Service–Driven Approaches to Architecture and Enterprise Integration

Raja Ramanathan
Independent Researcher, USA

Kirtana Raja
Independent Researcher, USA
Chapter 6
Maintaining Transactional Integrity in Long Running Workflow Services: A Policy-Driven Framework

Stephan Reiff-Marganiec
University of Leicester, UK

Manar S. Ali
University of Leicester, UK

ABSTRACT
This chapter presents a framework to provide autonomous handling of long running transactions based on dependencies which are derived from the workflow. Business Processes naturally involve long running activities and require transactional behaviour across them. This framework presents a solution for forward recovery from errors by automatic application of compensation to executing instances of workflows. The mechanism is based on propagation of failures through a recursive hierarchical structure of transaction components (nodes and execution paths). The authors discuss a transaction management system that 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. One notable 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. Another novel feature is that in addition to dependencies arising from the structure of the workflow, the approach also permits the workflow designer to specify additional dependencies which will also be enforced. Thus, the authors introduce new techniques and architectures supporting enterprise integration solutions that cater to the dynamics of business needs. The approach is implemented through workflow actions executed by services and allows management of faults through a policy-driven framework.

DOI: 10.4018/978-1-4666-4193-8.ch006
INTRODUCTION

Enterprise integration is significantly eased by the use of Service-driven architectural approaches as diverse systems can be encapsulated as Services which in turn can communicate readily through standard interfaces. This is further enhanced by the concept of executable (and possibly dynamic) Business Processes or Workflows which add a technical layer between the services and the business process as seen by a business analyst (Montangero, Reiff-Marganiec, & Semini, 2011; Gorton et al., 2009). However, enterprise integration is challenging and many of the challenges persist even in the Service-driven environment. One crucial aspect is that of ensuring correct transactional behaviour – both in the sense of not failing in states that are undesirable, and also in terms of attempting to complete the process (suitable concepts would be backward and forward recovery).

The database community has always strived to ensure consistency of data and allow for transactions to complete (possibly at a later stage or later attempt), or to be rolled back to a previous consistent state, and many solutions currently exist. It would seem straightforward to simply employ existing techniques; however, this is hampered by a number of new challenges arising in business processes. The two most crucial are the long running nature of business transactions and the delicate and complex nesting that naturally occurs. Database transactions usually complete within a matter of seconds and the most common solution for addressing the transactional integrity challenge involves some form of locking of resources. This is not applicable to business transactions, also often called Long running transactions (LRTs), these often span several days or even weeks (they typically involve humans making decisions such as approval of applications or time intensive operations such as shipping of physical goods) which clearly forbids any resource locking approach. Also, business processes often perform many actions in parallel or inside nested structures with complex control operators and this must be reflected in the transactions.

One of the important aspects in managing Long Running Transactions is in 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 that systems are maintained consistently. This is made possible by ensuring that the execution of the LRT terminates in an accepted state from the business and the transaction modeling points of view. This will normally occur in the absence of a failure, but the same behaviour should manifest even if the LRT has not completed its normal path of execution due to a failure that causes termination of the LRT or diverts the execution to an abnormal path. This is usually achieved by adopting effective compensation and fault-handling techniques.

Workflows are usually composed out of workflow patterns (van der Aalst et al., 2000), and Bhiri, Perrin, and Godart (2006) have proposed transactional patterns to provide an understanding of the transactional consequences of workflow patterns. We have extended the concepts introduced in the transactional patterns to support multi-level nesting and we introduce the novel concept of vitality of actions, allowing the process designer to express the cruciality of activities in the workflow with respect to the business logic. We also present a framework based on propagation of failures through a recursive hierarchical structure of transaction components (nodes and execution paths). Our transaction management system 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 and policy rules enforce good transactional management at runtime.

We analysed a number of example business processes and derived the following list of aspects that we consider essential for a transaction...
management system to support. Aspects 1 to 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; aspect 4 facilitates to separate the actual process and any handling of exceptions in a clear and user-friendly way; and aspects 5 to 6 are requirements ensuring the practicality of the approach.

1. Multi-level nesting of transactions with reliable behavioral dependencies between transaction components and across hierarchy levels.
2. Definition of designer-order compensation patterns that reflect the business logic of the LRT.
3. Incorporating compensation logic into business logic of long running transactions through transactional dependencies.
5. Automated method for propagating failure events through the hierarchical 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 chapter focuses on handling failures that occur during the normal execution of the LRT (referred to as Failure Management) and the compensation handling mechanisms from a control flow perspective (referred to as Compensation Management). The failure-handling and compensation mechanism are incorporated into the business logic of the transaction through strict logical definitions of behavioral dependencies between transactional components. This chapter defines the structure of the workflow and the required dependencies, and explores the algorithms guiding the reactive controller. This will be presented in the context of the supporting architecture. The work will be placed in the context of the current state of the art in the domain.

The encoding of LRTs as workflows and hierarchical structures and the corresponding failure management has been presented before in Ali and Reiff-Marganiec (2012); however, the compensation mechanism presented in this chapter is novel.

This chapter is organized as follows. First, we describe the adopted transaction modeling paradigm, discussing workflow and transactional patterns. We then present the COMPMOD’s LRT attributes, dependencies, and management rules; the recursive failure handling propagation mechanism, and the compensation mechanism with its rules and dependencies. Through examples and a case study we illustrate the mechanisms. We conclude by summarising and reflecting on future work.

BACKGROUND

Transactions

Database centric transactional models are well understood, supported in practice and provide a strong theoretical foundation for transactions. Transactions can be composed of sub-transactions where tasks and activities are transactions on their own. Failure atomicity and concurrency control are inherent within the models usually enacted through temporary resource locking. Recovery is mainly based on the notion of roll-back and compensation to restore the state of the system to the state before the failure had happened. Coordination support for multi-tasking and collaborative activities across organisations is limited and thus they are not applicable to heterogeneous and loosely coupled systems.

Conventional transactions are ACID transactions (Lewis, Bernstein, & Kiefer, 2001; Gray & Reuter 1993; Gray, 1981). ACID stands for Atomic, Consistent, Isolated and Durable – which
represent four characteristics seen as desirable for transactions as they ensure stable results after a transaction completes (or fails). They have been developed in the context of tightly coupled systems, occur between trusted parties, and run over short periods of time (short-lived) – so they are generally the accepted model for databases. ACID transactions must either fully commit or fully roll back in case of failure.

In ACID transactions, any failure that occurs within the transaction will be rolled back and its effects are erased. In traditional database operations, a one-step operation is called an atomic operation where an operation does not conflict or interfere with other operations using the same database. More complex database operations are called transactions and involve multiple steps that must all be completed for the transaction to succeed. A traditional transaction is a single unit of work that is composed of two or more tasks. As databases are more distributed, some of the initial constraints have been loosened and protocols such as the two-phase commit (2PC) protocol (Mohan & Lindsay, 1985) have prolonged the lifespan and usefulness of ACID transactions allowing for transactions to span multiple local database systems.

However, these protocols and solutions do not address the needs encountered by long lived and complex transactions. For long-lived transactions, individual constituent sub-transactions maybe ACID, but the overall business transaction employs a compensatory approach to reverse or erase partial work. For this reason, a number of extended and relaxed transactional models have been proposed which relax some of the ACID requirements.

Nested Transactions (Moss, 1982) allow transactions to be nested within other transactions to form a tree structure, that is, a transaction is decomposed into a hierarchy of sub-transactions. A child can only start after its parent has started and a parent may terminate only after all its children have terminated. Each sub-transaction can either commit or roll-back. The ‘commit’ of a child will only take place if its parent commits. If a parent rolls back, all its children must roll back. This is applied in a recursive manner. This model has advantages: it provides full isolation, better failure handling, and allows concurrent execution of sub-transactions. The Open Nested Transactions Model (Weikum & Schek, 1992) is a generalization of nested transactions. It relaxes the isolation property by allowing the results of committed sub-transactions to be visible to all top level transactions. Sub-transactions can commit and release resources before their predecessor transaction successfully completes and commits. The abort of a top-level transaction requires roll-back for committed sub-transactions.

The Saga Transactional Model (Garcia-Molina & Salem, 1987) relaxes the full isolation requirement and provides an increased inter-transaction concurrency. A Saga divides a long running transaction into a sequence of ACID sub-transactions. Each sub-transaction has an associated compensating sub-transaction which can be executed in case the effects of its associate need to be undone. Saga describes a mechanism for handling LRT within relational databases. It supposes that a LRT is composed of a sequence of smaller inner transactions which could be interleaved with inner transactions from other Saga. Each inner transaction retains the ACID properties. The Saga itself is not ACID.

The most important feature of a Saga is its failure handling mechanism. There are two modes of recovery, backward recovery and forward recovery. Each inner transaction is provided by its own compensation handler which is responsible for cancelling or reversing the effects of its associated transaction. In backward recovery, if any of the inner transactions of a saga has failed, it is rolled back. The Saga then executes the compensation handler for all previously committed inner transactions, in reverse order; Forward recovery depends on the existence of save points. A save point is where the state of Saga is persistent. Hence, when
Maintaining Transactional Integrity in Long Running Workflow Services

a failure occurs, the Saga restarts execution from the save point. If one or more inner transactions have committed between the save point and the failure point, they first must do backward recovery by executing compensating transactions for these committed transactions and then restart from the save point.

Flexible transactions (Elmagarmid, 1992; Zhang et al., 1994; Mehrotra et al., 1992) propose approaches suitable for a multi database environment. Transactions are defined as global transactions and composed of sub transactions and a set of execution dependencies such as commit, alternative, and failure dependencies on sub-transactions. These models depend in their correctness on weak-atomicity of the global transaction by relaxing the 2-PC protocol. Failure recovery is handled by retriable and compensable transactions and hence flexible transactions provide better resilience to failures than traditional transaction models.

ACTA model (Chrysanthis & Ramamritham, 1990) presents a framework for specifying and reasoning about transaction structure, concurrency and recovery. The model formalizes the effects of transactions on other transactions and on objects through commit and abort dependencies, using predicate logic.

Business Processes and Workflows

Business Processes are usually defined by business analysts to capture the activities and their respective order and dependencies required to achieve some larger business goal. The result of such a process definition is a workflow. Modeling of such workflows is usually conducted in some graphical notation such as BPMN (White, 2004) or UML activity diagrams. However, there are other notations, such as YAWL (van der Aalst & ter Hofstede, 2005) which are graphical and textual and have formally defined semantics.

Workflow systems integrate, automate and manage business processes through flexible representations of the control flow of their tasks. A Service-driven workflow process is composed of Web Services that relate to each other through workflow constructs such as sequence, split and join; to allow for sequencing, parallelism or choices in the control flow. A workflow management system is required to coordinate the sequence of service invocations within a process, to manage control flows and data flows between Web Services, and to ensure execution of the process as a reliable transaction unit (Yan et al., 2005).

(Kiepuszewski, ter Hofstede, & Bussler, 2000) provide formal definitions for arbitrarily structured and well behaved workflows. A structured workflow consists of symmetrical blocks of AND-split followed by AND-join or OR-split followed by an OR-join. A workflow is well behaved if “it can never lead to deadlock nor can it result in multiple active instances of the same activity.” Their work shows that every structured workflow is well behaved and provides transformation techniques to transform non-structured workflows into structured ones.

(Reichert & Dadam, 1997; Reichert & Dadam, 1998), present a framework (ADEPTflex) to support ad-hoc and dynamic deviations from premodeled Workflow (WF) activities. This framework is based on well-structured workflows and proposes a minimal and complete set of change operations to support dynamic structure modification of a running workflow. Change operations are supported with correctness properties which ensures the correctness and consistency of the resulting WF graph by construction.

The workflow management system in (Müller, Greiner, & Rahm, 2004) is a step towards dynamic and automatic workflow adaptations in case of failure events. The management model is based on temporal rule-based approach to specify exceptions such as logical failures and perform necessary workflow adaptations as a failure recovery mechanism.

The work in (Casado et al., 2012) proposes an abstract model for dynamically modeling Web
Service transactions, based on BTP (Business Process Protocol) and WS-transaction standards. They apply a model-based testing tool to generate test scenarios, and evaluate the reliability of WS-standards in terms of failures.

The work in Qiu et al. (2005) presents a formal operational semantics BPEL (Business Process Execution Language) as a simplified version of BPEL4WS (omitting data handling semantics) to highlight its fault handling and compensation semantics. Activities are enclosed by scopes and each scope is associated with fault handler and compensation handler (default or programmed). An exception within a scope invokes its fault handler and if compensation is required, the fault handler invokes the compensation handler which associates compensation context with each activity. The compensation mechanism is based on accumulating compensation contexts of completed activities such that when a scope is compensated, compensation contexts are invoked in reverse order of their installation.

The work in (Butler, Hoare, & Ferreira, 2005; Butler & Ripon, 2005) proposes a compensating CSP (cCSP) modeling approach for LRTs based on Process Algebra (Hoare, 1978). Operational semantics of cCSP are modelled as follows: atomic actions are aggregated through sequencing, choice, and parallel operators to compose standard processes where processes can be aggregated as well to form a business process transaction. A compensable standard process is a process that is paired with compensating actions. The model provides execution and compensation primitives to control execution flow; SKIP for successful termination, THROW for throwing an interrupt, and YIELD to indicate that an interrupt is willing to yield between the execution of two processes. When an atomic transaction fails, sequential compositions of compensable processes are executed in reverse order while compensations of parallel compositions are accumulated in parallel.

Parallel Sagas are proposed in Bruni, Melgratti, and Montanari (2005) by adding increased expressiveness to LRT representations and the work is supported by a hierarchy of transaction calculi to model parallelism, nesting, and choices. The definition of compensation is part of the Saga (compensation pairs like cCSP) but provide a richer form of exception than cCSP. The model provides primitives to allow execution of alternatives to an aborted sub-transaction as well as discriminator choices.

In Kokash and Arbab (2011) REO (a channel-based exogenous coordination language) 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. For example, an activity cancelled in a sequential flow leads to all previous activities being compensated by passing a cancel token.

Control Flow Intervention (CFI) (Moller & Shuldt, 2010) 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.

Workflow and Transactional Patterns

Bhiri, Godart and Perin (Bhiri, Godart, & Perrin, 2006; Bhiri, Perrin, & Godart, 2006; Bhiri, Perrin, & Godart, 2005) introduced transactional
Maintaining Transactional Integrity in Long Running Workflow Services

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. Galoul, Bhiri, and Rouached (2010) propose 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.

Workflow patterns have been developed as part of an initiative commenced in 2000 by van der Aalst et al. (2000). 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 modeling. Workflow patterns consider workflow specifications from a control-flow perspective and characterize a range of control flow patterns that might be encountered when modeling a business workflow. Following the initial work (van der Aalst et al. 2000), 43 control patterns were proposed in Russel, ter Hofstede, and Mulyar (2006). 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, COMP-MOD, 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 Bhiri, Godart, and Perrin (2006). Transactional patterns are aimed at specifying flexible and reliable Composite Web Services. They are a convergence concept between workflow patterns and advanced transactional models (Elmagarid, 1992), and thus they combine the flexibility of work flow control patterns with the reliability of transactional models to ensure transactional consistency of service compositions. Transactional patterns define orchestrations between services in a 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. Services can change their state based on internal behaviour or on external stimuli – both would be transitions, the latter being externally triggered and often referred to as external transitions.

The general definition of a dependency is:

**Dependency:** A dependency from service s1 to service s2 exists if a transition of s1 can fire an external transition of s2. (Bhiri, Godart, & Perrin, 2006).

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()`).

**Policies**

Management rules (or policies) incorporate autonomy into systems by describing how a system is to adapt its behaviour under certain circumstances. The most common form in which the rules are described is that of an ECA (event condition action) rule, which presents an event-driven approach.

Policies have been used in many systems, with the most common occurrence being in access control (e.g., Siewe, Cau, & Zedan, 2003; Halpern & Weissman, 2003), in usage control (Zhang et al., 2005), in telecommunications (Turner et al.,
and in service-oriented computing (Buscemi et al., 2007; Gorton & Reiff-Marganiec, 2006) are documented.

One of the first attempts in applying ECA rules approach in management of transactions in WF systems was in (Dayal et al., 1990) by using triggers for organizing long running activities. ECA rules have been used to adapt workflows and provide more fine-grained specification for service selection for tasks in (Müller, Greiner, & Rahm, 2004) and in database management systems (Paton, 1999; Widom & Ceri, 1996).

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.

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 Bhiri, Perrin, and Godart (2006). 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 operator creates a scope with parallel paths where only a subset of these paths are executed during runtime, the executed paths are those whose enabling conditions are satisfied. An OR scope successfully completes if all its enabled activity 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 hasAlternative, 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. Informally, a scope groups semantically related nodes together and we can formally define a scope node as follows:

Scope Node:

\[
\forall_{i=1..m} p_i, nodeList = splitNode, \forall_{i=1..m} nodeList . type = \{ATOMIC, SCOPE\} : scope \\
= (operator, [splitNode_1 . . . splitNode_m]) \\
\rightarrow scope.pathList = [p_1 . . . p_m]
\]

where operator \(\in\) \{AND,OR, XOR\}.

Consider some of the notations introduced here as they are used throughout this work. Components, i.e. scopes and paths, consist of ordered list of nodes. An object-oriented attribute dot-type notation is used to identify attributes of components. Nodes are either atomic or scope
nodes, which is reflected in their type. $p$ denotes a path (there might be indices to differentiate different paths).

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 $p_i.nodeList$.

**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 (an 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 chapter.

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 at the business level. Therefore, it is possible for a designer to define a scope node as a non-vital node, even when it encapsulates vital paths, without leading to an incorrect model.

Note that we make a fundamental assumption that workflows are structured in a well formed way where scopes are completely enclosed inside other scopes and do not overlap in random ways. This assumption makes the approach easier to explain and many of the practical workflows that we have encountered do fulfil this requirement, with the others that we came across allowing for easy syntactical rewrites bringing them into this structure.

However, the following logical restrictions are assumed by the approach with respect to design of scopes (and they are assumptions that could be considered for relaxation in the future):

**Assumption 1:** In an exclusive scope, all exclusive paths should have the same vitality measure, which is they must all be vital or non-vital.

**Assumption 2:** If all paths in a scope are non-vital, their encapsulating scope should be non-vital by specification.

Assumption 1 might be seen as quite restrictive, however, considering it at a more business-oriented level, it essentially says that if there is a choice, each of the alternatives that one could choose from are ultimately of equal vitality. Not having this assumption would allow for a kind of free choice between maybe doing something (or not) – the non-vital route – and a strong requirement of doing something and succeeding in it – the vital route. This seems simply wrong: consider a
human decision in a notification task: you must send a letter (vital) or alternatively you could send an email (non-vital).

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. Figure 1 shows the state transition diagram of atomic nodes. State transitions are triggered by events. For example in Figure 1, when a completion of an atomic node is triggered, the execution state of the node changes from ACTIVATED to COMPLETED.

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\}

Representations of Nested LRTs

We use two main representations of the workflows in our work: a workflow representation which 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 transactional behaviour between transaction components. Transactional dependencies are defined between a component and its neighbours.

**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 demonstrates a two level nested LRT that consists of atomic nodes and nested scopes. Considering execution \( p_1 \) in \( \text{scope}_2 \), the path consists of an atomic node \( n_5 \) followed in sequence by a scope node \( \text{scope}_{2.1} \) which 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, \( p_1.\text{nodeList}=\{n_5, \text{scope}_{2.1}\} \). 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 \( p_0.\text{nodeList}=\{n_1, n_2, \text{scope}_1, \text{scope}_2, \text{scope}_3\} \) in sequential order (see Figure 3).

**Hierarchical Structure Model**

Transaction components –nodes and execution paths-- are linked together in a hierarchical structure (see Figure 4). Each component has a single superior and an ordered set of one or more inferiors.

*Figure 2.*
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 the path encloses. The root of the recursive hierarchy is the main execution path of the LRT $p_0$.

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 such as 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 leads to an event being fired. In the scope of this chapter, we focus on failure and force-fail dependencies.

As we are using an event based mechanism to control state changes, it is meaningful to also express transactional behaviour in the same way. For that we allow components to raise events to notify other parts of the system of transactional requirements; such events are called transactional events in our approach.

The general definition for a behavioral dependency is:

**Behavioral Dependency:** A behavioral dependency exists from component $j$ to component $i$ if a state transition in component $i$ can fire a transactional event for component $j$:

\[
\text{Dep}(\text{component } j) := \text{preCond}(\text{component } i.\text{state})
\]

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:

\[
\text{ActDep}(\text{succNode}) = (\text{PredNode.State} = \text{COMPLETED}) \lor (\text{PredNode.Vital} = \text{FALSE} \land \text{PredNode.State} = \text{FAILED})
\]

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

\[
\text{Dep}(\text{superior}) = \text{preCond}(\text{Isibling..state..sibling..state})
\]

Also note that the behavioral dependency defines a trivial compensation dependency, such that a state transition in component $i$ can fire a compensation event for component $j$.

### 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 later. Table 1 shows a complete list of failure and force fail dependencies. Failure of all vital nodes in a path will fail the path ($\text{path.nodeName} \leftarrow \text{fail; path; FD1}$); failure of paths in a scope will lead to failure of the scope ($\text{scope.pathName} \leftarrow \text{fail scope}$), 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 cancellation. 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 $\text{component.superior} \leftarrow \text{forcefail component}$ 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.
FAILURE MANAGEMENT

Management Rules

Event Condition Action (ECA) rules in COMPMOD are used 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:

\[
\text{ON event IF condition DO action}
\]

The event part of the rule can be (a) an internal system generated event such as completion, failure, or cancellation 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 executing a 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 chapter focuses on failure handling, we only list failure and failure propagation ECA rules in Table 2. Note that \textit{fail} and \textit{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:

\[
\text{if component.state=ACTIVATED}
\]

\[
\text{then component.state:=FAILED}
\]

ECA rules of COMPMOD reflect the following:

- The business logic of the LRT (e.g. FR4 states that if a node is vital and failed, its superior path fails).
- 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).

Table 1. Fail and force-fail dependencies

<table>
<thead>
<tr>
<th>Dependency</th>
<th>Dependency Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>Failure Dependencies</td>
<td></td>
</tr>
</tbody>
</table>
| FD1 | \[\text{FailDep}(\text{path}) = \bigwedge_{1 \leq i \leq m} \left[ \text{nodeList}_i . \text{State} = \text{FAILED} \right] \]
| where | \[\text{path} . \text{nodeList}\] |
| FD2 | \[\text{FailDep(\text{ANDscope})} = \bigwedge_{1 \leq i \leq m} \left[ \text{path}_i . \text{State} = \text{FAILED} \right] \]
| where | \[\text{ANDscope} . \text{pathList}\] |
| FD3 | \[\text{FailDep(\text{ORscope})} = \bigvee_{1 \leq i \leq m} \left[ \text{path}_i . \text{State} = \text{FAILED} \right] \]
| where | \[\text{ORscope} . \text{pathList}\] |
| FF1 | \[\text{ForceFailDep}(\text{path}) = \text{immediateSuperior}.\text{State} = \text{FAILED} \]
| FF2 | \[\text{ForceFailDep(node)} = \text{immediateSuperior}.\text{State} = \text{FAILED} \]
Maintaining Transactional Integrity in Long Running Workflow Services

Table 2. Failure and propagation rules

<table>
<thead>
<tr>
<th>Rule</th>
<th>Pseudo ECA-Rule statement</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Failure Rules</strong></td>
<td></td>
</tr>
<tr>
<td>FR1</td>
<td>ON “internal failure/cancellation event fired for atomic node” DO fail(node)</td>
</tr>
<tr>
<td>FR2</td>
<td>ON FailDep(node)=TRUE IF node.type=SCOPE DO fail(node)</td>
</tr>
<tr>
<td>FR3</td>
<td>ON FailDep(path)=TRUE DO fail(path)</td>
</tr>
<tr>
<td>FR4</td>
<td>ON fail(node) IF node.vital=TRUE DO fail(node.superior)</td>
</tr>
<tr>
<td>FR5</td>
<td>ON fail(path) IF path.hasAlternative = FALSE and path.vital = TRUE DO fail(path.superior)</td>
</tr>
<tr>
<td>FR6</td>
<td>ON fail(p0) DO fail(LRT)</td>
</tr>
<tr>
<td><strong>Failure Propagation Rules</strong></td>
<td></td>
</tr>
<tr>
<td>FFR1</td>
<td>ON ForceFailDep(node) IF node.type=SCOPE and node.state=activated DO fail(node)</td>
</tr>
<tr>
<td>FFR2</td>
<td>ON ForceFailDep(node)=TRUE IF node.type=ATOMIC and node.state=activated DO abort(node)</td>
</tr>
<tr>
<td>FFR3</td>
<td>ON ForceFailDep(path)=TRUE IF path.state=activated DO fail(path)*</td>
</tr>
</tbody>
</table>

- 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).

**Failure Propagation Mechanism**

This work presents a recursive method for propagating vital failure events through the recursive hierarchical 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 hierarchical structure, the recursive failure propagation mechanism entails a combination of three types of propagation methods:

- **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.

- **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 ($p_0$).

- **Downwards Recursive Propagation:** Originates from a failure of a scope node (vital or non-vital) by repeating a top-down
Maintaining Transactional Integrity in Long Running Workflow Services

propagation to its immediate activated paths until the propagation reaches all active atomic nodes within the failed scope’s sub-hierarchy. This represents a means 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:** 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 5 and 6 illustrate key parts of the failure propagation mechanism linked to dependencies and ECA rules (Tables 1 and 2). Figures 5 and 6 include compensation mechanisms that tie in with work presented in the next section.

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 node is the last node in the path and all other nodes in the path have failed. Failure of a non-vital path (Figure 6) 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

**Figure 5. Propagation of vital atomic node failure**
Maintaining Transactional Integrity in Long Running Workflow Services

Figure 6. Failure handling mechanism for non vital nodes

and 2 earlier, 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 briefly illustrate the propagation mechanism, consider the LRT presented in Figure 4. Assume an execution instance with the following states of its components: \( n_1, n_2, \text{scope}_1 \) and \( \text{scope}_2 \) have completed, and \( \text{scope}_a \) is activated. \( n_{17} \) is a vital node and has failed to complete. All other nodes and paths are vital apart from \( p_2 \) and \( n_{16} \). All nodes and paths in the subtree are activated.

Following the propagation mechanism of Figure 4, failure of \( n_{17} \) will fail its superior path \( \text{scope}_a.p_a \). 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. \( \text{scope}_a.p_a \) is vital by \textit{evaluation} since it encapsulates vital node \( n_{17} \). Therefore, the immediate scope of \( \text{scope}_a.p_a \) which is \( \text{scope}_a \) fails \( \text{scope}_a \) is vital by \textit{specification}, hence two actions take place: (a) the failure is propagated recursively one level up in the hierarchy to path \( p_a \) (b) Force fail is recursively propagated in top-down order to cancel all activated components encapsulated by \( \text{scope}_a \). Failure of \( p_a \) will fail the LRT (FR6). Failure of \( \text{scope}_a \) will force fail all its activated paths. At this point of execution, \( \text{scope}_a.p_a \) has already failed while \( \text{scope}_a.p_j \) and \( \text{scope}_a.p_2 \) 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 \( \text{scope}_{3,1} \) are forced to fail. \( \text{scope}_{3,1} \) is a scope node and hence the force fail mechanism is recursively repeated one level down in the hierarchy to force fail \( \text{scope}_{3,1} \)'s activated components in same manner as \( \text{scope}_3 \)'s activated components were forced to fail. In this example, failure of a vital node \( \text{scope}_a \) on \( p_o \) 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.

COMPENSATION MANAGEMENT

While executing a long running transaction, faults might occur – often this is not a problem as alternatives can be probed and often the long running transaction can be successfully completed. Consider the scenario of booking a holiday; if one hotel is full one might find an alternative hotel and still get the desired rest. However, usually
when faults occur, a path in the business process has been travelled along, possibly for quite some time, making commitments to specific services along the route. However, sometimes there is simply no alternative (or all alternatives have been explored – it is a small town on a remote island and all hotels are fully booked) and the long running transaction will fail.

In both cases we require compensation, and COMPMOD caters for this by supporting two types of compensation modes:

1. **Partial Compensation**: Occurs where some compensation actions take place while the LRT is executing in its normal mode (in the model the LRT state is *activated*). Partial compensation is applied to nodes, paths, and scopes in tolerance with failures and it primarily reflects WF semantics.

2. **Comprehensive Compensation**: Needed when an explicit consensus is reached about the failure of the LRT. The LRT starts its global compensation applying it to all successfully completed atomic nodes in a customised-order that is defined by the business process designer at design time. Comprehensive compensation mainly reflects the compensation logic of the business process.

We will now explore the two types of compensations in more detail. We will focus on Partial Compensation but also provide an overview of the Comprehensive Compensation and custom-order aspects.

### Partial Compensation

Partial compensation is triggered by failure of an exclusive path that has an alternative. Exclusive scopes encapsulate paths that alternate each other in execution such that only one path is allowed to succeed. If an activated path has failed to successfully complete, which is mainly triggered by a failure of a vital node on the path or by failure of all its encapsulated nodes, then all nodes on the path that have successfully completed (if any) are compensated. When the failed path has completed its compensation actions, an activation event is fired for its alternative path.

When compensating a path, the current state of its encapsulated nodes at the time the failure happened is important for deciding the compensating actions to be performed on these nodes, so we consider the following possible situations:

- The failed exclusive path might contain nodes that have succeeded, failed, or not been activated (i.e. the failure occurred before the node has been activated).
- Nodes might be scopes, and hence, if they were activated and some tasks had succeeded within the scope, then their work has to be compensated.
- The path is an atomic path that encapsulates a single node, and its failure has caused the failure of the path; the node could be either atomic or a scope.

We adopt two widely used terminologies in Transaction Processing: Forward Compensation and Backward Compensation and give them a precise definition in COMPMOD.

1. **Forward Compensation**: Used to refer to the compensation process of an exclusive path that has an alternative but failed to complete. Forward compensation starts by compensating the last node on the path and completes when the first node on that path has completed its compensating actions at which point the alternative path will be attempted.

2. **Backward Compensation**: Used to refer to the compensation process of a scope node that has previously succeeded or failed (i.e., some partial work could have succeeded within the scope). We define
backward compensation for scopes that are contained within potentially compensable paths. Backward compensation of a scope starts by compensating all its encapsulated paths concurrently in backward order. The backward compensation of each path is processed in the same manner as in forward order, that is, starting from last node and cascading compensation events along the nodes on the path until the first node on the path has completed its compensating actions.

A potentially compensable path is a path that can possibly, in case of tolerable failures and during the normal execution mode of the LRT, have some compensating actions applied to it. Hence, a forward compensable path and a backward compensable path (a path within a backward compensable scope) are both potentially compensable paths. Analogously, a node is potentially compensable if it is encapsulated with a potentially compensable path. In COMP-MOD, all potentially compensable components are defined with compensation dependencies. However, compensations of nodes on a compensating path are always performed in reverse order of their activations. Therefore, whether a path is in forward or backward compensation mode, the order by which nodes are compensated is always in reverse order of their activations.

We require some preliminary artifacts before we can consider compensation dependencies, one of these is concerned with formalizing the notion of being compensable, while the other looks at identifying nodes that must be compensated. For the former, we define an attribute for LRT components, \( \text{IsCompensable} \), and its value is computed as follows:

1. The main execution path is not compensable since if it fails, the LRT has failed and we leave the normal execution mode of the LRT to start comprehensive compensation:

\[
\text{path} = p_0 \rightarrow \text{path.IsCompensable} = \text{FALSE}
\]

2. A path \( \text{IsCompensable} \) if the path has an alternative:

\[
\begin{align*}
\text{path.hasAlternative} &= \text{TRUE} \rightarrow \\
\text{path.IsCompensable} &= \text{TRUE}
\end{align*}
\]

3. A scope node \( \text{IsCompensable} \) if its superior path \( \text{IsCompensable} \):

\[
\begin{align*}
\text{scopeNode.superior.IsCompensable} &= \text{TRUE} \rightarrow \text{scopeNode.IsCompensable}
\end{align*}
\]

4. A path that has no alternative \( \text{IsCompensable} \) if it’s superior scope \( \text{IsCompensable} \). This applies to the case of concurrent paths (e.g. AND), and the last exclusive path in an exclusive scope (e.g. XOR).

\[
\begin{align*}
\text{scope.IsCompensable} &= \\
&= \text{TRUE} \land \neg \text{scopeInferior.hasAlternative} \rightarrow \\
&= \text{scopeInferior.IsCompensable} = \text{TRUE}
\end{align*}
\]

Items 3 and 4 do of course include the notion that compensation of some inner components depends on their enclosing environment, so if that environment can offer alternatives then they will be compensable; if the environment does not offer alternatives they will not be compensable.

It is intuitive that compensation is only required to undo actions of atomic nodes that succeeded (it is only those that might have an effect on the world), we need to be able to identify such nodes. Such atomic nodes can only exist on a successful path or failed path (there might be some nodes earlier on the path that succeeded) but never on not-activated paths or previously compensated paths. This can be further extended to enclosing scopes, where not-activated scopes will only contain not-activated paths.
So, we have two rules telling us which components can be skipped when considering compensations:

**CR1:** If the component was an atomic node that has not succeeded (i.e. it failed, was not-activated, or was aborted) or it was a scope node that was not activated, the node is skipped (i.e. its state is marked as SKIPPED).

**CR2:** If the component was a not-activated or previously compensated path, no action is taken for its compensation event; hence the state of the path does not change.

### Compensation Dependencies

Considering compensations, we have two types of dependencies: those that we refer to as compensation dependencies and those that are called compensation completion dependencies. The former capture the targets for compensation events, while the latter are concerned with notifications of completed compensations. Compensation dependency exists

- Between a node and its successor (if any) if its superior path *IsCompensable*,
- Between a path and its superior scope if the superior scope *IsCompensable*, and
- Between the last node and its encapsulating path if the path *IsCompensable*.

Table 3 details the compensation dependencies. A compensation event is fired for the last node on a compensable path when the path has commenced its compensation (CompD.1) and is fired for a path when its superior scope has commenced its compensation (CompD.3). (CompD.2) enforces the reverse order of compensation activation such that a compensation event is fired for a node if its successor on the path has been compensated or skipped.

Compensation completion dependencies are defined for compensable paths and scopes to signal the end of their compensation process (Table 4) such that when fired, they are marked by a completion policy as COMPENSATED. Atomic nodes raise an event when compensation is completed (we assume that compensation completion is guaranteed to succeed) and the state of the node will be COMPENSATED. A path ends its compensation process when the first node in the path has either compensated or skipped (CpCompLD.1). To reach a consensus about compensation completion of a scope, we have to evaluate all possible states of its encapsulated paths at the time the scope has SUCCEEDED or FAILED.

### Partial Compensation Mechanism

As with fault handling, partial compensation is automated through compensation policies (Table 5) and compensation completion policies (Table 6). All compensation policies contain a
Maintaining Transactional Integrity in Long Running Workflow Services

Table 4. Compensation completion dependencies

<table>
<thead>
<tr>
<th>Dep #</th>
<th>Dependency</th>
<th>Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>CpCompi.D.1</td>
<td>$\text{CpComplDep(path) = (firstNode.State = COMPENSATED) v firstNode.State = SKIPPED}$</td>
<td>Compensable Path</td>
</tr>
<tr>
<td>CpCompi.D.2</td>
<td>$\text{CpComplDep(scope) = scope.state = compensating } \land \text{(pathList..state = COMPENSATED v pathList..state = NOT - ACTIVATED)}$</td>
<td>Compensable scope with m paths</td>
</tr>
</tbody>
</table>

Table 5. Compensation policies

<table>
<thead>
<tr>
<th>Rule#</th>
<th>Compensation Policies</th>
<th>Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>CompR.1</td>
<td>ON fail(path) &lt;br&gt; IF path.hasAlternative and node.superior.state=ACTIVATED &lt;br&gt; DO compensate(path)</td>
<td>Exclusive path with alternative</td>
</tr>
<tr>
<td>CompR.2</td>
<td>ON CompDep(path) &lt;br&gt; IF LRT.State=ACTIVATED and (path.state=SUCCEEDED or path.state=FAILED) &lt;br&gt; DO compensate(path)</td>
<td>Compensable path previously succeeded or failed</td>
</tr>
<tr>
<td>CompR.3</td>
<td>ON CompDep(node) &lt;br&gt; IF LRT.State=ACTIVATED and node.Type=ATOMIC and node.State=SUCCEEDED &lt;br&gt; DO compensate(node)</td>
<td>succeeded Atomic node</td>
</tr>
<tr>
<td>CompR.4</td>
<td>ON CompDep(node) &lt;br&gt; IF LRT.State=ACTIVATED and node.Type=SCOPe and (node.State=FAILED or node.state=NOT-ACTIVATED Or nodeState=ABORTED) &lt;br&gt; DO skip(node)</td>
<td>Non-succeeded atomic node</td>
</tr>
<tr>
<td>CompR.5</td>
<td>ON CompDep(node) &lt;br&gt; IF LRT.State=ACTIVATED and node.Type=SCOPe and (node.State=SUCCEEDED or node.state=FAILED) &lt;br&gt; DO compensate(node)</td>
<td>Succeeded or failed scope</td>
</tr>
<tr>
<td>CompR.6</td>
<td>ON CompDep(node) &lt;br&gt; IF LRT.State=ACTIVATED and node.Type=SCOPe and node.state=NOT-ACTIVATED &lt;br&gt; DO skip(node)</td>
<td>Not activated scope</td>
</tr>
</tbody>
</table>

Table 6. Compensation completion policies

<table>
<thead>
<tr>
<th>Rule#</th>
<th>Compensation Policies</th>
<th>Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>CpCompi.R.1</td>
<td>ON &quot;internal compensation completion event of atomic node&quot; &lt;br&gt; IF LRT.State=ACTIVATED &lt;br&gt; DO compensated(node)</td>
<td>Atomic node</td>
</tr>
<tr>
<td>CpCompi.R.2</td>
<td>ON CpComplDep(path) &lt;br&gt; DO compensated(path)</td>
<td>Path</td>
</tr>
<tr>
<td>CpCompi.R.3</td>
<td>ON CpComplDep(node) &lt;br&gt; IF node.Type=SCOPE and node.state#SKIPPED &lt;br&gt; DO compensated(node)</td>
<td>Concurrent Scope</td>
</tr>
<tr>
<td>CpCompi.R.4</td>
<td>ON compensated(path) &lt;br&gt; IF path. IsExclusive and Path.superior.state=compensating &lt;br&gt; DO compensated(path.superior)</td>
<td>Exclusive path</td>
</tr>
</tbody>
</table>
consistency condition ($LRT\.state=\text{ACTIVATED}$) to differentiate between the partial compensation mode and comprehensive compensation mode ($LRT\.state=\text{COMPENSATING}$), such that compensation events are handled reliably in the correct mode of compensation.

The partial compensation mechanism operates as follows (Figure 7 and Figure 8 contain a graphical representation of some of the steps). Please note that the steps indicate which rules are applicable, however the reactive controller will be based on events available in the system and apply the correct rules automatically:

1. When a failure event is fired for an exclusive path with an alternative, the event is assessed by (CompR.1 - Table 5) and the path is marked COMPENSATING.
2. A compensation event fired for a compensable path is assessed by (CompR.2 - Table 5) and the path is marked COMPENSATING.
3. When a path commences its compensation, a compensation event is fired for the last node in the path (CompD.1 – Table 3).
4. When a compensation event is fired for an atomic node, if the node has succeeded, the event is assessed by (CompR.3 – Table 5) and the nodes start COMPENSATING.
5. When a compensation event is fired for an atomic node, if the node has not been succeeded, the event is assessed by (CompR.4 – Table 5) and the node is SKIPPED.
6. When an internal compensation completion event is fired for an atomic node, the node is marked as COMPENSATED (CpCompLR.1 – Table 6).
7. A compensation event fired for a non-activated scope is assessed by (CompR.6 – Table 5) and the scope is SKIPPED.
8. A compensation event fired for a SUCCEEDED or FAILED scope is assessed by (CompR.5 – Table 5) and the scope starts compensating.

Figure 7. Compensation of path

![Diagram of compensation process](chart13.png)

* A compensable path within a compensating concurrent scope
** An exclusive path within an activated exclusive scope
9. When a scope commences its compensation, a compensation event is fired for all its encapsulated paths (CompD.3 – Table 3) and control goes to step 2.

10. When a node is COMPENSATED or SKIPPED, if the node was the first node in the path, a compensation completion event is fired for the path (CpCompLD.1 – Table 4) and the path is marked compensated by (CpCompLR.2).

11. If a COMPENSATED or SKIPPED node has a predecessor node, a compensation event is fired for the preceding node (CompD.2 – Table 3) and control goes to step 4 or 5.

12. When a compensation completion event is fired for an exclusive path within an activated exclusive scope, an activation event is fired for the next alternative path (ActD.6) and the node is activated by activation policy (ActR.3).

13. When a compensation completion event is fired for an exclusive path within a compensating exclusive scope, the scope is marked COMPENSATED by policy (CpCompLR.4 – Table 6).

14. When a compensation completion event is fired for a concurrent scope (CpCompLD.2 – Table 4), the scope is marked compensated by policy (cpCompLR.3 – Table 6).

For further illustration of the mechanism, let us consider the scope of a larger process depicted in Figure 9. Consider a scenario, where \( n_2 \) and \( n_6 \) succeed, and non-vital \( n_7 \) fails which fails \( \text{scope}_{2.1} \) by FailR.7, but \( \text{scope}_{2.1} \) succeeds by CompLR.5. \( n_1 \) is vital, but fails and thus \( p_j \) fails by propagation.

In this scenario, \( \text{scope}_{2.1} \) has succeeded and thus it is explored in the following manner: both paths \( p_j \) and \( p_{j'} \) start compensating, \( n_k \) and \( n_j \) are explored because they are the last nodes on the
paths, $n_5$ is compensated (CompR.3) and $n_7$ is skipped. Subsequently $p_1$, $p_2$ and their enclosing scope $scope_{2.1}$ are all marked COMPENSATED by compensation completion events and policies.

**Comprehensive Compensation**

We will only briefly touch upon comprehensive compensation, highlighting the main ideas. The overall management in COMPMOD is similar to that presented for failure handling and partial compensation: that is a number of rules and dependencies are defined and enacted by the reactive controller.

Comprehensive compensation is engaged when a global failure of the long running transactions is recognized – that is there is no possibility to recover and complete the LRT in some alternative way. These failures are triggered by a failure of a vital node that is preceded by a hierarchy of vital ancestor components towards the top of the hierarchy ($p_0$), such that the failure propagates up the hierarchy structure and reaches the main execution path and consequently the transaction fails globally.

From the business point of view, a failed transaction means that it has failed to achieve its expected outcome. Both from a business perspective and also from a consistency of transaction’s point of view, any task that has succeeded must be compensated. The question raised is how to apply compensations and in which order. The most common way is to apply compensations to tasks in reverse order of their completions which is commonly referred to as rolling back or backward compensation. In backward compensations, rolling back is enforced by the management model of the transactions and results in a long running compensating transaction.

With Web Services, tasks in a transaction can mean anything from a database update operation to sending email to a client. Thus, rolling back a transaction must depend on what the transaction was about. In real B2B applications, it is the case that business process logic requires that compensation logic diverges from the standard backward compensation order by freely incorporating compensation logic into business logic. The restricted backward recovery mechanism makes implementing an arbitrary order for compensations not a straight forward process and might force the business process designer to change the business logic of the transaction to comply with compensation logic requirements (or build very complicated compensation schemes into the original workflow making the workflow more complicated and distracting from the actual process).

COMPMOD model supports a customized compensation method that provides transaction designers with the flexibility of expressing their business process logic without putting compensation in mind. Compensation logic can then be
Maintaining Transactional Integrity in Long Running Workflow Services

mapped onto the business process in a very flexible way to meet business needs. The designer is allowed to specify compensation patterns on a subset or subsets of atomic nodes (component services) of an LRT. In Figure 10, such dependencies are indicated by dashed arrows. A compensation pattern decides the order by which the specified services are compensated, and will only be defined for parts of the LRT where the designer cares about the compensation order. Any other services that are not involved in any compensation pattern are compensated concurrently. This will increase the performance of the system in terms of time spent on the compensation process. Assignment of compensation patterns is restricted by validity rules to avoid deadlocks and violation of logic integrity. The general mechanism of comprehensive compensations guarantees the following:

- Each atomic node in the LRT is traversed.
- Each succeeding atomic node in the LRT is compensated.
- If there are customized compensation patterns, then the order of each pattern is enforced.
- Achieving (1-3) guarantees an explicit compensation completion state of the transaction.

We will not discuss the technical details of the implementation here, but mention that nodes are sorted into groups depending on whether:

- They can readily be compensated (i.e. they do not form part of a defined pattern or are the source node of a pattern (that is no user defined dependency points to it).
- They are a target node in a user defined dependency.

In the latter case, which includes nodes that might be both target and source, the node will have to wait until the nodes on the user defined path leading to them have been compensated and

Figure 10. A sample LRT with customized compensation dependencies
then they will move to the group of nodes that can readily be compensated. Once all the atomic nodes in the LRT are visited, a customized compensation completion event is fired for the LRT and it is marked as COMPENSATED.

**e-Booking Example**

We demonstrate our management and failure handling mechanism on an e-booking example depicted in Figure 11 to illustrate how an LRT can succeed in case of non-vital node failures. In this scenario, it is required to book a flight, a hotel room, and a car for a specific period as received by the BookingOrder activity. It is necessary to find a flight booking and a hotel room for requested dates and thus the nodes Flight and Hotel are assigned as vital nodes. It is desirable for MakeBookings scope that a car rental is booked for the same dates, but this booking is not essential. In other words, if a car rental was not available, the MakeBookings is still successful from the business point of view – this is reflected in the non-vital nature of the node. All other nodes are vital. So, by evaluation \( p_1 \) and \( p_2 \) in MakeBookings scope are vital, while \( p_a \) is non-vital.

Activation of the LRT (ActR.1) triggers an activation event for \( p_o \) (ActD.1). Activation of a path triggers the activation of the first node BookingOrder (ActD.2). The system waits for the BookingOrder to finish its execution. We assume that a completion event has been fired for the node and the BookingOrder is marked SUCCEEDED (CompLR.1). Successful completion of BookingOrder activates MakeBookings scope (ActD.3) since BookingOrder is not the last node on \( p_o \). Activation of MakeBookings, fires activation events (ActD.4) for \( p_1, p_2, \) and \( p_a \) encapsulated by MakeBookings and they are all activated by (ActR.3). Subsequently, and in the same manner as illustrated above, the first nodes on the concurrent paths are activated; Flight, Hotel, and Car and are executed concurrently. Assume that Flight succeeded and Hotel succeeded and the system is waiting for the Car node to finish its execution. Note \( p_1 \) and \( p_2 \) have succeeded by (CompLR.2).

To demonstrate how the completion and successful completion of concurrent scopes are dealt with in case of non-vital failures, we assume that the Car node fails to complete. Failure of the non-vital Car node fires a failure event for \( p_a \) (FailD.1) and thus \( p_a \) fails (FailR.7). Failure of \( p_a \) fires a completion event for MakeBookings since it is the last path to complete and hence \( \text{CompLDep(MakeBookings)=True} \). MakeBookings has not failed since all its vital components succeeded and there is no failure event fired for the path since \( p_1 \) and \( p_2 \) have succeeded, hence MakeBookings succeeds by the completion policy (CompLR.5). Successful completion of MakeBookings activates Payment. If we assume

![Figure 11. e-Booking example](image-url)
that Payment succeeds, then a completion event is fired for \( p_o \) (CompLD.1) and policy (CompLR.2) succeeds \( p_o \). The successful completion of the main execution path leads to the success of the LRT by (CompLR.3).

**CONCLUSION AND FUTURE RESEARCH DIRECTIONS**

We presented an approach for modeling and enacting failure recovery and compensation 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 where failure does not matter. We also introduced the idea of custom defined compensation dependencies in the context of final failure of an LRT. The designed propagation rules are enforced through a novel rule based management system, allowing for monitoring and controlling LRTs. Nested workflows are used as examples throughout.

One of the motivations for this work was the perceived lack of high level approaches to compensation handling: compensations are part of the business process and are best understood at the design level. Existing support in some BPM tools (e.g. TIBCO BW or IBM Process Server) and also existing work in exception handling for processes (Russell, van der Aalst, & ter Hofstede, 2006). Lerner, Christov, and Osterweil (2010) address the issue of “things going wrong” in a way that is akin to programming level solutions. They require detailed consideration of each individual case of possible failure and then a deliberate exploration of how to handle this. The presented work lays a foundation for abstracting away from specific errors and considering how failure and compensation should be handled in the situations that are meaningful to address for the business analyst while dealing with all other cases automatically in standard ways defined through the framework and its policies. Programmatically this might mean that the tools implement the details of the framework through an exception handling mechanism, but this would be transparent to the user.

Direct future work includes implementation of an operational system reflecting this approach and its use in some larger case studies. There is also a growing interest in risk-aware business processes and our notion of vitality (combined with the proposed framework) could be one way of addressing this. However, this requires further study.

More generally, there are two areas of work that are required to better support transactions: workflow or business process design standards and workflow execution environments. For the former, much work has been done over the last few years with the introduction of BPEL (more as an implementation oriented mechanism) and BPMN (more targeted as a business requirements capture mechanism) in formulating and designing workflows. These efforts consider ideas of compensation and alternatives that can be engaged when repair is needed due to partial failure, but they are somewhat cumbersome to describe. In our work, we provide a good solution in terms of dependencies that automatically takes care of many of the issues that arise, letting the business analyst focus on the parts of the process where more customized dependencies are needed. Also, none of the mechanisms support the distinction of vital and non-vital parts of the process (with the only option being an alternative scope to capture non-vital aspects, making the flow less intuitive).

Regarding the execution environments, these are currently more of interpreters for workflows that largely leave transaction handling aside at the high level and assume that transactions are managed at lower levels in the execution environment, and possibly through the aforementioned repair
routes. It would be desirable to include transaction management as a more native part of the workflow engines – and again as much of these work in an event based fashion, our approach should be able to provide a solution for ready implementation.

REFERENCES


Maintaining Transactional Integrity in Long Running Workflow Services


### KEY TERMS AND DEFINITIONS

**Compensation:** The undoing of the primary effects of having completed an activity.

**Failure:** The incapability of successfully completing an activity or workflow – failure can be partial with the possibility to recover or total with the outcome that the process overall fails.

**Long Running Transaction:** A transaction that spans a long period of time that can run into days, weeks, months, or years.

**Management Rules:** Rules describing the actions to be undertaken to manage failure and compensation under certain occurring events and conditions.

**Task:** An activity to be undertaken to achieve a business goal.

**Vitality of Tasks:** The contribution of the success of a task to the overall outcome of the business process.

**Workflow:** An artifact describing a process in terms of the activities to be undertaken and their relationships.

**Workflow Pattern:** A structure of activities and the flow between them that represents a specific business need.

### ENDNOTES

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