authors present a method that can handle exceptions in sequence diagrams.

EPC. The Event-driven Process Chain (EPC) method was developed within the framework of the Architecture of Integrated Information Systems (ARIS) in the early 1990s. The merit of EPC is that it provides an easy-to-understand notation. OR, AND, and XOR nodes are used to depict logical operations in the process flow. The main elements of a process description include events, functions, organization, and material (or resource) objects. The EPC does not have specific notation support for exception processing. Instead, EPC uses the logical operations to specify the handling of events and exceptional conditions. Recent work has modified the EPC notation to provide better support for process modeling. For example, in (Mendling, Neumann, & Nüttgens, 2005), yEPC provides a cancellation notation to model either an activity or a scope cancellation process.

Petri Nets. Petri Nets were originally invented by Carl Adam Petri (Petri, 1966) for the purpose of describing chemical processes. In 1970's, Petri Nets were soon recognized as one of the most adequate and sound methods to model a system of synchronization, communication, and resource sharing between concurrent processes. A Petri net is a directed, connected, and bipartite graph in which nodes represent places and transitions, and tokens occupy places. Van Der Aalst (1998) discussed the use of Petri nets in the context of workflow management. Petri nets can be used not only as a design language for the specification of complex workflows, but also provides powerful analysis techniques to verify the correctness of workflow procedures. However, using plain Petri Nets to model workflow has some drawbacks. First, plain Petri Nets lack the data concept which is very important in some workflow. Second, due to the strict rule that transitions must be connected through places, a plain Petri Net of a complex workflow might be excessively large. Third, there are no hierarchy concepts in plain Petri Nets. Colored Petri Nets (Jensen, 1997) removed some drawbacks of plain Petri Nets by enabling the tokens to carry values and introducing the concept of *pages*. Therefore, rules can be checked in Colored Petri Nets by using the data values carried by tokens. Pages, which are smaller Colored Petri Nets, can be used to compose a large hierarchical Colored Petri Nets. YAWL (Van Der Aalst & Ter Hofstede, 2005) was designed based on Petri nets providing comprehensive support for workflow patterns, such as complex data, transformations, integration with organizational resources and Web Service integration. YAWL defines new control-flow symbols. By using the new control-flow symbols, a complex workflow can be explicitly transformed.

Other Related Methods. FlowMake is presented by Sadiq and Orlowska in (Sadiq

& Orlowska, 1999). FlowMake models a workflow using a graphical language, including workflow constraints that can be used to verify the syntactic correctness of a graphical workflow model. Reichert and Dadam (Reichert & Dadam, 1998) present a formal foundation for the support of dynamic structural changes of running workflow instances. ADEPTflex is a graph-based modeling methodology that supports users in modifying the structure of a running workflow, while maintaining its correctness and consistency (Reichert & Dadam, 1998). ActivityFlow (L. Liu & Pu, 1997) provides a uniform workflow specification interface to describe different types of workflows and helps to increase the flexibility of workflow processes in accommodating changes. ActivityFlow also allows reasoning about correctness and security of complex workflow activities independently from their underlying implementation mechanisms.

An advantage of graph-based languages is that they are based on formal graph foundations that have rich mathematical properties. The visual capabilities also enhance process design for users and designers. The disadvantage is that graph-based modeling methods are not agile for dynamic runtime issues, requiring the use of specialized notations that can cause the model to become more complex.

2.4.2 Rule-Based Modeling Techniques

In a rule-based modeling approach, business rules are defined as statements about guidelines and restrictions that are used to model and control the flow of a process (Herbst, Knolmayer, Myrach, & Schlesinger, 1994). More recently, rules are used together with agent technology to provide more dynamic ways of handling processes.

Use of Rules in Workflow and Service Composition: Active databases extend traditional database technology with the ability to monitor and react to circumstances that are of interest to an application through the use of Event-Condition-Action (ECA) rules (Widom & Ceri, 1996).

The work of Dayal, Hsu, and Ladin (1991) was one of the first projects to use ECA rules to dynamically specify control flow and data flow in a workflow. In the CREW project (Kamath & Ramamritham, 1998), ECA rules are used to implement control flow. The TrigSFlow (Kappel, Proll, Rausch-Schott, & Retschitzegger, 1995) model uses active rules to decide activity ordering, agent selection, and worklist management. Database representation of workflows (Jean, Cichock, & Rusinkiewicz, 1996) uses Event-Condition-Message rules to specify workflows and utilize database logging and recovery facilities to enhance the fault-tolerance of the workflow application. Migrating workflows (Cichocki & Rusinkiewicz, 1998) provide dynamics in workflow instances. A migrating workflow transfers its code (specification) and its execution state to a site, negotiates a service to be executed, receives the results, and moves on to the next site (Cichocki & Rusinkiewicz, 1998). ECA rules are used to specify the workflow control.

Active rules also provide a solution for exception handling in workflow systems. ADOME (Chiu, Li, & Karlapalem, 1999) and WIDE (Ceri, Grefen, & Sanchez, 1997) are commercial workflow systems that use active rules in exception handling. Rules are also used in workflow systems to respond to ad-hoc events that have predefined actions. Other workflow projects that use active rules are described in (Cichocki, 1998; Doğaç, 1998). Active rules have been used to generate data exchange policies at acquaintance time among peer databases (Kantere, Kiringa, Mylopoulos, Kementsietsidis, & Arenas, 2004). Urban et al. (2001) introduced the Integration Rules (IRules) approach which can interconnect distributed software components.

Agent-Based Techniques. Agent technology has been introduced to model business processes. Agents are autonomous, self-contained and capable of making independent decisions, taking actions to fulfill design goals and to model elements in a business process. Agents also support dynamic and automatic workflow adaptations, thus providing flexibility for unexpected failures. ADEPT (Jennings, Faratin, Norman, Odgers, & Alty, 2000) is an agent-based system for designing and implementing processes. The process logic is defined by a service definition language, where agents have sufficient freedom to determine which alternative path should be executed at runtime. AgentWork (Müller, Greiner, & Rahm, 2004) is a flexible workflow support system that provides better support for exception handling using an event monitoring agent, an adaptation agent, and a workflow monitoring agent. Events represent exceptional conditions, with rule conditions and actions used to correct the workflow. The adaptation agent performs adjustments to the implementation. The workflow monitoring agent checks the consistency of the workflow after adaptation implementation. If the workflow is inadequate, the workflow monitoring agent will re-estimate the error and invoke a re-adaptation of the workflow.

Rule and agent-based modeling methods provide better support for flexibility and adaptability in process modeling. Rule-based methods support modifications at run-time much easier than graph-based methods. It is easy to modify a process model by rule-based methods, and, unlike graph-based methods, rule-based methods do not need new notations to express exception handling processes. Rule-based methods, however, can be difficult to use and understand.

2.5 Failure Recovery Strategies for Web Services

In a service-oriented architecture, a business process can terminate successfully if all activities in it complete successfully or if the process is in a consistent state and the failed activities have been substituted by alternative execution paths. In practice, the great majority of business processes may encounter numerous and diverse failures. Failures can occur anywhere at any time due to the loosely-coupled, autonomous, and heterogeneous characteristics of the execution environment. An activity can fail in many ways, such as an undesirable return value, an unavailable resource, or even hardware failures. Therefore, the failure recovery techniques in web service have been widely discussed. As investigated in (Peltz, 2003), nearly 80% of the time is spent on handling exception (failure) when a business process is executed. Considering parallel process execution, failure recovery and exceptional handling become considerably more difficult. One important reason is that a failed activity may already affect another activity before recovery. In (Greenfield, Fekete, Jang, & Kuo, 2003), the authors used an e-procurement example to show that in many situations, compensation is not enough. In this paper, the authors discussed the shortages of the standard model for handling failures and cancellation and point out that under many environments, even if some aspects of an activity can be undone, it is not always the case that we can return exactly to the original state. The authors also pointed out that the failed activity may affect the concurrent activities. So in many cases, just doing a compensation for the failed activity is not enough. This section summarizes the existing failure recovery techniques in web service composition.

2.5.1 Re-do strategy

One approach to keep a process running is to re-do (re-try) the failed activity. Re-do is the easiest way to handle fault and keep running. However, sometimes re-do procedures are difficult to define in a business process.

Some fault-handling methods for job flow management were presented in (Tan, Fong, & Bobroff, 2010). The authors proposed a new business process execution model called BPEL4JOB. In this model, three fault-handling policies are designed. The cleanup policy gets the failure report and deletes the failed flow instance (assuming no side effect). The re-try policy uses a signal to indicate the job execution state and adds a while loop to each scope. The job will be executed repeatedly if the signal indicates false. The third policy is the re-submission and instance migration method. This policy means exporting job flow instance data in one flow engine, and importing it into another one so that the flow instance can resume in it. The challenge for this policy is to collect sufficient data from the source flow engine.

Another method for handling a re-do mechanism in BPEL was described by (Modafferi & Conforti, 2006). In this method, a re-do procedure is achieved by the event-handler and the compensation-handler. In the compensation-handler, both re-do and compensation procedures are defined following a select structure since only one compensation-handler can be defined for an activity. The aim of the event-handler is to set the variable that will drive the choice between redo and compensation.

In (Vaculín, Wiesner, & Sycara, 2008), based on OWL-S, a recovery mechanism of semantic web service was introduced. In this recovery mechanism, a retry is used as a form of recovery action. A retry action can be defined in a fault-handler, or a constraint violation handler.

2.5.2 Un-do Strategy

Once a crucial error occurs, it is important to clean all of the incorrect data that were generated by the failed activity. Typically, the overall business is recovered to the previous consistent state. The recovery procedure is usually done through the use of compensation.

In (Lakhal, Kobayashi, & Yokota, 2006), the authors used definition rules, composability rules and ordering rules to build a flexible web service composition model. In this model, for each compensatable activity, the users define a compensating procedure, which will be invoked in case of a failure later in the execution of activity that makes it necessary. A concept of vitality degree is also defined in this model. The vitality degree indicates that where some activities are identified as optional, others are tailored as crucial for the overall process.

A concept of automatic compensation was presented in (Wiesner, Vaculín, Kollingbaum, & Sycara, 2008). Since the information of the effect about a service is defined as a process definition in OWL-S, it is possible to discover an automatic compensation based on the effect information. This technique is achieved by searching a service with effect ϵ^{-1} to undo the failed service which has the effect ϵ .

In a loosely-coupled execution environment which allows concurrent activity execution without isolation guarantee, the un-do strategy becomes more difficult to implement, because a failed activity may cause cascaded compensations. Dialani, Miles, Moreau, De Roure, and Luck (2002) proposed a transparent fault tolerance architecture for web services. The authors define a two layered model which consists of an application layer and a service layer. For failure recovery, the application layer implements two key components which are the global fault manager and the fault detector. The service layer includes a local fault manager which is a set of libraries that can be bound dynamically to the service code. In case of a failure, the local fault manager tries to recover the fault first. In case a full recovery is not possible, the local fault manager recovers to a maximal state and escalates the fault notification to the global fault manager. Then the global fault manager initiates a roll back by notifying the affected services. The functionalities of the fault detector are sending the fault notification to the global fault manager and providing a dependency set for the current fault.

A fault handling method in decentralized web service composition was proposed by Chafle, Chandra, Kankar, and Mann (2005). The authors use a partition technique to decentralize web services. The decentralization algorithm partitions a scope in such a manner that the start and end of each scope reside in the same partition which is referred to as the root partition of that scope and the fault handlers and compensation handlers are in the end of scope. In addition to this, each partition except root has an inserted scope start and an inserted scope end which includes an inserted fault handler. When a fault occurs, the fault needs to be propagated to the root partition since the corresponding fault handler only resides in the root partition. Then the fault handler of the root partition of the scope in which the fault occurred, sends a DataCollection control message to its next partition(s) according to the control flow. The message flows along the path traversed by the fault (as per the fault propagation scheme). Now each root partition except the top level scope for the composite service, enters a wait state. At last, the fault handler and compensation handler in the top level root partition will address the fault and compensate completed inner scopes respectively.

2.5.3 Alternative Strategy

Another approach used in failure recovery is to execute an alternative process instead of running the failed process. In many situations, alternative execution paths can totally substitute the failed activity so that the whole business can continue running.

2.5.3.1 Alternative with Un-do

An alternative method is a form of contingency procedure. After failure recovery, the whole process backs up to a consistent state. However, in many cases, the re-do of the failed activity is still unsuccessful. Hence, an alternative execution path is a good solution for maximizing forward recovery.

In (Xiao & Urban, 2009), the authors defined the atomic group and composite group which may include contingency procedures as alternative execution paths. After failure recovery, the contingency procedure of the failed group will be invoked if the contingency exists. Otherwise, the outer group of the failed group will be compensated in order to find an existing contingency procedure at the outer level. In other words, once a group fails, the whole process will be compensated recursively until a contingency procedure is found. Another replace operation was provided in (Vaculín et al., 2008). A ReplaceBy(otherProcess) tries to use another process as a substitute for a failed process. In (Wiesner et al., 2008), a more flexible operation named ReplaceByEquivalent was introduced. Because the OWL-S process model defines inputs, outputs, preconditions, and effects by using existing algorithms for automatic web service discovery (matchmaking) (Sycara, Paolucci, Ankolekar, & Srinivasan, 2003), the information is used to dynamically find an alternative service. Based on ReplaceByEquivalent, the authors also give a definition of advanced backward and forward recovery. After a failure, a rollback is performed first for all processes that have finished at the same level. Then, if ReplaceByEquivalent can find an alternative service, it is executed. Otherwise, the back recovery is repeated at one higher level in the hierarchy.

2.5.3.2 Alternative without Un-do

In some failure recovery approaches, the failed activity is not crucial and alternative execution can be used to keep the business process running. Generally, each activity has several back-up activities in these approaches.

The Primary-Backup method (Zhang, Zagorodnov, Hiltunen, Marzullo, & Schlichting, 2004) uses a backup service to substitute the failed primary service in grid services. In this model, each primary service has one or more backup services. Before replying to the client, the primary service needs to send the execution state to every backup service. If these backup services receive a failure notification, or do not receive a heartbeat message after a certain period of time, these backups need to cooperate to elect a new primary service. The newly elected primary service then sends a failover notification to the client so it can obtain a new server instance handle.

The merit of this approach is that it saves the expensive rollback or compensation of the failed activities. However, because the failed activity is just abandoned, the alternative strategy does not support the atomicity point of the ACID properties of the traditional transaction. Traditional transaction concepts require either all operations have completed or otherwise none has happened. Furthermore, an alternative without un-do method usually does not consider concurrent process execution errors. For example, if other concurrent activities have data dependencies on the failed activity, all results of these activities become unbelievable.

2.5.4 Other Techniques in Executable Process Language

The WS-BPEL standard provides fault-handler, compensation-handler, and termination-handler attached to a scope to handle execution exception. To continue the process execution in case an exception occurs, the fault-handler might invoke the compensation-handler first to un-do the completed portion in the scope. If the corresponding compensation-handlers are not specified, then the default compensation-handler is assigned to the scope. However, in some cases, the default compensation-handler may cause complications and return unexpected results. Khalaf, Roller, and Leymann (2009) highlight 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, and 2) high complexity of the default compensation order due to the default handler behavior. Instead of the standard fault and compensation mechanism in BPEL, Khalaf et al. (2009) proposed a new and deterministic mechanism to better handle default compensation for scopes. In the new mechanism, the relationships between scopes include both structured nesting and graph-based links. Therefore, in case of an execution exception, the model can calculate the default compensation order before starting the compensation procedure.

Several efforts have been made to enhance the BPEL fault and exception handling capabilities. The work in (Modafferi & Conforti, 2006) proposed mechanisms like external variable setting, future alternative behavior, rollback and conditional re-execution of the flow, timeout, and redo mechanisms for enabling recovery actions using BPEL. The work in (Modafferi, Mussi, & Pernici, 2006) presented the architecture of the SH-BPEL engine, a Self-Healing plug-in for WS-BPEL engines that augments the fault recovery capabilities in WS-BPEL with mechanisms such as annotation, pre-processing, and extended recovery. The Dynamo (Baresi, Guinea, & Pasquale, 2007) framework for the dynamic monitoring of WS-BPEL processes weaves rules such as pre/post conditions and invariants into the BPEL process.

In checkpointing systems, consistent execution states are saved during the process flow. During failures and exceptions, the activity can be rolled back to the closest consistent checkpoint to move the execution to an alternative platform (Luo, 2000). The work in (Dialani et al., 2002) uses the means of checkpointing and rollback to detect and recover the faults. The rule-based technique can be also used with checkpointing systems. Marzouk, Maalej, Rodriguez, and Jmaiel (2009) introduced a periodic checkpointing based approach for strong mobility of orchestration processes. With a set of rules, WS-BEPL processes can be transformed to equivalent mobile ones.

This approach can be used as a self-healing mechanism that supports recovering the execution of instances stopped following a failure and resuming the execution starting from the last checkpoint.

Aspect-oriented programming (AOP) is another way of modularizing and adding flexibility to service composition through dynamic and autonomic composition and runtime recovery. In AOP, aspects are weaved into the execution of a program using join points to provide alternative execution paths (Charfi & Mezini, 2007). 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. The work in (Charfi & Mezini, 2006) illustrates the application of aspect-oriented software development concepts to workflow languages to provide flexible and adaptable workflows. AO4BPEL (Charfi & Mezini, 2007) is an aspect-oriented extension to BPEL that uses AspectJ to provide control flow adaptations (Kiczales et al., 2001). Business rules can also be used to provide more flexibility during service composition. AO4BPEL enhances the limited capabilities of BPEL in terms of modularity and dynamic adaptability. XML files are used to provide functionality in AO4BPEL to avoid changing the service composition during runtime. In contrast with the standard WS-BPEL, AO4BPEL provides better support for functional modularization.

2.6 Summary

In this section, many past and ongoing research projects that support dynamic web service composition are presented. However, due to the distributed nature of services, flexibility in service composition is still a challenging research topic. Even BPEL, the de-facto standard for composing web services, still lacks sophistication with respect to handling faults and events. The focus of the proposed research is to use event-driven and rule-based techniques to provide better flexibility to dynamically respond to failure in web service composition. The proposed research is different than the related work by using event-driven techniques to trigger rule checking to determine the recovery actions. In this way, how to respond to a process exception is fully dependent on the current execution status and rule checking result. Therefore, rather than uniform compensation defined in the process specification, different process instances might respond to the same exception through different actions. In addition, by analyzing the data dependency in a decentralized manner, this research provides better recovery support for failures among concurrent processes in terms of correctness and efficiency.

CHAPTER 3
MOTIVATION AND STATEMENT OF RESEARCH OBJECTIVES

This chapter provides background and motivation for the proposed research. Section 3.1 gives an overview of the foundational work with the decentralized data dependency analysis project. The research challenges of this foundational work are discussed in Section 3.2. Finally, the statement of objectives of the proposed research is itemized in Section 3.3.

3.1 Overview of the Decentralized Data Dependency Project

As illustrated in the previous section, most process modeling techniques are aligned with either a procedural approach, specified as a flow graph, or a rule-driven approach, where events and rules are used to control the flow of execution. Rules provide a more dynamic way to respond to events that represent a need to change the normal flow of execution. The use of rules in process modeling is especially important considering the growing prevalence of complex events, event-driven applications, and business activity monitoring.

A dynamic approach to process modeling for service-oriented environments requires a combination of graph and rule-based techniques, where graph-based techniques provide a means for specifying the main application logic and events are used to interrupt or branch off of the main flow of execution, triggering rules that check constraints, respond to exceptions, and initiate parallel activity. Events and rules should also play an increased role in supporting failure and recovery activity. Planning for failure and recovery should be an integral component of process modeling for service-oriented architectures, especially in the context of concurrently executing processes that access shared data and cannot enforce traditional transactional properties.

The proposed research is a component of the decentralized data dependency analysis project (NSF Grant No. CCF-0820152), which addresses consistency checking as well as failure and recovery issues for service-oriented environments through the use of integration rules, invariants, and application exception rules, used together with a checkpointing concept known as assurance points. To motivate the use of these concepts, consider a decentralized execution environment consisting of Process

Execution Agents (PEXAs) as shown in Figure 3.1. In this research, a PEXA is responsible for monitoring the execution of different processes. In Figure 3.1, PEXA 1 is responsible for the execution of P_1 and P_4, PEXA 2 is responsible for the execution of P_2, and PEXA 3 is responsible for the execution of P_3. Each process invokes services at various locations within the network. As shown in Figure 3.1, P_1 invokes operation_a at the site of PEXA 1, operation_b and operation_c at the site of PEXA 2, and operation_d at the site of PEXA 3.

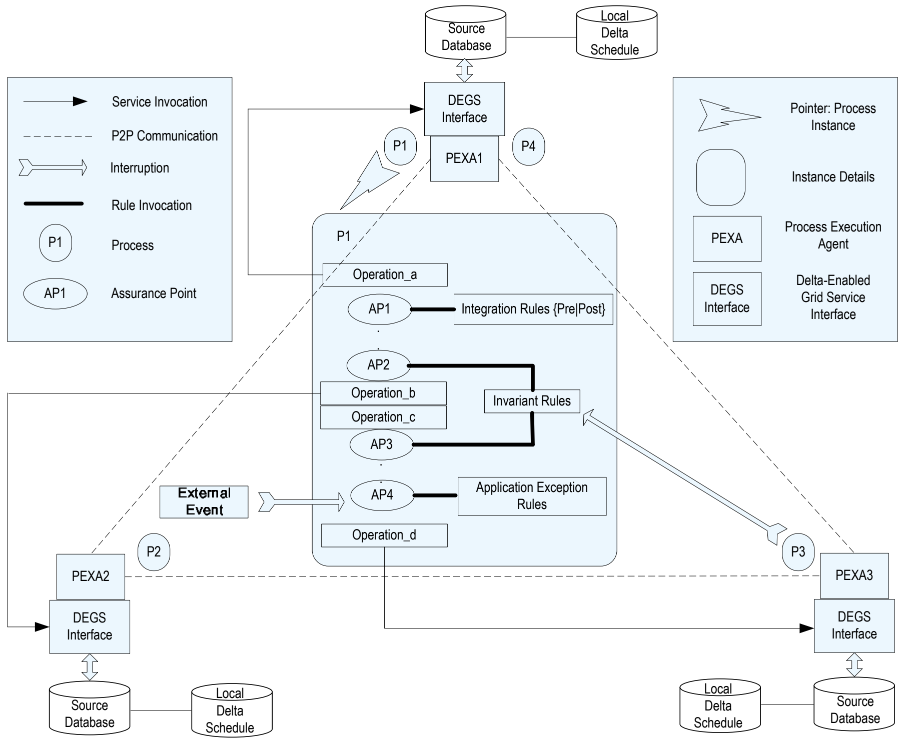


Figure 3.1. Decentralized Process Execution Agents with Events and Rules (Urban, Gao, Shrestha, & Courter, 2011)

Figure 3.1 also illustrates that PEXAs are co-located with Delta-Enabled Grid Services (DEGS). A DEGS is a Grid Service that has been enhanced with an interface that stores the incremental data changes, or *deltas*, that are associated with service execution in the context of globally executing processes (Blake, 2005; Urban, Xiao, et al., 2009). A DEGS uses an OGSA-DAI Grid Data Service for database interaction. The database captures deltas using capabilities provided by most commercial database systems. The original implementation of DEGS (Blake, 2005; Urban, Xiao, et al., 2009) has experimented with the use of triggers as a delta capture mechanism, as well as the Oracle Streams capability (Urban, Xiao, et al., 2009). Oracle Streams is a feature that monitors database redo logs for changes and publishes these changes to a queue to be used for replication or data sharing.

3.1.1 Decentralized Data Dependency Analysis

Deltas captured using DEGS are stored in a delta repository that is local to the service. The past work in (Xiao, 2006; Xiao, Urban, & Dietrich, 2006) has experimented with the creation of a centralized Process History Capture System (PHCS) that included deltas from all DEGSs in the environment and the process runtime context generated by the process execution engine. Deltas are dynamically merged using timestamps as they arrive in the PHCS to create a time-ordered schedule of data changes from the DEGS. The global delta object schedule is used to support recovery activities when process execution fails (Xiao, 2006; Xiao & Urban, 2009, 2008; Xiao, Urban, & Liao, 2006), where the global delta object schedule provides the basis for discovering data dependencies among processes.

The most recent work with the decentralized data dependency project has transformed the global delta object schedule into a distributed schedule with a decentralized algorithm for discovering data dependencies (Urban, Liu, & Gao, 2009; Z. Liu, 2009). As a result, each PEXA in Figure 3.1 has its own local delta object schedule. Z. Liu (2009) developed a decentralized approach to analyzing data dependencies among concurrently executing processes in an SOA using PEXAs to control the execution of processes and to build local delta object schedules. At each execution site, the PEXA contains a local PHCS which only captures the local deltas. A data dependency graph is built at each PEXA according to the loacl delta object schedule. PEXAs then communicate with each other about data dependencies in a peer-to-peer manner.

Figure 3.2 shows the interleaved execution view of each process and operation from a data access point of view when op_{12} fails at time $\mathsf{t8}$. There are four processes that are executing concurrently. Each process involves multiple operations and each operation accesses a specific data object. In Figure 3.2, D1, D2, and D3 represent three independent service providers. Each provider involves one or more data objects. For instance, D1 has three data objects which are X1, Y1, and Z1. The global process dependency graph for the four active processes is shown in the upper right part of Figure 3.3, indicating that the process dependency graph is $p1 \leftarrow p2 \leftarrow p3 \leftarrow p4$. The recovery process is invoked when op_{12} fails at site D1 and invokes the compensation of p_1, which is controlled by PEXA 1. Figures 3.2 and 3.3 together illustrate that PEXA 1 can detect that p_2 is dependent on p_1 due to modification of X1. PEXA 1 can also detect that p_4 is dependent on p_3 due to modification of Y1, but PEXA 1 cannot identify this dependency as part of the global graph for p_1 because of the distributed nature of the execution. As shown in Figure 3.3, p_3 is not dependent on p_1, p_2, or p_4 based on data access patterns at D1, but p_3 is dependent on p_2 based on data accessed at D2. Disconnected graphs such as those in PEXA 1 of Figure 3.3 are referenced to as *hidden dependencies*. Additional execution information must be recorded to link together all distributed components of the graph and to identify hidden dependencies within a single PEXA. Therefore, a PEXA that controls a process that invokes a service at a different site must create a *link object* to record information about the site where the service is executed. In Figure 3.3, PEXA 2 creates a link object to indicate that op_{21} of process p_2 is executed at the site of PEXA 1. PEXA 3 creates two link objects to record the fact that op_{31} executes at PEXA 2 and op_{32} executes at PEXA 1. Used in combination, link objects together with an indication of internal or external process invocation can be used to dynamically discover global, distributed process dependency graphs.

Two different algorithms for decentralized data dependency analysis were developed in (Z. Liu, 2009). The Lazy approach uses a passive method to build the data dependency graph. When a process encounters an error, the PEXA where the failure occurs starts to build the local data dependency graph and then communicates with other PEXAs to recover the error. In contrast, the Eager approach builds the data dependency graph at run time. Once an error occurs, PEXAs use the existing local data dependency graphs and communicate with each other to recover the error.

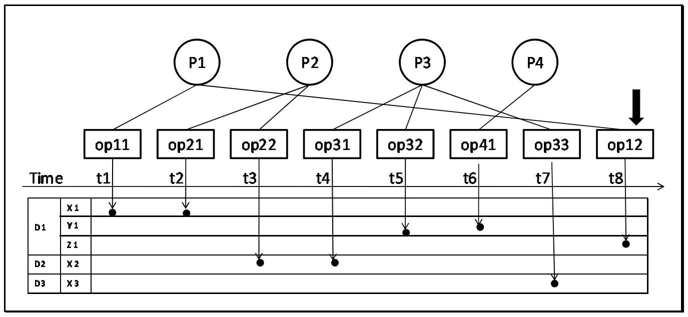


Figure 3.2. Data Access View of Interleaved Execution
(Urban, Liu, & Gao, 2009)

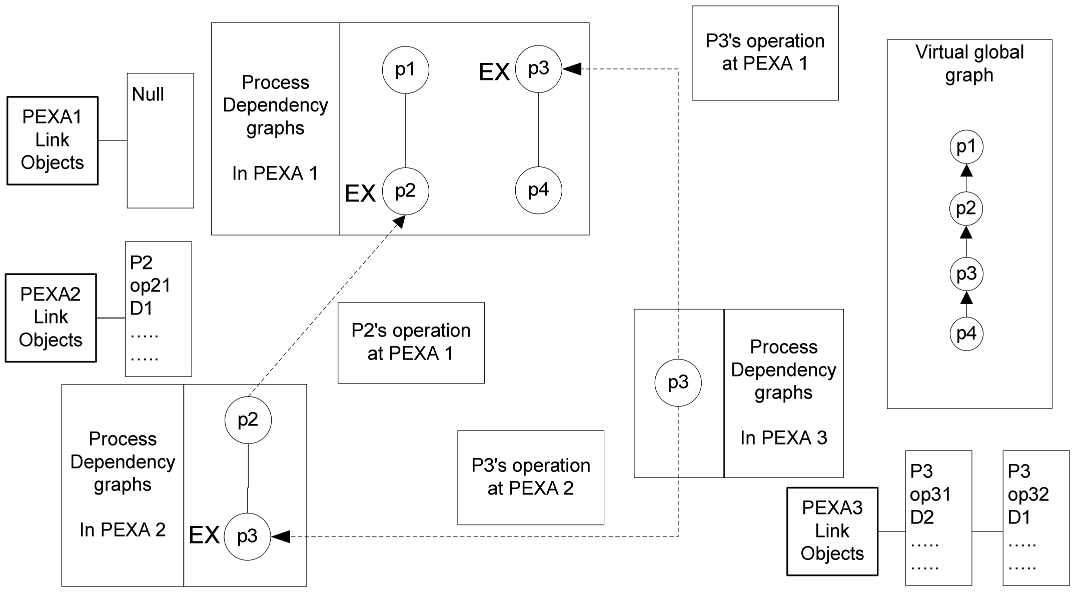


Figure 3.3. Global, Distributed Process Dependency Graph
(Urban, Liu, & Gao, 2009)

3.1.2 The Assurance Points Model with Integration and Invariant Rules

Given that processes can execute in an environment where decentralized PEXAs can monitor data changes and communicate about data dependencies among concurrently executing processes, the decentralized data dependency analysis project focuses on how to enhance the ability to monitor data consistency in the failure and recovery process. To illustrate this approach, the expanded view of P1 in Figure 3.1 shows the use of assurance points (APs) and the different types of rules that have been defined to support data consistency and process recovery. As shown for P1, APs can be placed at strategic locations in a process, where an AP is a combined logical and physical checkpoint that can be used to store execution data, alter program flow, and support recovery activity.

In (Xiao & Urban, 2009), a process is hierarchically structured by several 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. Atomic groups and composite groups 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. A composite group can have its own compensation and contingency as optional elements. A process is essentially a top-level composite group. Contingency is always tried first upon the failure of a group. The compensation process will only be invoked if there is no contingency or if the contingency fails.

Urban, Gao, Shrestha, and Courter (2010) extended the model by introducing APs in the execution of a process, providing the capabilities of checking pre/post conditions through the use of integration rules (IRs). An IR as shown in Figure 3.4, which is based on the work in (Urban et al., 2001), is triggered by a process reaching a specific AP during execution. Upon reaching an AP, the condition of an IR is evaluated. If the condition evaluates to true, the action specification is executed to invoke a recovery action. As part of the recovery process, there is a possibility for the process to execute through the same pre or post condition a second time, where *action* 2 is invoked rather than *action* 1.

Three different forms of backward recovery are described in (Urban, Gao, Shrestha, & Courter, 2010), with the different forms supporting either full backward recovery or a combination of backward and forward recovery. APRetry is used when the running process needs to be backward recovered to a previously-executed AP. APRollback is used when the overall process has more severe errors and must be recovered back to the beginning of the process. APCascadedContingency is a hierarchical backward recovery that continues to compensate nested processes, checking each AP that is encountered for a possible contingent procedure that can be used to correct an execution error.

Figure 3.5 shows a portion of an online shopping process. In Figure 3.5, composite group cg_2 contains two atomic groups, shown as the solid line rectangles. The optional compensations and contingencies are shown in dashed line rectangles, denoted as cop and top. The two APs, which are OrderPlaced(orderId) and CreditCardCharged(orderId, cardnumber, amount), are placed before and after cg_2. The OrderPlaced AP has a pre-condition IR 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 IR that further guarantees the in-stock quantity must be in a reasonable status after the decInventory operation.

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

Figure 3.4. Integration Rule Structure

Another use of an AP is to activate invariant rules. As shown in Figure 3.1, Invariants indicate conditions that must be true during process execution between two different APs. As shown for P_1 in Figure 3.1, an invariant is monitored during the execution between AP2 and AP3, where the invariant represents a data condition that is critical to the correct execution of P_1. P_1, however, may not be able to lock the data associated with the invariant during the service executions between AP2 and AP3. Given that DEGS can be used to monitor data changes, P_1 can activate the invariant condition, but still allow concurrent processes to access shared data. P_1 can then be notified if data changes violate the invariant condition. For example, if P_3 modifies data associated with the invariant of P_1, P_1 can re-evaluate the invariant

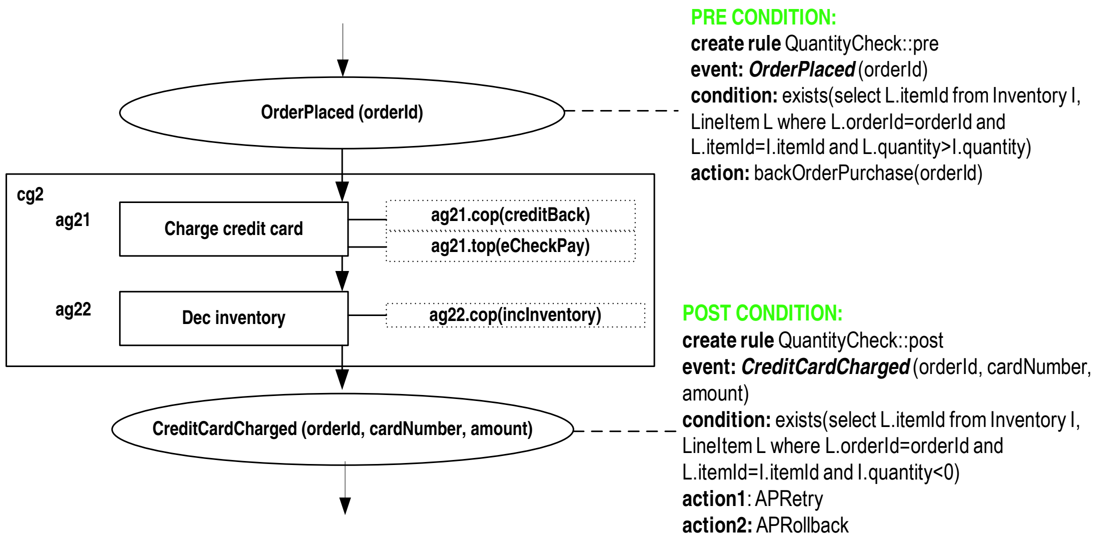

PRE CONDITION:
create rule QuantityCheck::pre
event: OrderPlaced (orderId)
condition: exists(select L.itemId from Inventory I, LineItem L where L.orderId=orderId and L.itemId=I.itemId and L.quantity>I.quantity)
action: backOrderPurchase(orderId)
OrderPlaced (orderId)
cg2
ag21
Charge credit card
ag21.cop(creditBack)
ag21.top(eCheckPay)
ag22
Dec inventory
ag22.cop(incInventory)
POST CONDITION:
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
CreditCardCharged (orderId, cardNumber, amount)

Figure 3.5. APs in Online Shopping Process

condition and invoke recovery actions if needed.

3.1.3 Application Exception Rules

Application exception rules (AERs) are another type of rule used in combination with APs (Ramachandran, 2011). AERs have a case-based structure and are specifically used to respond to interrupts caused by events external to the process. Within a process, each AP represents the fact that a process has passed certain critical points in the execution. Application exception rules are then written to respond to exceptions according to the AP status of a process. Based on the most recently passed AP, a corresponding condition/action pair will be examined. Therefore, AERs enable a process to dynamically respond to the event depending on its current execution status. In Figure 3.1, P_1 also illustrates the use of application exception rules at AP4. A process should be capable of responding to external events that may affect execution flow. The response to the event, however, may depend on the current status of the process. For example, P_1 may respond one way if the process has passed AP4, but may respond differently if the process is only at AP1.

Figure 3.6 shows a basic use of AERs. Application exception rules have a case structure, defining recovery actions based on APs. When an exception occurs, application exception rules are triggered. The exception handling procedure to execute varies according to the AP status of the process, where recovery actions can query the execution state associated with the most previous AP. As shown in Figure 3.6, one instance of process A executes recoveryAction1 since the process has passed AP1 but not AP2. The other instance of process A executes recoveryAction2 since the process has passed AP2.

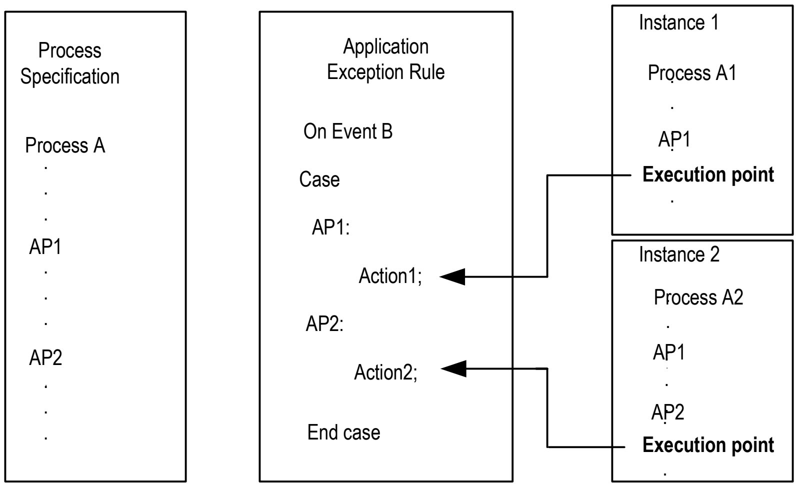


Figure 3.6. The Use of Application Exception Rules

AERs also have an additional functionality. Since PEXAs can communicate about data dependencies among concurrently executing processes, when a process P_j invokes recovery procedures in response to integration rules, invariant conditions, or application exception rules, event notifications can be sent through P2P communication to dependent processes that are controlled by other PEXAs. Application exception rules can be used by a process P_i to intercept such events, determine how the failure and recovery of P_j potentially affects the correctness conditions of P_i, and respond in different ways depending on the AP status of the process.

3.2 Research Challenges for Existing Work

There are still many challenges in the existing work described in the Section 3.1.

i) From an application point of view, a complete model must support sequence, if-else, parallel, and loop control structures. In the current service composition and recovery model, however, execution and recovery for processes with complex control structures have not been fully developed. A full suite of control structures will complicate the semantics of the service composition, especially for recovery execution. In addition, assurance points and application exception rules will also need further improvements to assure the correctness of execution with complex control structures.

ii) Existing discussion of the semantics of APs and the different rule structures and recovery actions is explained through prototyping and examples. The semantic definition of the assurance point approach still needs formal specifications and verifications. Petri Nets are good candidates to formally validate web service compositions. Some high-level Petri Net theories, such as colored Petri Nets (Jensen, 1987), timed Petri Nets (Ramchandani, 1973), the workflow net (Van Der Aalst, 1998) and YAWL (Van Der Aalst & Ter Hofstede, 2005), can provide a more concise way to describe the AP service composition model.

iii) The AP model can recover a single process but does not address consistency issues for processes that are data dependent on the recovered process. Ideally, the AP model should be integrated with the decentralized data dependency analysis process. However, due to the partial recovery of a process in the AP model, PEXAs must be able to build dependency graphs based on only the recovered portion of a process.

3.3 Statement of Objectives

The global objective of this research is to formally define a hierarchical web service composition and recovery model based on the use of assurance points, integration rules and application exception rules. To provide a more robust model, the rule-based failure recovery approach will be integrated with the decentralized data dependency analysis algorithm, so that decentralized PEXAs can communicates about dependencies associated with the partial process recovery. To achieve this objective, the following sub-objectives will be investigated:

1) Fully define the execution and recovery semantics of the AP model with event-driven and rule-triggered techniques in the context of programming control structures. The investigation will focus on the execution and recovery with if-else, parallel, and loop control structures integrated with the use of APs, integration rules and AERs.

2) Formalize the execution and recovery semantics of the AP approach by using high level Petri Nets. The workflow language, YAWL (Van Der Aalst & Ter Hofstede, 2005) will also be used to provide a more abstract view of the Petri Nets specifications.

3) Investigate the integration of the decentralized data dependency analysis algorithm with the AP recovery model. This investigation will involve the development of an algorithm for building data dependency graphs based on the partial recovery of a process and the use of AERs to trigger the write/read dependent events.

4) Evaluate the performance and functionality of the decentralized data dependency analysis approach integrated with the AP model. A simulation environment for the decentralized data dependency analysis approach will be developed to evaluate the performance and functionality of the system.

The proposed research focuses on defining a new paradigm for service execution that uses rule-based techniques for testing user-defined semantic conditions. With the new web service model, a dynamic and intelligent approach can be developed to monitoring failures, detecting data dependencies, and responding to failures and exceptional events. The methodology for conducting this research is presented in the next section.

CHAPTER 4
DISCUSSION OF RESEARCH OBJECTIVES

This chapter provides a more detailed discussion of the research objectives. Section 4.1 discusses recovery issues associated with the use of programming control structures in the AP model. A Petri Net and YAWL formalization of the AP model is discussed in Section 4.2. Section 4.3 discusses research issues for decentralized data dependency analysis within the AP model.

4.1 The AP Model with Programming Control Structures

In the current web service composition model with APs, the execution and recovery semantics of if-else, parallel, and loop control structures have not been fully defined. Furthermore, due to the complexity of the control structures, adequate and meaningful points in the control structures for the placement of APs needs to be investigated. This section first provides an overview of recovery actions in the AP model. Extensions needed in the context of additional control structures are then addressed.

4.1.1 Recovery Actions in the AP Model

Figure 4.1 shows an abstract view of a sample process definition, where the boxes represent different components of the process composition. Ovals represent APs, while the broad, curved arrows denote recovery actions. APs and recovery actions will be addressed in the following paragraphs.

The main process in Figure 4.1 is the top-level composite group cg_0. This composite group is composed of three composite groups cg_{01}, cg_{02}, and cg_{03} followed by two atomic groups ag_{04} and ag_{05}. Similarly, cg_{01}, cg_{02}, and cg_{03} 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 ag_{05}, 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.

Contingency is always tried first upon the failure of a group. The compensation process will only be invoked if there is no contingency or if the contingency fails. For

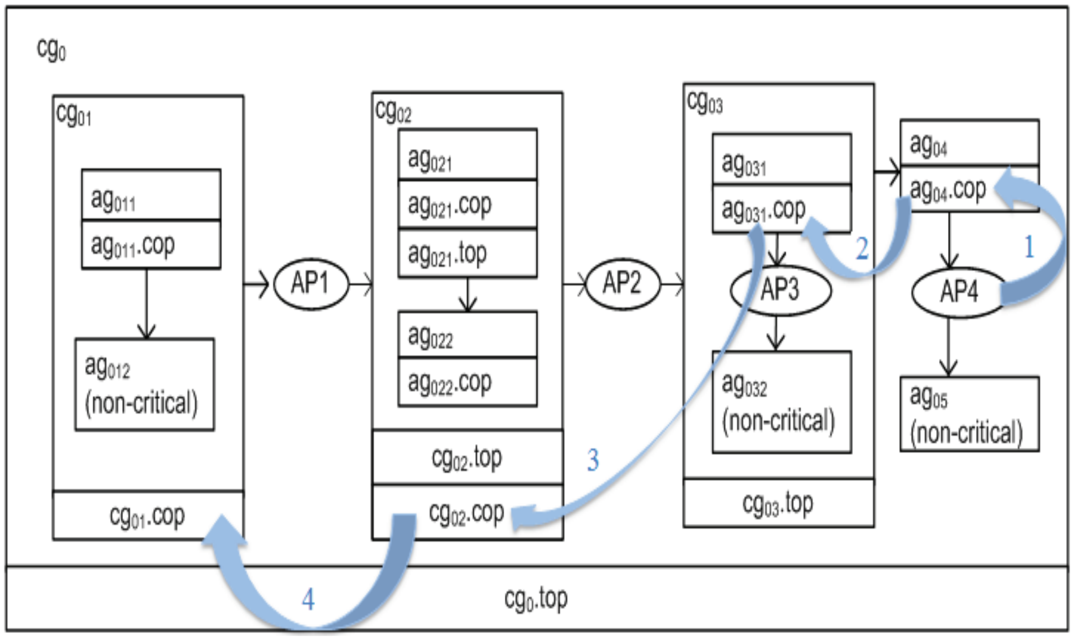


Figure 4.1. Generic Process: Scenario 1 (APRollback)
(Urban, Gao, Shrestha, & Courter, 2010)

example in Figure 4.1, if ag_{021} fails, ag_{021}.top will be executed.

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 4.1, if the contingent procedure ag_{021}.top fails, the recovery process will first try to compensate cg_{01} using the associated compensating procedure, cg_{01}.cop. If the shallow compensation fails, deep compensation will be invoked by executing ag_{011}.cop. Note that ag_{012} is non-critical and does not require compensation. After compensating cg_{01}, the contingent procedure for the top-most composite group (i.e., cg_{0}.top) will be executed.

An AP is defined with a unique identifier, a set of parameters which list the critical data items to be stored and checked, and a set of pre and post conditions defined as integration rules (IRs). An IR is triggered by a process reaching a specific AP during execution. The condition represents a user-defined constraint. If the constraint is violated, the action is executed to trigger a recovery action. 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 to logically reverse the current state of the entire process using shallow and deep compensation.

  Scenario 1 (APRollback): Assume that the post-condition fails at AP4 in Figure 4.1 and that the IR action is APRollback. Since APRollback is invoked, the process compensates all completed atomic and/or composite groups. The APRollback execution sequence is numbered in Figure 4.1. First the process invokes $\mathsf{ag}_{04}\mathsf{.cop}$ to compensate ag_{04}. Second, the APRollback process will deep compensate ag_{031} by invoking $\mathsf{ag}_{031}\mathsf{.cop}$ since 1) there is no shallow compensation for cg_{03} and 2) ag_{032} is non-critical and therefore has no compensating procedure. Finally, APRollback invokes shallow compensation $\mathsf{cg}_{02}\mathsf{.cop}$ and $\mathsf{cg}_{01}\mathsf{.cop}$.

  The APRollback procedure is a standard way of using compensation in past work. The originality of the rollback process in our work is the way in which it is used together with APs in the retry and cascaded contingency recover actions.

- ***APRetry.*** APRetry is used to recover to a specific AP and then retry the recovered atomic and/or composite groups. If the AP has an IR that is a pre-condition, then the pre-condition will be re-examined. If the pre-condition fails, the action of the rule is executed, which either invokes an alternate execution path for forward recovery or a recovery procedure for backward recovery. By default, APRetry will go to the most recent AP. APRetry can also include a parameter to indicate the AP that is the target of the recovery process.

  Scenario 2 (APRetry-default): Assume that the post-condition of an IR fails at AP4 in Figure 4.2 and that the action of the IR is APRetry. This action compensates to the most recent AP within the same scope by default. In Figure 4.2, APRetry first invokes $\mathsf{ag}_{04}\mathsf{.cop}$ to compensate ag_{04} at step 1. The process then deep compensates cg_{03} by executing $\mathsf{ag}_{031}\mathsf{.cop}$ at step 2. At this point, AP2 is reached and the pre-condition of the IR is re-evaluated shown as step 3. If the pre-condition fails, the process executes the recovery action of IR. If the pre-condition is satisfied or if there is no IR, then execution will resume again from cg_{03}. In this case, the process

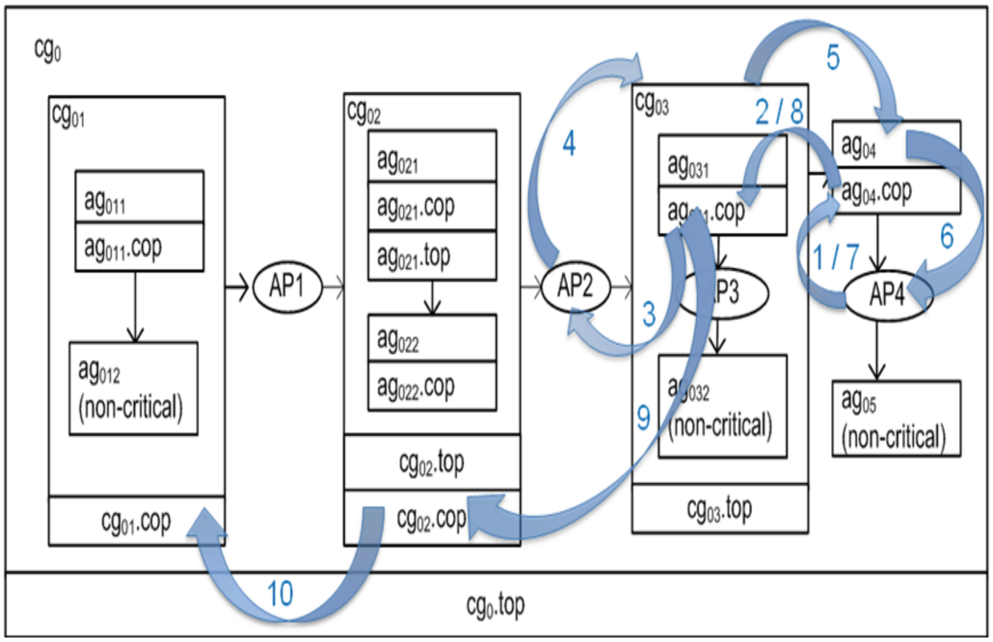


Figure 4.2. CScenario 2 (APRetry-default)
(Urban, Gao, Shrestha, & Courter, 2010)

will reach AP4 a second time through steps 4, 5 and 6, where the post-condition is checked once more. If failure occurs for the second time, the second action defined on the rule is executed rather than the first action (IRs can specify multiple actions for the case when a retry fails). If a second action is not specified, the default action will be APRollback as steps 7 through 10.

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

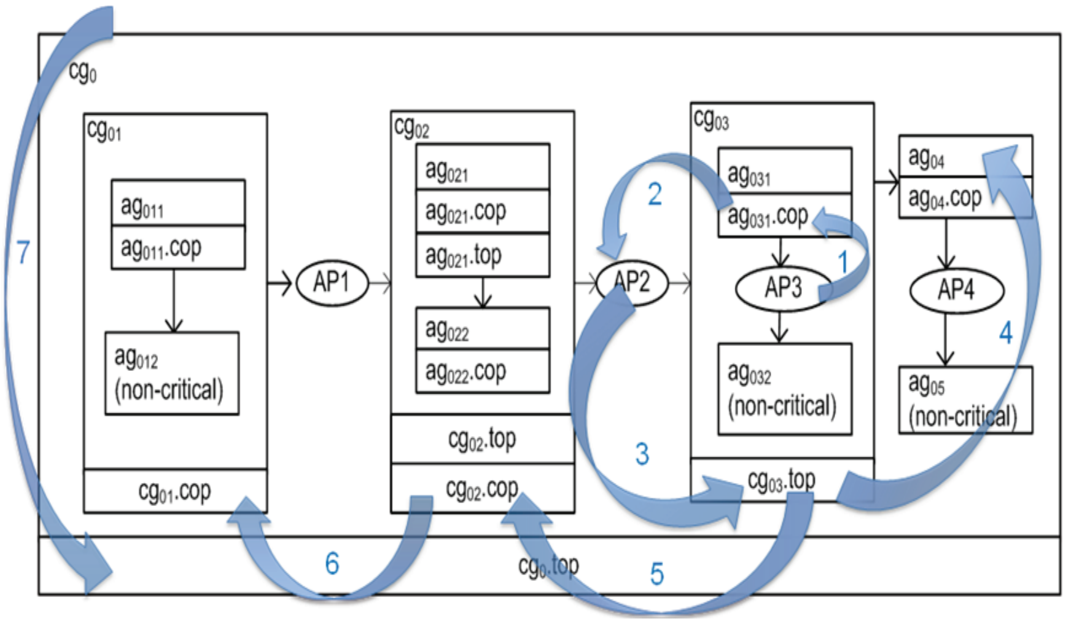


Figure 4.3. Scenario 4 (APCC)

Scenario 3 (APCC): Assume that the post-condition fails at AP4 in Figure 4.1 and that the IR action is APCC. The process starts compensating until it reaches the parent layer. In this case, the process will reach the beginning of cg0 after compensating the entire process through deep or shallow compensation through the same steps as shown in Figure 4.1. Since there is no AP before cg_0, cg_{03}.top is invoked.

Scenario 4 (APCC): Assume that the post-condition fails at AP3 in Figure 4.3 and that the IR action is APCC. Since AP3 is in cg_{03}, which is nested in cg_0, the APCC process will compensate back to the beginning of cg_{03}, executing ag_{031}.cop at step 1. The APCC process finds AP2 with an IR pre-condition for cg_{03} at step 2. As a result, the pre-condition will be evaluated before trying the contingency for cg_{03}. If there is no pre-condition or if the pre-condition is satisfied, then cg_{03}.top is executed at step 3 and the process continues shown as step 4. Otherwise, the recovery action of the IR pre-condition for AP2 will be executed and the process quits APCC mode.

If cg_{03}.top fails at step 3, then the process will still be under APCC mode, where the process will keep compensating through steps 5 and 6 until it reaches the cg_0 layer, where cg_0.top is executed at step 7.

4.1.2 Recovery Issues for Additional Control Structures

The recovery actions for the AP model will be extended for use in the context of additional control structures, such as parallel, if-else, and loop control structures.

In BPEL, the parallel control structure is supported by the flow activity. The flow activity specifies multiple threads that can execute in parallel. The flow activity completes when all threads have completed. For example, a loan application process can contain a flow activity that sends the loan requests to two different banks simultaneously. In flow activity, all threads are running independently and do not need to wait for others to complete.

The parallel control structure involves multiple concurrently running threads. In a single process, concurrently running paths are normally data-independent. Therefore, in this research, data-independence among concurrent paths is assumed in parallel activity. A new group, known as a flow group, will be added to the service composition model in addition to atomic and composite groups. A flow group can contain multiple composite groups and atomic groups executed in parallel and independently. Figure 4.4 shows an example of a flow group that contains two composite groups and one atomic groups. A flow group succeeds only when all groups have succeeded. A flow group can also have optional shallow compensation (fg_1.cop) and contingency (fg_1.top). A shallow compensation will compensate the effects done by all threads involved in the flow group. A contingency will be used as an alternative execution path to the entire flow group. Similar to a composite group, shallow compensation of a flow group involves the execution of a compensating procedure attached to an entire flow group, while deep compensation involves the execution of compensating procedures for each group within a flow group.

An initial investigation of the flow group in the AP model has been presented in (Friedman, Urban, Gao, & Shrestha, 2010). As shown in Figure 4.5, an AP can be placed:

- immediately before the flow group (AP_n),

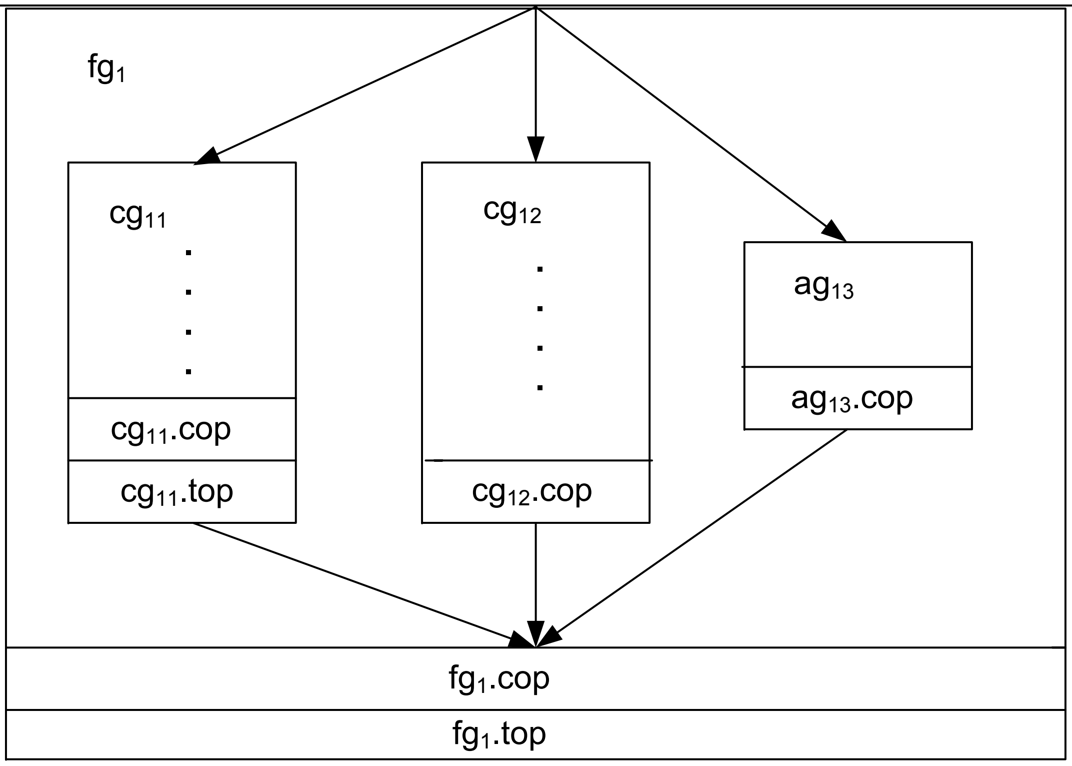


Figure 4.4. An Example of Flow Group

- inside of the flow group before the start of a specific composite group (AP_{n1}, ... , AP_{ni}),
- or as the first element of a composite group (AP_{n11}, ..., AP_{ni1}) within a flow group.

In a flow group, if a constraint violation or an execution failure occurs in a thread, the thread will first try to recover the error inside of the thread by an APRetry or APCC action. If the recovery action succeeds, all other threads in the flow group will not be affected. However, if the failed thread cannot recover from the error, the flow group will be under APCC mode and all threads must be recovered to invoke the contingency of the entire flow group. In addition, if at any point a thread is marked for APRollback as a form of recovery, all concurrent threads must also be flagged for APRollback and the entire process will be recovered.

As describe in (Friedman et al., 2010), more than 10 execution and recovery cases have been identified based on the placement of APs in Figure 4.5. As an example of

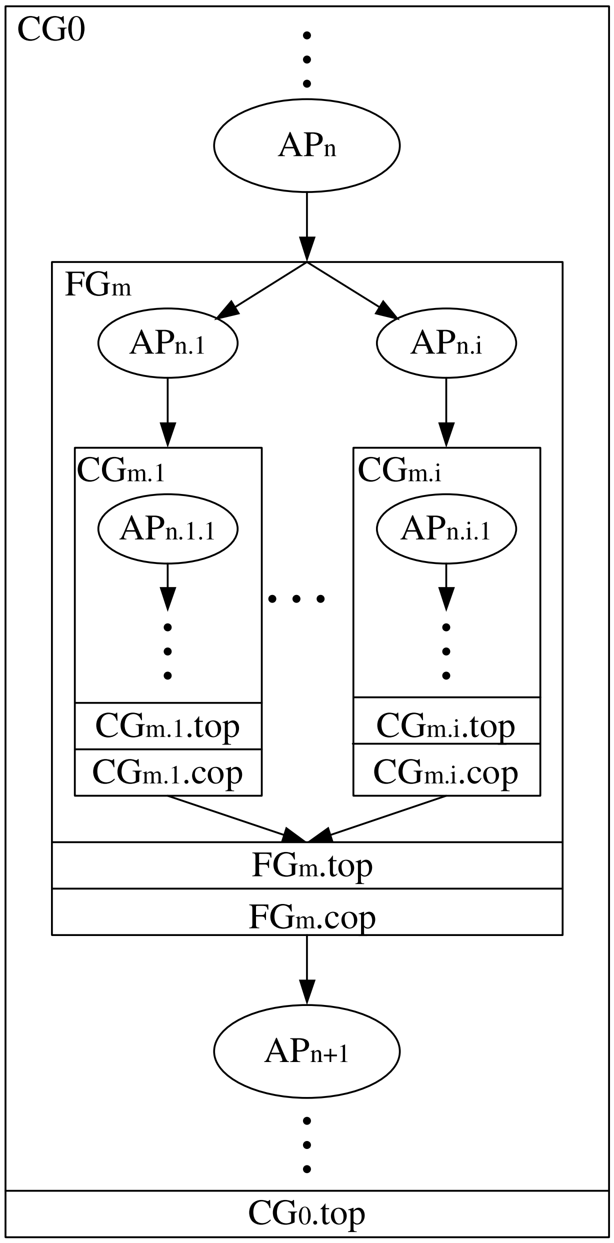


Figure 4.5. General Flow Group with Possible AP Placement

the recovery issues that must be addressed, consider the following two cases:

- **Case 1** Innermost AP only with APRetry

  Assumptions: AP_n undefined, AP_{ni} undefined, AP_{ni1} defined.

In the case where AP_n and AP_{ni} do not exist and AP_{ni1} is present, if at some point during the execution of CG_{mi}, an APRetry is encountered, execution of CG_{mi} will halt and execute compensation procedures for completed tasks within CG_{mi}, returning the state to AP_{ni1} . There, the precondition of AP_{ni1} will be checked. If it is still valid, the procedure will attempt to re-execute CG_{mi} without affecting the concurrent threads. Otherwise, the recovery action for AP_{ni1} will be performed, which will require exiting to the outer scope of FG_m to perform APRollback or APCC. As a result, all other threads will also be compensated according to the recovery action at AP_{ni1}.

- **Case 2** Middle and innermost APs with APCC

  Assumptions: AP_n undefined, AP_{ni} defined, AP_{ni1} defined.

  Suppose a failure occurs in CG_{mi} with a recovery action of APCC. In this case, all of CG_{mi} will be compensated, and the precondition at AP_{ni} (i.e., AP_{ni1} in this case is ignored) will be rechecked. If the condition is satisfied, a contingency for CG_{mi} will be attempted. In the event that CG_{mi} has no contingency or the contingency fails, recovery flow will reach the beginning of FG_m, necessitating the compensation of the remaining threads. Once all compensation is completed, a contingency for FG_m will be attempted. If that still fails, recovery will continue following the APCC semantics, compensating the outer scope of FG_m to find a contingent procedure in the outer scope.

The cases in (Friedman et al., 2010) need to be further analyzed for conformance to realistic scenarios. For example, in Figure 4.5, it may not be realistic for AP_{n1} and AP_{n11} to both exist in a process. In the AP model, because the recovery procedure is tightly associated with the APs enclosed in the process, an effective and efficient way of using APs in the presence parallelism is important. The proposed research will investigate rules for simplifying the use of APs with flow groups to reduce the complexity of the recovery actions associated with flow groups.

This research will also address APs in the presence of if-else and looping control structures. The if-else control structure defines two execution paths. Depending on the selection condition, only one path will be executed. This naturally makes the if-else control structure easy to address the recovery process. In the case of compensation, the process should only need to recover the path which has been executed. The loop control structure involves iterative execution. In BPEL, a scope associated with a compensation handler can be enclosed in a loop structure, such as a while activity. To compensate the completed while loop structure, the number of times that the associated compensation handler is invoked must be the same as the number of times of successfully completed scopes in the repeatable structure. Because the recovery procedure in the AP model is based on the presence of APs in a process, APs inside of a loop control structure can potentially complicate the recovery process. In this research, the necessity and complexity of inserting APs in the loop control structure will be investigated.

4.2 Formalization of the AP model

To precisely describe the execution semantics of the AP model, Petri Nets (Peterson, 1981) and YAWL (Van Der Aalst & Ter Hofstede, 2005) will be used to formalize the AP model in this research. Petri Nets can present a precise definition of the execution semantics of the AP model. YAWL is based on Petri Nets, but provides a more abstract representation. Section 4.2.1 presents examples of a portion of the AP model that has already been formalized by Petri Nets (Urban, Gao, Shrestha, Xiao, et al., 2010). Section 4.2.2 addresses the work that will be done with YAWL.

4.2.1 Execution Semantics of Assurance Points by Petri Nets

The initial execution semantics of the AP approach to service composition and recovery without if-else, looping, and parallel control structures have been defined in (Urban, Gao, Shrestha, Xiao, et al., 2010) by using Petri Nets (Peterson, 1981). A Petri Net is a directed, connected, and bipartite graph in which nodes represent places and transitions, and tokens occupy places. A directed arc in a Petri Net connects a place to a transition or a transition to a place. The places that have arcs running to a transition are called input places of the transition. The places that have arcs coming from a transition are called output places of the transition. A transition is enabled when each of its input places has at least one token. After firing a transition, exactly one token at each of its input places has been consumed, while one token at each of its output places has been generated.

4.2.1.1 General Approach

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

Before discussing the details of the Petri Net formalization, the notation used in the Petri Net diagrams is introduced. All transitions are labeled as T_n inside a transition node. Each place in a Petri Net has a short phrase beside the place node. Short phrases are used to label places due to limited room in the Petri Net graph. The complete set of all places that appear in the graphs that follow for atomic and composite execution groups are shown in Table A.1 in Appendix A, while Table A.2 in Appendix A indicates the places that are associated with graphs for APs. The left column of each table contains the short phrase of each place. The middle column contains the actual meaning of places. The right-most column indicates the type of the place, which is specified as status, condition, or resource.

4.2.1.2 AP Model Represented by Petri Nets

Figures 4.6 and 4.7 present the execution semantics of an atomic group and an assurance point respectively. In Figure 4.6, the normal atomic group invocation starts from place A and ends at either place S or places US and AP_CC. The end place S means execution success, while places US and AP_CC indicate the execution failure. Similarly, in Figure 4.7, a token at place A activates the AP and a token generated at place P indicates the successful pass of the AP. Because of numerous potential violations of pre/post condition, different recovery actions can be invoked through transitions T11 to T22.

The semantics of a composite group is shown in Figure 4.8. The execution of a

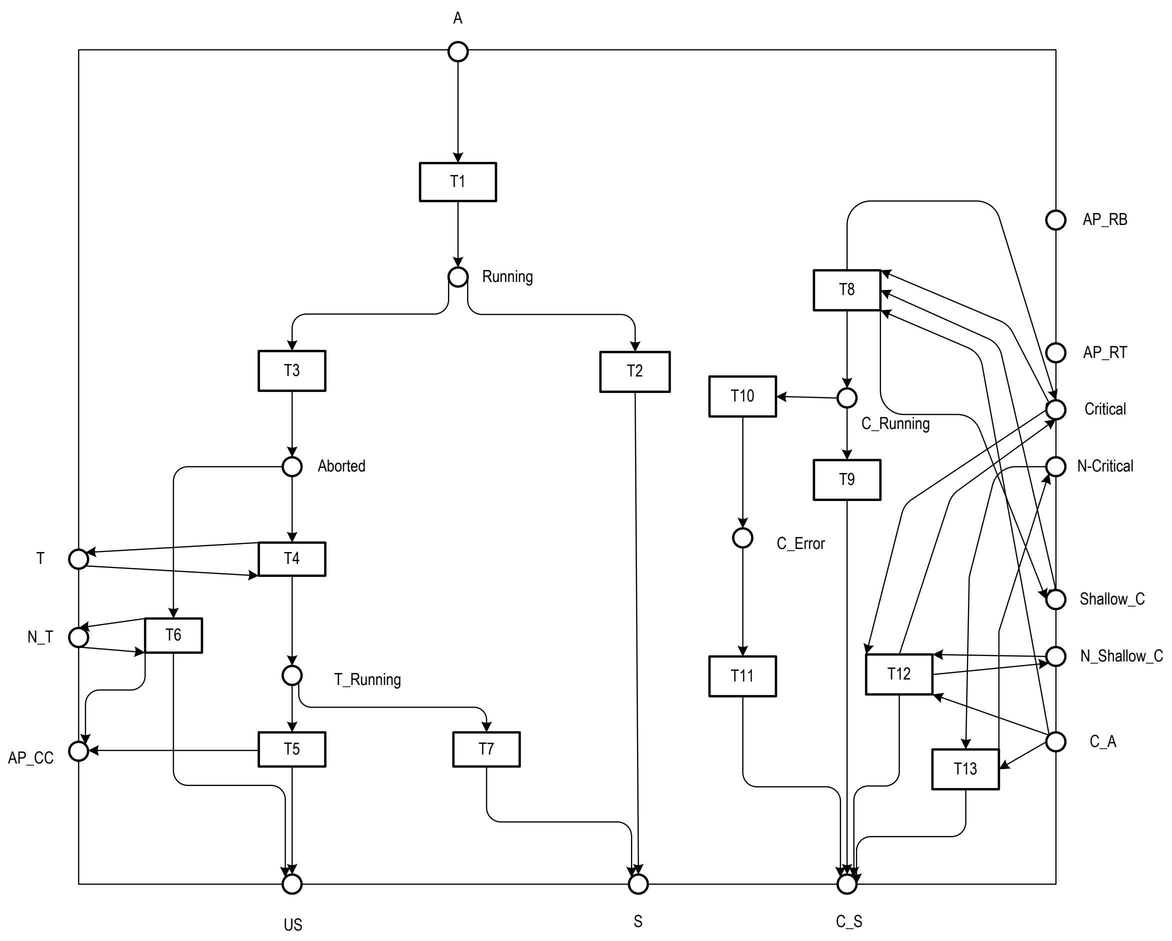


Figure 4.6. Semantics of Atomic Group
(Urban, Gao, Shrestha, Xiao, et al., 2010)

composite group involves multiple sub-groups and APs. In Figure 4.8, specifically, a dashed-line quadrilateral represents either an atomic or a composite group and a dashed-arc connecting a transition and a place represents repeating the same token movement pattern described at the current level.

The semantics of all three recovery actions defined in (Urban, Gao, Shrestha, & Courter, 2010) were also transformed into Petri Nets. For example, Figure 4.9 represents the semantics of the default APRetry action which recovers the process back to the most recent AP and checks the pre-condition before the re-execution. In Figure 4.9, when a status place AP_RT at an AP is marked, the APRetry mode is triggered.

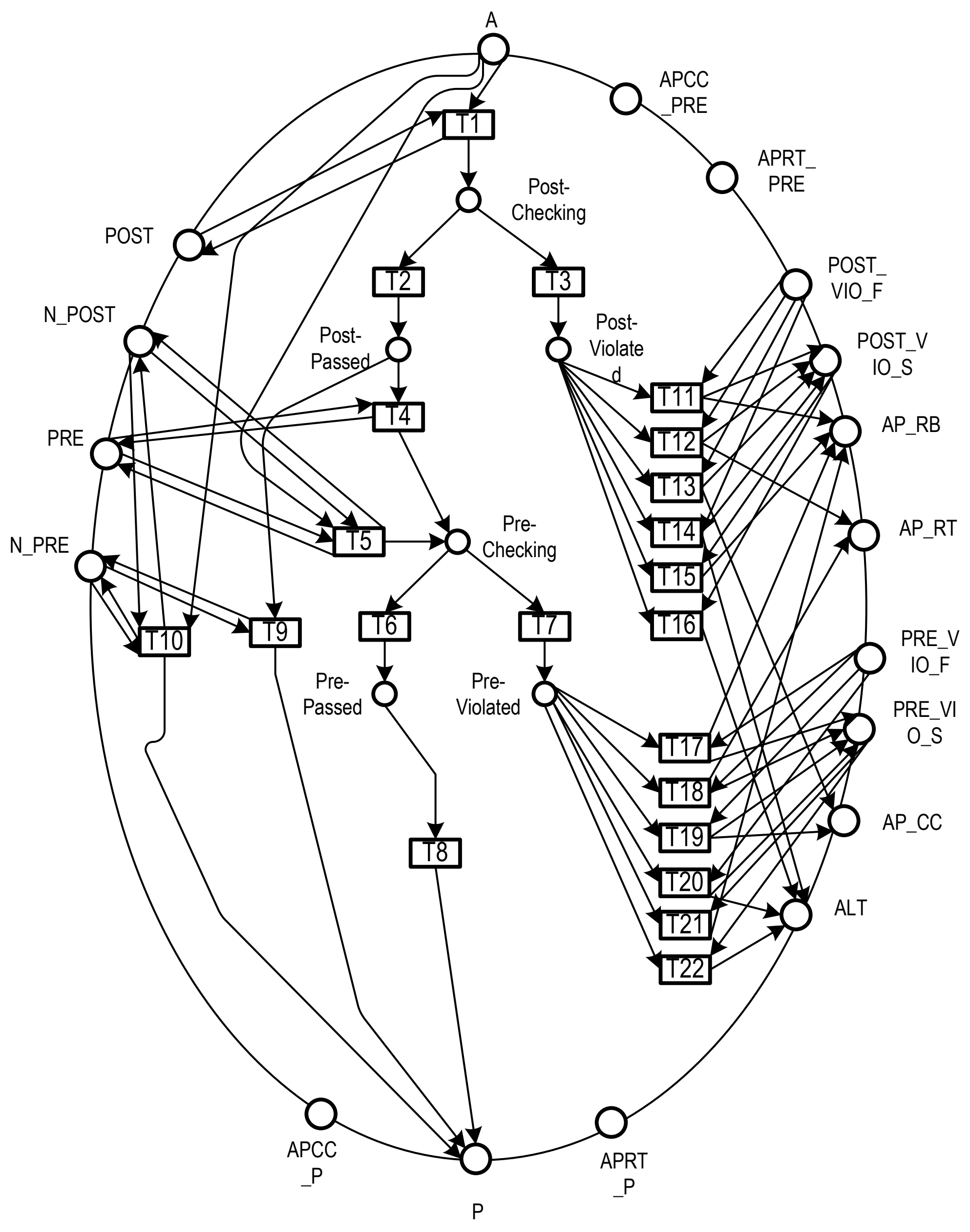


Figure 4.7. Semantics of Assurance Point
(Urban, Gao, Shrestha, Xiao, et al., 2010)

Then transition T1 fires to start the recovery. The status place AP_RT at a completed group is marked when the group is compensating. When the group just after the most recent AP is compensated, transition T3 fires and the place APRT_PRE at

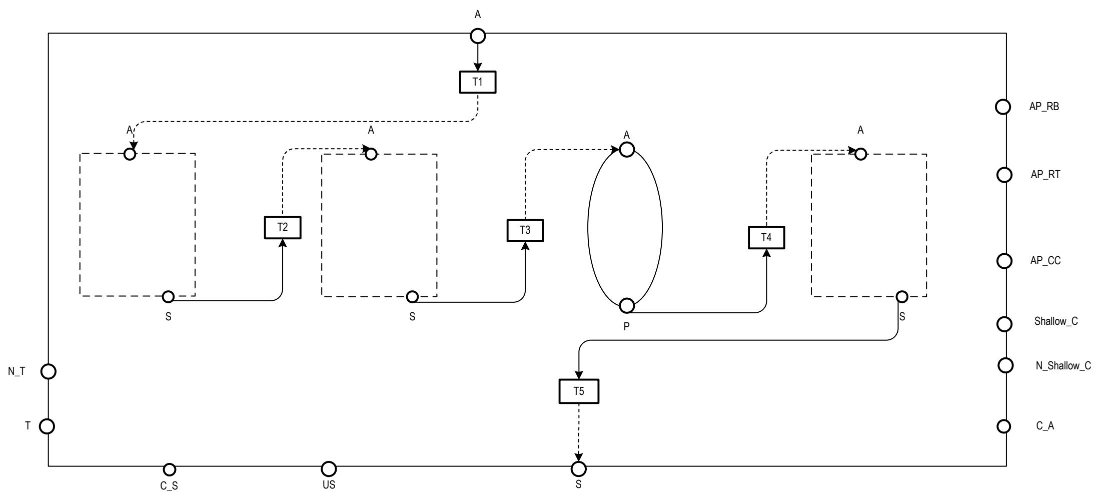


Figure 4.8. Semantics of Composite Group
(Urban, Gao, Shrestha, Xiao, et al., 2010)

the most recent AP is marked. Then the pre-condition defined at the most recent AP is re-checked. If the pre-condition is satisfied, the status place APRT_P is marked and transition T4 is enabled to start the retry process. If the pre-condition fails, another action will take place depending on the action specified in the rule.

Petri Nets can provide a graphical and formal representation of the execution semantics of the AP model. Besides the figures shown in Section 4.2.1.2, other diagrams developed in (Urban, Gao, Shrestha, Xiao, et al., 2010) for the current Petri Net specification of the AP model without if-else, looping, and parallel control structures are shown in Appendix A in Figures A.1-A.13 .

An obvious deficiency of Petri Nets is state explosion. Because there are many conditions and resources in the model, many places are needed to precisely define the semantics using regular Petri Nets, which makes the specification difficult to develop and understand.

4.2.2 Semantics of AP model by YAWL

In the proposed research, YAWL (Van Der Aalst & Ter Hofstede, 2005) will be adopted to improve the formalization of the AP model. In comparison with Petri Nets, YAWL has several merits. For example, YAWL does not have the rule that a place must exist between two transitions, which avoids state explosion.

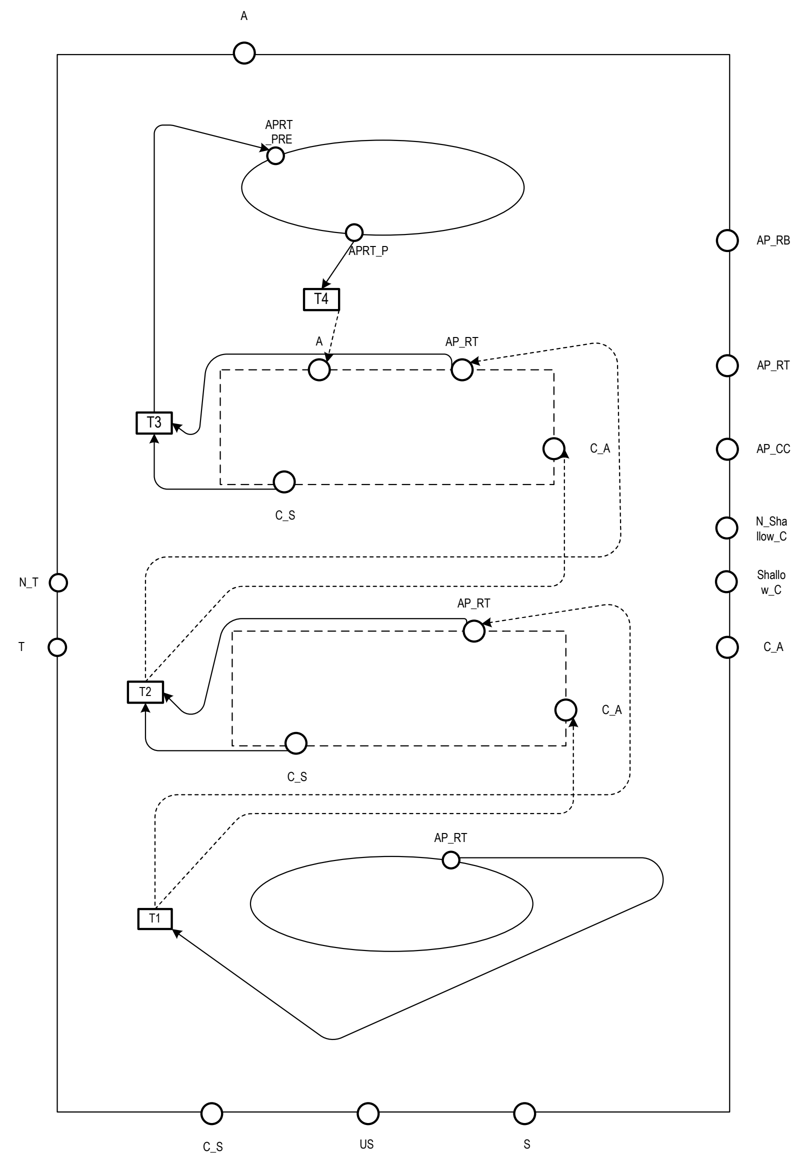


Figure 4.9. Semantics of default AP-Retry (Urban, Gao, Shrestha, Xiao, et al., 2010)

4.2.2.1 Introduction of YAWL

YAWL is inspired by Petri Nets, but it is not an extension of Petri Nets. YAWL has its own symbols and independent semantics. The symbols used in a YAWL net are shown in Table 4.1. Each YAWL net has one unique input and output condition. Similar to the AP model, a task is either an atomic task or a composite task. In the

YAWL net, six join and split constructs may be associated with tasks.

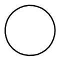
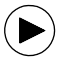
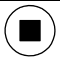
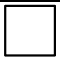
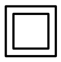
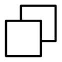
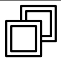
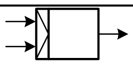
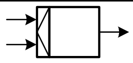
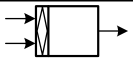
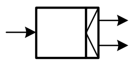
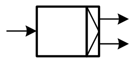
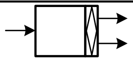
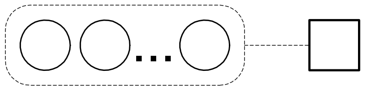
* AND-join - A task is invoked when all of the incoming arcs have been enabled.
* OR-join - A task is invoked when either (1) all of the incoming arcs have been enabled or (2) any incoming arcs that have not been enabled will never be enabled at any future time with the current marking continuing to be fired.
* XOR-join - A task is invoked when one of the incoming arcs has been enabled.
* AND-split - When a task completes, the thread of control is passed to all of the outgoing arcs.
* OR-split - When a task completes, the thread of control is passed to one or more of the outgoing arcs depending on the evaluations of the conditions associated with each arcs.
* XOR-split - When a task completes, the thread of control is passed to exactly one outgoing arc depending on the evaluations of the conditions associated with each arcs.

Additionally, YAWL introduces the notion of a cancellation region. A cancellation region is connected to a specific task in a YAWL net. When the task completes executing, all tasks currently executing in the corresponding cancellation region are withdrawn. Any tokens that reside in the corresponding cancellation region are also withdrawn.

4.2.2.2 Atomic Group Represented by YAWL

Figure 4.10 presents an example of a YAWL net to represent the execution semantics of an atomic group in the AP model. The atomic group fires task Running to execute. Depending on the condition evaluation, the thread of control may be passed to different branches. If the task Running succeeds, the token is passed to condition Successful. If the task Running fails and the contingency exists, task T_Running fires to execute the contingency. If the the task Running fails and no contingency exists, conditions Unsuccessful and APCC are marked. Similarly, if task T_Running fires,

Table 4.1. Symbols used in YAWL

Symbol	**Type**
	Condition
	Input condition
	Output condition
	Atomic task
	Composite task
	Multiple instances of an atomic task
	Multiple instances of a composite task
	AND-join task
	XOR-join task
	OR-join task
	AND-split task
	XOR-split task
	OR-split task
	Cancellation region

depending on the execution result, either condition Successful or conditions Unsuccessful and APCC are marked. The atomic group finalizes when condition Successful is marked or conditions Unsuccessful and APCC are marked.

The compensation semantics of an atomic group is formalized by YAWL as shown in Figure 4.11. After initializing of the compensation, different execution paths might be invoked depending on the condition evaluation. If the atomic group is non-critical,

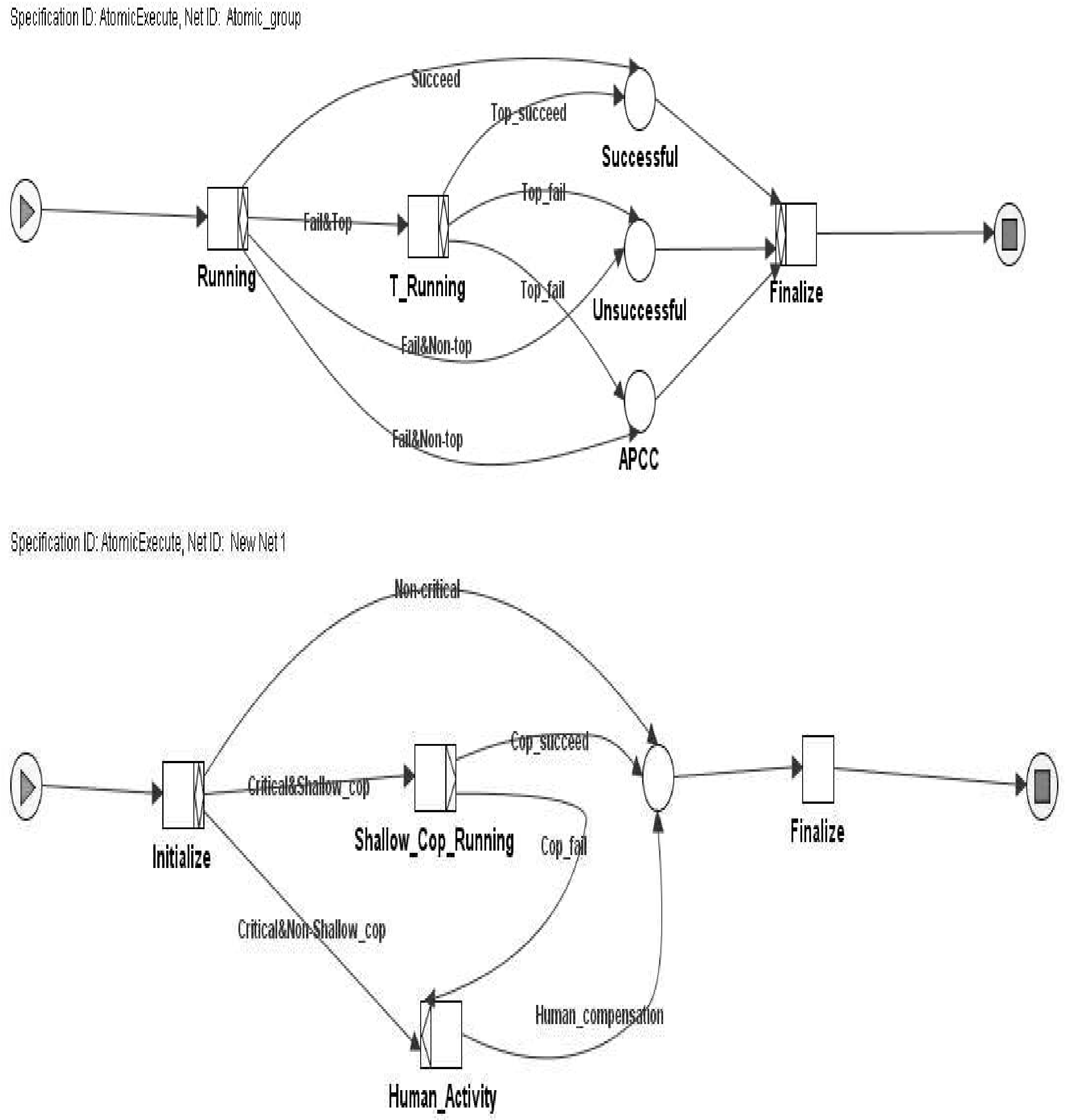


Figure 4.11. Compensation of Atomic Group by YAWL

the condition Cop_Successful is marked directly. If the atomic group is critical and the shallow compensation exists, the task Shallow_Cop_Running fires to execute the shallow compensation. If the atomic group is critical and no shallow compensation exists, the task Human_Activity fires to manually compensate the group. The task Shallow_Cop_Running also has two possible outputs. If the shallow compensation succeeds, the condition Cop_Successful is marked. If the shallow compensation fails, the task Human_Activity fires. The compensation of an atomic group finalizes when the condition Cop_Successful is marked.

In comparison with Petri Nets, YAWL provides a more visualized representation of the semantics in the AP model. YAWL also provides functionality to validate the semantics by checking dead lock and reachability in the net. In this research, YAWL will be used to formalize the complete AP model with if-else, looping, and parallel control structures. The analysis of YAWL nets for the AP model will focus on soundness property. A net is strictly sound if and only if the following requirements are met:

* All instances of the net must eventually terminate.
* There must be exactly one token at the end place when a instance terminates.
* Any tasks in the net may be executed in some instances.

Since the AP model supports three different recovery actions, any YAWL net that represents of the semantics of the AP model must terminate with an acceptable status.

4.3 Integration of the AP Model with Decentralized Data Dependency Analysis

The third component of this research is to integrate the decentralized data dependency analysis algorithm with the AP model. The integration to be addressed in this research will develop an intelligent event response environment that handles the data dependency notifications between PEXAs. By responding to different events with different actions, the impact caused by data dependencies between a failed process and other concurrent executing processes will be minimized.

As presented in Section 3.1.1, decentralized data dependency analysis raises new research challenges, such as discovering the *hidden dependencies* among concurrent processes that are accessing data that is distributed throughout the SOA. In decentralized data dependency analysis, process execution agents (PEXAs) are key components that are responsible for controlling the execution of processes, building local dependency graphs, and also sending and receiving notifications among PEXAs. However, the initial version of the algorithms in (Urban, Liu, & Gao, 2009; Z. Liu, 2009) operate in a naive manner and assume that all processes involved in the dependency graphs need to be fully recovered.

Figure 4.12 provides an example to illustrate the integration of the AP model and decentralized data dependency analysis. As shown in the figure, process P_1 and process P_2 are controlled by PEXA 1 and PEXA 2, respectively. Process P_1 is partial recovered due to the APRetry action invoked at AP2. Since the recovered operations, op_{12} and op_{13}, are both executed at PEXA 1, PEXA 1 is responsible for building the local data dependency graphs based on these two recovered operations. Unlike the approach in (Urban, Liu, & Gao, 2009; Z. Liu, 2009) that builds the data dependency graphs on a process level, decentralized data dependency analysis using the AP model requires that a PEXA builds the local data dependency graphs based on the partial recovery portion of a process, which means building the graphs at the operation level. As shown in Figure 4.12, two data dependency graphs on the operation level are formed in PEXA 1.

A major research challenge is the communication about data dependencies between PEXAs. If a data dependency graph at PEXA_1 contains an operation that is an external operation controlled by another PEXA, such as PEXA_2, a notification must be sent from PEXA_1 to PEXA_2. It is important, however, for PEXA_2 to respond to the data dependency notification in an intelligent way, rather than always doing a total rollback as in (Urban, Liu, & Gao, 2009; Z. Liu, 2009).

Recall that AERs provide a case-based structure to dynamically respond to interrupts caused by events external to the process. Different recovery actions are defined under different APs in an AER. Based on the most recently passed AP, a corresponding condition/action pair will be examined. This rule-based and event-driven technique of the AP model can be used to dynamically respond to any execution exceptions, including the data dependency interruptions. By using this event-driven technique, the data dependency communications between PEXAs can also be recognized as a external event. As a result, processes can design AERs that can respond to data dependency events caused by certain critical data items and ignore events associated with other less critical data items. Also with the help of the integration rules and AERs, PEXAs can respond to the external data dependency events with different actions based on the current process execution status or the constraint checking results of the process.

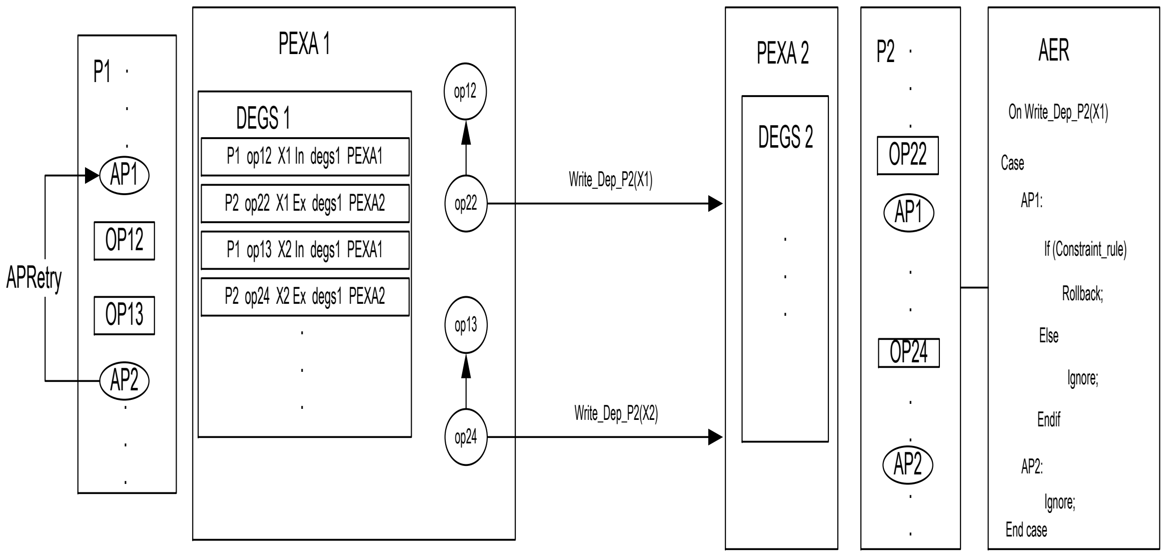


Figure 4.12. Dependent Event in Decentralized Data Dependency Analysis

In Figure 4.12, since operations op_{22} and op_{24} are external operations from PEXA 2, PEXA 1 must send the notifications to PEXA 2. The notifications are represented in event formats. For the event Write_Dep_P2(X1), since the process p_2 has an corresponding AER on this event, PEXA 2 might respond to it in different ways based on the execution status of p_2.

* if process p_2 has passed AP1 but not AP2

- if the constraint rule returns true, rollback of process p_2 will be performed. As a result, PEXA2 will start to build its own dependency graphs based on the rollback.
- if the constraint rule returns false, the event will be ignored.

* if process p_2 has passed AP2, the event will be ignored.

For the other event Write_Dep_P2(X2), since the process p_2 does not have an AER on this event, this event will be ignored since this data item is not critical to the successful execution of process p_2.

In summary, this research will develop a decentralized data dependency analysis approach that supports building data dependency graphs at the operation level based on partial recovery of a process. This approach will also define data dependency events as a communication method between PEXAs and use AERs to determine the necessity of responding to data dependency events. A simulation environment for the decentralized data dependency analysis approach will be developed to evaluate the performance and functionality of the system. The evaluation will focus on the efficiency of the algorithm with event-driven technique. Since AERs will be used to respond to data dependency events, a comparison of recovery complexity between approaches with and without AERs will be performed.

CHAPTER 5
DISSERTATION OUTLINE

Based on discussion of the research issues in previous sections, the dissertation will be structured as follows:

1) **Introduction**

   Present a general overview of the main research issues, the motivation for the research, and a roadmap to the rest of the dissertation.

2) **Related Work**

   Discuss the existing work in related areas, including advanced transaction models, transactional workflow, data consistency approaches for web service composition, dynamic modeling of business processes and failure recovery strategies for web services.

3) **Overview of the Assurance Point Model with Decentralized Data Dependency Analysis**

   Introduce the global view of the decentralized data dependency project, including the concepts of APs, IRs, AERs and PEXAs.

4) **The Assurance Point Web Service Model**

   Present the specification of the AP model in the context of programming control structures, including the placement of APs, execution and recovery semantics with the use of APs, IRs, and AERs.

5) **Formalization of the Assurance Point Model**

   Present the formalization of execution and recovery semantics of the AP model by using YAWL, including the graphical specifications and the analysis based on the graphical specifications.

6) **Integration of the Assurance Point Model with Decentralized Data Dependency Analysis**

Present the integration of decentralized data dependency analysis with AP Model, including the algorithms of building data dependency graph at operation level and using AERs to respond to data dependency events.

7) **Demonstration and Evaluation of Decentralized Data Dependency Analysis with Assurance Point Model**

   Present the prototype of decentralized data dependency analysis with AP Model. Performance and complexity issues will also be addressed.

8) **Summary and Future Work**

   Summarize the results of this research, highlight the main contributions of this research, and discuss future directions based on the existing research result.

The timetable of the research is presented in the Table 5.1.

Table 5.1. Tentative Timeline of this Research

Task Number	**Task**	**Start Date**	**End Date**
1	Investigate the AP model with if-else, parallel and loop control structures	Aug-2011	Oct-2011
2	Formalize and analyze the semantics of AP model by using YAWL	Oct-2011	Nov-2011
3	Integrate decentralized data dependency analysis with AP model	Nov-2011	Jan-2012
4	Prototype and test decentralized data dependency analysis with AP model	Jan-2012	Mar-2012
5	Dissertation Writing	Mar-2012	May-2012
6	Dissertation Defense	Jun-2012	Jun-2012

CHAPTER 6
SUMMARY

This proposal has presented plans for the development of a robust web service composition model with decentralized data dependency analysis and rule-based failure recovery capability. This research is expected to define a new paradigm for service execution that uses rule-based techniques for testing user-defined semantic conditions. With the AP model, a dynamic and intelligent approach can be used to monitor failures, detect data dependencies, and respond to failures and exceptional events. By integrating rule-based and event-driven techniques into web service composition, process failures and exceptions can be maximumly forward recovered. Past work on transactional workflows is inadequate for service composition since most techniques that support relaxed isolation do not actively address the impact that the failure and recovery of one process can have on other data dependent processes. The integration of decentralized data dependency analysis with the AP model will not only provide a way to detect the data dependency, but also minimize the impact caused by data dependencies between a failed process and other concurrent executing processes.

References

Alves, A., Arkin, A., Askary, S., Barreto, C., Bloch, B., Curbera, F., et al. (2007). Web services business process execution language version 2.0. *OASIS Standard http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html, 11.*

Anderson, T., & Lee, P. (1981). *Fault tolerance: principles and practice.* Prentice/Hall International.

Andrews, T., Curbera, F., Dholakia, H., Goland, Y., Klein, J., Leymann, F., et al. (2003). Business process execution language for web services, version 1.1. *Standards proposal by BEA Systems, International Business Machines Corporation, and Microsoft Corporation.*

Baresi, L., Guinea, S., & Pasquale, L. (2007). Self-healing BPEL processes with Dynamo and the JBoss rule engine. In *International workshop on engineering of software services for pervasive environments: in conjunction with the 6th esec/fse joint meeting* (pp. 11–20).

Blake, L. (2005). *Design and Implementation of Delta-Enabled Grid Services.* Unpublished master's thesis, MS Thesis, Deptment of Computer Science and Engineering, Arizona State Univ.

Booch, G., Rumbaugh, J., & Jacobson, I. (2005). *Unified Modeling Language User Guide, The (Addison-Wesley Object Technology Series).* Addison-Wesley Professional.

Bpmn, O. (2009). BPMN 2.0 beta 1. *http://www.omg.org/cgi-bin/doc?dtc/09-08-14.pdf.*

Cabrera, F., Copeland, G., Cox, B., Freund, T., Klein, J., Storey, T., et al. (2002). Web services transaction (WS-transaction). *joint specification by BEA, IBM, and Microsoft.*

Cabrera, F., Copeland, G., Freund, T., Klein, J., Langworthy, D., Orchard, D., et al. (2002). Web services coordination (WS-Coordination). *joint specification by BEA, IBM, and Microsoft.*

Cabrera, L., Copeland, G., Feingold, M., Freund, R., Freund, T., Joyce, S., et al. (2005). Web services business activity framework (ws-businessactivity). *joint specification by BEA, IBM, and Microsoft.*

Ceri, S., Grefen, P., & Sanchez, G. (1997). WIDE-a distributed architecture for workflow management. In *proceedings of 7th int. workshop on research issues in data engineering* (pp. 76–79).

Chafle, G., Chandra, S., Kankar, P., & Mann, V. (2005). Handling faults in decentralized orchestration of composite web services. In (pp. 410–423). Springer.

Charfi, A., & Mezini, M. (2006). Aspect-oriented workflow languages. *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE*, 183–200.

Charfi, A., & Mezini, M. (2007). Ao4bpel: An aspect-oriented extension to bpel. *World Wide Web*, *10*(3), 309–344.

Chiu, D., Li, Q., & Karlapalem, K. (1999). Exception handling with workflow evolution in ADOME-WFMS: a taxonomy and resolution techniques. *ACM Siggroup Bulletin*, *20*(3), 8.

Chiu, D., Li, Q., & Karlapalem, K. (2000). Facilitating exception handling with recovery techniques in ADOME workflow management system. *Journal of Applied Systems Studies*, *1*(3), 467–488.

Cichocki, A. (1998). *Workflow and process automation: concepts and technology.* Kluwer Academic Pub.

Cichocki, A., & Rusinkiewicz, M. (1998). Migrating workflows. *NATO ASI series. Series F: computer and system sciences*, 339–355.

Courter, A. (2010). *Supporting Data Consistency in Concurrent Process Execution with Assurance Points and Invariants.* Unpublished master's thesis, Master's Thesis, Texas Tech University.

Dayal, U., Hsu, M., & Ladin, R. (1991). A transactional model for long-running activities. In *Proceedings of the 17th international conference on very large data bases* (pp. 113–122).

Dialani, V., Miles, S., Moreau, L., De Roure, D., & Luck, M. (2002). Transparent fault tolerance for web services based architectures. *Euro-Par 2002 Parallel Processing*, 107–201.

Doğaç, A. (1998). *Workflow management systems and interoperability.* Springer Verlag.

Eder, J., & Liebhart, W. (1995). The workflow activity model WAMO. In *Proc. 3rd international conference on cooperative information systems,vienna.*

Elmagarmid, A. (1992). *Database transaction models for advanced applications.* Morgan Kaufmann.

Elmagarmid, A., Leu, Y., Litwin, W., & Rusinkiewicz, M. (1990). A multidatabase transaction model for interbase. In *Proceedings of the 16th international conference on very large data bases* (pp. 507–518).

Engels, G., Förster, A., Heckel, R., & Thöne, S. (2005). Process modeling using UML. *Process Aware Information Systems: Bridging People and Software Through Process Technology*, 85–118.

Friedman, Z., Urban, S., Gao, L., & Shrestha, R. (2010). *Extending the Assurance Point Approach to Process Recovery for Use With Flow Groups* (Tech. Rep.). Texas Tech University.

Garcia-Molina, H., & Salem, K. (1987). Sagas. *ACM SIGMOD Record*, *16*(3), 249–259.

Greenfield, P., Fekete, A., Jang, J., & Kuo, D. (2003). Compensation is not enough [fault-handling and compensation mechanism]. In *Enterprise distributed object computing conference, 2003. proceedings. seventh ieee international* (pp. 232–239).

Hagen, C., & Alonso, G. (2002). Exception handling in workflow management systems. *Software Engineering, IEEE Transactions on*, *26*(10), 943–958.

Halvorsen, O., & Haugen, O. (2006). Proposed notation for exception handling in UML 2 sequence diagrams. In *Software engineering conference, 2006. australian* (p. 10).

Herbst, H., Knolmayer, G., Myrach, T., & Schlesinger, M. (1994). The specification of business rules: A comparison of selected methodologies. In *Proceedings of the ifip wg8* (Vol. 1, pp. 29–46).

Jang, J., Fekete, A., & Greenfield, P. (2007). Delivering Promises for Web Services Applications. In *Ieee international conference on web services, salt lake city, utah, usa* (pp. 599–606). IEEE Computer Society.

Jean, D., Cichock, A., & Rusinkiewicz, M. (1996). A database environment for workflow specification and execution. In *Proc. intl symposium on cooperative database systems kyoto.*

Jennings, N., Faratin, P., Norman, T., Odgers, B., & Alty, J. (2000). Implementing a business process management system using ADEPT: A real-world case study. *Applied Artificial Intelligence*, *14*(5), 421–463.

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

Jensen, K. (1997). *Coloured Petri nets: basic concepts, analysis methods, and practical use*. Springer.

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*.

Kamath, M., & Ramamritham, K. (1998). Failure handling and coordinated execution of concurrent workflows. In *Proceedings of the international conference on data engineering* (pp. 334–341).

Kantere, V., Kiringa, I., Mylopoulos, J., Kementsietsidis, A., & Arenas, M. (2004). Coordinating peer databases using ECA rules. *Databases, Information Systems, and Peer-to-Peer Computing*, 108–122.

Kappel, G., Proll, B., Rausch-Schott, S., & Retschitzegger, W. (1995). TriGS/sub flow: Active object-oriented workflow management. In *Proc. of hicss* (p. 727).

Karnath, M., & Ramamritham, K. (1998). Failure handling and coordinated execution of concurrent workflows. In *Data engineering,. proceedings., 14th international conference on* (pp. 334–341).

Khalaf, R., Roller, D., & Leymann, F. (2009). Revisiting the behavior of Fault and Compensation handlers in WS-BPEL. In (pp. 286–303). Springer.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., & Griswold, W. (2001). An overview of AspectJ. In (pp. 327–354). Springer.

Kiepuszewski, B., Muhlberger, R., & Orlowska, M. (1998). FlowBack: providing backward recovery for workflow management systems. *ACM SIGMOD Record*, *27*(2), 555–557.

Lakhal, N., Kobayashi, T., & Yokota, H. (2006). Dependability and flexibility centered approach for composite web services modeling. *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE*, 163–182.

Liu, L., & Pu, C. (1997). Activity flow: Towards incremental specification and flexible coordination of workflow activities. *Conceptual ModelingER'97*, 169–182.

Liu, Z. (2009). *Decentralized Data Dependency Analysis for Concurrent Process Execution*. Unpublished master's thesis, MS Thesis, Texas Tech University.

Lu, R., & Sadiq, S. (2007). A survey of comparative business process modeling approaches. In *Business information systems* (pp. 82–94).

Luo, Z. (2000). Checkpointing for workflow recovery. In *Proceedings of the 38th annual on southeast regional conference* (pp. 79–80).

Marzouk, S., Maalej, A., Rodriguez, I., & Jmaiel, M. (2009). Periodic checkpointing for strong mobility of orchestrated web services. In *2009 congress on services-i* (pp. 203–210).

Mendling, J., Neumann, G., & Nüttgens, M. (2005). Yet another event-driven process chain. *Business Process Management*, 428–433.

Mikalsen, T., Tai, S., & Rouvellou, I. (2002). Transactional attitudes: Reliable composition of autonomous Web services. In *Dsn 2002, workshop on dependable middleware-based systems (wdms).*

Modafferi, S., & Conforti, E. (2006). Methods for enabling recovery actions in WS-BPEL. *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE*, 219–236.

Modafferi, S., Mussi, E., & Pernici, B. (2006). SH-BPEL: a self-healing plug-in for Ws-BPEL engines. In *Proceedings of the 1st workshop on middleware for service oriented computing (mw4soc 2006)* (pp. 48–53).

Müller, R., Greiner, U., & Rahm, E. (2004). AW: a workflow system supporting rule-based workflow adaptation. *Data & Knowledge Engineering*, *51*(2), 223–256.

Ouyang, C., Dumas, M., Aalst, W., Hofstede, A., & Mendling, J. (2009). From business process models to process-oriented software systems. *ACM transactions on software engineering and methodology (TOSEM)*, *19*(1), 2.

Peltz, C. (2003). Web services orchestration and choreography. *Computer*, 46–52.

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

Petri, C. (1966). *Communication With Automata: Volume 1 Supplement 1* (Tech. Rep.). Applied Data Research Inc Princeton NJ.

Pintér, G., & Majzik, I. (2005). Modeling and analysis of exception handling by using UML statecharts. *Scientific Engineering of Distributed Java Applications*, 58–

67.

Ramachandran, J. (2011). *Integrating Exception Handling and Data Dependency Analysis through Application Exception Rules.* Unpublished master's thesis, Master's Thesis, Texas Tech University.

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

Reichert, M., & Dadam, P. (1998). ADEPT flexsupporting dynamic changes of workflows without losing control. *Journal of Intelligent Information Systems*, *10*(2), 93–129.

Rolf, A., Klas, W., & Veijalainen, J. (1997). *Transaction management support for cooperative applications.* Kluwer Academic Pub.

Sadiq, W., & Orlowska, M. (1999). On capturing process requirements of workflow based business information systems. In *Proceedings of the 3rd international conference on business information systems (bis99).*

Scheer, A., Thomas, O., & Adam, O. (2005). Process modeling using event-driven process chains. *Process-aware information systems: bridging people and software through process technology*, 119–145.

Singh, M., & Huhns, M. (2005). *Service-oriented computing: semantics, processes, agents.* John Wiley & Sons Inc.

Sycara, K., Paolucci, M., Ankolekar, A., & Srinivasan, N. (2003). Automated discovery, interaction and composition of semantic web services. *Journal of Web Semantics*, *1*(1), 27–46.

Tan, W., Fong, L., & Bobroff, N. (2010). Bpel4job: a fault-handling design for job flow management. *Service-Oriented Computing–ICSOC 2007*, 27–42.

Urban, S., Dietrich, S., Na, Y., Jin, Y., Saxena, S., Urban, S., et al. (2001). The irules project: using active rules for the integration of distributed software components. In *Proceedings of the 9th ifip 2.6 working conference on database semantics: Semantic issues in e-commerce systems, hong kong* (pp. 265–286).

Urban, S., Gao, L., Shrestha, R., & Courter, A. (2010). Achieving Recovery in Service Composition with Assurance Points and Integration Rules. *On the Move to Meaningful Internet Systems: OTM 2010*, 428–437.

Urban, S., Gao, L., Shrestha, R., & Courter, A. (2011). The dynamics of process modeling: new directions for the use of events and rules in service-oriented computing. *The evolution of conceptual modeling*, 205–224.

Urban, S., Gao, L., Shrestha, R., Xiao, Y., Friedman, Z., & Rodriguez, J. (2010). *The Assurance Point Model for Consistency and Recovery in Service Composition.* Book Chapter Under Review.

Urban, S., Liu, Z., & Gao, L. (2009). Decentralized data dependency analysis for concurrent process execution. In *Enterprise distributed object computing conference workshops, 2009. edocw 2009. 13th* (pp. 74–83).

Urban, S., Xiao, Y., Blake, L., & Dietrich, S. (2009). Monitoring data dependencies in concurrent process execution through delta-enabled grid services. *International Journal of Web and Grid Services*, *5*(1), 85–106.

Vaculín, R., Wiesner, K., & Sycara, K. (2008). Exception handling and recovery of semantic web services. In *Networking and services, 2008. icns 2008. fourth international conference on* (pp. 217–222).

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

Van Der Aalst, W., & Ter Hofstede, A. (2005). YAWL: yet another workflow language. *Information Systems*, *30*(4), 245–275.

Wächter, H., & Reuter, A. (1991). *The contract model.* Universität, Fakultät Informatik.

Weikum, G. (1991). Principles and realization strategies of multilevel transaction management. *ACM Transactions on Database Systems (TODS)*, *16*(1), 132–180.

White, S., et al. (2004). Business Process Modeling Notation (BPMN) Version 1.0. *Business Process Management Initiative, BPMI. org.*

Widom, J., & Ceri, S. (1996). *Active database systems: Triggers and rules for advanced database processing.* Morgan Kaufmann Pub.

Wiesner, K., Vaculín, R., Kollingbaum, M., & Sycara, K. (2008). Recovery mechanisms for semantic web services. In *Distributed applications and interoperable systems* (pp. 100–105).

Wodtke, D., Weißenfels, J., Weikum, G., & Dittrich, A. (1996). The Mentor project: Steps towards enterprise-wide workflow management. In *Proceedings of the* *twelfth international conference on data engineering* (pp. 556–565).

Worah, D., & Sheth, A. (1997). Transactions in transactional workflows. In *Transactions in transactional workflows* (pp. 3–34).

Xiao, Y. (2006). *Using deltas to analyze data dependencies and semantic correctness in the recovery of concurrent process execution.* Unpublished doctoral dissertation, Ph.D Dissertation, Arizona State University Tempe, AZ, USA.

Xiao, Y., & Urban, S. (2007). Process dependencies and process interference rules for analyzing the impact of failure in a service composition environment. In *Business information systems* (pp. 67–81).

Xiao, Y., & Urban, S. (2008). Using Data Dependencies to Support the Recovery of Concurrent Processes in a Service Composition Environment. In *Proceedings of the cooperative information systems conference (coopis), monterrey, mexico* (pp. 139–156).

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

Xiao, Y., Urban, S., & Dietrich, S. (2006). A process history capture system for analysis of data dependencies in concurrent process execution. *Data Engineering Issues in E-Commerce and Services*, 152–166.

Xiao, Y., Urban, S., & Liao, N. (2006). The DeltaGrid abstract execution model: service composition and process interference handling. *Conceptual Modeling-ER 2006*, 40–53.

Zhang, X., Zagorodnov, D., Hiltunen, M., Marzullo, K., & Schlichting, R. (2004). Fault-tolerant grid services using primary-backup: feasibility and performance. In *cluster* (pp. 105–114).

Zhao, W., Moser, L., & Melliar-Smith, P. (2005). A reservation-based coordination protocol for Web Services. In *Ieee international conference on web services, orlando, florida , usa* (pp. 49–56). Published by the IEEE Computer Society.

APPENDIX A
FORMALIZATION USING PETRI NETS

Table A.1. Places in an Execution Group Petri Net

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

Table A.2. Places in an AP Petri Net

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

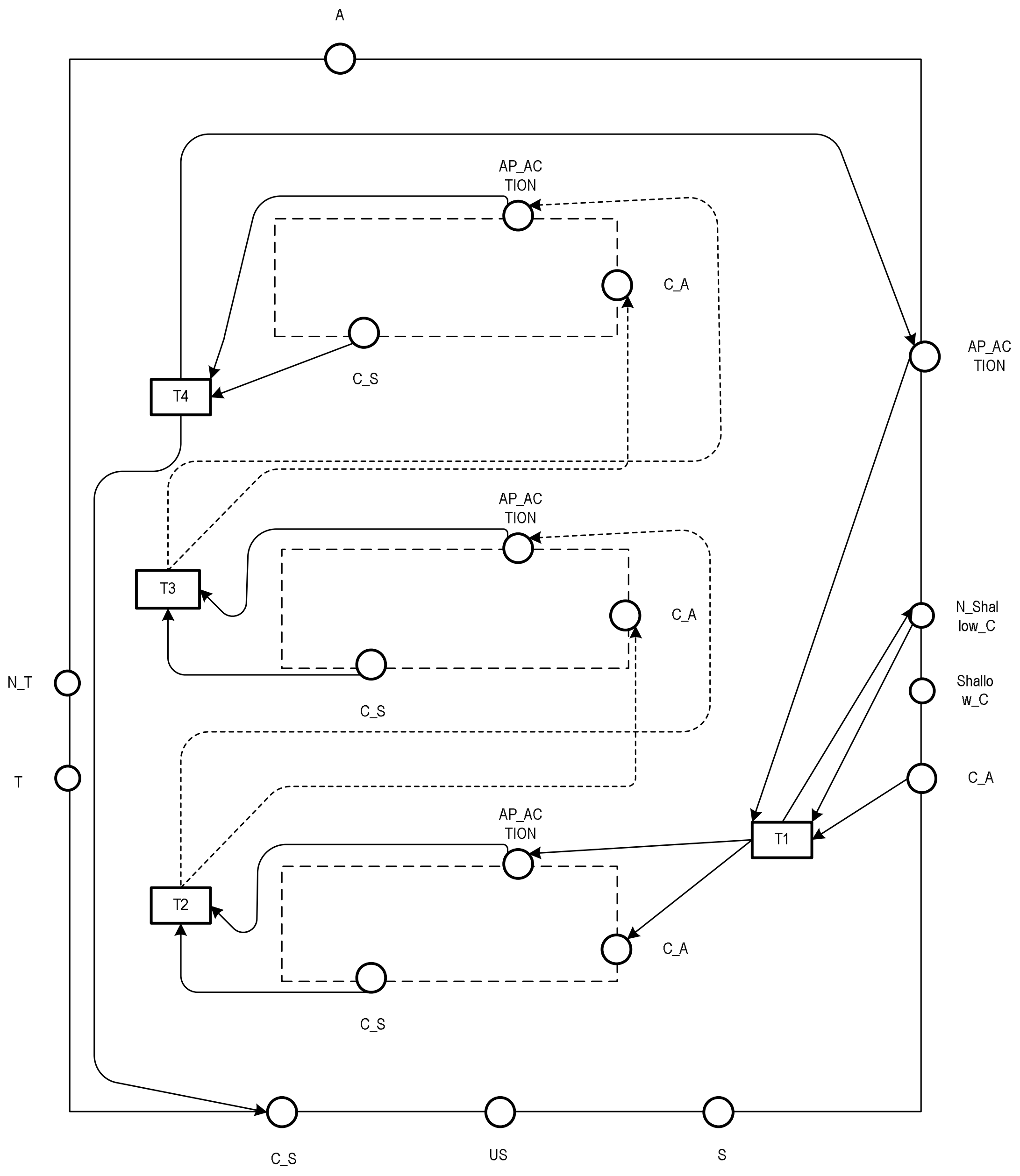


Figure A.1. Petri Net of Deep Compensation

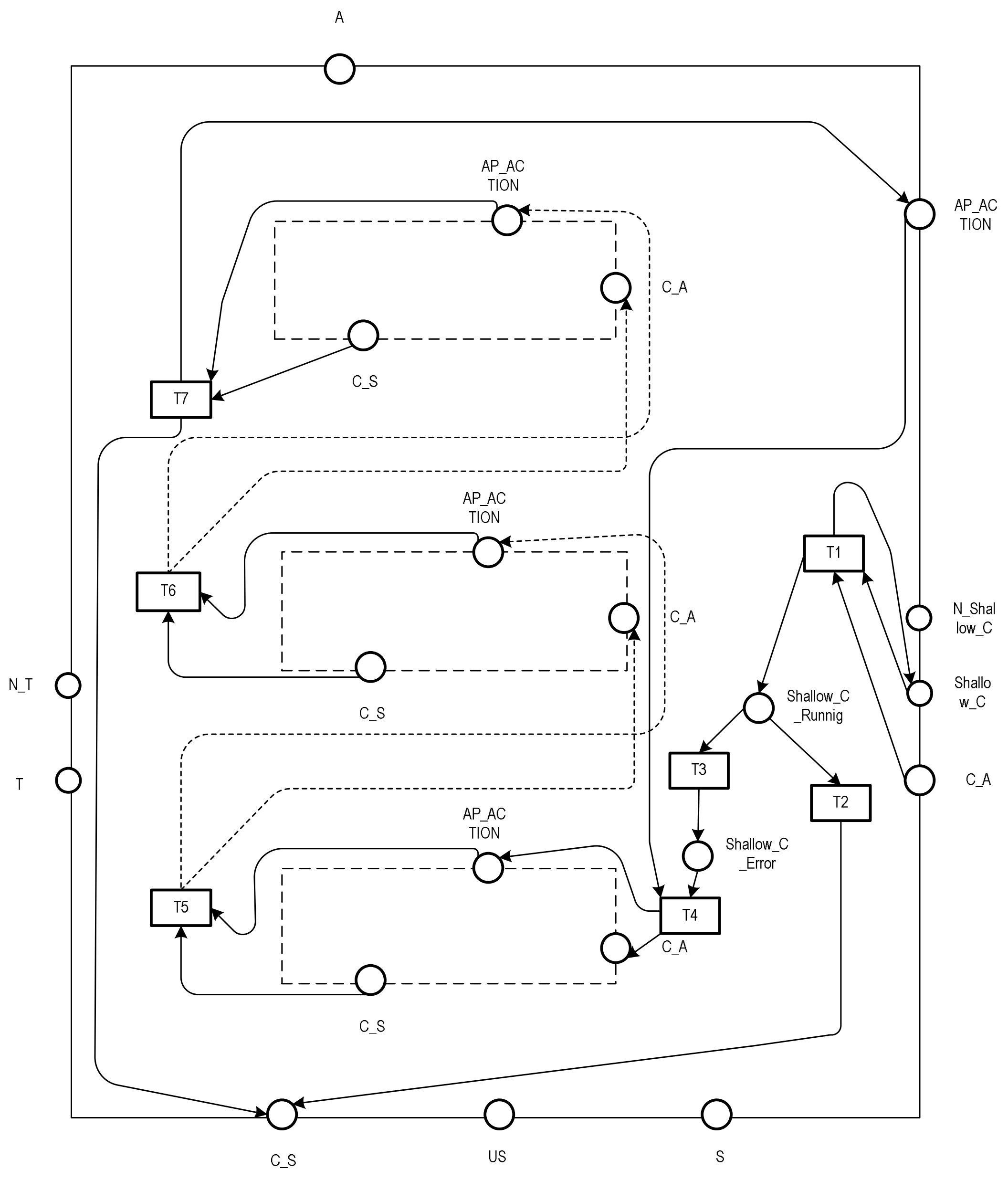


Figure A.2. Petri Net of Shallow Compensation

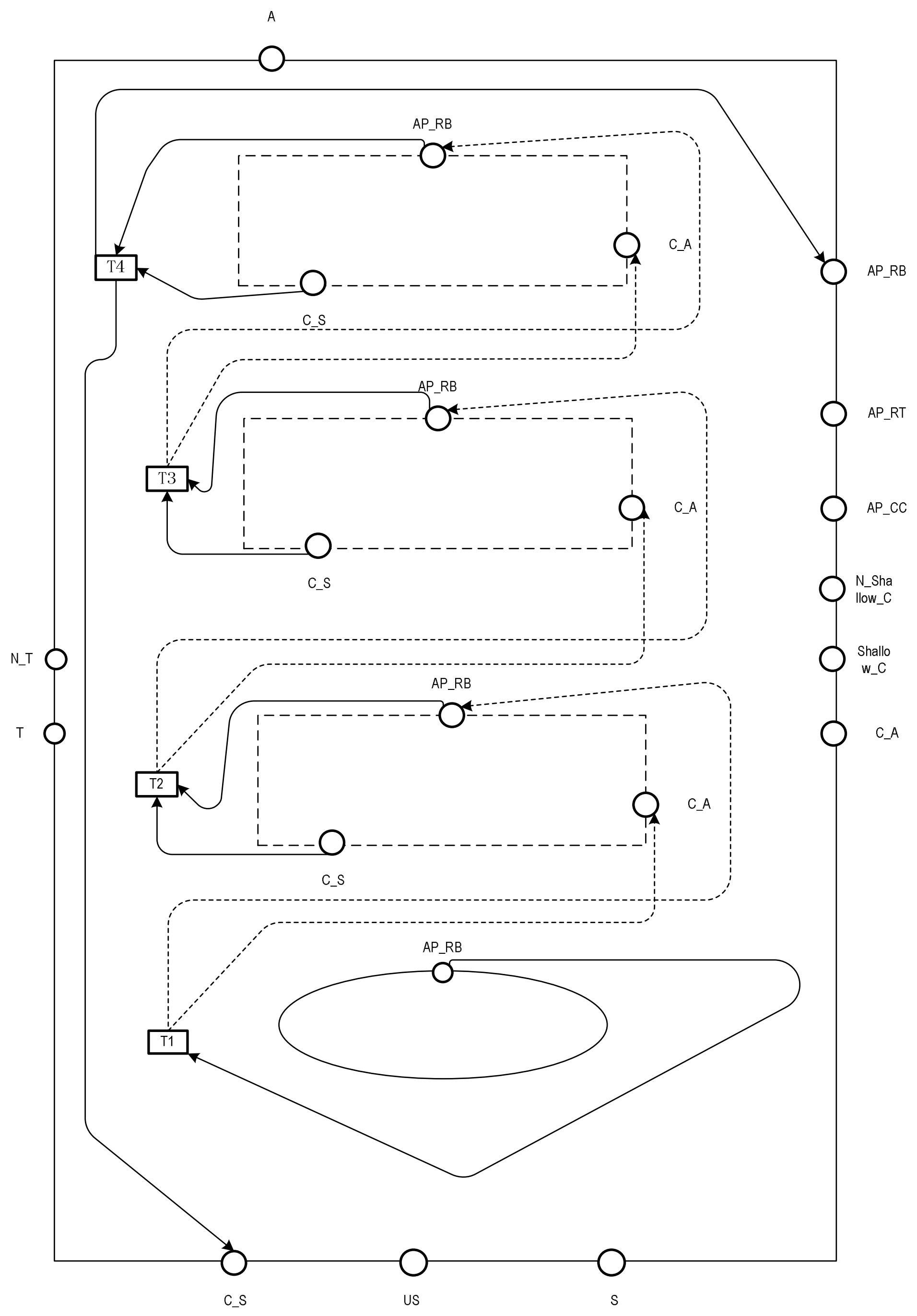


Figure A.3. Petri Net of APRollback

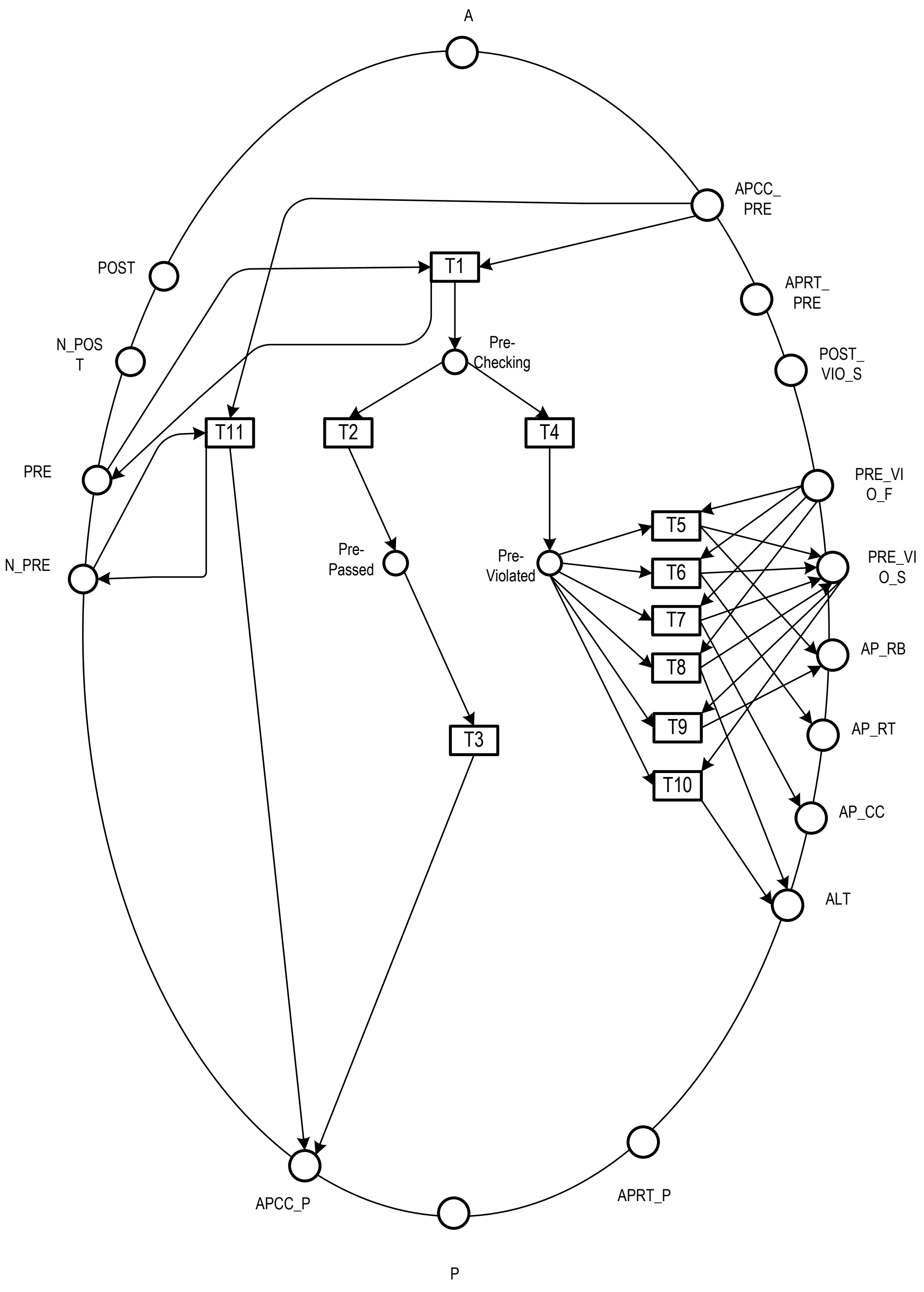


Figure A.4. Petri Net of Re-checking Pre-condition (APCC)

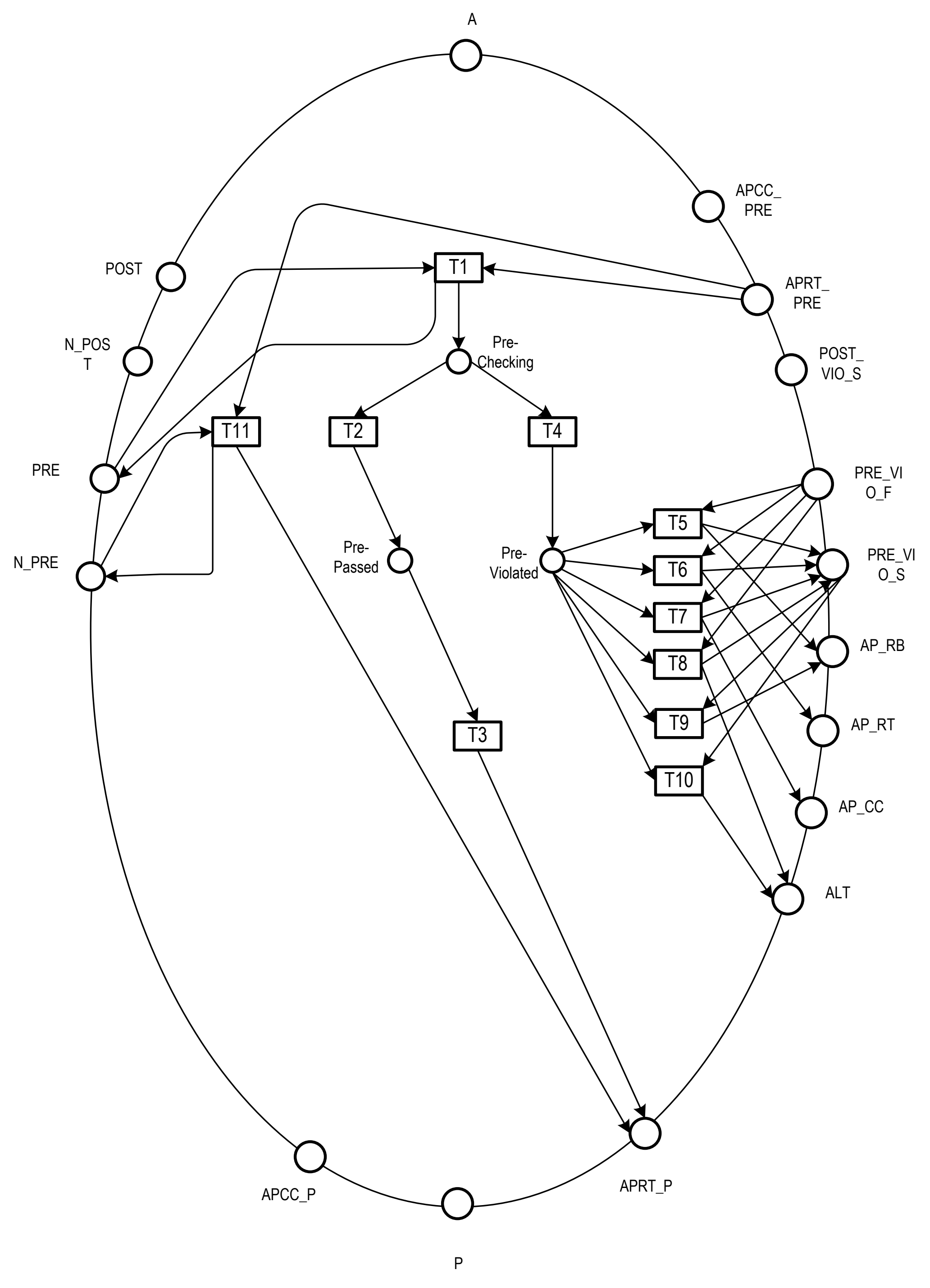


Figure A.5. Petri Net of Re-checking Pre-condition (APRT)

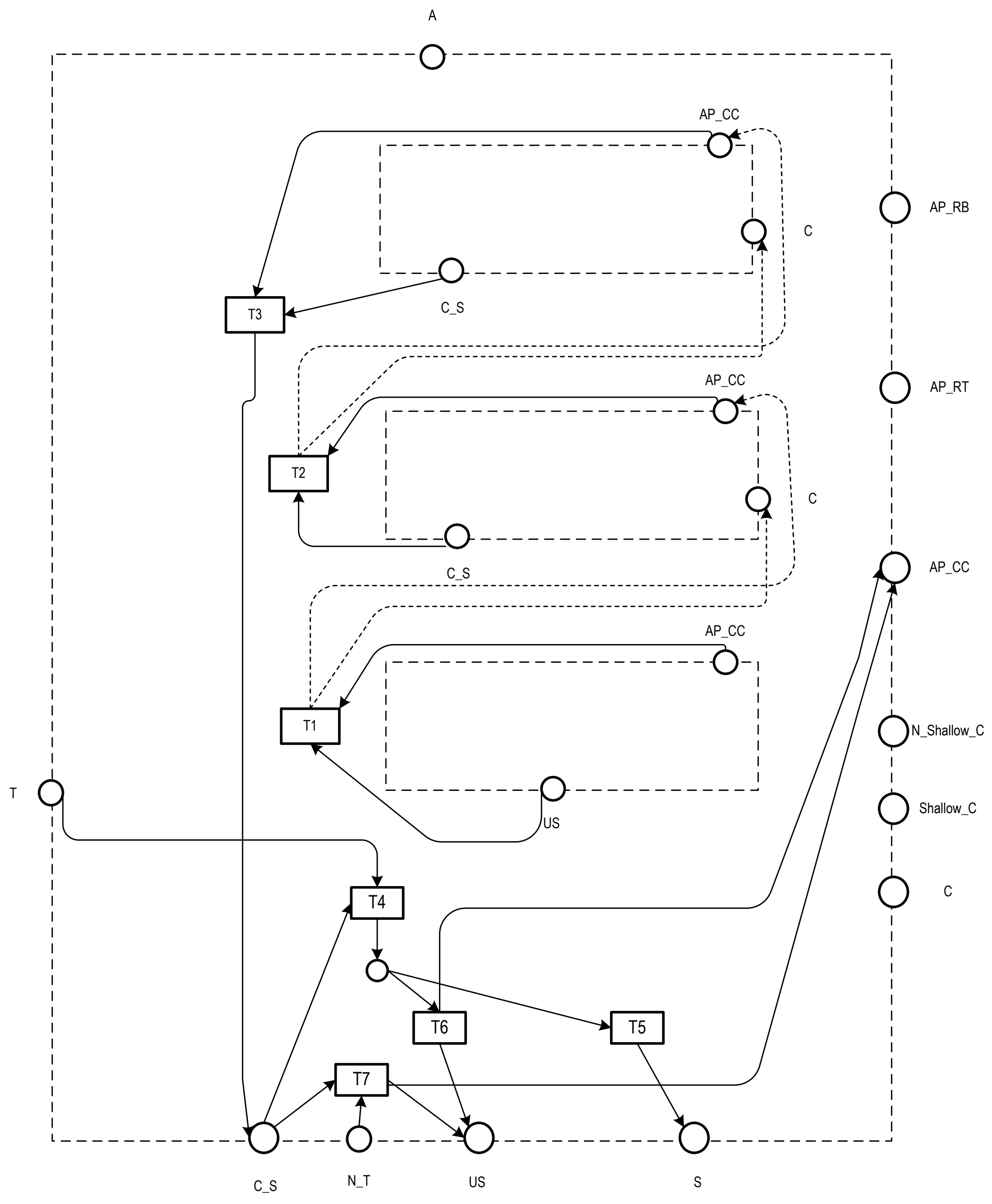


Figure A.6. Petri Net of APCC (1)

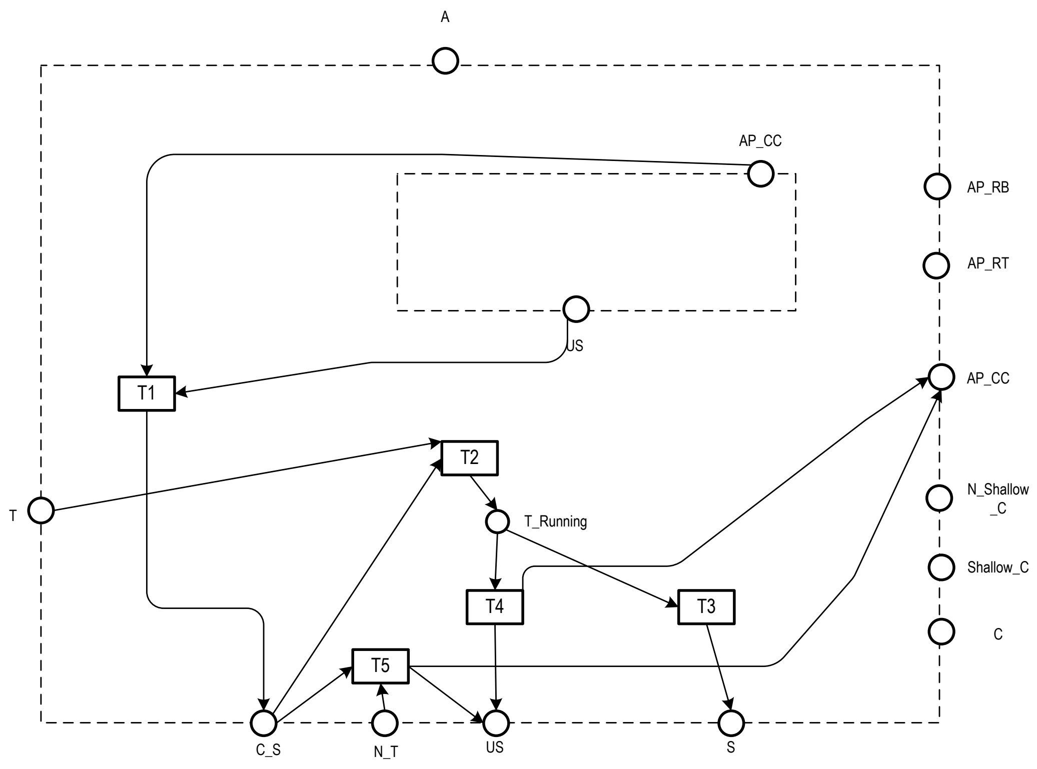


Figure A.7. Petri Net of APCC (2)

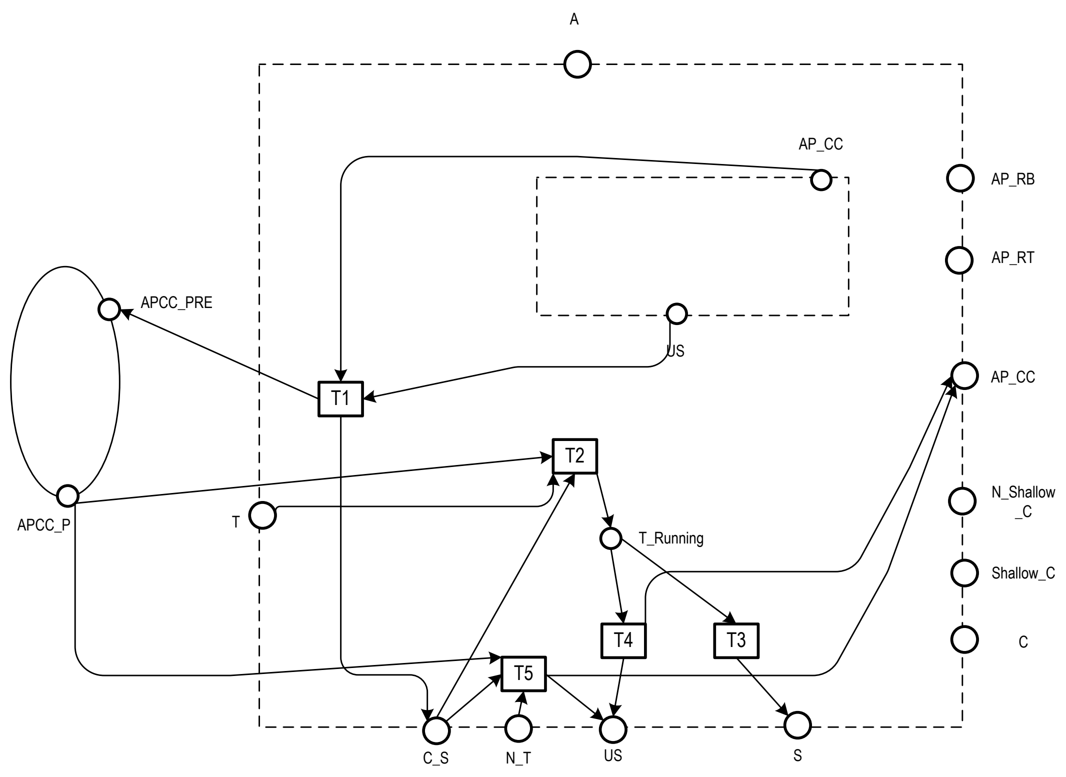


Figure A.8. Petri Net of APCC (3)

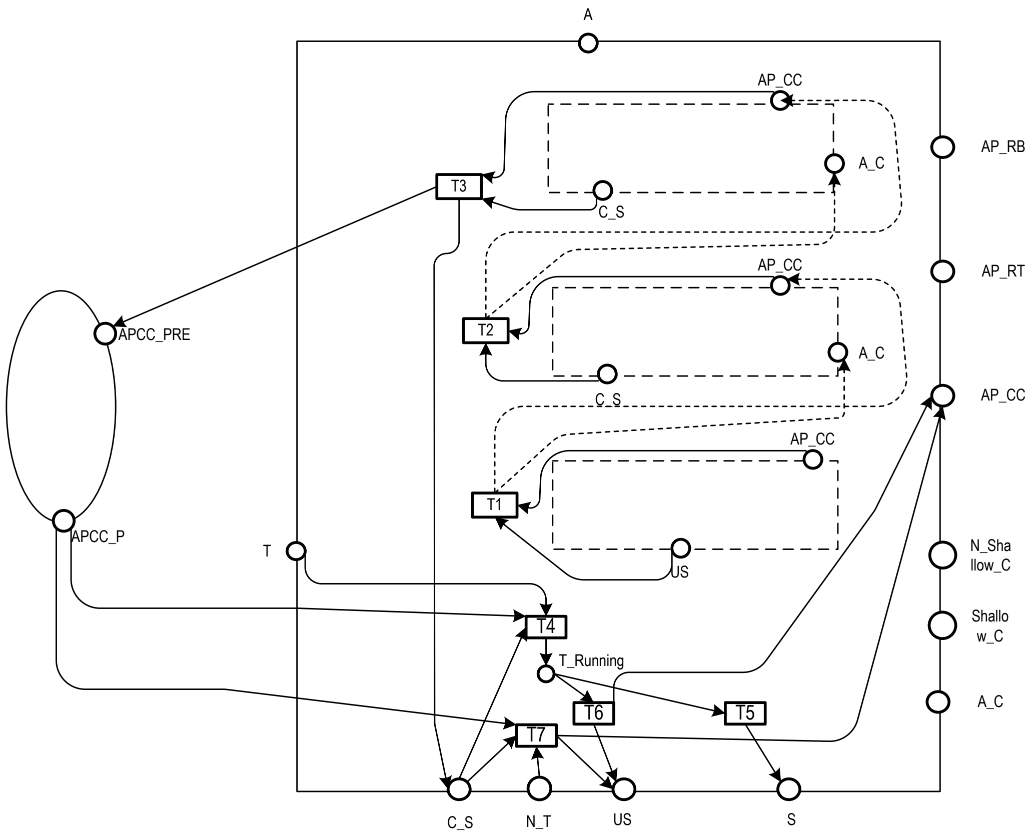


Figure A.9. Petri Net of APCC (4)

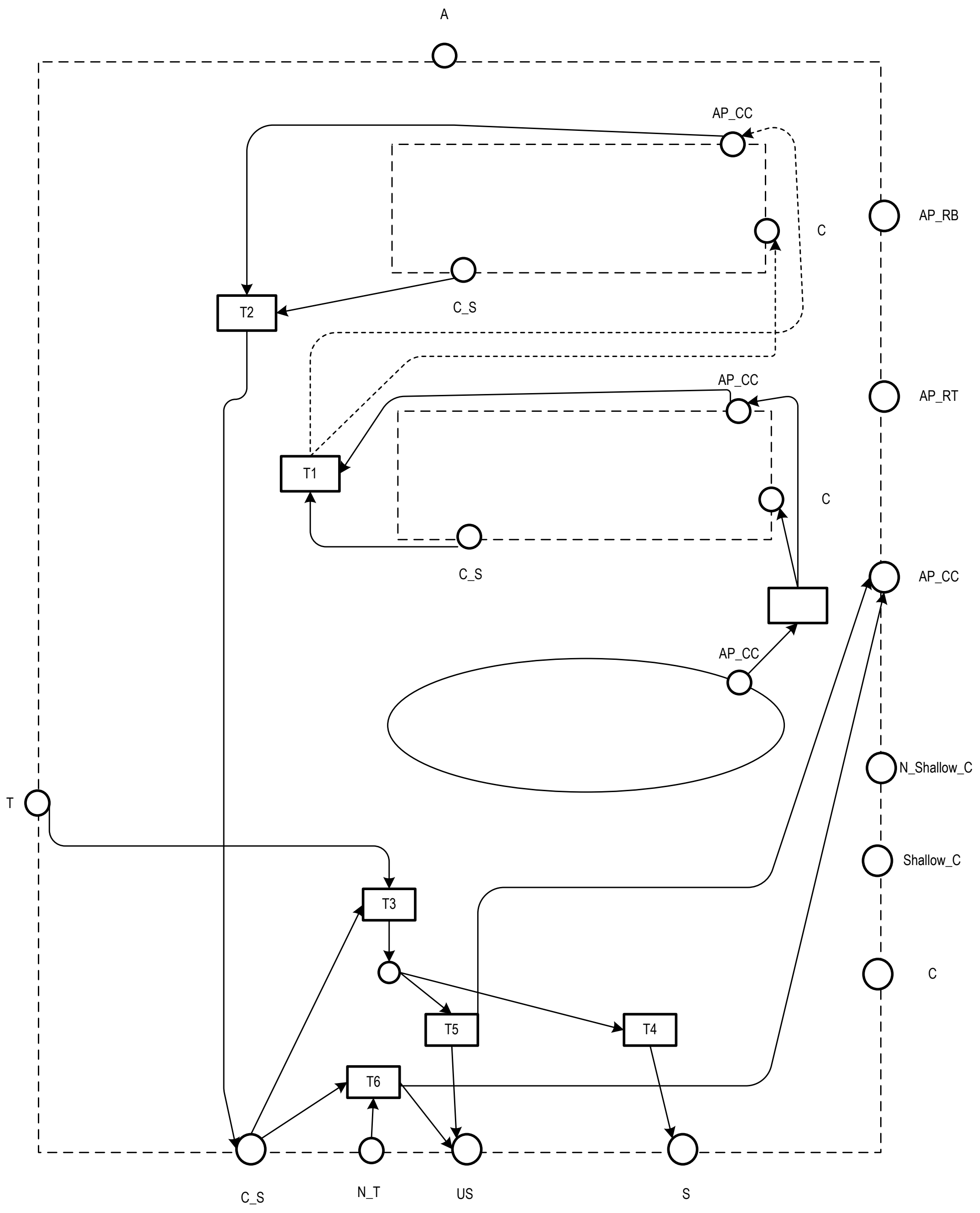


Figure A.10. Petri Net of APCC (5)

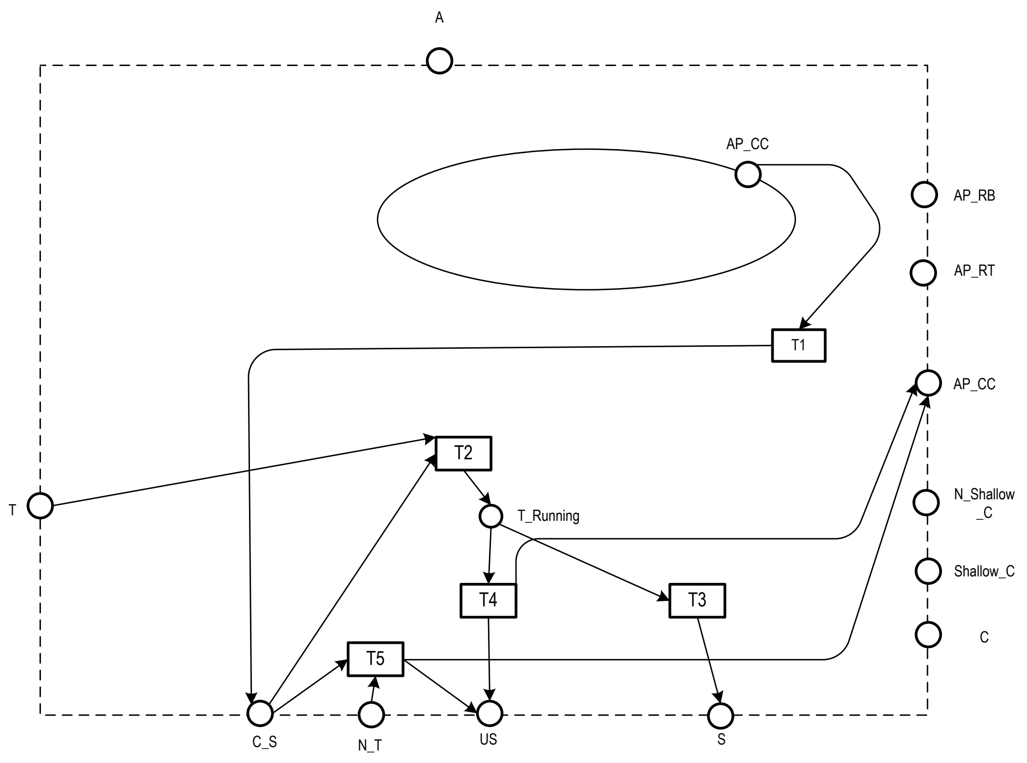


Figure A.11. Petri Net of APCC (6)

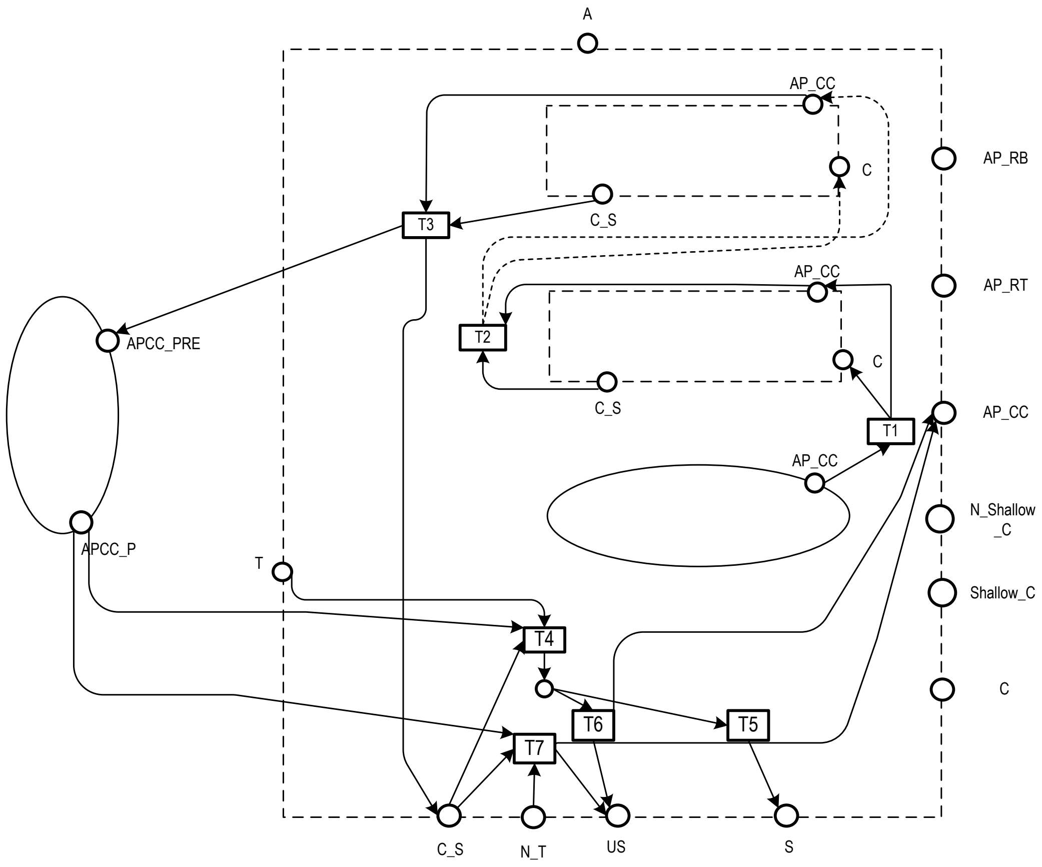


Figure A.12. Petri Net of APCC (7)

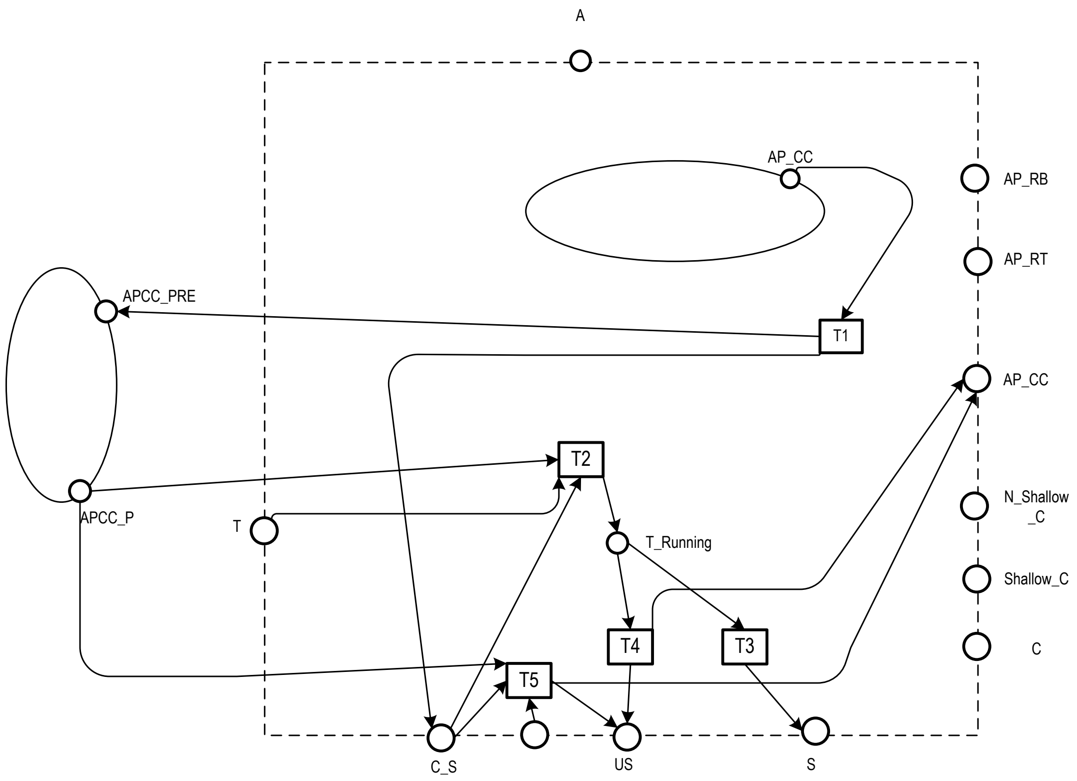


Figure A.13. Petri Net of APCC (8)