Achieving Flexibility in Service Composition with Assurance Points and Integration Rules

Susan D. Urban, Le Gao, Rajiv Shrestha, and Andrew Courter
Texas Tech University
Department of Computer Science
Lubbock, TX 79409
+1-806-742-2484
{susan.urban | le.gao | rajiv.shrestha | s.courter} @ ttu.edu

Abstract

This paper defines the concept of Assurance Points (APs) together with the use of integration rules to provide a flexible way of checking constraints and responding to execution errors in service composition. An AP is a combined logical and physical checkpoint, providing an execution milestone that stores critical data and interacts with integration rules to alter program flow and to invoke different forms of recovery depending on the execution status. During normal execution, APs invoke rules that check pre-conditions, post-conditions, and other application rules. When execution errors occur, APs are also used as rollback points. Integration rules can invoke backward recovery to specific APs using compensation as well as forward recovery through rechecking of preconditions before retry attempts or through execution of contingencies and alternative execution paths. APs combined with the use of integration rules provide an increased level of consistency checking as well as greater flexibility for recovery actions.

1. Introduction

With the development of the Internet, Web Services and service-oriented computing are becoming widely used for business-to-business integration. These processes are often collaborative in nature, involving long-running activities based on loosely-coupled, multi-platform, service-based architectures. Prevalent techniques such as the Unified Modeling Language (UML) [1], the Business Process Modeling Notation (BPMN) [2], and Event-Driven Process Chains (EPC) [3] have been widely adopted for process modeling, with execution engines based on standards such as the Business Process Execution Language (BPEL) [4] providing a framework for execution of conceptual process designs. Service composition for business integration, however, creates challenges for traditional process modeling techniques.

In a service execution environment, a process must be flexible enough to respond to errors, exceptions, and interruptions. Backward and forward recovery mechanisms [5] can be used to respond to such events. 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. Nevertheless, most service composition techniques do not provide flexibility with respect to the combined use of compensation and contingency. This absence of flexibility often does not do enough to keep processes running in a forward manner. Furthermore, most process modeling techniques for service composition do not make adequate use of pre-conditions, post-conditions, and other constraint checking techniques that can be used to validate the correctness of execution, especially considering that most processes execute in an environment that does not support traditional transaction processing guarantees for correctness and consistency of data. Service composition models need to be enhanced with features that allow processes to assess their execution status to support more dynamic ways of responding to failures, while at the same time validating correctness conditions for process execution.

This paper presents our investigation of Assurance Points (APs) and integration rules to provide a more flexible way of checking constraints and responding to execution failures. APs are illustrated as an extension to the service composition and recovery model in [6]. An AP is a combined logical and physical checkpoint, providing an execution milestone that stores critical data and interacts with integration rules to alter program flow and invoke different forms of recovery depending on the execution status. During normal execution, APs invoke integration rules that check pre-conditions, post-conditions, and other application conditions. Failure of a pre or post-condition can invoke several different forms of recovery, including backward recovery of the entire process, backward recovery to a specific AP for retry attempts, or a dynamic backward recovery process, known as cascaded contingency, in an attempt to recover to a previous AP that can be used to invoke contingent procedures or alternate execution paths. When failures occur, APs are also used as rollback points for rechecking preconditions and determining whether to invoke further forward or backward recovery actions.
In this paper, we describe the semantics of APs, integration rules, and the different forms of recovery actions as defined in [23], illustrating the functionality using an online shopping scenario. We also outline a prototype of an execution environment that we have developed to test the AP and integration rule model. The primary contribution of our work is found in the definition of a service composition and recovery model with explicit support for user-defined constraints, contingency, and compensation that is embedded in well-defined recovery actions that make use of the execution state supported by assurance points to provide flexibility in the recovery process.

The remainder of this paper is structured as follows. Section 2 presents related work. An overview of the service composition and recovery model, APs, integration rules, and recovery actions is given in Section 3. Section 4 elaborates on the semantics of the recovery actions, while Section 5 discusses a prototype execution environment with a comparison to BPEL fault and compensation handlers. The paper concludes in Section 6 with a summary and discussion of future research directions.

2. Related Work

Events and rules can be used to dynamically specify control flow and data flow in a process by using Event Condition Action (ECA) rules [7]. ECA rules have also been successfully implemented for exception handling in work such as [8], [9]. The work in [9] uses ECA rules to generate reliable and fault-tolerant BPEL processes to overcome the limited fault handling capability of BPEL.

Several efforts have been made to enhance the BPEL fault and exception handling capabilities. BPEL4Job [10] addresses fault-handling design for job flow management with the ability to migrate flow instances. The work in [11] proposes mechanisms like external variable setting, future alternative behavior, rollback and conditional re-execution of the Flow, timeout, and redo mechanisms for enabling recovery actions using BPEL. The work in [12] presents the architecture of the SH-BPEL engine, a Self-Healing plug-in for WS-BPEL engines that augments the fault recovery capabilities in WS-BPEL with mechanisms such as annotation, pre-processing, and extended recovery. The Dynamo [13] framework for the dynamic monitoring of WS-BPEL processes weaves rules such as pre/post conditions and invariants into the BPEL process. Most of these projects do not fully integrate constraint checking with a variety of recovery actions to support more dynamic and flexible ways of reacting to failures. Our research demonstrates the viability of variegated recovery approaches within a BPEL-like execution environment.

In checkpointing systems, consistent execution states are saved during the process flow. During failures and exceptions, the activity can be rolled back to the closest consistent checkpoint, resuming the execution from that point to prevent a restart of the whole process [14]. The work in [15] provides a fault tolerant architecture for Web services to detect faults and recover by means of checkpointing and rollback. The AP concept presented in this paper also stores critical execution data that can be used for constraint checking and passing parameters to rules that invoke different types of recovery actions.

Aspect-oriented programming (AOP) is another way of modularizing and adding flexibility to service composition through dynamic and autonomic composition and runtime recovery. In AOP, aspects are weaved into the execution of a program using join points. The behavioral code specified in the join point is known as advice, which can be executed before, after, or instead of the joint points [16]. The work in [17] illustrates the application of aspect-oriented software development concepts to workflow languages to provide flexible and adaptable workflows. AO4BPEL [16] is an aspect-oriented extension to BPEL that uses AspectJ to provide control flow adaptations [18]. Aspects are written in XML in different files, helping to minimize the need for changing the composition during runtime. Business rules can also be used to provide more flexibility during service composition. APs as described in this paper are similar to join points, with a novel focus on using APs to access process history data in support of constraint checking as well as flexible and dynamic recovery techniques.

3. Service Composition and Recovery Model with Assurance Points

This section summarizes the service composition and recovery model from [6]. We then define and illustrate the use of assurance points and integration rules.

3.1 Overview of the Model

In [6], a process is a top-level execution entity that is composed of other execution entities. A process is denoted as \( p \), where \( p \) represents a process and the subscript \( i \) represents a unique identifier of the process. An operation represents a service invocation, denoted as \( op_i \), such that \( op \) is an operation, \( i \) identifies the enclosing process \( p \), and \( j \) represents the unique identifier of the operation within \( p \). Compensation \( (comp_j) \) is an operation intended for backward recovery, while contingency \( (cont_j) \) is an operation used for forward recovery.

An atomic group and a composite group are logical execution units that enable the specification of processes with complex control structure, facilitating service execution failure recovery by adding scopes within the
context of a process execution. An atomic group (denoted \(ag_0\)) contains an operation, an optional compensation, and an optional contingency. A composite group (denoted \(cg_{i,k}\)) may contain multiple atomic groups, and/or multiple composite groups that execute sequentially or in parallel. A composite group can have its own compensation and contingency as optional elements. A process is essentially a top-level composite group.

![Figure 1. An Abstract View of a Sample Process](image)

Figure 1 shows an abstract view of a sample process definition. The process \(p_1\) is the top-level composite group \(cg_1\). The process \(p_1\) is composed of two composite groups \(cg_{1,1}\) and \(cg_{1,2}\), and an atomic group \(ag_{1,3}\). Similarly, \(cg_{1,1}\) and \(cg_{1,2}\) are composite groups that contain atomic groups. Each atomic and composite group can have an optional compensation plan and/or contingency plan. Some operations, such as \(op_{1,4}\), can also be marked as non-critical, meaning that the failure of the operation does not invoke any recovery activity and that the process can proceed even if the operation fails.

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

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

3.2 Assurance Point and Rule Extensions

Our work has extended the model described in the previous section with the concept of assurance points. An AP is a process execution correctness guard. Given that concurrent processes do not execute as traditional transactions in a service-oriented environment, inserting APs at critical points in a process is important for checking consistency constraints and potentially reducing the risk of failure or inconsistent data. An AP also serves as a milestone for backward and forward recovery activities. When failures occur, APs can be used as rollback points for backward recovery, rechecking pre-conditions relevant to forward recovery. In the current version of our work, we assume that APs are placed at points in a process where they are only executed once, and not embedded in iterative control structures.

An AP is defined as: \(AP = \langle apId, apParameters*, IR_{pre}, IR_{post}, IR_{cond}\rangle\), where:

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

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

\(IR_{pre}\), \(IR_{post}\), and \(IR_{cond}\) are expressed as Event-Condition-Action (ECA) rules using the format shown in Figure 2, which is based on our previous work with using integration rules to interconnect software components [21], [22]. An IR is triggered by a process reaching a specific AP during execution. Upon reaching an AP, the condition of an IR is evaluated. The action specification is executed if the condition evaluates to true. For \(IR_{pre}\) and \(IR_{post}\), a constraint \(C\) is always expressed in a negative form (not(C)). The action (action 1) is invoked if the pre or post condition is not true, invoking a recovery action or an alternative execution path. If the specified action is a retry activity, then there is a possibility for the process to execute through the same pre or post condition a second time. In such a case, action 2 is invoked rather than action 1, to invoke a different recovery action.

In its most basic form, a recovery action simply invokes an alternative process. Recovery actions can also be one of the following actions:

- **APRollback**: APRollback is used when the entire process needs to compensate its way back to the start of the process according to the semantics of the service compensation model.
- **APRetry**: APRetry is used when the running process needs to be backward recovered using compensation to a specific AP. By default, the backward recovery process will go to the first AP reached as part of the shallow or deep compensation process within the same scope. The pre-condition defined in the AP is re-checked. If the pre-condition is satisfied, the process execution is resumed from that AP by re-trying the recovered operations. Otherwise, the action of the pre-condition rule is executed. The APRetry command can optionally specify a parameter indicating the AP that is the target of the backward recovery process.

- **APCC**: APCC is a hierarchical backward recovery process when an AP is reached, the pre-condition defined in the AP will be re-checked before invoking any contingent procedures for forward recovery.

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

**Figure 2. Integration Rule Structure**

The most basic use of an AP together with integration rules is shown in Figure 3, which shows a process with three composite groups and an AP between each composite group. The shaded box shows the functionality of an AP using AP2 as an example. Each AP serves as a checkpoint facility, storing execution status data in a checkpoint database (AP data in Figure 3). When the execution reaches AP2, IRs associated with the AP are invoked. The condition of an IRpost is evaluated first to validate the execution of CG2. If the post-condition is violated, the action invoked can be one of the pre-defined recovery actions as described above. If the post-condition is not violated, then an IRpre rule is evaluated to check the pre-condition for the next service execution. If the pre-condition is violated, one of the pre-defined recovery actions will be invoked. If the pre-condition is satisfied, the AP will check for any additional, conditional rules (IRcond) that may have been expressed. IRcond rules do not affect the normal flow of execution but provide a way to invoke additional parallel activity based on application requirements. Note that the expression of a pre-condition, post-condition or any additional condition is optional.

### 3.3 Case Study: Assurance Points and Rules

This section provides an example of assurance points, integration rules, and conditional rules using an online shopping application in Figure 4. All atomic and composite groups are shown in the solid line rectangles, while optional compensations and contingencies are shown in dashed line rectangles, denoted as rop and rop, respectively. APs are shown as ovals between composite and/or atomic groups, where the AP identifiers and parameters are OrderPlaced(orderId), CreditCardCharged(orderId, cardnumber, amount), UPSShipped(orderId, UPSShippingDate), USPSShipped (orderId), Delivered(orderId, shippingMethod, deliveryDate).

Table 1 shows the integration rules and conditional rules associated with the APs in Figure 4. The components of an assurance point are explained below using the APs in Figure 4 and the rules in Table 1.

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

**Component 2 (Integration Rules)**: An integration rule is optionally used as a transition between logical components of a process to check pre and post conditions. In Table 1, the orderPlaced AP has a pre-condition that guarantees that the store must have enough goods in stock. Otherwise, the process invokes the backOrderPurchase process. The CreditCardCharged AP has a post-condition that further guarantees the in-stock quantity must be in a reasonable status after the deinventory operation.

**Component 3 (Conditional Rule)**: In Table 1, the CreditCardCharged AP has a conditional rule that sends a text message notification for large charges. Since no pre or post condition is specified for the Delivered AP, only the conditional rule shippingRefund is evaluated. Assume the delivery method was overnight through UPS with an
extra shipping fee. If UPS has delivered the item on time, then the Delivered AP is complete and execution continues. Otherwise, the action refundUPSShippingCharge is invoked to refund the extra fee while the main process execution continues. If backward recovery with retry takes place, it is possible that the process will execute the same conditional rule a second time. The action of the rule will only be executed during the retry process if the action was not executed the first time through.

4. Semantics of AP Recovery Actions

This section illustrates the semantics of the APRollback, APRetry, and APCC recovery actions using the generic sample process in Figure 5 as well as the Online Shopping example in Figure 4. Detailed algorithms can be found in [23]. In the following, assume that each AP in Figure 5 has an IRpre and an IRpost rule.

4.1 Recovery Actions for Pre and Post Conditions

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

_scenario 1 (APRollback):_ Assume that the post-condition fails at AP4 in Figure 5 and that the action of IRpost is APRollback. Since APRollback is invoked, the process compensates all completed atomic and/or composite groups as described in Section 3.1. Here, the process invokes cg04.cg0 to compensate cg04. The APRollback process will then deep compensate cg03 by invoking cg03.cg0 since 1) there is no shallow compensation for cg03 and 2) cg03 is non-critical and therefore has no compensating procedure. APRollback then invokes shallow compensation cg02.cg0, with no specific action to cg01 since it is non-critical.

APRetry is used to recover to a specific AP and then retry the recovered atomic and/or composite groups. If the AP has an IRpre, then the pre-condition will be re-examined. If the pre-condition fails, the action of the rule is executed, which either invokes an alternate execution path for forward recovery or a recovery procedure for backward recovery. Otherwise, the relevant section of code is re-executed. APRetry can only recover to an AP that is within the same scope of the failed pre or post condition. 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):_ Assume that the post-condition fails at AP4 in Figure 5 and that the action of IRpost is APRetry. This action compensates to the most recent AP within the same scope by default. APRetry first invokes cg04.cg0 to compensate cg04. The process then deep compensates cg03 by executing cg03.cg0. At this point, AP2

**Figure 4. Online Shopping Process with APs**

**Table 1. AP Rules in the Online Shopping Process**

<table>
<thead>
<tr>
<th>Integration Rule</th>
<th>Conditional Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>create rule</td>
<td>create rule</td>
</tr>
<tr>
<td>event: QuantityCheck::pre</td>
<td>event: Notice::cond</td>
</tr>
<tr>
<td>condition: exists(select L.itmId from Inventory I, LineItem L where L.orderId=orderId and L.itmId=I.itmId and L.quantity=I.quantity)</td>
<td>condition: amount &gt; $1000</td>
</tr>
<tr>
<td>action: backOrderPurchase(orderId)</td>
<td>action: highExpenseNotice(cardNumber)</td>
</tr>
<tr>
<td>create rule</td>
<td>create rule</td>
</tr>
<tr>
<td>event: CreditCardCharged::post</td>
<td>event: Delivered::cond</td>
</tr>
<tr>
<td>condition: exists(select L.itmId from Inventory I, LineItem L where L.orderId=orderId and L.itmId=I.itmId and L.quantity=0)</td>
<td>condition: shippingMethod = UPS &amp; deliveryDate = UPS:Shipped.UPSShippingDate+1</td>
</tr>
<tr>
<td>action1: APRetry</td>
<td>action: refundUPSShippingCharge(orderId)</td>
</tr>
<tr>
<td>action2: APRollback</td>
<td>action:</td>
</tr>
</tbody>
</table>
is reached and the pre-condition of $\text{IR}_{\text{pre}}$ is re-evaluated. If the pre-condition fails, the process executes the recovery action of $\text{IR}_{\text{pre}}$. If the pre-condition is satisfied or if there is no $\text{IR}_{\text{pre}}$, then execution will resume again from $\text{cg}_0$. In this case, the process will reach $\text{AP}4$ a second time, where the post-condition is checked once more. If failure occurs for the second time, the second action defined on the rule is executed rather than the first action. If a second action is not specified, the default action will be $\text{APRollback}$.

Now assume that the action of the pre-condition for $\text{AP}4$ is parameterized as $\text{APRetry}(\text{AP}1)$, indicating that the retry activity should rollback to $\text{AP}1$. The process will then compensate the procedure back to the point of $\text{AP}1$ for the retry process, ignoring all APs in between.

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

**Scenario 3 (APCC):** Assume that the post-condition fails at $\text{AP}4$ in Figure 5 and that the $\text{IR}_{\text{post}}$ action is APCC. As soon as APCC is invoked, the process starts compensating until it reaches the parent layer. In this case, the process will reach the beginning of $\text{cg}_0$ after compensating the entire process through deep or shallow compensation. Since there is no AP before $\text{cg}_0$, then $\text{cg}_0$.top is invoked.

**Scenario 4 (APCC):** Assume that the post-condition fails at $\text{AP}3$ in Figure 5 and that the $\text{IR}_{\text{post}}$ action is APCC. Since $\text{AP}3$ is in $\text{cg}_0$, which is nested in $\text{cg}_0$, the APCC process will compensate back to the beginning of $\text{cg}_0$, executing $\text{cg}_0$.top. The APCC process finds $\text{AP}2$ with an $\text{IR}_{\text{pre}}$ rule for $\text{cg}_03$. As a result, the pre-condition will be evaluated before trying the contingency for $\text{cg}_03$. If there is no pre-condition or if the pre-condition is satisfied, then $\text{cg}_03$.top is executed and the process continues. Otherwise, the recovery action of $\text{IR}_{\text{pre}}$ for $\text{AP}2$ will be executed. If $\text{cg}_03$.top fails then the process will still be under APCC mode, where the process will keep compensating until it reaches the $\text{cg}_0$ layer, where $\text{cg}_0$.top is executed.

### 4.2 Recovery Actions in Execution Errors

When process execution encounters an internal error, the running operation first tries the most immediate contingency, as defined in Section 3.1. If the contingency succeeds, the recovery is complete and the execution continues. If the contingency fails or if there is no immediate contingency, then the execution goes into APCC mode as described in Section 4.1.

**Scenario 5 (Online Shopping Example - Failure at ChargeCreditCard):** Returning to the Online Shopping Example of Figure 4, assume the process fails while executing $\text{chargeCreditCard}$. The process then executes the contingency $\text{cg}_21$.top ($\text{eCheckPay}$). If $\text{cg}_21$.top fails, then APCC process begins, during which the process reaches the $\text{orderPlaced}$ AP, where the pre-condition of the AP is re-checked (rule $\text{QuantityCheck}$ in Table 1). If the pre-condition is violated, the action $\text{backOrder}$ is invoked, which means there are not enough goods in stock.

**Scenario 6 (Online Shopping Example – Failure at UPSShipping):** From Figure 4, assume the process fails on the operation $\text{UPSShipping}$. Since there is no immediate contingency, the process invokes the APCC process, rolling back to the $\text{CreditCardCharged}$ AP at the outer level. Since there is no pre-condition defined at the $\text{CreditCardCharged}$ AP, the contingency $\text{cg}_3$.top ($\text{FedexShipping}$) will be executed. If $\text{cg}_3$.top fails, the process will be still under APCC mode, compensating its way back to the beginning of the transaction.

### 5. Prototype and Comparison to BPEL

We have prototyped an execution environment to demonstrate the extended service composition and recovery model with APs and integration rules. We are not directly using BPEL since the broader scope of our research is addressing techniques for decentralized data dependency among distributed Process Execution Agents (PEXAs) [19, 20, 24]. Existing BPEL engines do not provide the flexibility needed to experiment with this form of decentralized communication among process execution engines. The process specification framework, however, is based on BPEL using our previous work with a Process Modeling Language (PML) described in [25].

The process specification framework uses a minimal set of activities such as assign, invoke, and switch to **Figure 5. Generic Process for Recovery Actions**
illustrate the functionality of APs and the different forms of recovery. Figure 6 shows a sample process in XML to illustrate the syntax for defining atomic (<ap>...) and composite (<cg>...) groups with compensating (<cop>...) and contingent (<cop>...) procedures. The syntax for APs and their parameters is also illustrated (<ap>...>). Figure 7 shows the XML rule specification associated with the orderPlaced AP in Figure 6. The rule indicates the event (i.e., assurance point) that triggers the rule (<event ap = ...>), whether the rule is a pre (<pre>), post (<post>), or conditional (<cond>) rule, as well as the condition (<condition>...) and action (<action>...) of the rule, where rule conditions are implemented in web services.

```
<cg name=“cg0”>
  ...
  <ap name=“OrderPlacedAP”>
    <apDataIn variable=“orderId” />
  </ap>
  ...
  <cg name=“cg1”>
    ...
    <invoke name=“makePayment” serviceName=“creditCard1”
      portType=“cc:CreditCardPortType” operation=“makePayment”
      inputVariable=“makePaymentInput”
      outputVariable=“makePaymentOutput” />
  </cg>
</cg>
```

**Figure 6. Activity Syntax**

The parser for the XML Java binding process has been implemented in the execution engine using XMLBeans which fully utilizes the XML Schema definition for unmarshalling and validating XML input documents. XMLBeans creates the Java types that represent schema types. For each activity defined, a wrapper class has been developed that implements the semantics of the activity within the processor. AP data is stored in a db40 object-oriented database. We have fully implemented the functionality described in this paper to test and demonstrate all algorithms associated with the creation of APs and the use of the rules and recovery procedures.

We have conducted a comparison of the AP model with BPEL fault and compensation handlers. BPEL does not explicitly support a contingency feature other than fault, exception, and termination handlers. The designer is also responsible for complex fault handling logic, which, as pointed out in [26, 27], has the potential to increase complexity and create unexpected errors. The work in [27] specifically points out the confusion that can arise about compensation order in BPEL. The AP concept defines explicit recovery semantics for atomic and composite groups. Furthermore, whereas the compensation handler in BPEL does not directly know the current status of the process, the AP model allows the designer to save relevant state, thus helping to compensate the process to a previous state. The AP model also provides explicit contingency activities so that forward recovery is possible. Compared to BPEL, the AP logic allows designers to have a clearer notion of how recovery actions take place and at the same time provide flexibility through different recovery actions depending upon the status of execution and user-defined integration rule conditions. See [23] for a more complete comparison of the AP model with BPEL.

```
<rules>
    ...
    <event ap=“orderPlacedAP”>
      ...
      <ecaRule>
        <condition name=“QuantityCheck”>
          <invoke name=“checkQuantity” serviceName=“ruleConditions”
            portType=“rule:ruleConditionsPortType” operation=“checkQuantity”
            inputVariable=“quantity”
            outputVariable=“result” />
        </condition>
        <actions>
          <action name=“BackOrderPurchase”>
            ...
          </action>
        </actions>
      </ecaRule>
    </event>
  </rules>
```

**Figure 7. Rules Syntax**

7. Conclusions and Future Directions

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

There are several directions for future work. One direction involves formalization of the assurance point concept using Petri-nets and model-checking. Methodological issues for the specification of APs, integration rules, and recovery procedures will also be addressed, together with refinement of recovery actions for concurrent and iterative activity. We are also investigating the integration of assurance points with our work on decentralized data dependency analysis [24] in Process Execution Agents (PEXAs), where PEXAs communicate about data dependencies so that when one process fails and recovers, other data dependent processes can be notified of potential data inconsistencies. The AP concept can be used to enhance decentralized PEXAs with greater flexibility for process recovery options.

8. Acknowledgements

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

9. References