Autonomous Failure-handling mechanism for WF Long Running Transactions

Manar S Ali and Stephan Reiff-Marganiec
Department of Computer Science
University of Leicester
Leicester, UK
e-mail: {mssa1, srm13}@le.ac.uk

***Abstract*— Business Processes naturally involve long running activities and require transactional behaviour across them. The work presented in this paper is a proposal for a novel autonomous failure handling mechanism for long running nested transactions (LRT) and forms part of a general management and compensation model for long running transactions in workflows. The mechanism is based on propagation of failures through a recursive hierarchical structure of transaction components (nodes and execution paths). The management system of transactions (COMPMOD) is implemented as a reactive system controller, where system components change their states based on rules in response to triggering of events such as activation, failure, force-fail, completion, or compensation events. A notable new feature of the model is the distinction of vital and non-vital components, allowing the process designer to express the cruciality of activities in the workflow with respect to the business logic.**

Keywords-LRT, compensation and failure handling, rule based reactive approach.

I. INTRODUCTION

Business processes naturally involve long running activities and require transactional behaviour across these. Such business processes are typically represented by workflows and these in turns are composed out of workflow patterns [11].

One of the important management aspects in managing long running transactions (LRTs) is preserving consistency of the systems being involved in the LRT. This is done by guaranteeing that an LRT will always ensure that the data integrity is preserved and systems are maintained consistently. This guarantees that the execution of the LRT terminates in an accepted state from the business and the transaction modelling points of view. In the absence of failure this should occur, but even if the LRT has not completed its normal path of execution due to a failure causing termination of the LRT or an abnormal path being chosen the same guarantee needs to be given. This is usually achieved by adopting effective compensation and fault-handling techniques. Transactional patterns [3] have been proposed as a reliable paradigm for orchestrating transaction component behaviours in alignment with workflow patterns.

We can distinguish two execution modes of an LRT, namely the normal execution mode and the compensation execution mode. During the normal execution mode, failures of component services could occur. Failure handling and recovery mechanisms should be implemented to process failures of component services and try to save the transaction from failing completely.

In web service based LRT, reliability of transactions behaviour is a more critical issue due to autonomy and heterogeneity of web services. Transactions should be reliable in case of failure occurrence. However, even in such loosely coupled applications, failure handling and compensation mechanisms must be implemented to ensure predictable behaviour of transactions in case of hardware or software failures.

The management of LRTs must proceed in two parallel directions: (a) the management of the LRT during its normal execution path, which must embrace a reliable and efficient fault-handling mechanism and (b) the management of the LRT during the execution of its compensation path, in case the LRT has failed to complete.

To handle LRTs a modelling and management system would ideally support the following aspects. 1-3 are motivated by the structure of transactions and the fact that it is at the business level where a full understanding of the implications exists; 4 allows to separate the actual process and any handling of exceptions in a clear and user-friendly way; and 5-6 are requirements ensuring the practicality of the approach.

1. Multi-level nesting of transactions with reliable behavioural dependencies between transaction components and across hierarchy levels.
2. Definition of designer-order compensation patterns that reflects the business logic of the LRT.
3. Incorporating compensation logic into business logic of long running transactions through transactional dependencies.
4. Rule-based actions for managing execution and compensation control flow.
5. Automated method for propagating failure events through the hierarchy structure as a failure handling mechanism.
6. Automated method for performing compensation actions while the LRT execution is in progress, through backward and forward order compensations.

The work in this paper focuses on handling failures that occur during the normal execution of the LRT, focusing on the control flow perspective. The failure-handling mechanism is incorporated into the business logic of the transaction through strict logical definitions of behavioural dependencies between transactional components. Compensation management is not covered in this paper and

therefore points 2, 3 and 6 are not discussed. However, similar mechanisms to those proposed can be adopted for the handling of compensations.

The mechanism is supported by a general management system (COMPMOD) which is modelled as a reactive system controller [1] that changes its components' execution states and its actions in response to stimuli/events. An event is fired as a result of a behavioural dependency satisfaction. A stimulus is triggered as a result of a transition in the execution state of a transaction component or result from the application of a rule leading to event propagation in the hierarchy. In other words, COMPMOD is an Event/Control driven WF management system that reacts continuously to stimuli/events until the LRT execution finally terminates in a state that is meaningful from both a system as well as a business perspective.

In this paper we present the work as follows: first, we describe the adopted transaction modelling paradigm, discussing WF and transactional patterns. We then present the novel contributions of (1) COMPMOD's LRT attributes, dependencies, and management rules and (2) the recursive failure handling propagation mechanism. All stages are supported by running an example.

II. WORKFLOW AND TRANSACTIONAL PATTERNS

Workflow patterns have been developed as part of an initiative commenced in 2000 by [2]. They classify the core architectural constructs inherent in workflows in a language and technology independent way, thus allowing definition of fundamental requirements of business process modelling. Workflow patterns consider workflow specifications from a control-flow perspective and characterize a range of control flow patterns that might be encountered when modelling a business workflow. Following the initial work [2], 43 control patterns were proposed in [4]. The patterns are classified as (a) basic control-flow patterns, (b) advanced branching and synchronization patterns, (c) structural patterns, (d) state-based patterns, and (e) cancellation patterns. Our approach, COMPMOD, so far implements the basic control-flow patterns: sequence, AND-split, AND-join, OR-split, OR-join, XOR-split, and XOR-join.

The concept of transactional patterns was introduced in [3]. Transactional patterns are aimed at specifying flexible and reliable composite web services. They are a convergence concept between workflow patterns and advanced transactional models [5], and thus they combine the flexibility of work flow control patterns with the reliability of transactional models to ensures transactional consistency of service compositions. Transactional patterns define orchestrations between services in composite web service by using dependencies to define how services are combined and how the behaviour of some given services influences the behaviour of some others. Dependencies are used to express the relationships that exist between services such as sequence, alternative, compensation, activation, or cancellation. They also associate preconditions with service operations. The general definition of a dependency is:

***Def.1 [3]:* A *dependency* from service *s1* to service *s2* exists if a *transition* of *s1* can fire an *external transition* of *s2*.**

It is assumed that a transition can be an internal or external transition with internal transitions being fired by the service itself (e.g. *complete(), fail(),* or *retry()*) and external transitions being fired by external entities (e.g. *abort(), cancel(),* or *compensate()*).

We extend the notion of transactional patterns to model multi-nested transactions by the introduction of the following concepts:

- Atomic nodes, scopes, and nested scopes and their transactional dependencies and attributes.
- Execution paths and their transactional attributes.
- A hierarchical structure that mirrors the workflow structure of the LRT.
- Vitality of nodes, scopes, and execution paths.
- Encapsulation of dependencies on the scope and execution path level to facilitate automated propagation of events.

In the following sections, we provide a description of the above concepts and show how they are implemented to automate the recursive failure-handling mechanism of COMPMOD.

III. LRT'S TRANSACTIONAL ATTRIBUTES AND DEPENDENCIES

An LRT is executed as a flat transaction, i.e. a sequence of nodes that are executed sequentially. A node can be an atomic node representing an atomic task (a single web service), or, a scope node starting with a split pattern and ending with a join pattern of the same type. Each scope creates two or more execution paths that start from the split point and end at the join point (or synchronizer) of the scope. Each execution path is a sequence of one or more nodes executed in sequential order where nodes along the path again can be atomic or scopes. Through the rest of the discussion we will use the term component to refer to both nodes (atomic/scope) and execution paths.

A. Transactional operators and scopes

A scope starts with a split operator (OR, AND, or XOR) that is explicitly assigned while constructing the LRT. The model implicitly specifies a join operator of the same type to mark the end point of a scope. The join point is represented by a synchroniser in the WF schema. The type of operator used to define a scope influences the definition of transactional attributes and dependencies of its encapsulated components. Semantics of operators are adopted from the definitions of WF-patterns in [4]. An AND operator creates a scope with parallel execution paths, and the scope is successfully completed if all its execution paths are successfully completed. An OR operators creates a scope with parallel paths where only a subset of these paths are executed during runtime, the executed paths are those whose enabling condition are satisfied. An OR scope successfully completes if all its enabled activated paths are successfully completed. An XOR scope creates exclusive paths, the first

path has the highest priority and therefore execution starts with the path with the highest priority. If an exclusive path failed to complete, it is compensated in forward order until the split point of the scope is reached, and then next path (if one exists) is executed. Therefore, execution paths are assigned with the following transactional attributes: an execution path *hasAnalternative*, if it was an exclusive path that has a path with lower priority in the same scope. In an OR scope, a path is *enabled* if and only if its branching condition is satisfied at runtime and hence, only enabled paths are activated. Each execution path has an ordered list of one or more nodes denoted by *nodeList*. A scope is formally defined as:

***Def.2:* A *scope node* is defined as follows:**
$\forall_{i=1..m}\ p_i.nodeList = splitNode_i\ and$
$\forall_{i=1..m}\ nodeList_i.type = \{ATOMIC, SCOPE\}$:
$scope = (operator, [splitNode_1..splitNode_m])$
$\rightarrow\ scope.pathList = [p_1..p_m]$
where operator∈{AND,OR.XOR}

When a scope is initially defined, a split operator and a list of split nodes are specified. The number of split nodes corresponds to the number of execution paths encapsulated within the scope. A split node can be an atomic node, or a scope node which facilitates the construction of nested scopes. When a node is appended to an existing execution path p_i, the node is appended to $p_i.nodeList$.

B. Vitality of components

Each LRT component has a *vitality* attribute, allowing to specify whether a component is vital or non-vital. A vitality value {TRUE/FALSE} is assigned to each component either by *specification* or by *evaluation*. Vitality of atomic and scope nodes is assigned by specification, that is, according to the business logic of the LRT. Essentially vitality allows the workflow designer to express whether a failure of the specific service can be tolerated and the workflow can proceed (and example of a non-vital task might be one sending a progress message to the invoking user – nothing in the process will be broken if the message is not sent). Vitality of execution paths is assigned by evaluation according to the following rules. A path is

- **vital** if it encapsulates at least one vital node.
- **non-vital** if all the nodes it encapsulates are non-vital.

The transactional implication of the vitality measure of a component expresses the impact of unsuccessful completion of a component on its immediate superior[1]. For example, the failure of a vital node will fail its enclosing execution path. Vitality of components is utilised in the failure handling propagation mechanism proposed in this paper.

Note that the decision of assigning the vitality value to nodes (atomic and scope) is based on the business logic of the LRT. It is important to note that our management/ compensation model does not investigate or analyse the business logic of the LRT. It is always assumed by the model that the logic provided for the LRT at design time is what it is required from the transaction by the business level. Therefore, for a designer it is possible to define a scope node as a non-vital node, while it could encapsulate vital paths, without leading to an incorrect model.

However, the following logical restrictions are assumed by the approach with respect to design of scopes:

- **Assumption 1**: In an exclusive scope, all exclusive paths should have the same vitality measure, that is they must all be vital or all be non-vital.
- **Assumption 2.** If all paths in a scope are non-vital, their encapsulating scope should be non-vital by specification.

C. Execution states

During the execution life cycle of the transaction, the LRT and its components go through different execution states and they are marked with their current execution state. We list below the set of execution states for the LRT and each component and Figure 1, shows the state transition diagram of atomic nodes. State transitions are triggered by events. E.g. in Figure 1, when a completion of an atomic node is triggered, the execution state of the node changes from ACTIVATED to COMPLETED.

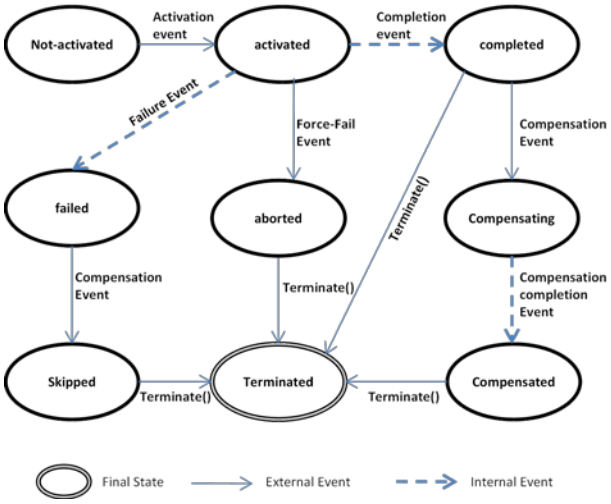

Figure 1. State transition diagram for atomic nodes

LRT.state = {*not-activated, activated, completed, failed, compensating, compensated, terminated*}
AtomicNode.state = {*not-activated, activated, completed, failed, compensating, compensated, skipped, aborted, terminated*}
ScopeNode.state = {*not-activated, activated, completed, failed, compensating, compensated*}
ExecutionPath.state = {*not-Activated, activated, completed, failed, compensating, compensated*}

[1]The immediate superior of node is its enclosing path and the immediate superior of a path is its enclosing scope (section 5).

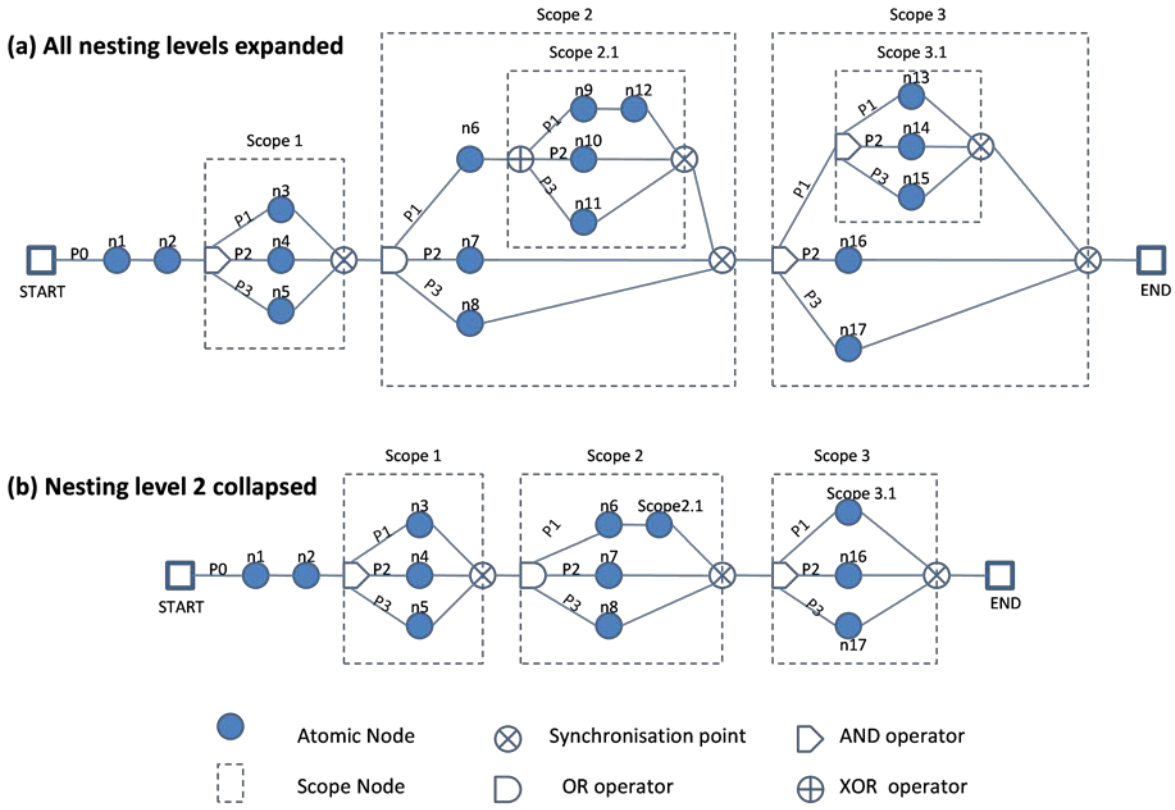

Figure 2. A sample workflow

IV. REPRESENTATIONS OF NESTED LRTS

We use two main representations of the workflows in our work: a workflow representation, that allows to abstract away from sub workflows and a tree representation that is used by the propagation algorithm.

In our model we have two basic components: nodes and execution paths. A node can be an atomic node (a single web service) or a scope node – a set of semantically connected nodes (atomic and/or scope). An execution path represents a trail of nodes that are executed in sequential order. An execution path reading of a scope node that it encapsulates is the same as an atomic node. In other words, scope nodes on an execution path are like black boxes that encapsulate execution paths and other nodes. Transactional dependencies are employed to model the transaction behaviour between transaction components. Transactional dependencies are defined between a component and its neighbours.

A. Workflow Model

The modeling method allows for multi-level nested transactions to address demands occurring in real cooperative business processes. In the representation model itself we see alternating levels of paths and nodes.

Figure 2.(a), demonstrates a two level-nested LRT that consists of atomic nodes and nested scopes. Considering execution p_1 in $scope_2$, the path consists of an atomic node n_6 followed in sequence by a scope node $scope_{2.1}$ that in turn encapsulates three execution paths. As mentioned earlier, an execution path is a trail of nodes (atomic and/or scope) that are executed in sequential order and we provide a nodeList attribute on path objects to express this: for example $p_1.nodeList = [n_6, scope_{2.1}]$. Figure 2.(a) shows the LRT with all nesting levels expanded and Figure 2.b demonstrates the LRT with level 2 of the WF collapsed.

The main execution path of a transaction is regarded as level 0 in the workflow and denoted as p_0. If we collapse level 1 of the WF, the main execution path becomes a flat WF that executes the nodes in $p_0.nodeList = [n_1, n_2, scope_1, scope_2, scope_3]$ in sequential order.

B. Hierarchical Structure Model

Transaction components –nodes and execution paths-- are linked together in a hierarchical structure (see e.g. Figure 3). Each component has a single superior and an ordered set of one or more inferiors. Specifically:

Node component: A superior of any node is the execution path that encapsulates the node. An atomic node is a leaf node that has no inferiors. A scope node has two or more inferiors which represents the number of split execution paths it encapsulates.

Execution path component: The superior of any execution path is the scope node that encloses it. The main execution path of a LRT has a NULL superior. Each execution path has one or more inferiors. Inferiors of a path represent an ordered set of one or more nodes that

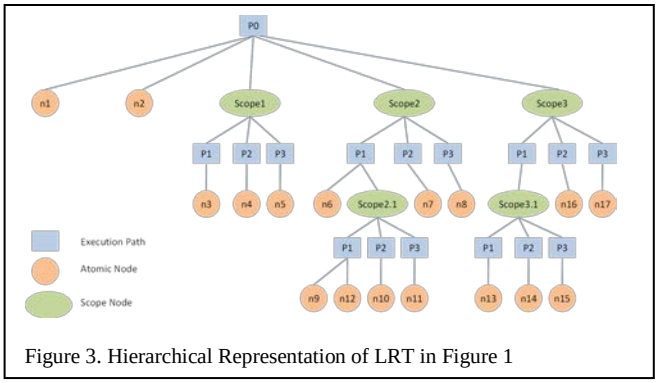

Figure 3. Hierarchical Representation of LRT in Figure 1

the path encloses. The root of the recursive hierarchy is the main execution path of the LRT p_0.

C. Hierarchical Transactional Dependencies

As stated, transaction behaviour between components is expressed through dependencies. Transactional dependencies are defined: (a) between an execution path and its immediate outer scope, (b) between a node and its immediate outer execution path and, (c) between any two successive nodes on a sequence of the same execution path. This imposes the hierarchical relationship between components and facilitates hierarchical propagation of events. We expect dependencies to be defined in the WF representation and then mapped into the hierarchical structure to enforce the propagation mechanism through and across hierarchy levels. In terms of the hierarchy structure, transactional dependencies are defined between a component and its immediate superior and between a node and its immediate siblings (if any exist).

Dependencies (activation, completion, failure, force-fail, compensation (forward/backward /designer-tailored) and compensation-completion) are defined in first order logic and in terms of sets of pre-conditions that, when satisfied at run time lead to an event being fired. In the scope of this paper we focus on failure and force-fail dependencies.

The general definition for a behavioural dependency is:

Def 3. A *behavioural dependency* exists from *component*$_j$ *to component*$_i$ iff a state transition in *component*$_i$ can fire a transactional event for *component*$_j$:

$$\boldsymbol{Dep(component_j) := preCond(component_i.state)}$$

As an example, for two successive nodes the activation dependency of the successor node stating that an activation event is fired for a successor node if its predecessor node has been completed or, if its predecessor node was not a vital node but failed to complete is defined as:

$$\boldsymbol{ActDep(succNode) := (PredNode.State = COMPLETED) \vee (PredNode.Vital=FALSE \wedge PredNode.State=FAILED)}$$

Behavioural dependencies can also be defined between a set of sibling components and their immediate superior component, essentially extending Def. 3 to allow for any of a number of sibling nodes to fire a transactional event for the superior component:

$$\boldsymbol{Dep(superior) := PreCond([sibling_1.state..sibling_n.state])}$$

D. Failure and Force-Fail dependencies:

Failure dependencies are defined for non-vital scope nodes and non-vital execution paths. Vital scopes and execution paths do not lead to events fired by dependencies, instead such failure is assessed by the management rules discussed in section 5. Table 1 shows a complete list of failure and force fail dependencies. Failure of all vital nodes in a path will fail the path ($\boldsymbol{path.nodeList \leftarrow_{fail} path;}$ FD1); failure of paths in a scope will lead to failure of the scope ($\boldsymbol{scope.pathList \leftarrow_{fail} scope}$), dependent on the semantics of the scope operator. For example, FD3 states that an OR scope fails if all its enabled paths failed.

Force-fail is a counterpart for cancelation. When a vital concurrent path fails, its immediate outer scope fails. Force-fail dependencies force all active paths within a failed concurrent scope to cancel their executions, and subsequently all active nodes on paths are forced to fail. Force-fail dependencies are defined between components and their immediate superiors. A force-fail dependency $\boldsymbol{component.superior \leftarrow_{forcefail} component}$ means that failure of an activated component's superior will force the component to fail. For example FF1 states that an activated path will fail if its enclosing scope has failed. Consequently, all concurrently activated paths within a scope will force-fail if their immediate superior scope fails.

TABLE I. FAIL AND FORCE-FAIL DEPENDENCIES

Dep	Dependency Formula
	Failure Dependencies
FD1	$FailDep(path) := \left(\bigwedge_{1\le i\le m} nodeList_i.State = FAILED\right)$ where $m = \lvert path.nodeList\rvert$
FD2	$FailDep(ANDscope) := \wedge_{1\le i\le m}(path_i.State = FAILED)$ where $m = \lvert ANDscope.pathList\rvert$
FD3	$FailDep(ORscope) := \bigwedge_{1\le i\le m}\left(path_i.Enabled = TRUE \wedge path_i.State = FAILED\right)$ where $m = \lvert ORscope.pathList\rvert$
FD4	$FailDep(XORscope) := \bigwedge_{1\le i\le m}(path_i.State = FAILED)$ where $m = \lvert XORscope.pathlist\rvert$
	Force-Fail Dependencies
FF1	$ForceFailDep(path) := immediateSuperior.State = FAILED$
FF2	$ForceFailDep(node) := immediateSuperior.State = FAILED$

V. FAILURE MANAGEMENT

A. Management Rules

Management rules (or policies) incorporate autonomy into systems. The most common form is that of ECA (event condition action) rules which present an event driven

approach. One of the first attempts in applying ECA rules approach in management of transactions in WF systems was in [6] by using triggers for organising long running activities. ECA rules have been used to adapt workflows and provide more fine-grained specification for service selection for tasks and in database management systems, e.g. [7]. ECA rules in COMPMOD are implemented to model the expected execution behaviour of the LRT. When an event is fired, it triggers an ECA rule, and if the condition holds, an appropriate action takes place. ECA rules have the following pseudo generic form:

***ON** event **IF** condition **DO** action*

The event part of the rule can be (a) an internal system generated event such as completion, failure, or cancelation of an atomic node or, (b) an external event fired as a result of a dependency condition satisfied for a component or, (c) a result of execution a state transition event of a component. The condition part is one or more connected Boolean expressions that need to hold for the rule to be applied. The action is a sequence of one or more actions to be performed in case the rule is applied, and can in turn introduce new events needing to be handled.

COMPMOD Rules are classified into: activation, completion, compensation, failure, and propagation rules. As this paper focuses on failure handling, we only list failure and failure propagation ECA rules in Table 2. Note that *fail* and *abort* are actions that lead to raising an event (fail or abort) but also have a side effect on the state of the respective component as follows:

if component.state=ACTIVATED
then component.state:=FAILED

ECA rules of COMPMOD reflect the following:

1. The business logic of the LRT (e.g. FR4 states that if a node is vital and failed, its superior path fails).
2. The semantics of a COMPMOD model (e.g. FFR2 states that if a force-fail event is fired for an activated atomic node, the node is aborted).
3. The semantics of WF patterns (e.g. FR5 states that failure of a vital path that has no alternative, i.e. a concurrent or last exclusive path, fails its enclosing scope).

TABLE II. FAILURE AND PROPAGATION RULES

Rule	Pseudo ECA-Rule statement
Failure Rules	
FR1	**ON** "internal failure/cancellation event fired for atomic node" **DO** fail(node)
FR2	**ON** FailDep(node)=TRUE **IF** node.type=SCOPE **DO** fail(node)
FR3	**ON** FailDep(path)=TRUE **DO** fail(path)
FR4	**ON** fail(node) **IF** node. vital = TRUE **DO** fail(node.superior)
FR5	**ON** fail(path) **IF** path. hasAlternative = FALSE and path. vital = TRUE **DO** fail(path.superior)
FR6	**ON** fail(p_o) **DO** fail(LRT)
Failure Propagation Rules	
FFR1	**ON** ForceFailDep(node) **IF** node.type=SCOPE and node.state=activated **DO** fail(node)
FFR2	**ON** ForceFailDep(node)=TRUE **IF** node.type=ATOMIC and node.state=activated **DO** abort(node)
FFR3	**ON** ForcefailDep(path)=TRUE **IF** path.state=activated **DO** fail(path)*

B. Failure propagation mechanism

This work presents a recursive method for propagating vital failure events through the recursive hierarchy structure of LRT components. Propagation is in parallel with rule-based actions in order to reach a consensus about the execution state of LRT components and the LRT itself.

Within the context of the proposed hierarchy structure, the recursive failure propagation mechanism entails a combination of three types of propagation methods:

1. *Bottom-up propagation* originates from failure of a vital atomic node and propagates up the hierarchy to its immediate superior path. If the failed atomic node exists on the main execution path p_0, the LRT fails.
2. *Upwards recursive propagation* originates from failure of a scope node by repeating a bottom-up propagation to its immediate superior execution path in recursive fashion until a non-vital component is reached in the hierarchy or until the failure reaches the root of the hierarchy structure (p_0).
3. *Downwards recursive propagation* originates from a failure of a scope node (vital or non-vital) by repeating a top-down propagation to its immediate activated paths until the propagation reaches all active atomic nodes within the failed scope's sub-hierarchy. This represents a mean of forcing failure/ cancellation of concurrently running nodes in a failed scope. Force fail only applies to concurrent scopes and in our model only applies to AND and OR scopes since a failed XOR is a result of a failure of all its exclusive paths.

Failure propagation is always initiated by the failure of a vital atomic node and propagates recursively through vital component ancestors in the hierarchy structure to stop when a non-vital ancestor component is reached or when the root of the hierarchy is reached. As for Top-down propagation of failures, both vital and non-vital active components are force-failed.

If a vital failure propagates through the hierarchy structure of the LRT and reaches the root of the hierarchy p_0, the LRT fails. Figures 4 and 5 illustrate the failure propagation mechanism linked to dependencies and ECA rules (Tables 1 and 2). Figures 4 and 5 include compensation mechanisms that are out of scope of this paper.

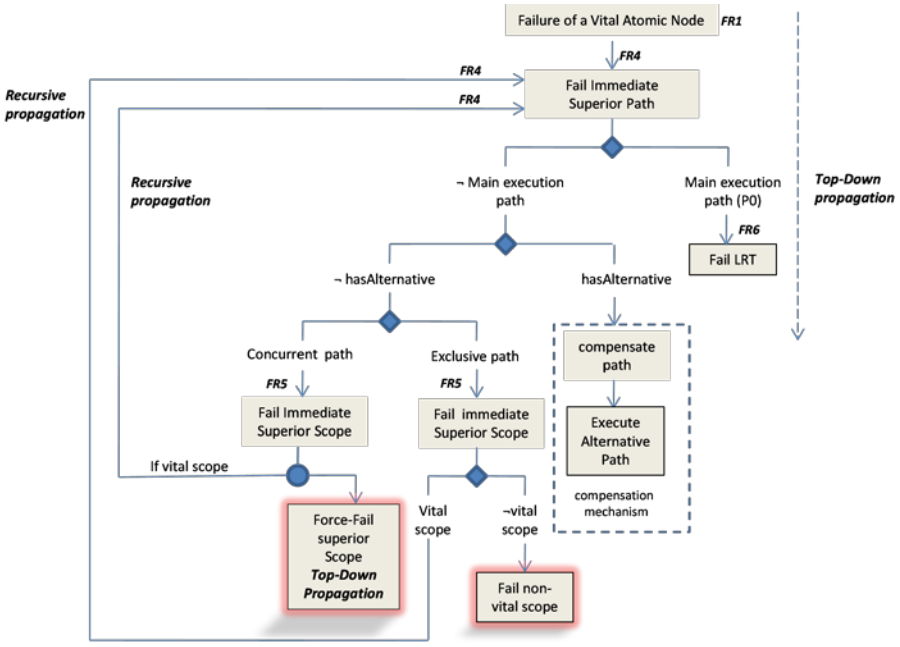

Figure 4. Propagation of vital atomic node failure

The failure mechanism also handles failures of non-vital components. Failure of a non-vital atomic node could fail its enclosing path if the enclosing path was a non-vital path under the following two conditions (1) the enclosing path is an atomic path, i.e. encapsulates one node only, or (2) the

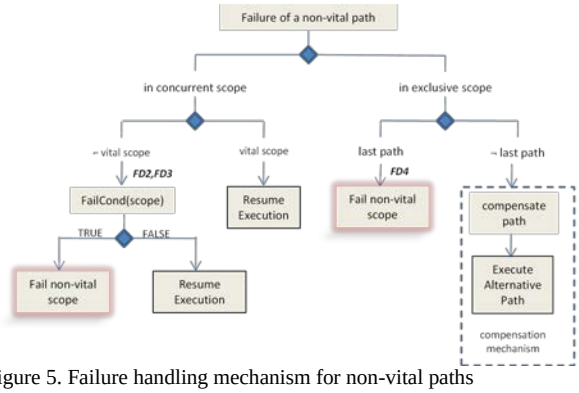

Figure 5. Failure handling mechanism for non-vital paths

node is the last node in the path and all other nodes in the path have failed. Failure of a non-vital path (Figure 5) will only fail its enclosing scope under two conditions: (1) it is an exclusive path (2) it has no alternative, i.e. it is the last exclusive path in the scope. From assumptions 1and 2 in section 3.1, failure of a last non-vital exclusive path will fail a non-vital exclusive scope. Recursively, failure of a non-vital scope is treated as a failure of non-vital node.

To further illustrate the propagation mechanism, we will consider the LRT presented in Figure 3. Assume an execution instance with the following states of its components:n_1, n_2, $scope_1$and $scope_2$ have completed, and $scope_3$ is activated. n_{17} is a vital node and has failed to complete. Table 3 shows $scope_3$'s sub hierarchy tree attribute values and execution states when node n_{17} failure event has been fired and we show how the failure propagation algorithm is employed.

TABLE III. EXECUTION INSTANCES OF $scope_3$

Component	Type	vital	Immediate superior	Has Alternative	Execution state
$scope_3$	AND scope	✓	p_0	-	activated
$scope_3.p_1$	path	✓	$scope_3$	✗	activated
$scope_3.p_2$	path	✗	$scope_3$	✗	activated
$scope_3.p_3$	path	✓	$scope_3$	✗	activated
n_{16}	node	✗	$scope_3.p_2$	-	activated
n_{17}	node	✓	$scope_3.p_3$	-	failed
$scope_{3.1}$	AND scope	✓	$scope_3.p_1$	-	activated
$scope_{3.1}.p_1$	path	✓	$scope_{3.1}$	✗	activated
$scope_{3.1}.p_2$	path	✓	$scope_{3.1}$	✗	activated
$scope_{3.1}.p_3$	path	✓	$scope_{3.1}$	✗	activated
n_{13}	node	✓	$scope_{3.1}.p_1$	-	activated
n_{14}	node	✓	$scope_{3.1}.p_2$	-	activated
n_{15}	node	✓	$scope_{3.1}.p_3$	-	activated

Following the propagation mechanism of Figure 4, failure of n_{17} will fail its superior path $scope_3.p_3$. This is not the main execution path and does not have an alternative but it is a concurrent path since its superior is an AND scope. $scope_3.p_3$ is vital by *evaluation* since it encapsulates vital node n_{17}. Therefore, the immediate scope of $scope_3.p_3$ which is $scope_3$ fails. $scope_3$. is vital by *specification*, hence two actions take place: (a) the failure is propagated recursively one level up in the hierarchy to path p_0. (b) Force fail is recursively propagated in top-down order to cancel all

activated components encapsulated by $scope_3$. Failure of p_0 will fail the LRT (FR6). Failure of $scope_3$ will force fail all its activated paths. At this point of execution, $scope_3.p_3$ has already failed while $scope_3.p_1$ and $scope_3.p_2$ are still activated and therefore both are forced to fail. Force failing a path, fails the activated node in that path. Therefore, activated nodes n_{16} and $scope_{3.1}$ are forced to fail. $scope_{3.1}$ is a scope node and hence the force fail mechanism is recursively repeated one level down in the hierarchy to force fail $scope_{3.1}$'s activated components in same manner as scope3's activated components were forced to fail.

In the above example, failure of a vital node $scope_3$ on p_0 caused the LRT to fail. Our management/compensation model applies a reliable mechanism that controls failure of the LRT in designer-specific order that reflects the business logic of the transaction. In case of force failing a scope that has un-activated paths, these paths can never activate since their enclosing scope state is failed ensuring correctness of the model and avoiding activation of paths in failed scopes.

VI. Related work

[1], [1] and [3] introduced transactional patterns. Control and transactional dependencies are defined for component web services and are mapped onto workflow patterns. Dependencies expressed in first order logic are employed to validate transactional behaviour of web service compositions. [6] proposes an event-driven approach where dependencies are defined in event calculus. These works discuss simple patterns such as AND-split or XOR-split, where a single service exists on each split branch. In addition, the way the dependencies are defined does not allow for nesting in the composite service. The failure handling and recovery mechanism is implemented through dependencies. We have drawn inspiration from that work, but provide solutions for multiple nested transactions.

In [7], REO is used to model the behaviour of LRTs. The approach uses a set of basic REO channels to implement connectors such as sequence and parallel routing. Control flow is monitored through signalling and flow of message tokens through the circuits. Exception handling is implemented by coordinating sequential and parallel activities with compensation activities where each activity is paired with a compensation activity. E.g. an activity cancelled in a sequential flow leads to all previous activities being compensated by passing a cancel token.

Control Flow Intervention (CFI) [8] presents a flexible and automatic failure handling mechanism for composite web services. If a failure of a service occurs at runtime, the failed service is dynamically replaced by a semantically equivalent service(s), thus achieving forward recovery. OWL-S profiles describing service semantics provide a formal framework to reason about semantically equivalent or similar services. The approach supports sequential executions only and parallelism is not addressed. In our approach, a failure of a component service does not necessarily fail the LRT. By applying a combination of forward recovery (implemented by exclusive routing) and a failure propagation mechanism, it is possible to tolerate failures and prevent the LRT from early failure.

VII. Conclusion and future work

In this paper we have presented an approach for modelling and enacting failure recovery on nested long running transactions. The approach provides a novel model that makes explicit the propagation of failure events through the transactions. It also distinguishes two types of nodes — vital and non-vital— that allow a process designer to include activities in the design that are useful but were failure does not matter. The designed propagation rules are enforced through a novel rule based management system, allowing for monitoring and controlling LRTs. A nested workflow is used as example throughout.

Ongoing work considers formalizing extensions to the approach to include compensation mechanisms. In this area we are specifically looking at customized-order compensations and incorporating compensation logic into business logic when designing a process with LRTs, as the designer will have the best understanding on what compensation will be required in the business process.

Acknowledgment

Manar Ali is supported by a PhD scholarship scheme of King Abdul Aziz University (Saudi Arabia).

References

[1] Bhiri, S., C. Godart, and O. Perrin. *Transactional patterns for reliable web services compositions*. Proceedings of ICWE06. p 137-144. ACM, 2006.

[2] Bhiri, S., O. Perrin, and C. Godart. *Ensuring required failure atomicity of composite Web services*. Proceedings of WWW05. p 138-147. ACM, 2005.

[3] Bhiri, S., Perrin, O., Godart, C.; Extending workflow patterns with transactional dependencies to define reliable composite Web services; Proceedings of AICT-ICIW '06; p. 145, IEEE, 2006

[4] Dayal, U., M. Hsu, and R. Ladin; Organizing long-running activities with triggers and transactions. Proceedings of SIGMOD '90. p. 204-214, ACM, 1990.

[5] Elmagarid, A.K.; Transaction models for advanced database applications; Morgan Kaufmann, 1992.

[6] Gaaloul, W., Bhiri, S. and Rouached, M.; *Event-based design and runtime verification of composite service transactional behavior*. Transactions on Services Computing, **3**(1); p. 32-45, IEEE, 2010.

[7] Kokash, N. and F. Arbab, *Formal Design and Verification of Long-Running Transactions with Eclipse Coordination Tools.* Transactions on Services Computing, 2011(99); p. 1-1, IEEE, 2011.

[8] Moller, T. and H. Schuldt. *OSIRIS Next: Flexible Semantic Failure Handling for Composite Web Service Execution*. Proceedings of ICSC10. p 212-217. IEEE, 2010.

[9] Russell, N., ter Hofstede, A.H.M. and Mulyar, N.; Workflow controlflow patterns: A revised view; Technical Report BPM-06- 22; BPM Centre, 2006.

[10] Tan, C. and A. Goh, Implementing ECA rules in an active database. Knowledge-Based Systems, 12(4); p. 137-144, Elsevier, 1999.

[11] Van der Aalst, W.M.P., Barros, A.P., ter Hofstede, A.H.M. and Kiepuszewski, B.; Advanced workflow patterns. Proceedings of CoopIS 2000; p. 18-29, Springer, 2000.

[12] Wieringa, R.; Design methods for reactive systems: Yourdon, statemate, and the UML; Morgan Kaufmann, 2003.