

Formal Specification of the Assurance Point Web Service Composition Model

Le Gao*

*Department of Computer Science, Texas Tech University
Lubbock, TX 79409, USA
legao82@gmail.com*

Susan D. Urban

*Department of Industrial Engineering, Texas Tech University
Lubbock, TX 79409, USA*

Jonathan Rodriguez and Abhijit Warkhade

*Department of Computer Science, Texas Tech University
Lubbock, TX 79409, USA*

Received 28 February 2014

Accepted 4 August 2014

Published 10 September 2014

This paper presents a formal specification of the Assurance Point (AP) web service composition model. The AP model provides 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 (IRs) to alter program flow and to invoke different forms of recovery depending on the execution status. In this paper, the execution and recovery semantics of assurance points have been fully defined in the context of if-else, parallel, and loop control structures. The activities and complex control structures of the AP model have been formalized and tested in the Yet Another Workflow Language (YAWL) engine. By doing so, the correctness of the execution and recovery semantics in the AP model has been verified, demonstrating that YAWL nets of the AP model satisfy the soundness property. Different from existing web service composition models, the AP model presented by this research provides multiple levels of protection against service execution failure with a combination of forward and backward recovery techniques.

Keywords: Service composition; exception handling; formalization.

1. Introduction

A service in a service-oriented architecture¹ (SOA) is a unit of work executed by a service provider to achieve desired results for a service consumer. In an SOA, a web service composition must be flexible enough to respond to individual service errors,

*Corresponding author.

exceptions, and interruptions. To respond such events, backward and forward recovery techniques^{2,3} 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.

From the software engineering point of view, web service composition is also an architecture of software development. As a result, verification of the execution semantics of the web service composition is another research challenge. 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),⁴ the Business Process Modeling Notation,⁵ and Event-Driven Process Chains (EPC)⁵ have been widely adopted for process modeling, with execution engines based on standards such as the Business Process Execution Language (BPEL)⁶ providing a framework for execution of conceptual process designs. Service composition for business integration, however, creates challenges for traditional process modeling techniques.

Our own research in this area has defined a hierarchical service composition and recovery model⁷ that combines the use of compensation and contingency operations to maximize the forward recovery of the process when failure occurs. To enhance flexibility in process execution, the concept of Assurance Points (APs) were introduced into the execution of a process.⁸ An AP is a combined logical and physical checkpoint, providing a milestone that stores critical data and invokes integration rules (IRs). IRs check pre and post conditions that can alter program flow and invoke different forms of recovery. An AP is also used as a rollback point in the recovery of a process. Three different forms of backward recovery are defined as actions triggered by IRs, where rule actions are capable of either full backward recovery or a combination of backward and forward recovery.

A limitation of this initial work with APs is that APs were only defined in the context of sequential control flow.^{8,9} This paper extends the AP model for use with if-else, parallel, and loop control structures, thus defining the use of APs in a more computationally complete context. In addition, the semantics of AP recovery actions invoked from within the complex control structures are defined.

To precisely describe and verify the execution and recovery semantics of the AP model, Petri Nets¹⁰ were initially used to formalize a subset of the AP model.^{8,11} Petri Nets provide graphical and formal representations of the execution and

recovery semantics of the AP model. However, 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, which makes the specification difficult to develop and understand. In addition, verification of the semantics of a Petri Net with too many places is complex. Yet Another Workflow Language (YAWL)¹² is a business process modeling system inspired by Petri Nets, providing better support for the business workflow modeling. Compared to Petri Nets, YAWL has several merits. First, YAWL does not have the rule that a place must exist between two transitions, which avoids state explosion. Second, YAWL provides a powerful analytical function to verify soundness property of a model defined in YAWL. Compared to BPMN, UML and EPCs, YAWL introduces more meaningful control constructs for modeling complex control-flows in a business process. In addition, YAWL models are executable. Compared to BPEL, YAWL has a graphical and formal representation which is easy to understand and verify.

The primary contribution of this research is the complete definition and formal specification of the AP model with full support for all primary programming language control structures. Different from existing web service composition models, the AP model provide multiple forms of protection against service execution failure. APs provide referenceable points that store critical execution status. APs also support user-defined constraints by checking pre and post conditions that are specified as IRs, with recovery actions that help to minimize backward recovery and maximize forward recovery. This research fully defines the use of these concepts for all primary programming language control structures with a mapping to YAWL nets that provides a clear specification of the semantics of the AP model and its recovery actions together with a demonstration of the soundness of the model. A prototype of the AP model has been developed as part of this research and is reported in another paper.¹³ The evaluation results show that the AP model is scalable and feasible for a large number of concurrently running processes.

In the remainder of this paper, Sec. 2 presents related work. An overview of the initial definition of the AP model is given in Sec. 3. The extension of the AP model with complex control structures is presented in Sec. 4. Section 5 then formalizes the execution and recovery semantics of the AP model by using YAWL. This paper concludes in Sec. 6, outlining contributions and future research directions.

2. Related Work

From a historical point of view, our work is founded on past work with Advanced Transaction Models (ATMs). ATMs provide better support for Long Running Transactions (LRTs) that need relaxed atomicity and isolation properties.¹⁴ Sagas¹⁵ were defined as a mechanism to structure long running processes, with each sub-transaction having a compensating procedure to reverse the affects of the saga when it fails. Other advanced transaction models have also made use of compensation for hierarchically structured transactions.¹⁶

The term transactional workflow was introduced to recognize the relevance of transactions to workflow activity that does not fully support ACID properties. The ConTract Model provides a classic example of work with transactional workflows,¹⁷ supporting the correct execution of non-atomic, long-lived applications with application-dependent consistency constraints. Other examples of transactional workflow models include the Workflow Activity Model,¹⁸ the Crew Project,¹⁹ and METEOR.² 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,²⁰⁻²² but do not fully support constraint checking and lack flexibility in the recovery process. The standard compensation mechanism used in web services has been discussed in work by Yang and Liu.²³ In addition, the authors have proposed a multiple-compensation mechanism to enrich the standard compensation mechanism. In work by Schafer, Dolog and Nejdl, the authors²⁴ proposed a contract-based approach to support flexible compensation operations in web services, which allows the specification of permitted compensations at runtime. Grefen, Vonk and Apers define another approach²⁵ that extends the standard compensation approach for web services to deal with arbitrary process structures to allow cycles in processes. Their work also defines safe-points to allow partial compensation of processes. The concept of AP presented in this paper extends the safe-points by checking pre/post conditions during the normal workflow execution, and before forward recovery, which further guarantees the correctness of the execution. A high-level, compensation based transaction model has also been presented by Vonk and Grefen,²⁶ which provides flexibility in rollback semantics by combining rollback modes and rollback scopes. To deal with the problem of atomicity and isolation in the context of processes, a unified model²⁷ for concurrency control and recovery of processes has been proposed. In the unified model, the authors proposed a dynamic scheduling protocol to achieve correctness for the concurrent execution of processes. Certain process locking mechanisms, however, are still adopted in the model that may be too restrictive for use with service-oriented computing.

WS-BPEL provides fault, compensation and termination handlers to handle execution exceptions. All three handlers are associated with scopes. A fault handler aims to correct the error in a scope such that a process can continue running or invoke an alternative process. A compensation handler is used to compensate a completed scope. A termination handler aborts a running scope. When an error occurs, all running activities in the scope in which the error occurs will be first terminated. If the activity is a non-scope activity, it is simply aborted. If the activity is a scope, the associated termination handler is activated. When all running activities have been terminated, the fault handler of the scope in which the error occurs is invoked. The fault handler will invoke compensation handlers to compensate all its nested completed scopes. In WS-BPEL, the compensation procedure in a scope follows the reverse order of enclosed scope completion. However, the dependency between enclosed scopes will potentially complicate the compensation

order. Because the default exception handling mechanism in WS-BPEL may activate handlers at different levels when scopes at different levels are being recovered, the “Zigzag” compensation behavior in WS-BPEL is difficult to understand.²⁸

More recently, events and rules have been used to dynamically specify control flow and data flow in a process by using Event Condition Action (ECA) rules.²⁹ ECA rules have also been successfully implemented for exception handling.^{30,31} Other work³¹ uses ECA rules to generate reliable and fault-tolerant BPEL processes to overcome the limited fault handling capability of BPEL. Our work with APs also supports the use of rules that separate fault handling from normal business logic. Combined with APs, IRs are used to integrate user-defined consistency constraints with the recovery process.

Several efforts have been made to enhance the BPEL fault and exception handling capabilities. BPEL4Job³² addresses fault-handling design for job flow management with the ability to migrate flow instances. Modafferi and Conforti³³ propose mechanisms like external variable setting, future alternative behavior, roll-back and conditional re-execution of the flow, timeout, and redo mechanisms for enabling recovery actions using BPEL. Modafferi *et al.*³⁴ present 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 Dynamo³⁵ framework for the dynamic monitoring of WS-BPEL processes weaves rules such as pre/post conditions and invariants into the BPEL process. Most of these projects do not fully integrate constraint checking with a variety of recovery actions as in our work to support more dynamic and flexible ways of reacting to failures. Our research demonstrates the viability of variegated recovery approaches within a BPEL-like execution environment.

In checkpointing systems, consistent execution states are saved during the process flow. During failures and exceptions, the activity can be rolled back to the closest consistent checkpoint to move the execution to an alternative platform.^{36,37} The AP concept presented in this paper also stores critical execution data, but uses the data as parameters to rules that perform constraint checking and invoke different types of recovery actions.

All or nothing (atomicity) is a key property to prevent inconsistency in service execution. One technique³⁸ has proposed a process algebraic framework to publish atomicity-equivalent public views from the backend processes. This method enables service consumers to achieve atomicity of a service composition by choosing suitable services before the service composition.

In another technique³⁹ based on Petri Nets, the authors proposed a Self-Adapting Recovery Net (SARN) model for specifying exceptional handling in business processes at design time. The authors also presented a set of high-level recovery policies of both single task and recovery region. However, the proposed recovery policies did not cover the recovery semantics in complex control structures of a hierarchical service composition.

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.⁴⁰ The work in Ref. 41 illustrates the application of aspect-oriented software development concepts to workflow languages to provide flexible and adaptable workflows. AO4BPEL⁴⁰ is an aspect-oriented extension to BPEL that uses AspectJ to provide control flow adaptations.⁴² Business rules can also be used to provide more flexibility during service composition. APs as described in this paper are similar to join points, with a novel focus on using APs to access process history data in support of constraint checking as well as flexible and dynamic recovery techniques.

Due to the distributed nature of services, service composition is often inflexible and highly vulnerable to errors. Even BPEL, the de-facto standard for composing Web services, still lacks sophistication with respect to handling faults and events. Our research is different than related work by providing comprehensive support with a cascaded compensation policy for user-defined constraints with the use of pre, post, and conditional rules. In addition, the AP model integrates the rules with different recovery actions as well as user-defined compensation and contingency. Thus, the concept of APs provides flexibility for process recovery. With the recovery actions provided by APs, a process can be forward recovered even if an execution error or an event interruption occurs, which is a combination of features that are not available in current or past research.

3. Overview of the AP Model and Recovery Actions

The AP model is an extension of a service composition model originally defined by Xiao and Urban,⁷ where a process is hierarchically composed of 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.

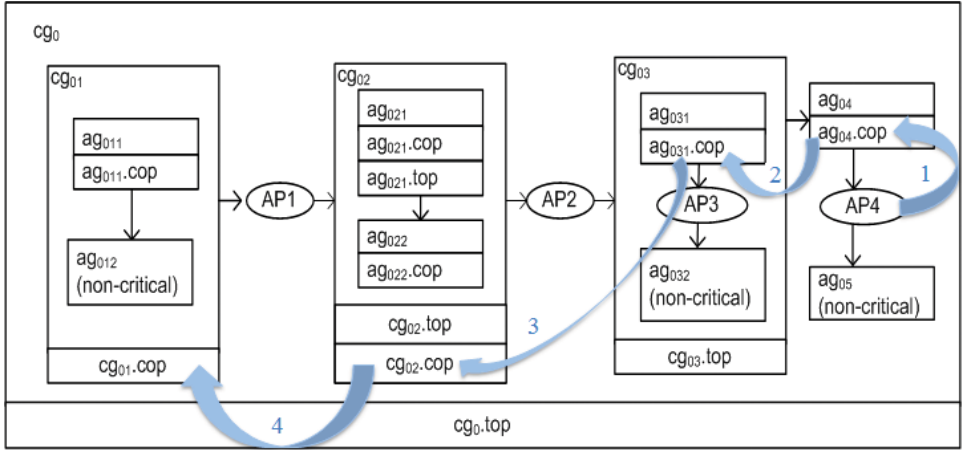


Fig. 1. Generic process: Scenario 1 (APRollback).⁴³

As an example of the service composition model, Fig. 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 Fig. 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 example in Fig. 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 Fig. 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.

The service composition model was extended by introducing APs in the execution of a process,⁸ providing the capability of checking pre and post conditions

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

Fig. 2. IR structure.

through the use of IRs. An AP is defined with a unique identifier, a set of parameters that list the critical data items to be stored and checked, and a set of pre and post conditions defined as IRs.⁴⁴ An IR as shown in Fig. 2, 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*.

Figure 3 shows a portion of an online shopping process. In Fig. 3, composite group *cg₂* 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 *cg₂*. 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.

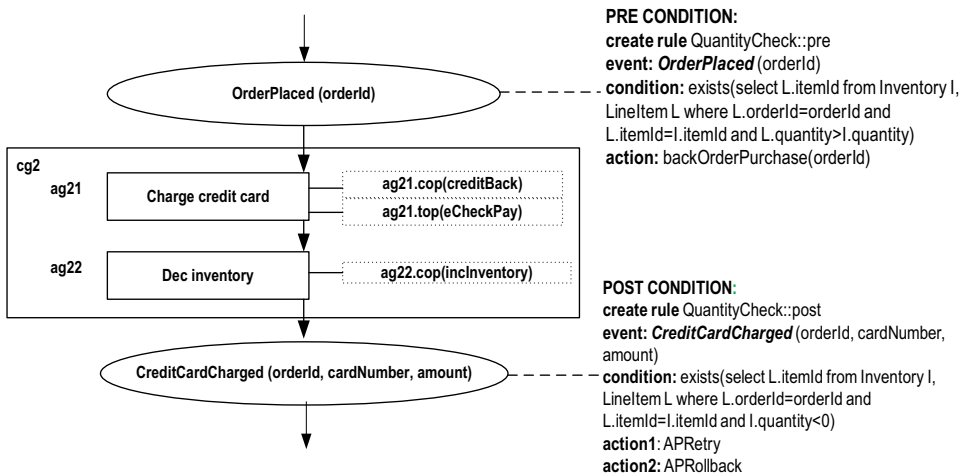


Fig. 3. APs in online shopping process.⁸

The condition defined in an IR 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 or deep compensation.

Scenario 1 (APRollback): Assume that the post-condition fails at AP4 in Fig. 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 Fig. 1. First the process invokes `ag04.cop` to compensate `ag04`. Second, the APRollback process will deep compensate `ag031` by invoking `ag031.cop` since 1) there is no shallow compensation for `cg03` and 2) `ag032` is non-critical and therefore has no compensating procedure. Finally, APRollback invokes shallow compensation `cg02.cop` and `cg01.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 Fig. 4 and that the action of the IR is APRetry. This action

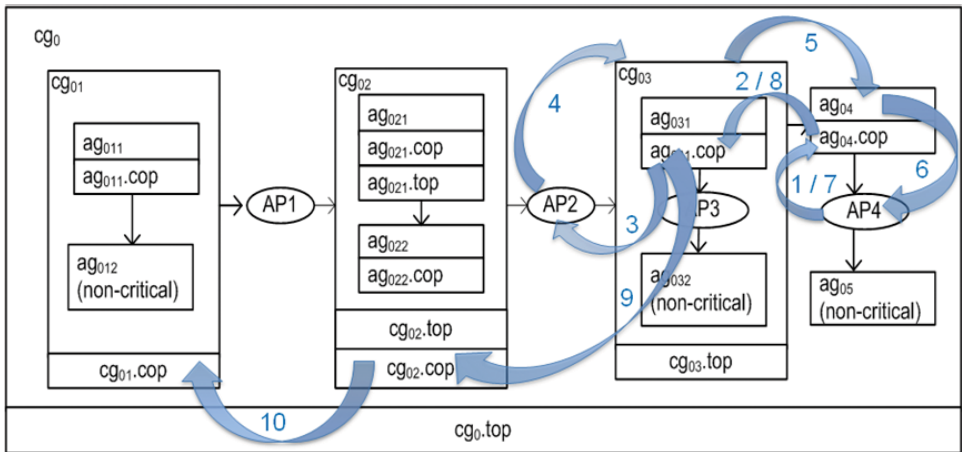


Fig. 4. Scenario 2 (APRetry-default).⁴³

compensates to the most recent AP within the same scope by default. In Fig. 4, APRetry first invokes `ag04.cop` to compensate `ag04` at step 1. The process then deep compensates `cg03` by executing `ag031.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 `cg03`. In this case, the process will reach AP4 a second time through steps 4–6, 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 7–10.

- **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.

Scenario 3 (APCC): Assume that the post-condition fails at AP4 in Fig. 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 Fig. 1. Since there is no AP before `cg0`, `cg03.top` is invoked.

Scenario 4 (APCC): Assume that the post-condition fails at AP3 in Fig. 5 and that the IR action is APCC. Since AP3 is in `cg03`, which is nested in `cg0`, the APCC process will compensate back to the beginning of `cg03`, executing `ag031.cop` at step 1. The APCC process finds AP2 with an IR pre-condition for `cg03` at step 2. As a result, the pre-condition will be evaluated before trying the contingency for `cg03`. If there is no pre-condition or if the pre-condition is satisfied, then `cg03.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 `cg03.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 `cg0` layer, where `cg0.top` is executed at step 7.

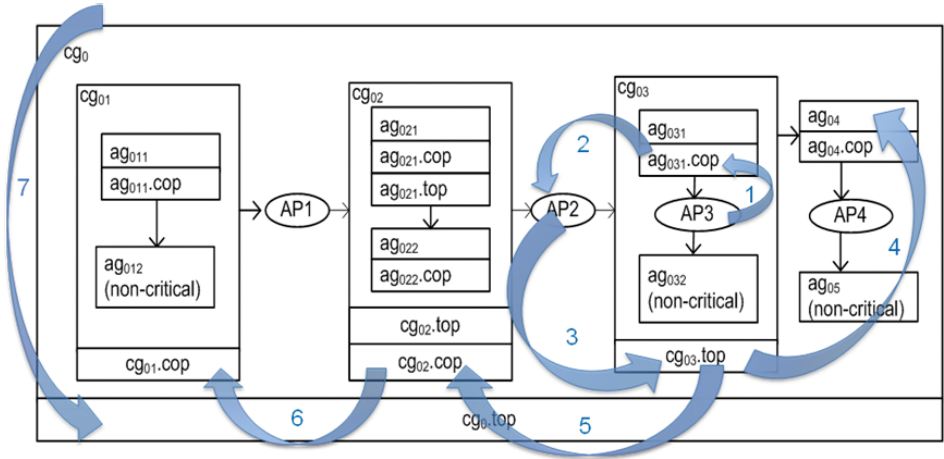


Fig. 5. Scenario 4 (APCC).⁴³

4. Extending the AP Model with Complex Control Structures

The AP model was originally defined in the context of sequential control flow for the purpose of defining AP functionality and recovery concepts. Sequential control flow obviously has limitations with respect to computational completeness. This research has extended the AP model for use with if-else, parallel, and loop control structures. In addition, the semantics of AP recovery actions invoked in the context of these complex control structures have also been defined. Section 4.1 informally introduces the parallel control structure, known as a flow group. The execution and recovery semantics in the if-else and loop control structures are given in Secs. 4.2 and 4.3, respectively. Section 4.4 then introduces a course enrollment case study to illustrate the use of the AP model in the context of these control structures. Formal specification of the complete AP model will be addressed in Sec. 5.

4.1. The parallel control structure and recovery semantics

In BPEL, parallel execution is supported through the use of 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 a flow activity, all threads are running independently and do not need to wait for others to complete.

In this section, a new group, known as a flow group, is introduced to the AP model, in addition to the atomic and composite groups that have already been defined. The flow group is similar to the flow activity in BPEL. Section 4.1.1 presents the concept of a flow group, while Sec. 4.1.2 presents the recovery semantics of a flow group in the AP model.

4.1.1. *The flow group control structure and recovery semantics*

A flow group is a parallel control structure that 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 flow group can contain multiple composite groups executed in parallel and independently. Figure 6 shows an example of a flow group that contains three composite 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. APs can be inserted in any points in a thread as needed. In addition, an AP is required at the end of a thread. There are two benefits of placing an AP at the end of a thread. First, an AP at the end of a thread will further guarantee the thread is in a good state and is ready for synchronization. Second, the AP can provide better support of cross-cutting concerns, such as synchronization of multiple threads.

4.1.2. *Recovery semantics in a flow group*

In the execution of concurrent composite groups, it is important to define the course of action to take when one of the threads of execution fails. In some cases,

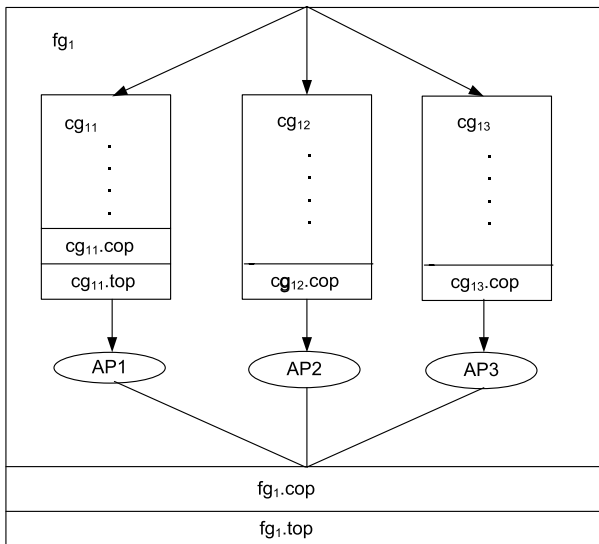


Fig. 6. An example of a flow group.

the other concurrently executing groups may also need to be halted and recovered using some form of backward recovery to the outer scope of the flow group. In other cases, it may be possible to recover only the failed thread of execution and to continue with a form of forward recovery for the failed thread. In this case, the other concurrently executing threads are not affected by the failure. All three recovery actions, APRollback, APRetry and APCC, can be performed in a flow group.

APRollback. If APRollback is invoked in a thread, the thread will be backward recovered. In this case, the flow group will notify all other threads to stop execution and start recovery. For example, if the APRollback action is invoked in any point in cg_{11} in Fig. 6, the cg_{11} will first be recovered and then cg_{12} and cg_{13} will be notified and recovered as well. Similarly, if the APRollback action is initiated by AP1, then cg_{11} will be recovered and cg_{12} and cg_{13} will then be recovered.

APRetry. In a flow group, APRetry is used to recover to a specific AP and then retry the recovered atomic and/or composite groups in a thread. Since the data used by each thread is completely independent from that of the others, APRetry performed in a thread will not affect other threads in a flow group. As a special AP, APRetry initiated by the ending AP in a thread will retry the entire thread. For example, in Fig. 6, APRetry initiated by AP1 will retry the entire cg_{11} .

APCC. In a flow group, if APCC is invoked in a thread, a contingent procedure must be executed instead of the failed thread. For example, if APCC is invoked in cg_{11} in Fig. 6, $cg_{11}.top$ will be executed after compensation of cg_{11} . However, compensation of other threads may be needed if the immediate contingency fails or does not exist. For instance, if $cg_{11}.top$ fails, cg_{12} and cg_{13} will need to be compensated. Then, $fg_1.top$ will be executed as a contingent procedure. APCC initiated by the ending AP in a thread, will cause compensation of all threads in a flow group. A contingent procedure of the entire flow group will be executed. For example, APCC initiated by AP1 in Fig. 6, will first compensate cg_{11} . Then cg_{12} and cg_{13} will be notified and compensated. Finally, $fg_1.top$ will be executed.

4.2. *The if-else control structure and recovery semantics*

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 for the recovery process. APs can be inserted in any position in a path as needed. In the case of compensation, the process should only need to recover completed atomic, composite, or flow groups on the path which has been executed.

APRollback. APRollback invoked in a path in a if-else control structure will recover all completed groups in the process. For example, if APRollback is invoked at AP2 in Fig. 7, all completed groups before AP2 on the “Y” path will be recovered. Then groups before the if-else control structure will be recovered.

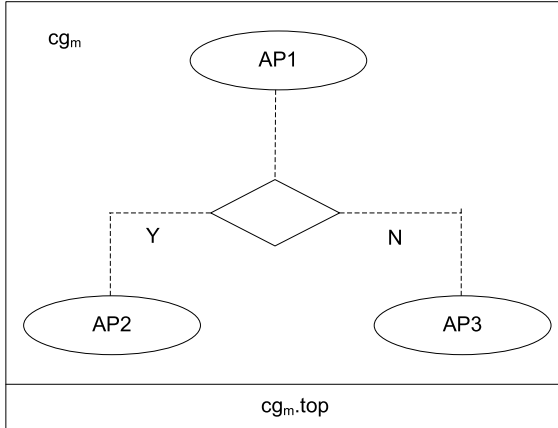


Fig. 7. An example of APs in an if-else control structure.

APRetry. In an if-else control structure, APRetry will recover the groups that are on the selected path and groups before the if-else control structure as needed. For example, if APRetry to AP1 is invoked at AP2 in Fig. 7, all completed groups before AP2 on the “Y” path will be recovered. Then groups that are before the if-else control structure and after AP1 will be recovered. The process resumes forward execution from AP1. The if-else control structure will be re-examined when the process reaches it.

APCC. If APCC is activated on a path in a control structure, a contingent procedure must be executed instead of the failed group. If a contingent procedure exists on the selected path and has been executed successfully, the process resumes forward execution. Otherwise, all completed groups on the selected path need to be recovered in order to search for another contingent procedure at the outer level. For example, if APCC is invoked at AP3 in Fig. 7, all completed groups before AP3 on the “N” path will be recovered. Then all groups in cg_m before the if-else control structure will be recovered. Finally, $cg_m.top$ will be executed.

4.3. The loop control structure and recovery semantics

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, ideally 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. However, if the execution of a loop control structure fails in a specific iteration, a question is raised as to whether the previous completed iterations need to be compensated or not before the loop control structure quits. In BPEL, if a fault is thrown inside a while loop activity, the compensation handler associated with the while loop may only execute once and then the while loop activity quits.⁴⁵

From the application point of view, if the iterations in a loop control structure have strong connections, the entire loop activity must have an all or nothing property; Otherwise if the iterations in a loop control structure are highly independent with each other, the failure of an iteration does not necessarily trigger the compensation of the previous completed iterations.

Because the recovery procedure in the AP model is based on the presence of APs in a process, APs inside a loop control structure can potentially complicate the recovery process. For example, in Fig. 8, APRetry is invoked at AP2 to backward recover the process to AP1 which is inside a loop control structure. Since AP1 inside the loop control structure might have been executed multiple times, it is unclear how many times the completed groups in the loop control structure need to be compensated. Furthermore, in an extreme case, AP1 in Fig. 8 may never be reached if the testing of the condition of the loop returns false directly. As a result, the APRetry action from an AP outside a loop to an AP inside a loop will unnecessarily complicate the recovery execution.

To prevent any inconsistent recovery in a loop control structure, all repeatable groups and APs in a loop must be embedded in the scope of a single composite group. As shown in Fig. 9, all repeatable groups and APs are embedded in cg_{mn} .

As discussed earlier in this section, if an iteration fails in the loop control structure, whether the previous completed iterations need to be compensated or not before the loop control structure quits depends on the application requirements. In the AP model, both scenarios are supported based on the logic of the contingency of the composite group in the loop control structure. For example, depending on the application requirements of Fig. 9, different logic can be set in the contingent procedure $cg_{mn}.top$.

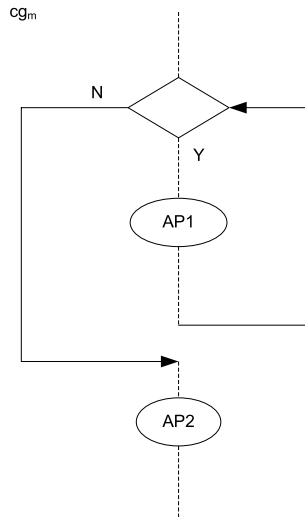


Fig. 8. Inconsistent APRetry in a loop control structure with APs.

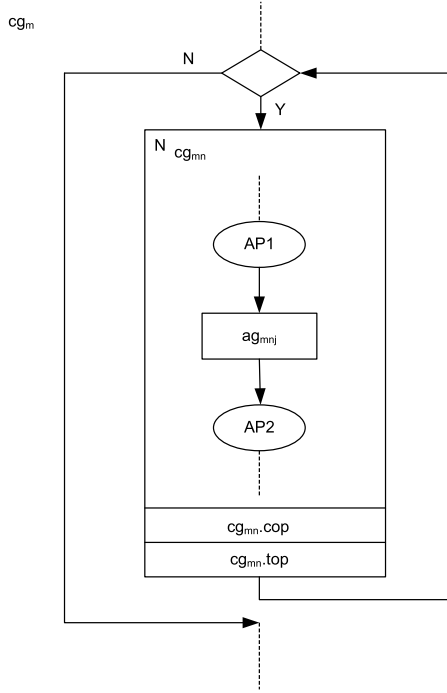


Fig. 9. A loop control structure with APs.

4.3.1. Exiting a loop with previous iterations compensated

As a default, if the composite group in the loop control structure has no contingent procedure, the scenario that the loop control structure quits with the previous iterations compensated is performed. To support this scenario, a **normal** contingency of the composite group, which is an alternative of the original composite group, is used. Different AP actions can be invoked in this first scenario:

APRollback. If APRollback is invoked in a loop, all groups that have finished in the current iteration will be compensated first. Second, the composite group in the loop will compensate itself the same number of times as it has been iterated. Then the APRollback will start recovery of all groups finished before the loop control structure. For example, in Fig. 9, if APRollback is invoked at AP2 in the third iteration, all finished groups before AP2 in the third iteration will be compensated first. After that, the shallow compensation $cg_{mn}.cop$ will run twice to compensate the first two iterations. Then all groups before the loop control structure will be recovered.

APRetry. In the AP model, since APRetry can only recover to an earlier AP at the same hierarchical level, the APRetry invoked in a loop control structure may only recover to an earlier AP which is also inside the loop. In addition, APRetry can only be performed in the current iteration. For example, if APRetry to AP1

is invoked at AP2 in Fig. 9, only \mathbf{ag}_{mnj} completed in the current iteration will be compensated. Then the current iteration will resume from AP1.

APCC. If APCC is invoked in a loop control structure, two situations may happen. One situation is that a contingent procedure is executed successfully in the current iteration. In this case, the loop control structure continues. For example, if \mathbf{ag}_{mnj} fails in the second iteration in Fig. 9, APCC is invoked. After compensating completed groups before \mathbf{ag}_{mnj} in the second iteration, the normal contingent procedure $\mathbf{cg}_{mn}.\mathbf{top}$ will be executed. After successfully executing $\mathbf{cg}_{mn}.\mathbf{top}$, the second iteration ends and the loop control structure continues. The other situation is that all possible contingent procedures in the current iteration fail or no contingent procedure in the current iteration is available. In this case, the loop control structure fails and the APCC will be propagated to the group at outer level. For example, if \mathbf{ag}_{mnj} fails in the second iteration in Fig. 9, APCC is invoked. After compensating completed groups before \mathbf{ag}_{mnj} in the second iteration, the normal contingent procedure $\mathbf{cg}_{mn}.\mathbf{top}$ will be executed. However, suppose $\mathbf{cg}_{mn}.\mathbf{top}$ fails also. Then the shallow compensation $\mathbf{cg}_{mn}.\mathbf{cop}$ needs to execute once to compensate the first iteration. After that, the APCC is propagated to the composite group \mathbf{cg}_m .

4.3.2. *Exiting a loop without previous iterations compensated*

A second scenario is that the loop control structure quits without the previous iterations compensated. In this case, a **break** contingency of the composite group is used, which simply breaks the loop control structure. A **break** contingency is assumed to be always successful since a **break** contingency only performs the break of the loop control structure. Different AP actions can also be invoked in the second scenario:

APRollback. APRollback performs the same as in the first scenario.

APRetry. APRetry performs the same as in the first scenario.

APCC. If APCC is invoked in a loop control structure, the **break** contingency of the composite group will be invoked to break the loop control structure without compensating the previous completed iterations. For example, if \mathbf{ag}_{mnj} fails in the second iteration in Fig. 9, APCC is invoked. After compensating completed groups before \mathbf{ag}_{mnj} in the second iteration, the **break** contingent procedure $\mathbf{cg}_{mn}.\mathbf{top}$ will be executed to break the loop control structure without compensating the previous completed iterations. After that, the APCC mode quits and the composite group \mathbf{cg}_m continues.

4.4. *Case study*

This section introduces a course enrollment case study to illustrate the use of the AP model to define processes. The course enrollment application, conducted by the students, the university and the financial institutes, contains typical business

processes that describe the activities to register for a single course. Figures 10(a)–10(d) presents a graphical view of the course enrollment process.

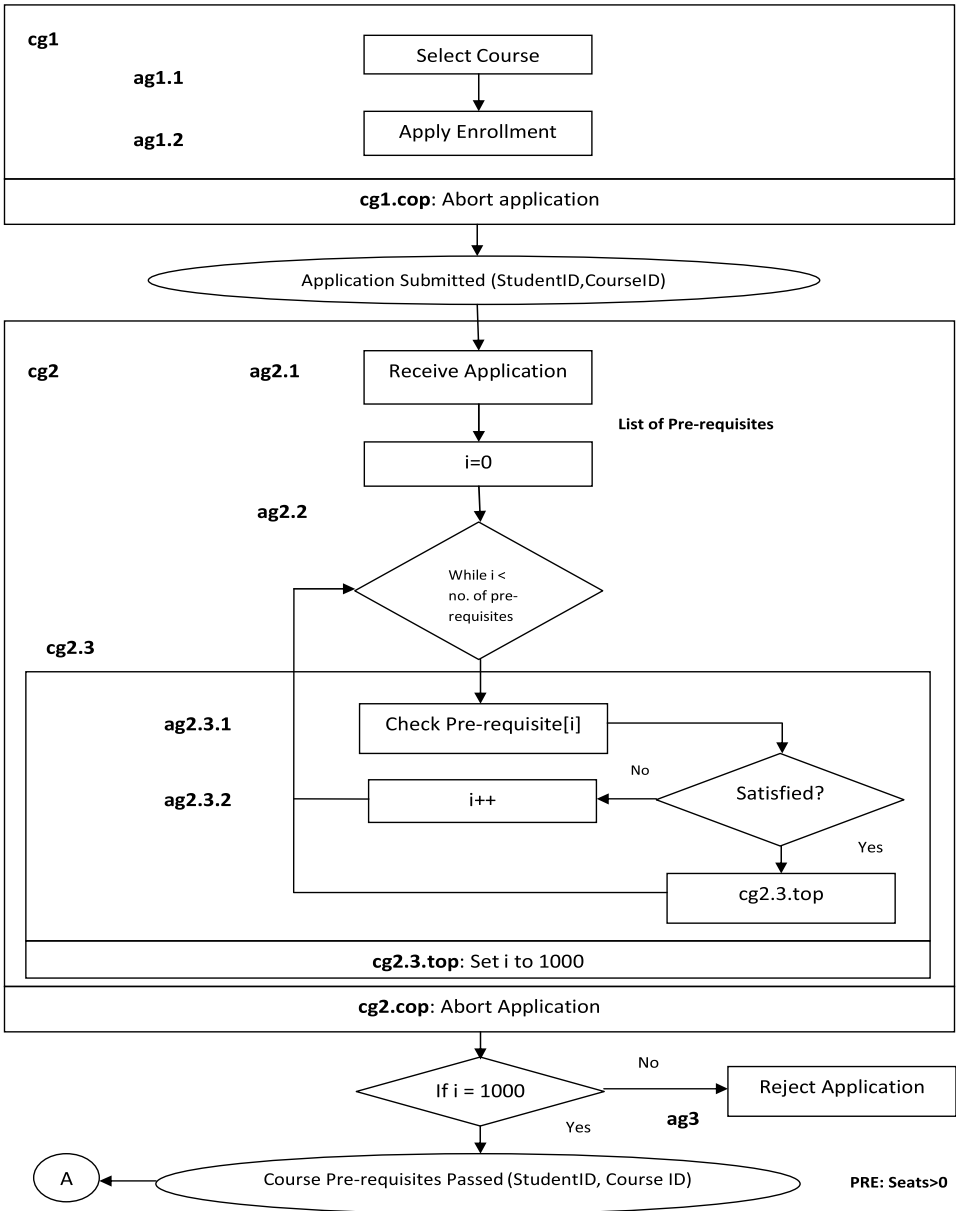
The process starts when a student selects a course and applies the enrollment. These two atomic groups constitute the composite group cg_1 in Fig. 10(a). The cg_1 has a compensation to abort the application. The process then reaches the first AP which is “Application Submitted”. At this AP, the application is submitted to the university and the process stores the `StudentID` and the `CourseID` into the database.

The composite group cg_2 consists of a loop control structure where a list of pre-requisites for the selected course is checked. To continue the application, one pre-requisite must be satisfied. If none of the pre-requisites are satisfied, the application is rejected. The loop control structure in the cg_2 presents the scenario that the loop control structure quits without compensating the previous iterations as only one pre-requisite is enough to accept the application. A `break` contingency $cg_{2.3.top}$ is defined to assign a high value to the iterating variable. To quit the loop when a pre-requisite is passed, the $cg_{2.3.top}$ is invoked so that the loop quits at the next iteration since the iterating variable has been set with a high value. The cg_2 also has a compensation that aborts the application. If a pre-requisite is satisfied, the second AP, which is “Course Pre-requisites Passed” is reached. A pre-condition is checked at this AP to ensure that the number of available seats of the selected course is greater than 0.

If the seats for the selected course are available, a flow group fg_4 in Fig. 10(b) is then executed to register the student into the university system. Two threads are executing in parallel in fg_4 . One thread $cg_{4.1}$ accepts the application, decreases the seat count and updates the transcript. An AP, “Records Updated”, is reached after the $cg_{4.1}$. At this AP, the process stores the `StudentID`, the `CourseID` and the `SeatCount` in the database. The other thread $cg_{4.2}$ adds the student to video access list and calculates the tuition amount. After execution of the $cg_{4.2}$, the process reaches the AP which is “Course Fees Calculated”. At this AP, the process stores the calculated `Amount` to the database. If any operations fail in any of these two threads, they both perform compensation for the completed operations and then execute the contingency $fg_4.top$ to register the student for an alternative distance class. After execution of the fg_4 , the process reaches the AP which is “Student Registered”. Before execution proceeds further, the process checks if the records have been updated using a `Post` condition.

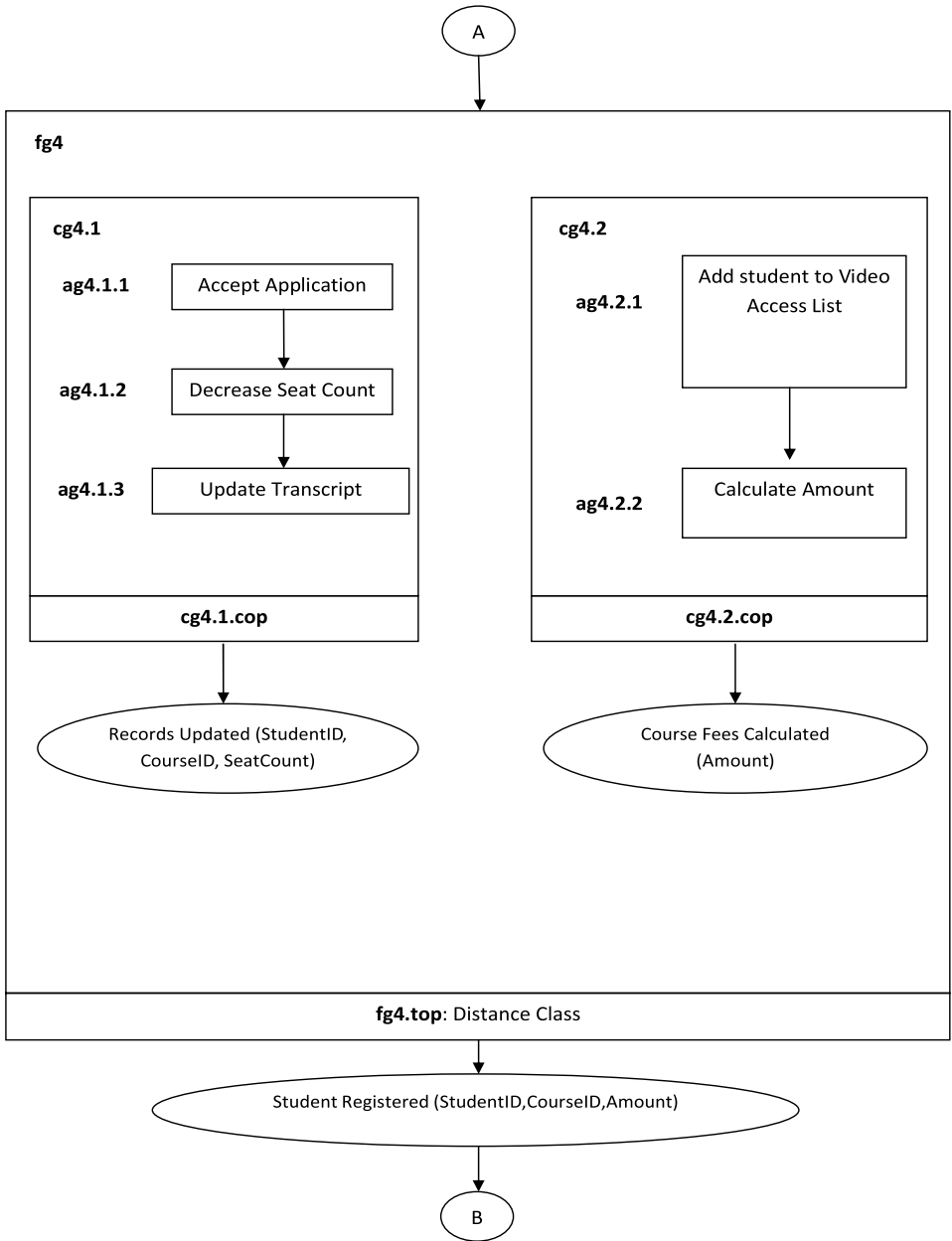
The composite group cg_5 in Fig. 10(c) consists of an if-else control structure. In this group, the process checks if any financial assist is available to the student account. If the scholarship is available, the composite group $cg_{5.1}$ applies the scholarship to the student account to calculate new amount, update the account, sends bill and requests payment. If scholarship is not applicable, the process checks if the student has departmental funding. If yes, the composite group $cg_{5.2}$ is executed to apply the departmental funding through operations: calculated, the account is updated, a bill is sent and payment is requested. If the student has no funding then the process directly sends the bill to the student by executing the composite group

cg0



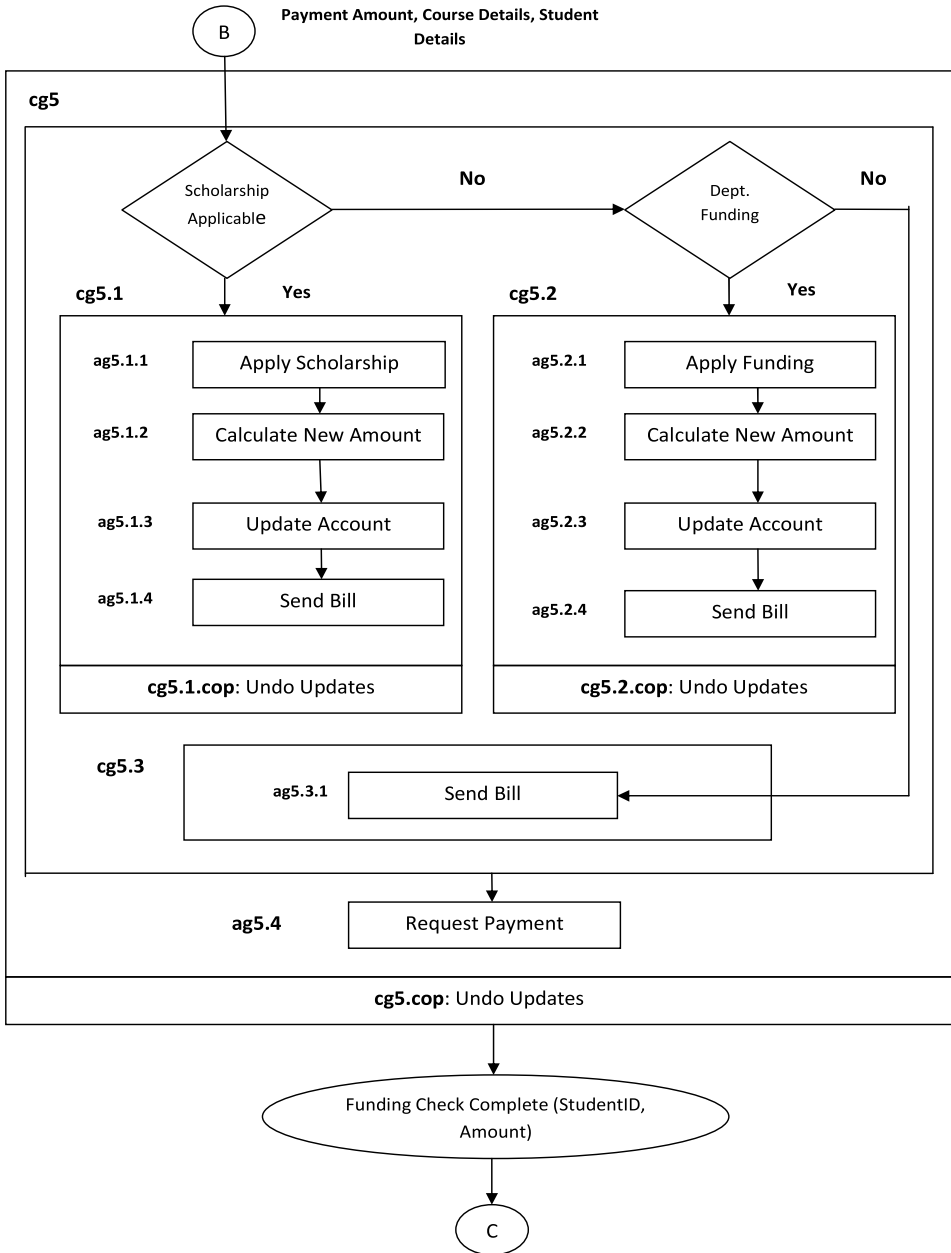
(a)

Fig. 10. Course enrollment process.



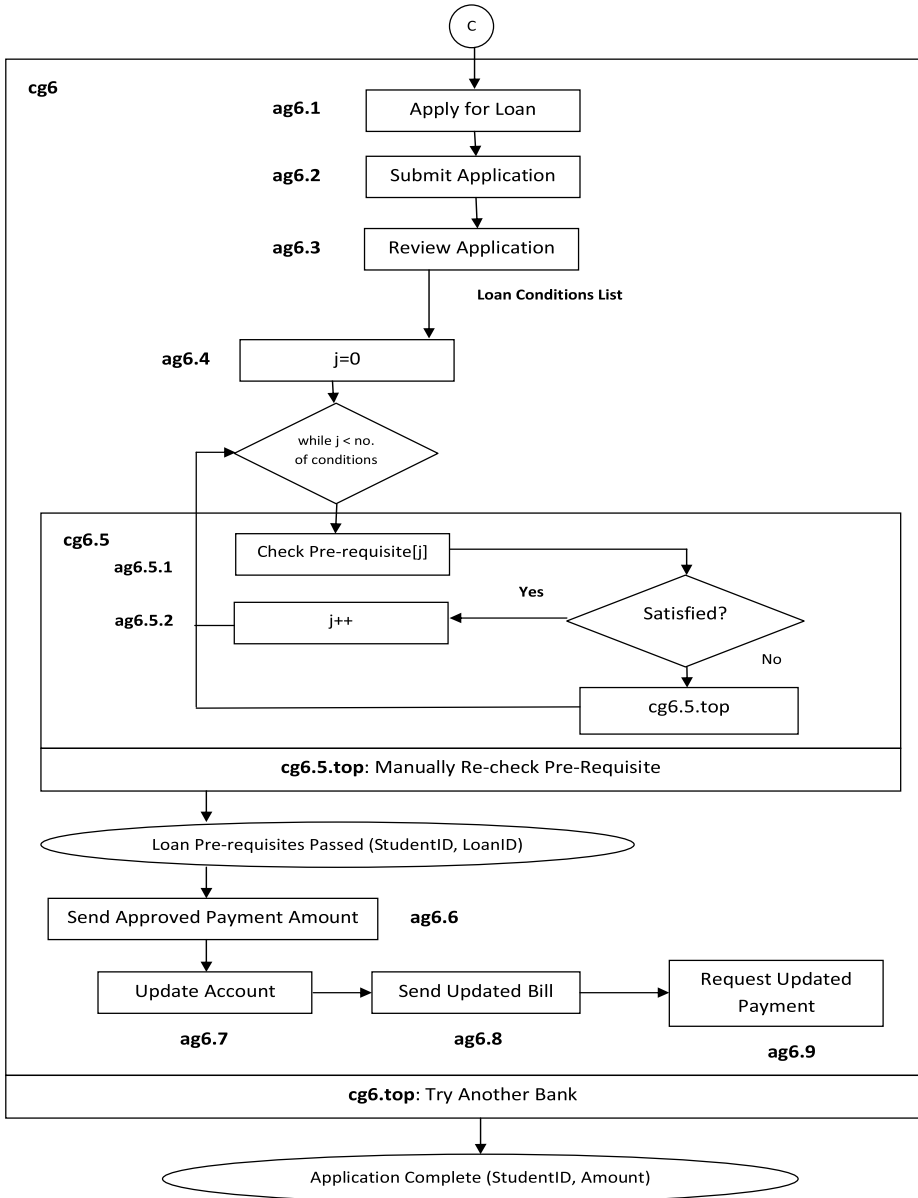
(b)

Fig. 10. (Continued)



(c)

Fig. 10. (Continued)



(d)

Fig. 10. (Continued)

cg5.3. Finally, after the if-else control structure, the atomic group ag5.4 is executed to request payment for the remaining balance in the student account. This composite group cg5 has a compensation cg5.cop to undo updates performed in the group. A “Funding Check Complete” is reached after executing this group. At this point,

the financial assist, if available, is applied to the student account and calculation of new tuition amount is completed.

The process then starts the execution of the composite group cg_6 in Fig. 10(d) to apply for a loan. After operations *Apply for Loan*, *Submit Application*, and *Review Application*, a list of the pre-requisites of the application must be checked. A loan is approved for the student only if the student passes all of the loan conditions. A loop control structure that quits with previous iterations compensated is used to test the scenario. The loop control structure iterates the composite group $cg_{6.5}$ to check each pre-requisite. If a single pre-requisite is not met, the contingency $cg_{6.5.top}$ is invoked to manually re-check the pre-requisite again. If a single pre-requisite fails after trying the contingency $cg_{6.5.top}$, the loop control structure here compensates. If the pre-requisites have passed, an AP which is “Loan Pre-requisites Passed” is reached. The process continues to finish the loan application through the atomic groups $ag_{6.6}$ to $ag_{6.9}$. If any step in the group cg_6 fails, the APCC mode is activated. After compensating the cg_6 , the contingency $cg_{6.top}$ is invoked to try apply the loan from another bank. If all the operations execute successfully, the process reaches the last AP in the workflow i.e. “Application Complete”. At this point the process ends.

5. Formal Specification and Verification of the AP Model








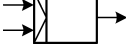
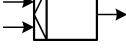
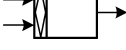
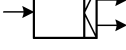
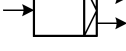
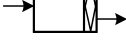
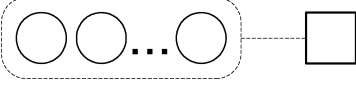
The execution and recovery semantics of the AP model for sequential control flow have been formalized using Petri Nets.^{9,46} In this research, YAWL¹² has been used to precisely describe and verify the execution and recovery semantics of the complete AP model. A set of mapping rules that are used to transform the AP model to YAWL nets have been defined. The soundness property of the execution recovery semantics of the AP model is also verified in the YAWL engine. By mapping the AP model to the YAWL model, the execution and recovery semantics can be tested and verified. In general, YAWL provides stronger and more formal representations for workflow patterns, especially for complex control-flow and the use of resources.

5.1. Overview of YAWL

YAWL is inspired by Petri Nets, but it is not a simple extension of Petri Nets. YAWL has its own symbols and independent semantics. The symbols used in a YAWL net are shown in Table 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 a YAWL net, six join and split control constructs may be associated with each task:

- 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 not be enabled at any future time with the current marking continuing to be fired.
- XOR-join — A task is invoked when only one of the incoming arcs has been enabled.

Table 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

- 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.

5.2. General approach

Several general rules of mapping the AP model to the YAWL model are as follows:

- Since each YAWL net has one unique input and output condition, each activity mapped to a YAWL net starts with an `Initialize` task and ends with a `Finalize` task.
- Since YAWL introduces six join and split control constructs, no other conditions except the input and output are needed in the mapping.
- In the mapping, each YAWL net has a net variable which stores a list of important parameters. If a YAWL net includes decompositions, for each decomposition, the YAWL net creates a net variable that matches the net variable in the decomposition.
- If a task has more than one outgoing arc, each arc must have a predicate. The selection of the outgoing arc is determined by examining of the predicate of each arc.

The verification of a YAWL net mapped from an activity of the AP model focuses on the examination of the soundness property. A YAWL net is sound if and only if the following requirements are met:

- (1) All instances of the net must eventually terminate.
- (2) There must be exactly one token at the end place when an instance terminates.
- (3) Any tasks in the net may be executed in some instances.

5.3. Mapping basic activities of the AP model to YAWL

There are two basic activities in the AP model: atomic group and AP. Sections 5.3.1 and 5.3.2 present the generic mappings of these two activities to YAWL nets.

5.3.1. Generic YAWL net of an atomic group

The atomic group is the basic executable entity in the AP model. A net variable `ag` is defined with an atomic group in YAWL. Seven parameters are stored in the `ag`:

- `Succeed(boolean)`: True if the atomic group succeeds, otherwise false.
- `Top(boolean)`: True if the contingency of the atomic group exists, otherwise false.
- `TopSucceed(boolean)`: True if contingency of the atomic group succeeds, otherwise false.
- `Cop(boolean)`: True if the compensation of the atomic group exists, otherwise false.
- `CopSucceed(boolean)`: True if the compensation of the atomic group succeeds, otherwise false.
- `Critical(boolean)`: True if the atomic group is critical, otherwise false.
- `Status(String)`: A string indicates the status of the atomic group.

Figure 11 shows the YAWL net of the atomic group in the AP model. After finishing task `Initialize`, the atomic group fires task `Running` to execute. Depending on the examination of the predicate on each outgoing arc of task `Running`, the thread of control may be passed to different branches. If task `Running` succeeds, the token is passed to task `Successful`. If task `Running` fails and the contingency exists, task `T-Running` fires to execute the contingency. If task `Running` fails and no contingency exists, task `US_APCC` fires. Similarly, if task `T-Running` fires, depending on the execution result, either task `Successful` or `US_APCC` fires. The atomic group finishes by firing task `Finalize`.

The compensation of an atomic group in YAWL is shown in Fig. 12. After initialization of the compensation by task `Initialize`, different execution paths may be invoked depending on the evaluation of the predicates. If the atomic group is non-critical, the YAWL net finishes by firing task `Finalize` directly. If the atomic group is critical and compensation exists, the task `Cop_Running` fires to execute the compensation. If the atomic group is critical and no compensation exists, the task `Human_Activity` fires to manually compensate the group. The task `Cop_Running` also has two possible outgoing arcs. If the compensation succeeds, task `Finalize` is enabled. If the compensation fails, task `Human_Activity` fires. Since task `Human_Activity` guarantees to compensate the atomic group successfully, task `Finalize` will be enabled after completing task `Human_Activity`.

The atomic group in YAWL satisfies the soundness property as shown at the bottom in both Figs. 11 and 12.

When an atomic group is executing in the YAWL engine, the net variable `ag` will be updated once after finishing a task. A predicate on an arc is evaluated based on the values of the parameters in the net variable. For example, Fig. 13 shows the predicates on outgoing arcs of task `Running`. The target task `Successful` is enabled if the parameter `Succeed` is true. If the parameter `Succeed` is false and the parameter `Top` is true, the predicate of the target task `T-Running` evaluates to true. In the YAWL net, for XOR-split, the predicates are evaluated in a specified sequence and once a predicate evaluates to true, then the thread of control is passed to the corresponding arc without any further evaluations of the remaining un-evaluated predicates. Therefore, in Fig. 13, if the evaluations of the first two predicates return false, the task `US_APCC` is enabled by default. Figure 14 presents an example of executing an atomic group in the YAWL engine when task `Running` is enabled. If task `Running` completes with the values of the parameters as shown in Fig. 14, task `Successful` will be enabled next since the parameter `Succeed` is true.

5.3.2. Generic YAWL net of an AP

An AP is also a basic activity in the AP model. A net variable `ap` is defined with an atomic group in YAWL. Seven parameters are stored in the `ap`:

- `Post(boolean)`: True if the post-condition exists, otherwise false.
- `Pre(boolean)`: True if the pre-condition exists, otherwise false.

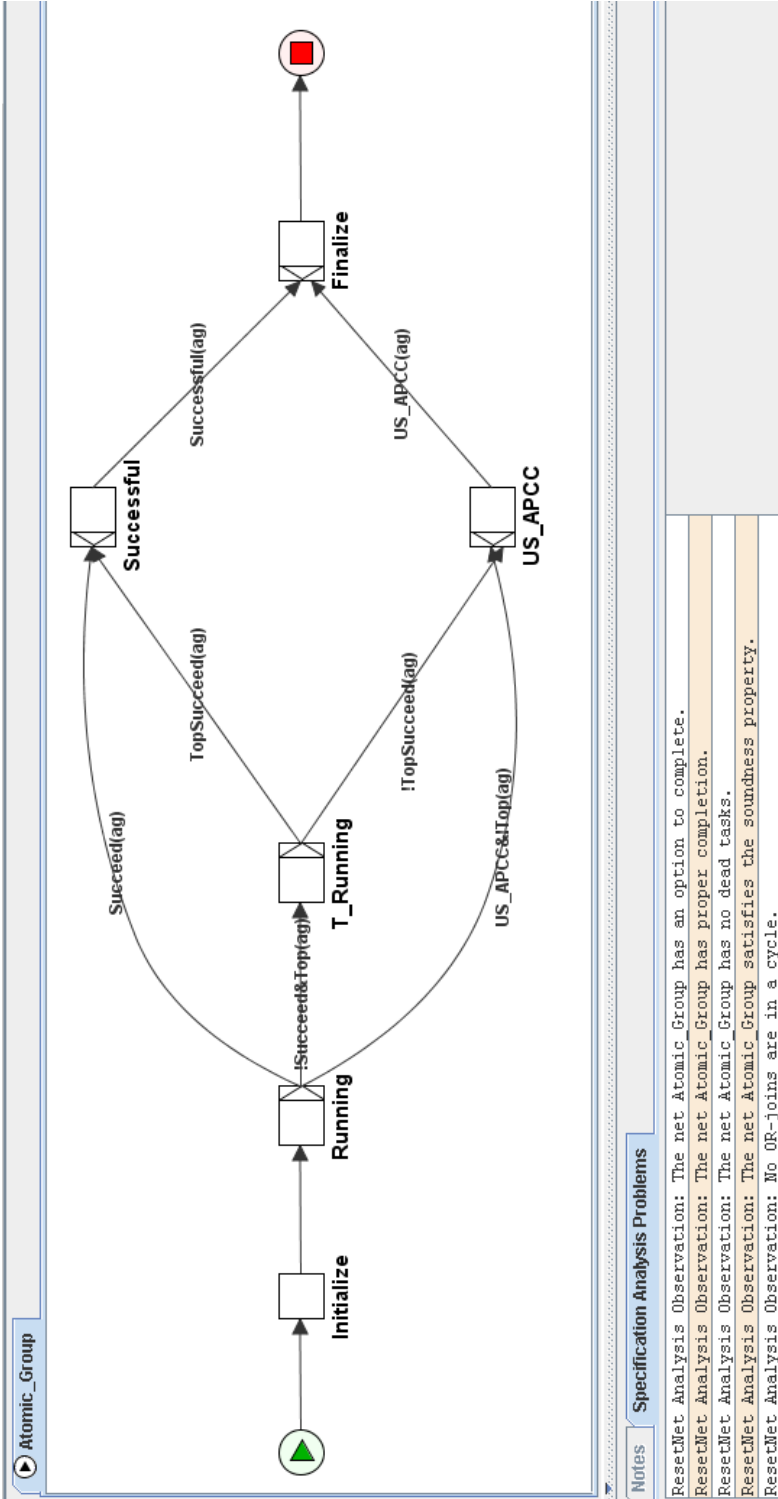


Fig. 11. Atomic group in YAWL.

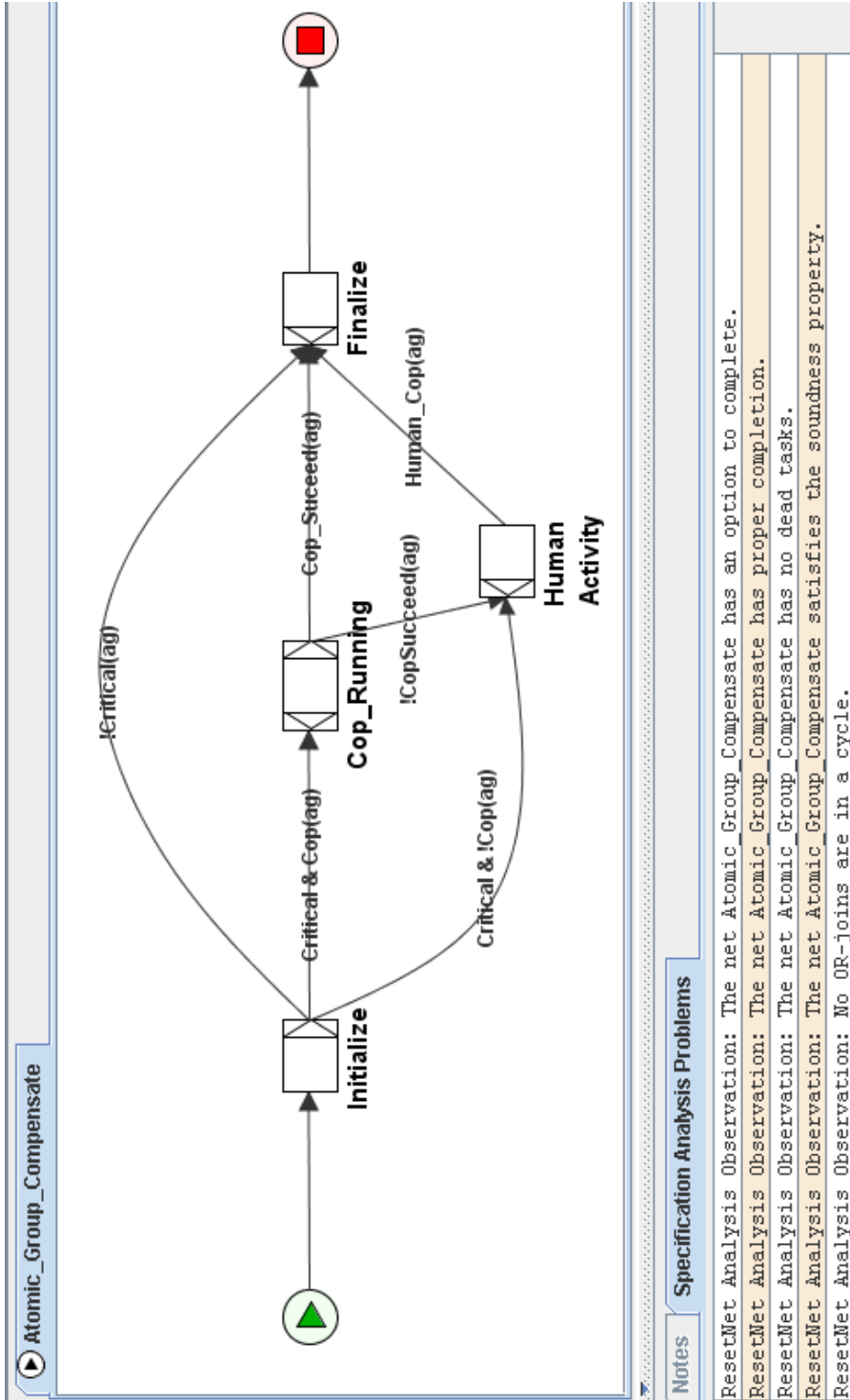


Fig. 12. Compensation of an atomic group in YAWL.

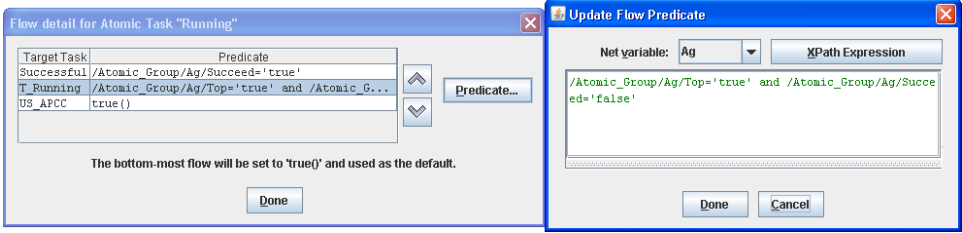


Fig. 13. Predicates on outgoing arcs of task running.

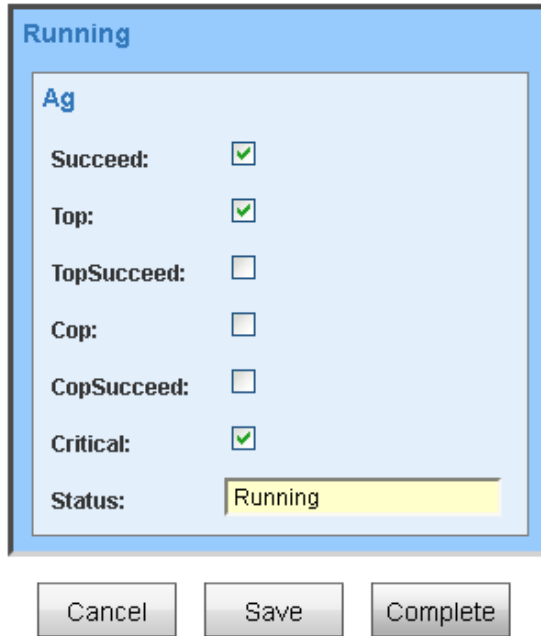


Fig. 14. Task Running in enabled in the YAWL engine.

- Post_Fail(boolean): True if the evaluation of the post condition returns a violation, otherwise false.
- Pre_Fail(boolean): True if the evaluation of the pre condition returns a violation, otherwise false.
- Post_Fail_F(boolean): True if the evaluation of the post condition has returned a violation once, otherwise false.
- Pre_Fail_F(boolean): True if the evaluation of the pre condition has returned a violation once, otherwise false.
- RecoveryAction(String): A string indicates the AP action that is invoked in the AP.
- Status(String): A string indicates the status of the AP.

Figure 15 represents a generic AP in YAWL. All execution semantics in an AP are supported in Fig. 15. After finishing task `Initialize`, the following cases can occur:

- Post and pre conditions both exist:
 - *Post and pre conditions are both satisfied*: Tasks `Post-Checking` and `Pre-Checking` pass successively, indicating both post and pre condition have passed. Task `AP_Pass` is then enabled.
 - *Post condition violated*: Task `Post-Checking` returns a violation. Depending on whether it is the first time violation or not, either task `Post_Fail_F` or `Post_Fail_S` is enabled. After checking the IR, one of the tasks `AP_RB`, `AP_Retry` and `AP_CC` is enabled.
 - *Post condition passed and pre condition violated*: Task `Post-Checking` passes. Tasks `Pre-Checking` then returns a violation. Depending on whether it is the first time violation or not, either task `Pre_Fail_F` or `Pre_Fail_S` is enabled. After checking the IR, one of the tasks `AP_RB`, `AP_Retry` and `AP_CC` is enabled.
- Only post condition exists:
 - *Post condition is satisfied*: Task `Post-Checking` passes. Task `AP_Pass` is then enabled.
 - *Post condition is violated*: Task `Post-Checking` returns a violation. Depending on whether it is the first time violation or not, either task `Post_Fail_F` or `Post_Fail_S` is enabled. After checking the IR, one of the tasks `AP_RB`, `AP_Retry` and `AP_CC` is enabled.
- Only pre condition exists:
 - *Pre condition is satisfied*: Task `Pre-Checking` passes. Task `AP_Pass` is then enabled.
 - *Pre condition is violated*: Task `Pre-Checking` returns a violation. Depending on whether it is the first time violation or not, either task `Pre_Fail_F` or `Pre_Fail_S` is enabled. After checking the IR, one of the tasks `AP_RB`, `AP_Retry` and `AP_CC` is enabled.
- Post and pre condition do not exist: Task `AP_Pass` is enabled.

Task `Finalize` completes the execution of an AP. The AP in YAWL satisfies the soundness property as shown in Fig. 15.

Figure 16 shows the predicates on outgoing arcs of task `Post-Checking`. After completing task `Post-Checking`, if the parameter `Post_Fail` is false and the parameter `Pre` is true, task `Pre-Checking` is enabled. If the parameter `Post_Fail` is true and parameter `Post_Fail_F` is false, task `Post_Fail_F` is enabled. If the parameter `Post_Fail` is true and parameter `Post_Fail_S` is true, task `Post_Fail_S` is enabled. If none of the first three predicates evaluate to true, task `AP_Pass` is enabled as default. Figure 17 presents an example of executing an AP in the YAWL engine when task `Post-Checking` is enabled. If task `Post-Checking` completes with the values of the parameters as shown in Fig. 17, task `Pre-Checking` will be enabled next since the parameter `Post_Fail` is false and the parameter `Pre` is true.

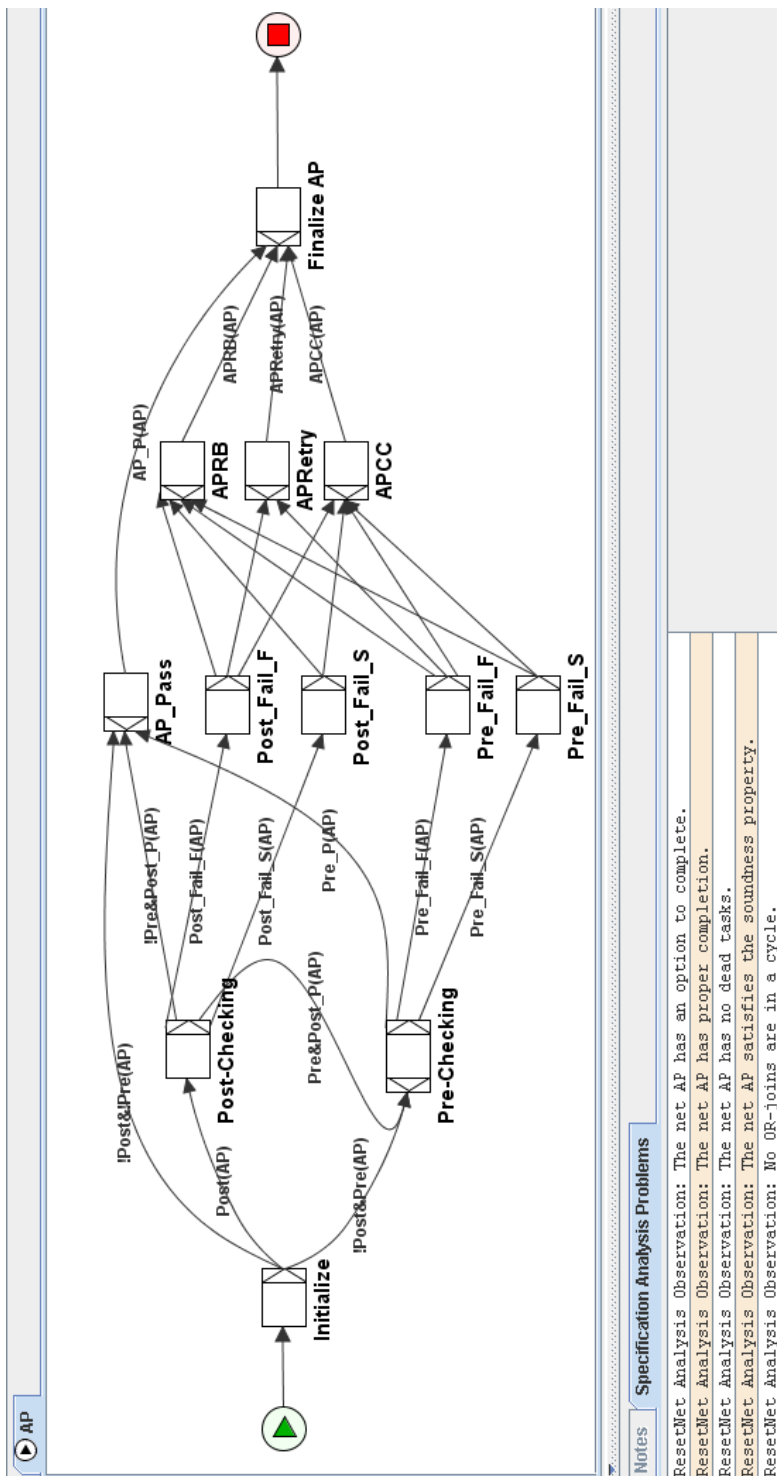


Fig. 15. AP in YAWL.

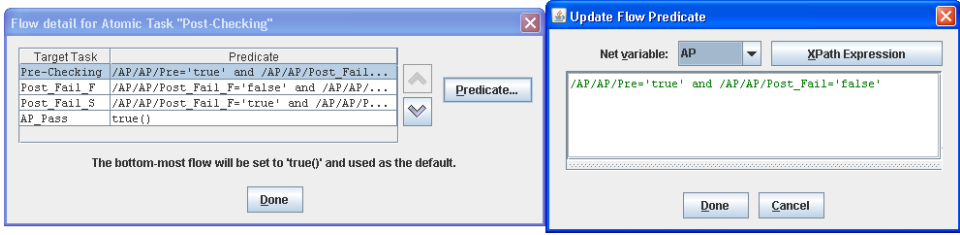


Fig. 16. Predicates on outgoing arcs of task Post-Checking.

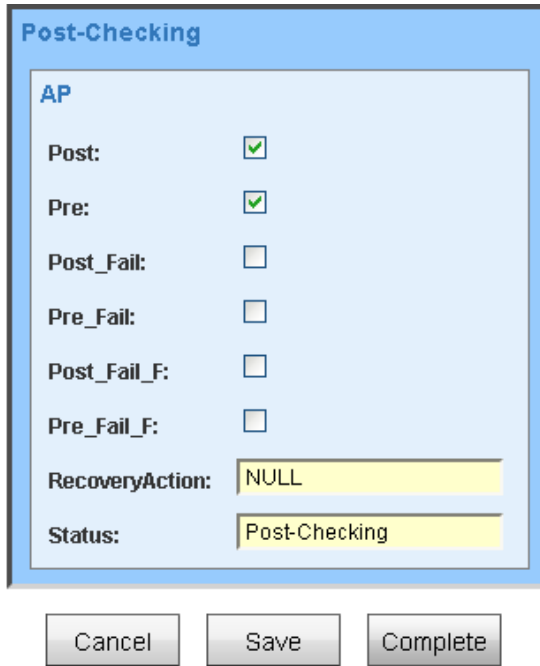


Fig. 17. Task Post-Checking in enabled in the YAWL engine.

5.4. Mapping composite groups of the AP model to YAWL

This section gives the rules of mapping a composite group to a YAWL net. Similarly to the AP model, a process in YAWL is also constructed hierarchically. In a YAWL net, a composite task refers to another YAWL net at a lower level in the hierarchy. Therefore, the composite groups in the AP model can be mapped hierarchically to YAWL nets. Since the composite groups do not have any generic structures in the AP model, the YAWL net presented in this section is modeled by a typical example that represents the characteristics of the composite groups in the AP model. In addition, the YAWL net presented in this section satisfies the soundness property.

Any specific composite groups in the AP model can be transformed into YAWL nets by imitating the mapping presented in this section.

A composite group is composed by two or more atomic and/or composite groups and APs. A composite group can also have optional compensation and contingency procedures. A typical example of a composite group, shown in Fig. 18, is used to illustrate the rules of mapping a composite group to a YAWL net.

As shown in Fig. 18, the composite group cg_1 contains two composite groups cg_{11} and cg_{12} , one atomic group ag_{13} , and two APs AP1 and AP2. If an IR is violated at AP1, the action APCC will be invoked. If an IR is violated at AP2, either the action APRetry or APRollback will be invoked.

The complete execution semantics of cg_1 is modeled in a YAWL net presented in Fig. 19. The YAWL net cg_1 not only describes the semantics of normal forward execution, but also represents the semantics of possible backward recoveries. In the YAWL net cg_1 , the composite groups cg_{11} and cg_{12} are represented as two composite tasks which refer to two YAWL nets cg_{11} and cg_{12} at a lower level. The YAWL nets cg_{11} and cg_{12} at a lower level, which are constructed following the same rules of mapping a composite group to a YAWL net, represent the execution semantics in the composite groups cg_{11} and cg_{12} , respectively. The APs AP1 and AP2 in the YAWL net cg_1 are depicted as two composite tasks AP1 and AP2 which refer to two YAWL nets AP1 and AP2 at a lower level. Both YAWL nets AP1 and AP2 at a lower level have the same structure as shown in Fig. 15. Similarly, the atomic group ag_{13} in the YAWL net cg_1 is also a composite task that refers to a YAWL net ag_{13} with the structure in Fig. 11 at a lower level.

A net variable cg_1 is created in the YAWL net cg_1 with the parameters below:

- Top(boolean): True if the contingency of the composite group exists, otherwise false.
- TopSucceed(boolean): True if contingency of the composite group succeeds, otherwise false.
- Shallow(boolean): True if the shallow compensation of the composite group exists, otherwise false.

cg_1

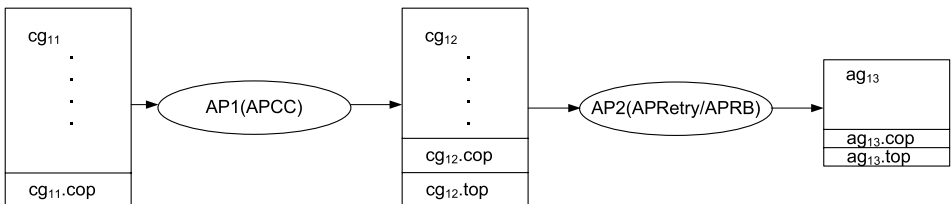


Fig. 18. An example of a composite group.

- CopSucceed(boolean): True if the shallow or deep compensation of the composite group succeeds, otherwise false.
- Status(String): A string indicates the status of the composite group.

In the YAWL net cg_1 , other than the net variable cg_1 , five more net variables (cg_{11} , AP1, cg_{12} , AP2, ag_{13}) are created associated with the five atomic/composite groups and APs. For each net variable associated with a composite task, the values of the parameters in the net variable are passed to the net variable in the YAWL net at a lower level before executing the task, and duplicated from the net variable in the YAWL net at a lower level after finishing the task. For example, in the YAWL net cg_1 , the values of the parameters in the net variable cg_{11} are passed to the net variable cg_{11} in the YAWL net cg_{11} at a lower level before executing task cg_{11} , and duplicated from the net variable cg_{11} in the YAWL net cg_{11} at a lower level after finishing task cg_{11} . In Fig. 19, if a task has more than one outgoing arcs, each arc is marked with a predicate with respect to the net variables.

The YAWL net cg_1 starts with task Initialize and ends with task Finalize. During the execution, different cases may happen:

- No error occurred during the execution: All composite tasks cg_{11} , AP1, cg_{12} , AP2, ag_{13} are finished normally. Task Successful then is enabled.
- Error occurred during the execution:
 - * Error returned after task cg_{11} :
 - APRB(cg_{11}): Task APRB is enabled.
 - US_APCC(cg_{11}): Since there is no contingency of cg_{11} , task US_APCC is enabled.
 - * Error returned after task AP1:
 - APCC(AP1): Task cop_cg_{11} fires to compensate the (cg_{11}). Then task US_APCC is enabled.
 - * Error returned after task cg_{12} :
 - APRB(cg_{12}): Task cop_cg_{11} fires to compensate the (cg_{11}). Then task APRB is enabled.
 - US_APCC(cg_{12}): Since there is an AP1 immediately before cg_{12} , before trying the contingency of cg_{12} , task AP1 fires to re-check the pre-condition. If AP1 passes, task top_cg_{12} fires to execute the contingency of cg_{12} . If the contingency succeeds, task AP2 is enabled. Otherwise, task cop_cg_{11} fires to compensate (cg_{11}). Then, task US_APCC is enabled.
 - * Error returned after task AP2:
 - APRetry(AP2): Task cop_cg_{12} fires to compensate the (cg_{12}). Then task AP1 is enabled to start the retry.
 - APRB(AP2): Tasks cop_cg_{12} and cop_cg_{11} fire successively to compensate cg_{11} and cg_{12} . Then task APRB is enabled.

* Error returned after task `ag13`:

- `US_APCC(AP1)`: Tasks `cop_cg12` and `cop_cg11` fire successively to compensate `cg11` and `cg12`. Then task `US_APCC` is enabled.

Figure 20 shows the YAWL net of the compensation of `cg1`. If the shallow compensation of `cg1` exists, task `Shallow Compensate cg1` fires. If the shallow compensation execute successfully, task `Shallow Compensate Successful` is enabled. If the shallow compensation of `cg1` fails or does not exist, the deep compensation executes by firing `cop_ag13`, `cop_cg12`, and `cop_cg11` in succession. Finally, task `Successful Compensate cg1` fires to enable task `Finalize`.

5.5. Mapping complex control structures of the AP model to YAWL

The AP model supports the executions of atomic/composite groups in parallel, if-else and loop control structures. For each complex control structure, a YAWL net can be created to represent the execution semantics in the complex control structure. In a YAWL net of a complex control structure, similar to the YAWL net of a composite group, the executable entities are represented by composite tasks that refer to YAWL nets at a lower level. For each composite task in a YAWL net, a net variable associated with the composite task is needed.

5.5.1. YAWL net of the parallel control structure

The flow group introduced in Sec. 4.1 presents the execution of the parallel control structure. In a flow group, two or more threads run concurrently. A flow group executes successfully if all threads succeed. A YAWL net `fg` of a flow group with two threads is presented in Fig. 21.

A net variable `fg` is defined in the YAWL net `fg`:

- `Top(boolean)`: True if the contingency of the flow group exists, otherwise false.
- `TopSucceed(boolean)`: True if contingency of the flow group succeeds, otherwise false.
- `Shallow(boolean)`: True if the shallow compensation of the flow group exists, otherwise false.
- `CopSucceed(boolean)`: True if the shallow or deep compensation of the flow group succeeds, otherwise false.
- `ErrorFlag(boolean)`: True if any thread returns an error, otherwise false.
- `Status(String)`: A string indicates the status of the flow group.

In Fig. 21, after finishing task `Initialize`, tasks `cg11` and `cg12` are enabled simultaneously. Take the upper thread as an example, if tasks `cg11` and `AP11` both finish successfully, task `Finalize Thread 1` is enabled to wait for other threads to complete.

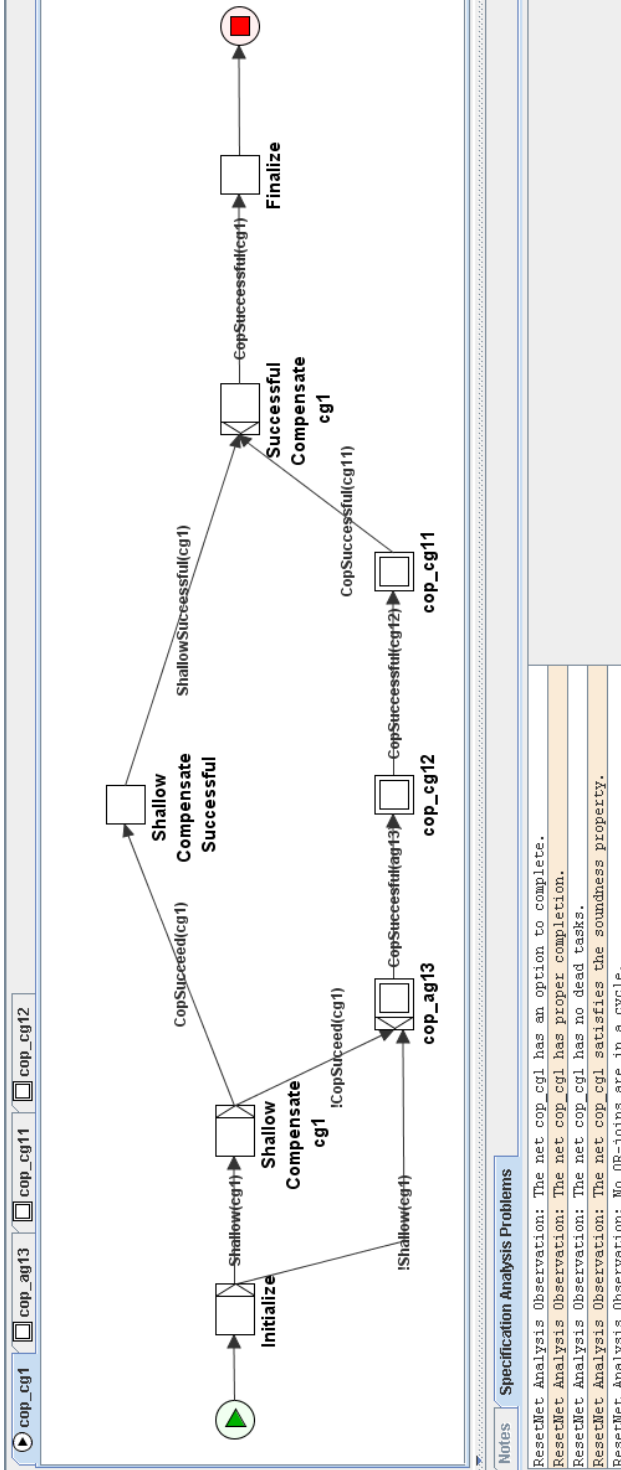


Fig. 20. The compensation of the composite group example in YAWL.

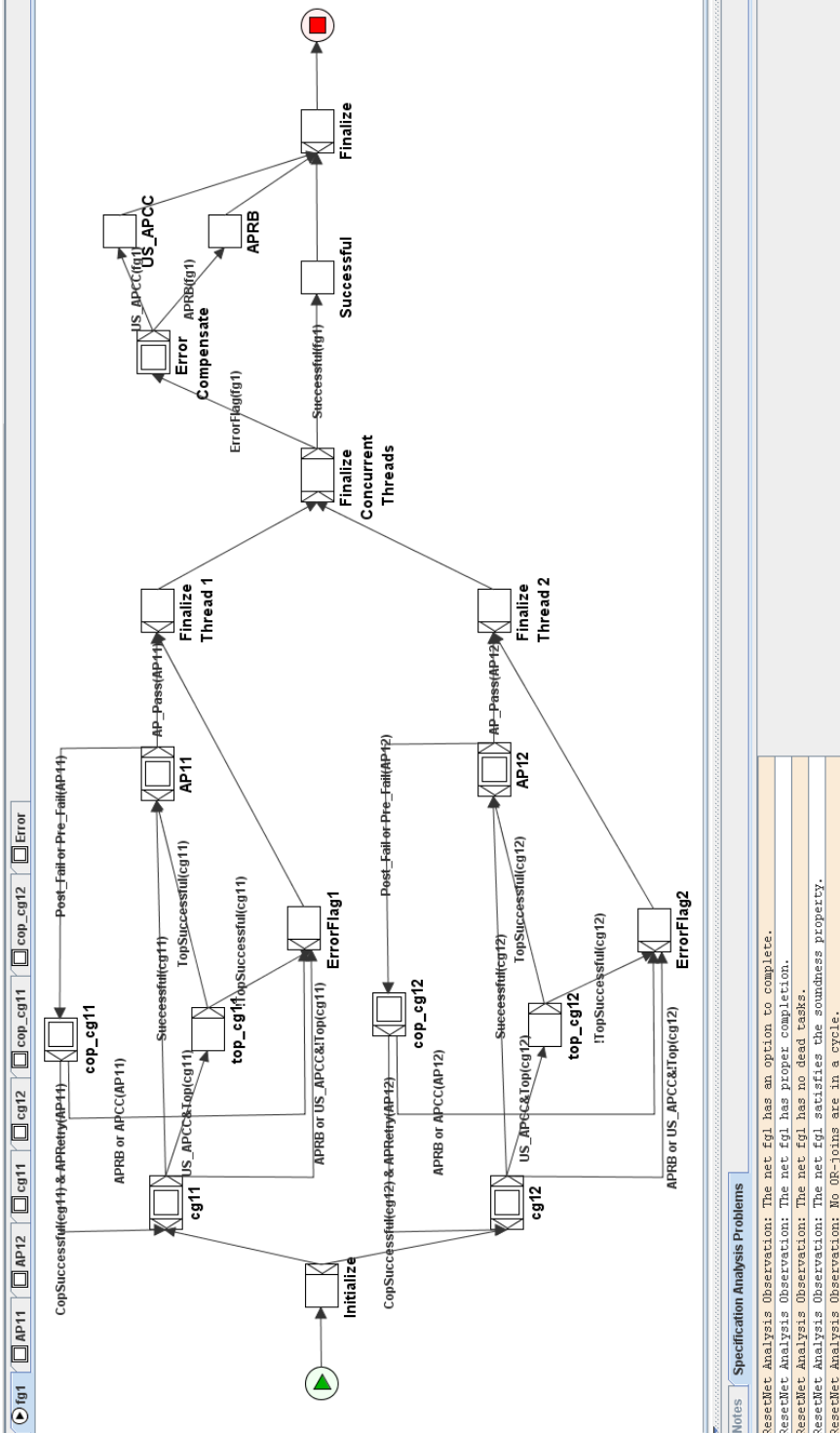


Fig. 21. A flow group with two threads in YAWL.

If an error occurs in the thread, different cases may happen:

- Error returned after task cg_{11} :
 - $APRB(cg_{11})$: Task $ErrorFlag1$ fires to update the parameter $ErrorFlag$ to true and the parameter $Status$ to “ARRollback”. Then task $Finalize Thread 1$ is enabled to wait for other threads to complete.
 - $US_APCC(cg_{11})$: If the contingency of cg_{11} is available, task top_cg_{11} fires to execute the contingency of cg_{11} . If the contingency succeeds, task $AP11$ is enabled. Otherwise, task $ErrorFlag1$ fires to update the parameter $ErrorFlag$ to true and the parameter $Status$ to “US_APCC”. If the contingency of cg_{11} is unavailable, task $ErrorFlag1$ fires directly to update the parameter $ErrorFlag$ to true and the parameter $Status$ to “US_APCC”. Finally, task $Finalize Thread 1$ is enabled to wait for other threads to complete.
- Error returned after task $AP11$:
 - $APRB(AP11)$: Task Cop_cg_{11} fires to compensate cg_{11} . Then task $ErrorFlag1$ fires to update the parameter $ErrorFlag$ to true and the parameter $Status$ to “ARRollback”. After that, task $Finalize Thread 1$ is enabled to wait for other threads to complete.
 - $APRetry(AP11)$: Task Cop_cg_{11} fires to compensate cg_{11} . Then task cg_{11} is enable to start re-try.
 - $US_APCC(AP11)$: Task Cop_cg_{11} fires to compensate cg_{11} . Then task $ErrorFlag1$ fires to update the parameter $ErrorFlag$ to true and the parameter $Status$ to “US_APCC”. After that, task $Finalize Thread 1$ is enabled to wait for other threads to complete.

When both threads in fg have finalized, task $Finalize Concurrent Threads$ is enabled. If the parameter $ErrorFlag$ is false, task $Successful$ is enabled. If the parameter $ErrorFlag$ is true, task $Error Compensate$ fires to compensate the threads that have executed successfully. Then depending on the parameter $Status$, either task $APRB$ or US_APCC is enabled. Task $Finalize$ ends the flow group.

Figure 22 presents the YAWL net of task $Error Compensate$ at a lower level. Depending on the execution status of each thread, an OR-split is used to compensate one or more threads as needed. The YAWL net of the compensation of fg is shown in Fig. 23. If the shallow compensation of fg exists, task $Shallow Compensation$ fires. If the shallow compensation executes successfully, task $Successful$ is enabled. If the shallow compensation of fg fails or does not exist, the deep compensation executes by firing the individual compensation for each thread. Finally, task $Finalize$ is enable to finish the compensation.

5.5.2. YAWL net of the if-else control structure

The if-else control structure selects one path from two to execute based on the evaluation of a condition. A YAWL net $lfeElse$ of an if-else control structure is presented

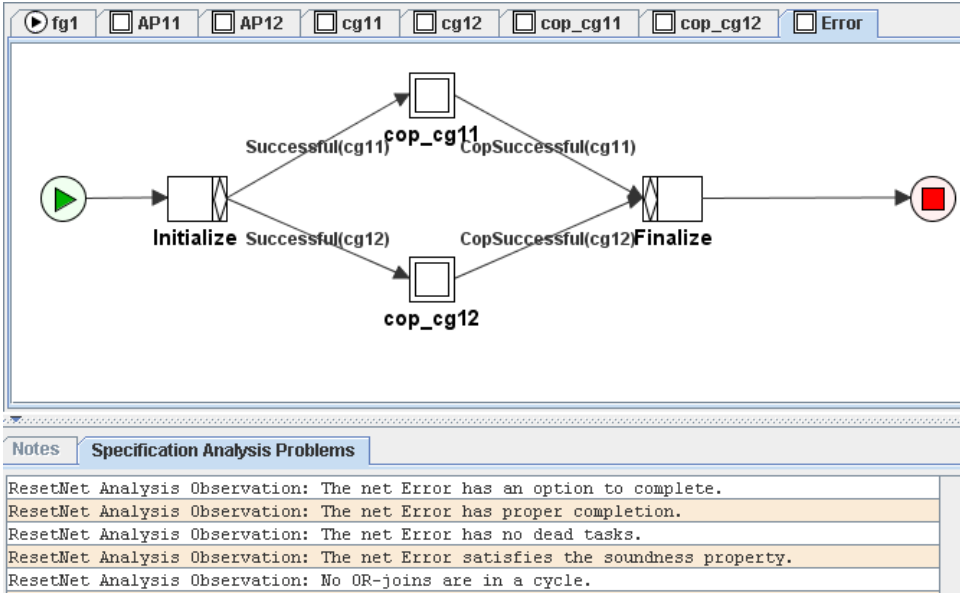


Fig. 22. Error compensation of the flow group in YAWL.

in Fig. 24. A net variable `lfElse` is defined in the YAWL net `lfElse`:

- Condition(boolean): If true, the upper path will be executed, otherwise if false, the lower path will be executed.
- Status(String): A string indicates the status of the of-else group.

In Fig. 24, after finishing task `Initialize`, either task `cg11` or `cg12` is enabled based on the value of the parameter `Condition`. If parameter `Condition` in the net variable `lfElse` is true, task `cg11` is enabled. If task `cg11` finishes successfully, task `Successful` is enabled. If an error occurs after task `cg11`, different cases may happen:

- `APRB(cg11)`: Task `ErrorFlag` fires to update the parameter `Status` to “ARRoll-back”. Then task `APRB` is enabled.
- `US_APCC(cg11)`: If the contingency of `cg11` is available, task `top_cg11` fires to execute the contingency of `cg11`. If the contingency succeeds, task `Successful` is enabled. If the contingency of `cg11` fails or does not exist, task `ErrorFlag` fires to update the parameter `Status` to “US_APCC”. Then task `US_APCC` is enabled.

Finally, task `Finalize` ends the if-else control structure. The other execution path of task `cg12` in Fig. 24 has the same execution semantics as discussed above.

Figure 25 gives the YAWL net of the compensation of `lfElse`. Depending on the value of the parameter `Condition`, the path that has been executed will be compensated.

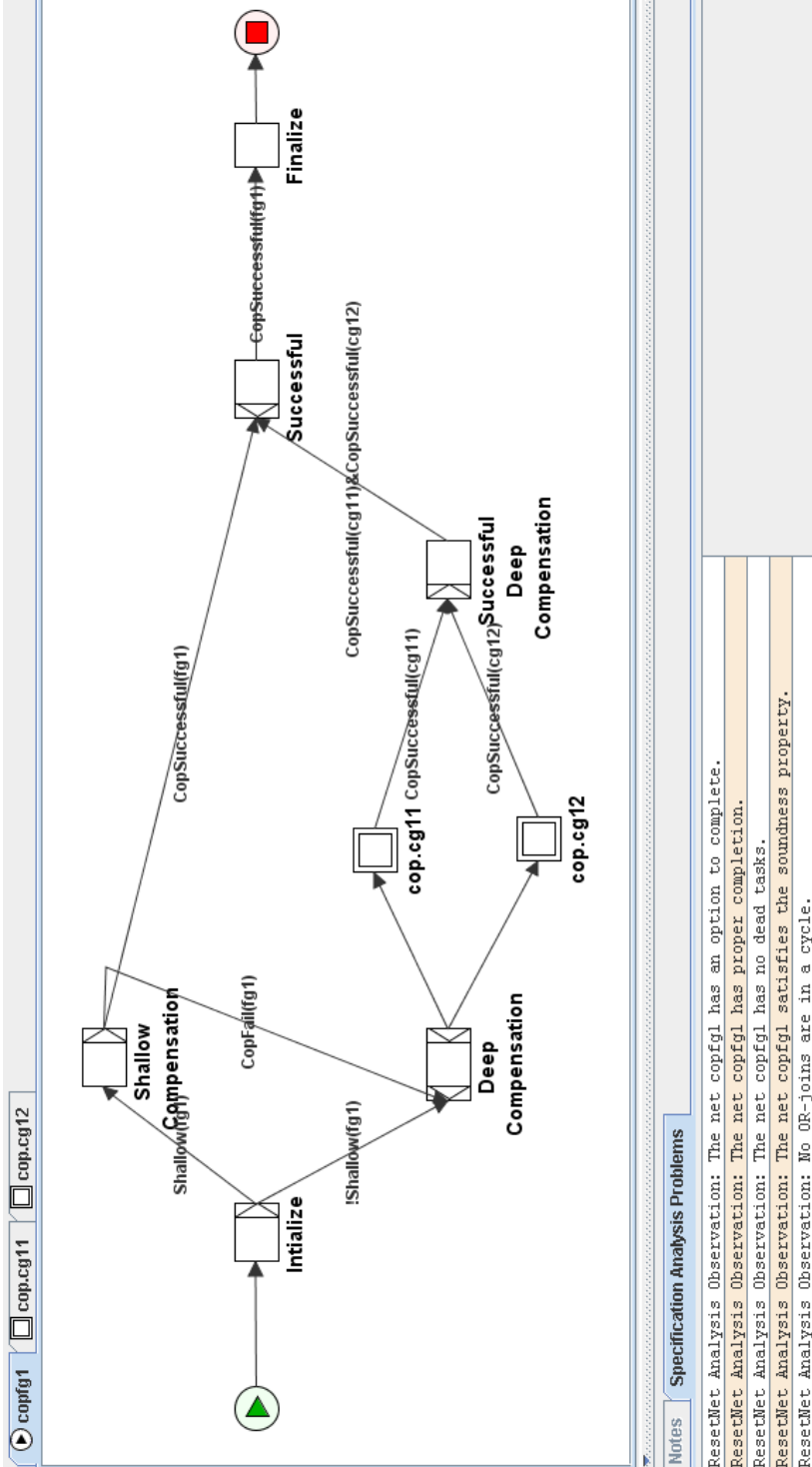


Fig. 23. Compensation of the flow group with two threads in YAWL.

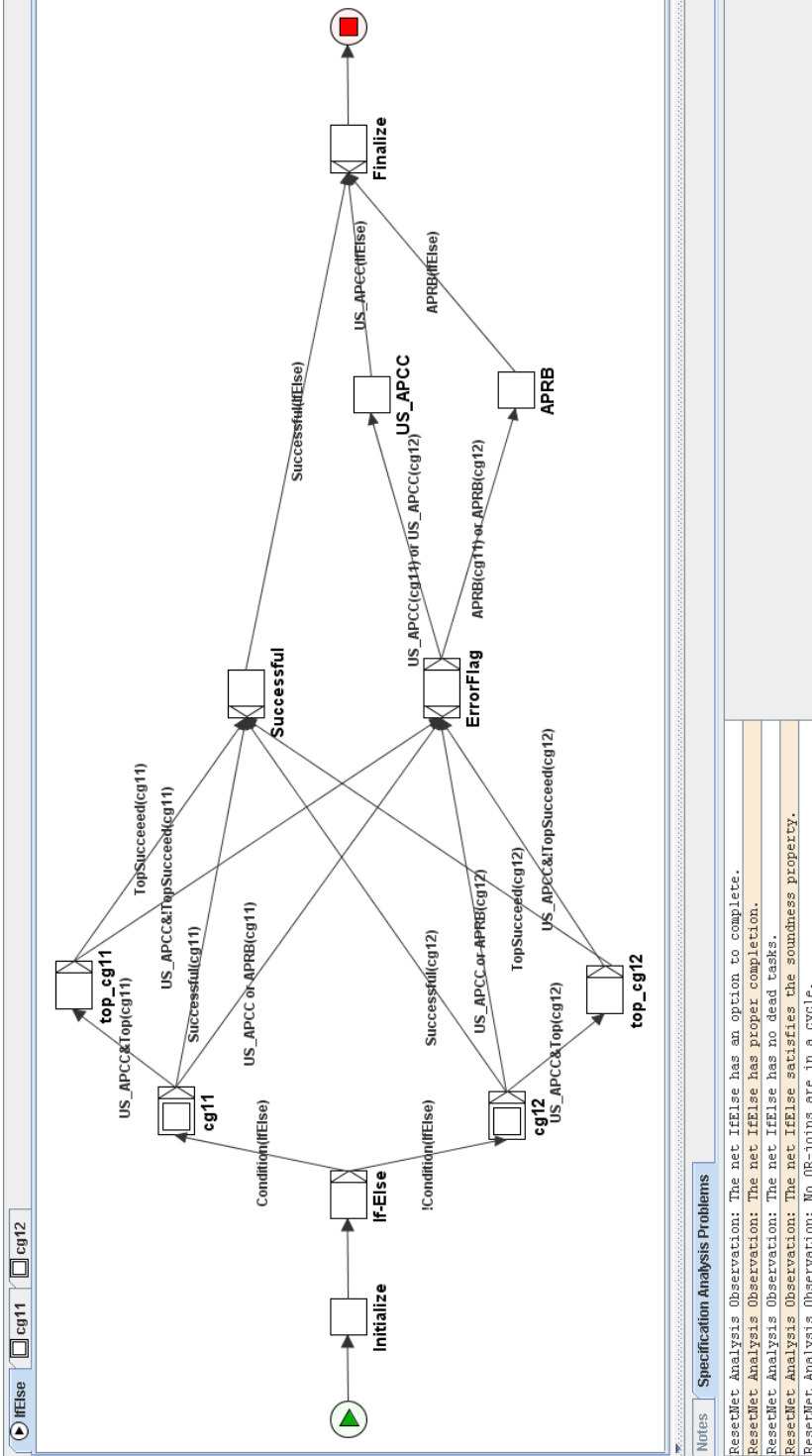


Fig. 24. The if-else control structure in YAWL.

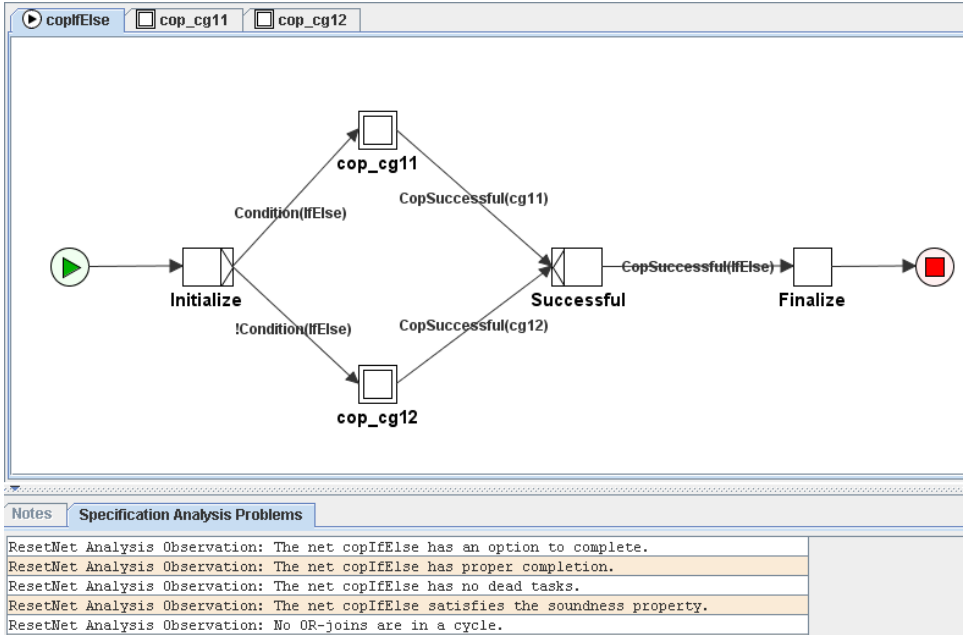


Fig. 25. The compensation of the if-else control structure in YAWL.

5.5.3. YAWL net of the loop control structure

The loop control structure repeatedly executes a composite group. In the AP model, a loop control structure can quit with three scenarios: quit with successful; quit without previous iterations compensated and quit with previous iterations compensated. A YAWL net looping of a loop control structure is presented in Fig. 26.

A net variable `Loop` is defined in the YAWL net looping:

- `Condition(boolean)`: Loop continues when `Condition` is true, otherwise quit when `Condition` is false.
- `Counter(Int)`: A number indicates the times of successful iterations, initialized with 0.
- `Status(String)`: A string indicates the status of the loop group.

In Fig. 26, after finishing task `Initialize`, if the parameter `Condition` is false, task `Successful` fires to finish the loop without any iteration. If the parameter `Condition` is true, task `cg11` is enabled to start the first iteration. Different cases may happen after task `cg11`:

- Task `cg11` succeeds, the parameter `Counter` increases by 1:
 - * The parameter `Condition` is true: task `cg11` fires to start the next iteration.
 - * The parameter `Condition` is false: task `Successful` is enabled to quit the loop with successful.

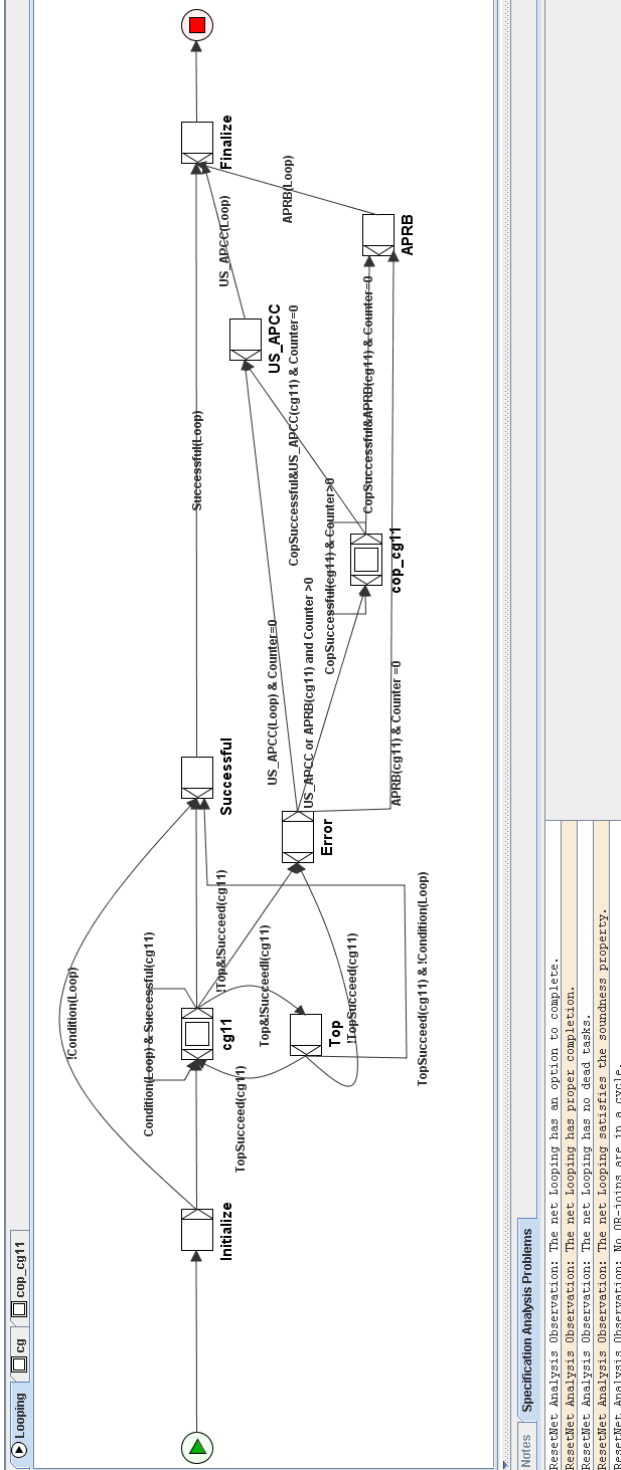


Fig. 26. The loop control structure in YAWL.

- Task cg_{11} fails:
 - * $APRB(cg_{11})$: Task $ErrorFlag$ fires. If the parameter $Counter$ is 0, task $APRB$ is enabled. If the parameter $Counter$ is greater than 0, task Cop_cg_{11} fires and the parameter $Counter$ decreases by 1. Task Cop_cg_{11} repeatedly fires until $Counter$ decreases to 0. Then task $APRB$ is enabled. The loop will quit with previous iterations compensated.
 - * $US_APCC(cg_{11})$:
 - No contingency of cg_{11} exists: Task $ErrorFlag$ fires. If the parameter $Counter$ is 0, task US_APCC is enabled. If the parameter $Counter$ is greater than 0, task Cop_cg_{11} fires and the parameter $Counter$ decreases by 1. Task Cop_cg_{11} repeatedly fires until $Counter$ decreases to 0. Then task US_APCC is enabled. The loop will quit with previous iterations compensated.
 - Normal contingency of cg_{11} exists: Task top_cg_{11} fires to try the contingency of cg_{11} . If task top_cg_{11} succeeds, the parameter $Counter$ increases by 1. After that, if the parameter $Condition$ is true, task cg_{11} fires to start next iteration. If the parameter $Condition$ is false, task $Successful$ is enabled to quit the loop with successful. However, if task top_cg_{11} fails, task $ErrorFlag$ fires. If the parameter $Counter$ is 0, task US_APCC is enabled. If the parameter $Counter$ is greater than 0, task Cop_cg_{11} fires and the parameter $Counter$ decreases by 1. Task Cop_cg_{11} repeatedly fires until $Counter$ decreases to 0. Then task US_APCC is enabled. The loop will quit with previous iterations compensated.
 - Break contingency of cg_{11} exists: Task top_cg_{11} fires to update the parameter $Condition$ in the net variable $loop$ to false and the parameter $TopSucceed$ in the net variable cg_{11} to true. Then task $Successful$ is enabled. The loop will quit without previous iterations compensated.

Figure 27 gives the YAWL net of the compensation of looping. If the parameter $Counter$ is 0, task $Finalize$ is enabled. If the parameter $Counter$ is greater than 0, task Cop_cg_{11} fires and the parameter $Counter$ decreases by 1. Task Cop_cg_{11} repeatedly fires until $Counter$ decreases to 0. Then task $Finalize$ is enabled.

5.6. Summary

This section has provided the formalization of the AP model by using Petri Nets and YAWL. The execution and recovery semantics of the atomic group, the AP, the composite group and the flow group have been presented using Petri Nets. Based on the formalization by Petri Nets, the rules of mapping a process in the AP model to YAWL nets have been defined. Other than the activities formalized in Petri Nets, the if-else and loop control structures in the AP model have also been formalized by YAWL nets.

The Petri Net specifications of the AP model are presented using generic execution and recovery semantics. The YAWL nets of the AP model, however, are

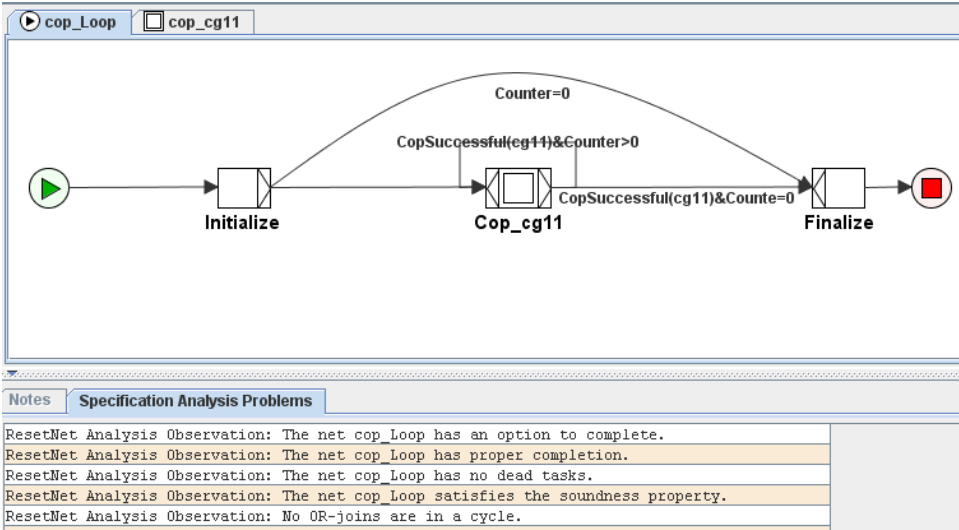


Fig. 27. The compensation of the loop control structure in YAWL.

modeled by some typical examples since the YAWL engine requires a complete executable process to be analyzed. Compare to the Petri Net specification of the AP model, using YAWL benefits the formalization of the AP model in three ways:

- (1) Activities modeled by YAWL net are easier to read and understand than Petri Nets.
- (2) A process modeled by YAWL nets is executable.
- (3) Activities and the complex control structures modeled by YAWL nets satisfy the soundness property.

With the help of the complete AP model, a new process execution agent (PEXA) has been developed,⁴⁶ using a dynamic and intelligent approach to monitor failures, detect data dependencies, and respond to failures and exceptional events. In particular, a decentralized data dependency analysis environment has been built by using the AP process execution engine. A complete set of algorithms to actively address the data dependency events among currently running processes has been developed and reported elsewhere.^{13,46}

6. Summary and Future Work

This paper has extended the original AP model with three complex control structures, providing more comprehensive support for business process modeling. A flow group has been introduced into the AP model to support parallel control structure. The discussions of the if-else and looping control structures have focused on the placement of the APs within each control structure. The rules for mapping a

process in the AP model to YAWL nets have been developed. Following the mapping rules, all activities and complex control structures in the AP model can be described and executed by YAWL nets. In addition, by mapping the activities and complex control structures to YAWL nets, the correctness of the execution and recovery semantics in the AP model has been verified since all YAWL nets satisfy the soundness property.

Different from existing web service composition models, the AP model presented by this research provides multiple levels of protection against service execution failure. Compared to WS-BPEL which uses a “Zigzag” compensation behavior to recover a failed process, the AP model uses shallow and deep compensations for cascaded recovery of a failed process. The “Zigzag” recovery manner in WS-BPEL may potentially violate the default compensation order when a control link is present between non-peer scopes.²⁸ In comparison, the AP model strictly compensates from the inner level to the outer level. This hierarchical recovery manner makes the recovery procedure more understandable and easy to design. Similar to aspect-oriented extensions to BPEL (AO4BPEL) that use business rules to provide more flexibility during service composition, the AP model checks pre/post conditions at each AP to actively monitor and evaluate the status of process execution. However, in contrast with AO4BPEL which creates business rules at different join points to avoid changing the service composition during runtime, the AP model embeds the IRs at different levels into the service composition. The AP model as a whole provides multiple levels of protection against service execution failure.

One future direction is to extend the functionalities of the AP model to support more cross-cutting concerns. Other than the functionalities of the AP provided in the current research, more functionalities can be performed at the assurance point. In the AP model, the AP is a referenceable point in the process that can create effects other than the main process flow to enhance the process execution. Thus, if the AP supports more functionalities, the process designer can easily specify a process that supports more cross-cutting concerns at the design time without changing the process execution engine. For example, a timer function can be added into the AP. By using the timer function in the AP, an AP can cancel the process when the set-time expires. Another example is that a parallel control structure may succeed if the number of successful threads reaches a threshold. In this example, the AP may perform as a counter to control the finish of a parallel control structure.

Acknowledgments

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.

References

1. M. P. Singh and M. N. Huhns, *Service-Oriented Computing: Semantics, Processes, Agents* (John Wiley & Sons Inc., Hoboken, New Jersey, USA, 2005).
2. D. Worah and A. Sheth, Transactions in transactional workflows, in *Advanced Transaction Models and Architectures*, eds. S. Jajodia and L. Kerschberg (Kluwer Academic Publishers, 1997), pp. 3–34.
3. T. Anderson and P. A. Lee, *Fault Tolerance: Principles and Practice* (Prentice/Hall International, 1981).
4. G. Booch, J. Rumbaugh and I. Jacobson, *Unified Modeling Language User Guide*, The Addison-Wesley Object Technology Series (Addison-Wesley Professional, Reading, MA, USA, 2005).
5. S. A. White et al., Business process modeling notation (BPMN) Version 1.0, *Business Process Management Initiative, BPMI. Org.* (2004).
6. T. Andrews et al., Business process execution language for web services, version 1.1, *Standards Proposal by BEA Systems, International Business Machines Corporation, and Microsoft Corporation* (2003).
7. Y. Xiao and S. D. Urban, The DeltaGrid service composition and recovery model, *International Journal of Web Services Research* **6**(3) (2009) 35–66.
8. S. D. Urban, L. Gao, R. Shrestha and A. Courter, Achieving recovery in service composition with assurance points and integration rules, in *Proc. 2010 Int. Conf. On the Move to Meaningful Internet Systems* (Springer-Verlag, Berlin-Heidelberg, 2010), pp. 428–437.
9. S. D. Urban, L. Gao, R. Shrestha, Y. Xiao, Z. Friedman and J. Rodriguez, The assurance point model for consistency and recovery in service composition, in *Innovations, Standards and Practices of Web Services: Emerging Research Topics, IGI Global publication* (2011), pp. 250–287.
10. J. L. Peterson, *Petri Net Theory and the Modeling of Systems* (Prentice Hall PTR Upper Saddle River, NJ, USA, 1981).
11. L. Gao, S. D. Urban, Z. Friedman and J. Rodriguez, Extending the assurance point (ap) approach to process recovery for use with flow groups, in *Proc. 26th IEEE Int. Parallel and Distributed Processing Symposium PhD Forum (IPDPSW)* (Shanghai, China, 2012), pp. 2201–2210.
12. W. M. P. Van Der Aalst and A. H. M. Ter Hofstede, YAWL: Yet another workflow language, *Information Systems* **30**(4) (2005) 245–275.
13. L. Gao, S. D. Urban and Z. Friedman, Decentralized data dependency analysis for concurrent process execution using the assurance point model, *International Journal of Web Services Research* (Under Review).
14. A. Cichocki, *Workflow and Process Automation: Concepts and Technology* (Kluwer Academic Pub., Norwell, MA, USA, 1998).
15. H. Garcia-Molina and K. Salem, Sagas, *ACM SIGMOD Record* **16**(3) (1987) 249–259.
16. A. Rolf, W. Klas and J. Veijalainen, *Transaction Management Support for Cooperative Applications* (Kluwer Academic Pub., Norwell, MA, USA, 1997).
17. H. Wächter and A. Reuter, The contract model, *Database Transaction Models for Advanced Applications* **7**(4) (1992) 219–263.
18. J. Eder and W. Liebhart, The workflow activity model WAMO, in *Proc. 3rd Int. Conf. Cooperative Information Systems* (Vienna, Austria, 1995), pp. 87–98.
19. M. Karnath and K. Ramamritham, Failure handling and coordinated execution of concurrent workflows, in *Proc. 14th Int. Conf. Data Engineering* (1998), pp. 334–341.

20. D. K. W. Chiu, Q. Li and K. Karlapalem, Facilitating exception handling with recovery techniques in ADOME workflow management system, *J. Appl. Syst. Studies* **1**(3) (2000) 467–488.
21. C. Hagen and G. Alonso, Exception handling in workflow management systems, *IEEE Transactions on Software Engineering* **26**(10) (2002) 943–958.
22. B. Kiepuszewski, R. Muhlberger and M. E. Orlowska, FlowBack: Providing backward recovery for workflow management systems, *ACM SIGMOD Record* **27**(2) (1998) 555–557.
23. Z. Yang and C. Liu, Implementing a flexible compensation mechanism for business processes in web service environment, in *ICWS '06. Int. Conf. Web Services, 2006*, September 2006, pp. 753–760.
24. M. Schäfer, P. Dolog and W. Nejdl, An environment for flexible advanced compensations of web service transactions, *ACM Transactions on Web* **2**(2) (2008) 14:1–14:36.
25. P. Grefen, J. Vonk and P. Apers, Global transaction support for workflow management systems: From formal specification to practical implementation, *The VLDB Journal* **10**(4) (2001) 316–333.
26. J. Vonk and P. Grefen, Cross-organizational transaction support for e-services in virtual enterprises, *Distribution of Parallel Databases* **14**(2) (2003) 137–172.
27. H. Schuldt, G. Alonso, C. Beeri and H.-J. Schek, Atomicity and isolation for transactional processes, *ACM Transactions on Database Systems (TODS)* **27**(1) (2002) 63–116.
28. R. Khalaf, D. Roller and F. Leymann, Revisiting the behavior of fault and compensation handlers in WS-BPEL, in *On the Move to Meaningful Internet Systems 2009* (2009), pp. 286–303.
29. J. Widom and S. Ceri, *Active Database Systems: Triggers and Rules for Advanced Database Processing* (Morgan Kaufmann Pub., San Francisco, CA, USA, 1996).
30. M. Brambilla, S. Ceri, S. Comai and C. Tziviskou, Exception handling in workflow-driven web applications, in *Proc. 14th Int. Conf. World Wide Web* (ACM, Chiba, Japan, 2005), pp. 170–179.
31. A. Liu, Q. Li, L. Huang and M. Xiao, A declarative approach to enhancing the reliability of BPEL processes, in *Proc. 2007 IEEE Int. Conf. Web Services*, Salt Lake City, Utah, USA, 2007, pp. 272–279.
32. W. Tan, L. Fong and N. Bobroff, BPEL4job: A fault-handling design for job flow management, in *Service-Oriented Computing 2007* (Springer-Verlag, Berlin-Heidelberg, 2010), pp. 27–42.
33. S. Modafferi and E. Conforti, Methods for enabling recovery actions in WS-BPEL, in *Proc. 2006 Confederated Int. Conf. On the Move to Meaningful Internet Systems: Coopis, DOA, GADA and ODBase* (Springer-Verlag, Berlin-Heidelberg, 2006), pp. 219–236.
34. S. Modafferi, E. Mussi and B. Pernici, SH-BPEL: A self-healing plug-in for WS-BPEL engines, in *Proc. 1st Workshop on Middleware for Service Oriented Computing* (Melbourne, Australia, 2006), pp. 48–53.
35. L. Baresi, S. Guinea and L. Pasquale, Self-healing bpel processes with dynamo and the jboss rule engine, in *Int. Workshop on Engineering of Software Services for Pervasive Environments* (Dubrovnik, Croatia, 2007), pp. 11–20.
36. V. Dialani, S. Miles, L. Moreau, D. De Roure and M. Luck, Transparent fault tolerance for web services based architectures, *Euro-Par Parallel Processing* (Springer, 2002), pp. 107–201.
37. Z. W. Luo, Checkpointing for workflow recovery, in *Proc. 38th Annual on Southeast Regional Conf.* (Clemson, South Carolina, USA, 2000), pp. 79–80,

38. C. Ye, S.-C. Cheung, W. K. Chan and C. Xu, Atomicity analysis of service composition across organizations, *IEEE Transactions on Software Engineering* **35**(1) (2009) 2–28.
39. R. Hamadi, B. Benatallah and B. Medjahed, Self-adapting recovery nets for policy-driven exception handling in business processes, *Distributed and Parallel Databases* **23**(1) (2008) 1–44.
40. A. Charfi and M. Mezini, AO4BPEL: An aspect-oriented extension to BPEL, *World Wide Web* **10**(3) (2007) 309–344.
41. A. Charfi and M. Mezini, Aspect-oriented workflow languages, in *Proc. 2006 Confederated Int. Conf. On the Move to Meaningful Internet Systems: Coopis, DOA, GADA and ODBase* (Springer-Verlag, Berlin-Heidelberg, 2006), pp. 183–200.
42. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm and W. Griswold, An overview of AspectJ, *Proc. 15th European Conf. Object-Oriented Programming* (Springer-Verlag, Berlin-Heidelberg, 2001), pp. 327–354.
43. R. Shrestha, Using assurance points and integration rules for recovery in service composition, Master's thesis, Department of Computer Science, Texas Tech University, Lubbock, TX, USA (2010).
44. S. D. Urban, S. W. Dietrich, Y. Na, Y. Jin, S. A. Saxena, S. D. Urban, S. W. Dietrich, Y. Na and Y. Jin, The irules project: Using active rules for the integration of distributed software components, in *Proc. 9th Working Conf. Database Semantics: Semantic Issues in E-Commerce Systems* (Hong Kong, 2001), pp. 265–286.
45. C. Ma, Q. Xu and J. W. Sanders, A survey of business process execution language (BPEL), *Int. Institute for Software Technology, UNU-IIST Report (425)* (2009) 102–108.
46. L. Gao, A robust web service composition model with decentralized data dependency analysis and rule-based failure recovery capability, Ph.D. thesis, Department of Computer Science, Texas Tech University, Lubbock, TX, USA (2012).

Copyright of International Journal of Cooperative Information Systems is the property of World Scientific Publishing Company and its content may not be copied or emailed to multiple sites or posted to a listserv without the copyright holder's express written permission. However, users may print, download, or email articles for individual use.