MAINTAINING TRANSACTIONAL INTEGRITY IN LONG RUNNING WORKFLOW SERVICES: A POLICY-DRIVEN FRAMEWORK

Thesis submitted for the degree of

Doctor of Philosophy

At the University of Leicester

By

Manar Salamah Ali

Department of Computer Science

University of Leicester

2012

ABSTRACT

Business to Business integration is enhanced by Workflow structures, which allow for aggregating web services as interconnected business tasks to achieve a business outcome. Business processes naturally involve long running activities, and require transactional behavior across them addressed through general management, failure handling and compensation mechanisms. Loose coupling and the asynchronous nature of Web Services make an LRT subject to a wider range of communication failures. Two basic requirements of transaction management models are reliability and consistency despite failures. This research presents a framework to provide autonomous handling of long running transactions, based on dependencies which are derived from the workflow. The framework presents a solution for forward recovery from errors and compensations automatically applied to executing instances of workflows. The failure handling mechanism is based on the 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 execution events. One practical 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. A novel feature of this research is that the approach permits the workflow designer to specify additional compensation dependencies which will be enforced. A notable feature is the extensibility of the model that is eased by the simple and declarative based formalism. In our approach, the main concern is the provision of flexible and reliable underlying control flow mechanisms supported by management policies. The main idea for incorporating policies is to manage the static structure of the workflow, as well as handling arbitrary failure and compensation events. Thus, we introduce new techniques and architectures to support enterprise integration solutions that support the dynamics of business needs.

DECLARATION

This thesis reports on work undertaken in the Department of Computer Science, University of Leicester, supervised by Dr. Stephan Reiff-Marganiec. I hereby declare that the contents of this submission have not previously been published for a degree or diploma at any other university or institute.

All the material submitted is the result of my own research, except where otherwise indicated.

The research work presented in some sections has been previously published: in particular:

- A brief description of COMPMOD architecture that is presented in Chapter 4 and the failure handling mechanism presented in (Chapter 5, section 5.4) has been published in (Ali and Reiff-Marganiec, 2012)
- The compensation mechanism presented in Chapter 6 has been published as a Book Chapter in "Service-driven Approaches to Architecture and Enterprise Integration", IGI Global, 2013, ISBN 978-1-4666-4193-8. The title of the book chapter is: “Maintaining Transactional Integrity in Long Running Workflow Services: A Policy-driven Framework" (Authors: Reiff-Marganiec and Ali.)

Manar Salamah Ali
Leicester, October 2012

ACKNOWLEDGMENTS

I would like to convey my sincere gratitude to my supervisor Dr. Stephan Reiff-Marganiec for his helpful advices, continuous support and valuable feedback on the context of my research and thesis, and his constant encouragement during the course of my PhD study. For his kind supervision and guidance, I'm sincerely grateful.

My exceptional obligations are to my dearest parents, Mr. Salamah and Mrs. Muazzez, for all their love, support and faith in me, for being my inspiration, and for making me everything I am. For all this, I will be forever thankful.

I would like to thank my loving family; my sisters Dr.Inas and Mrs.Nibras , my brother Mr. Abdullah, and my dear friends Mrs. Dina Mulla and Mr. Mustafa Jabbar for their encouragement, mostly for being there for me as well as for their patient listening to all my worries, over and over again without complaining. For all this, I owe them much.

I would like to acknowledge Dr. Ammar Amin, the Supervisor General of the Joint Supervision Program, King Abdul Aziz University. For his great support, I am truly thankful.

I would like to express my heartfelt appreciation and gratitude to Mr. Massimiliano Canali for his valuable support and encouragement during the final year of my PhD work. For his devotedness, I am genuinely indebted.

My love and admiration for my three kids, Layan, Lara, and Ammar. Without their patience, relentless praise and their shouldering of great responsibilities during my oft recurring absence from home, I would never have been able to complete this work. For their courage, I am full of pride and humble gratitude.

Thank you!

Manar

Contents

LIST OF FIGURES

LIST OF TABLES

LIST OF CHARTS

Chapter 1

Introduction

1.1 Motivation

Two widely demanding trends, both in web technologies and in the business world, drive and motivate the research in this thesis. In the business world, the trend is towards the collaboration of companies and enterprises as networked organizations. This is accomplished by adopting collaborative mechanisms of business processes integration within a large community of business partners. On the other hand, web technologies are transforming the web from an infrastructure for sharing information to an infrastructure where networked organizations can collaborate and integrate their business interests.

Essentially, Service Oriented Computing (SOC) has had a significant impact as the computing paradigm to support collaborative Business to Business (B2B) integration

over the internet (Papazoglou and Georgakopoulos, 2003). Service Oriented Architectures (SOA) forms a foundation for rapid application integration and automated business processes, ideally through web service implementations (Newcomer and Lomow, 2004).

In this chapter, we highlight the main challenges associated with the web service based business process modeling and management. These challenges enact modeling requirements that must be satisfied in order to achieve one general common objective: correctness and reliability of the management model. We discuss these requirements and state our research questions, research statement, model overview and assumptions.

1.2 Research Challenges

Web services are "*self-describing, open components that support rapid, low-cost composition of applications"* (Papazoglou and Georgakopoulos, 2003). Web services are offered by service providers (business organizations) by implementing services, together with their description and associated technical and business support. A B2B process can then be composed by aggregating web services to form a composite service, in order to achieve a required business outcome. Autonomy, loose coupling, the heterogeneous nature of web services and human interaction for some tasks makes a business process into a long running one.

A composite service would typically entail a complex structure of interrelated activities that would exhibit a high degree of concurrency and interrelationship. Therefore its composition requires flexibility in terms of the construction of the overall business process. **Workflow** systems integrate, automate and manage B2Bs and enable business processes to fulfill their business goals through flexible representations of the control flow of their tasks. A service-based workflow process is a long running workflow, composed of web services that relate to each other through workflow constructs such as 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).

One of the important aspects of management of B2B long running processes is to ensure their reliability, consistent outcomes and the correct execution of the composite services. In particular, in case of failure of some of the component services, it is required that the business task remains "stable". Autonomy and loose coupling of web services makes a composite service more prone to failure than other business processing environments, in that the failure of services can happen at any time, with a higher probability, and therefore an efficient failure handling mechanism is required. In addition, a collaborative B2B process usually involves different parties, and spans different organizations; thus, correct and reliable execution is an important aspect of business integration which

guarantees that all parties involved in the business process always maintain their systems consistently, especially in the case of failure occurrences.

Reliability, failure handling and correct execution behavior constitute the main properties on which **transaction management** models are based and where these properties are typically inherent within their execution semantics. Transaction management has been widely exploited in the literature as a mean of correct execution of database processes and has resulted in a plethora of proposed transaction models. The **ACID** correctness properties (Gray, 1981)(Haerder and Reuter, 1983, Özsu and Valduriez, 1991) establish the main properties on which other database transaction models have built their correctness.

In essence, an ACID transaction is (a) Atomic (all-or-nothing), by which all operations of a transaction are expected to either successfully commit or if the transaction fails (aborts), then all its effects are undone (rolled back), (b) Consistent: the transaction moves the state of the system from one consistent state to another consistent state, i.e. requires the transaction to be correct, (c) Isolated: this requires that correct concurrent transactions execute as if they are sequenced, and (d) Durable: this requires that once a transaction is committed then its outcome is made permanent in spite of future failures. To achieve overall correctness of transactions, different concurrency and recovery protocols have been proposed to ensure atomicity. These protocols mainly depend on the exclusive locking of shared resources for the duration of the transaction, e.g. the two phase locking protocol in (Moss, 1982).

As a result, in traditional database transactions, it is a requirement for transactions to enforce ACID properties, so as to ensure that only consistent state changes take place in the presence of concurrent access or failures. Even in complex business applications, ACID properties ensured that consistency of state is preserved. It is a very useful fault tolerance technique when multiple or remote resources are shared. The atomicity property ensures a reliable fault handling mechanism, but ACID transactions are regarded as "short-lived" entities, running on tightly coupled systems.

Applying ACID properties in long processing environments will oblige locking resources for long periods of time, which is inappropriate. Atomicity in long running transactions is not a straightforward notion, since it is not always possible to semantically undo the effects of all tasks in the transaction, due to the complexity of the transaction model and the nature of the business tasks – tasks can mean anything from a database update operation to sending email to a client or shipping goods. Instead, ACID properties are relaxed to suit long running transaction requirements where the atomicity requirement is replaced with the concept of Compensation. Compensation in long running transactions defines the behavior of the transactions in the case of occurrence of failures or cancellations. Failures need to be handled correctly, to ensure overall system consistency and data integrity.

Compensation was first introduced in the saga model (Garcia-Molina and Salem, 1987) where a long running transaction consists of a set of ACID transactions, the saga itself is not ACID. Failure atomicity is guaranteed for sub transactions such that when one fails, it

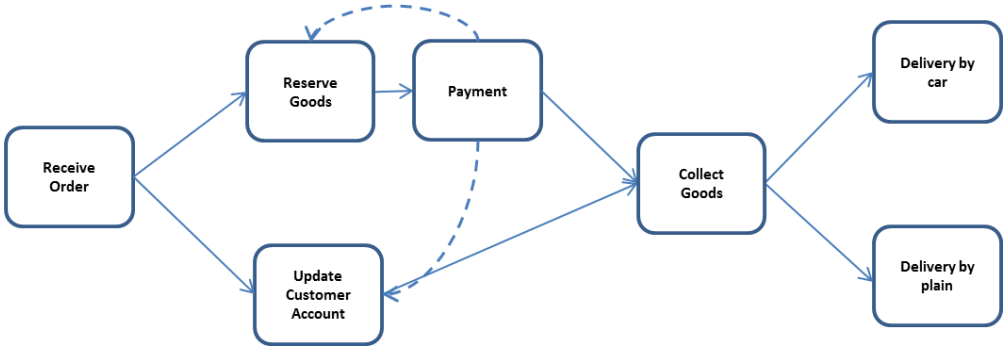
is aborted and retried through forward recovery. If the saga fails, it is aborted, and all committed sub transactions are compensated in backward order by running compensators associated with each sub transaction.

The compensation concept has been adopted in long running transaction models as a mean of recovery and reliability in the case of failure occurrences; primarily to relax atomicity of ACID transactions. Compensation tends to undo effects of previously completed tasks. Therefore, if an LRT failed, all previously completed activities are compensated by running their compensators. Generally, LRT management models apply compensation of activities using two standard methods: (1) Forward order: when a recoverable failure occurs, a subset of LRT activities are compensated in the reverse order of their completion order until a safe point is reached and then the same activities are retried and (2) Backward order: in the case of irrecoverable failures, the LRT fails and all previously completed activities are compensated in reverse order of their completions: that is, reverse order in case of sequenced tasks and parallel or any order in case of concurrent tasks. However, whatever the order by which compensations are executed, this order is always enforced by the structure of the LRT model being applied, and results in a long running compensating transaction.

In real B2B applications, it is the case that business process logic requires that compensation logic diverges from the standard compensation order that is obliged by the LRT structure 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. Furthermore, tasks that are not compensationally independent may execute their compensations in parallel.

A motivating example is the following e-procurement workflow case study 'Place Order' workflow:



The dashed arrows represent compensation logic

Figure 1.1 Place Order Business Process Scenario

The business\compensation logic in this e-procurement case study states as follows: if the transaction is compensated, then it has to be guaranteed that the compensators of *UpdateCustomerAccount* and *ReserveGoods* can be executed only after the *Payment* has completed its compensation. With the default backward compensation mechanisms, if *UpdateCustomerAccount* was completed after *Payment* has been completed, then, *UpdateCustomerAccount* will be compensated before *Payment* has been refunded, or they could both compensate concurrently which contradicts the compensation logic of this specific LRT. Therefore, and in the context of business process logic, we view an LRT as two transactions represented by one schema: the transaction in its normal intended form, and a compensating transaction.

Collaborative B2B business applications normally include tasks running **concurrently** as part of the overall transaction, which requires consistent and correct modeling of their behavior. Two important issues regarding concurrency execution are (a) the reliable modeling of the synchronization of concurrent tasks and (b) the prevention of possible deadlocks. Concurrency modeling is usually influenced by the underlying paradigm for representing transactions and the protocols used for the interactions between transaction tasks. The more flexible the transaction representation paradigm is, the more challenging it becomes to define a correct behavior for concurrency.

The increasing availability of business processes is an important feature to the practicality of a proposed model. This could be accomplished in many ways, but more importantly, by providing compensation techniques to allow for tolerable failures to be recovered without interrupting the normal processing path of the business process. Availability can be further increased by distinguishing between crucial tasks that must complete successfully and those tasks that are less crucial and their failure is tolerable and will not require any further actions.

Externalizing management aspects from actual execution aspects of process tasks increase the practicality of business process modeling. First, operational semantics can be captured at a higher abstract level than the actual executing tasks, allowing for implementing different methods for recovery without being tied to the underlying structure of the process. Second, it is a good way to provide the management model with extensibility of its operational semantics by adding new semantics. Event-Condition-

Action rules are a natural candidate to fulfill management externalization and for implementing this kind of functionality.

1.3 LRT Modeling Requirements

As an essential part of conducting this research, we have defined a set of LRT modelling requirements. These requirements are derived from the literature provided on LRT modelling approaches as well as from analysis of a number of example business processes.

Principally, our Long Running Transaction is:

1- Web service based: tasks in the LRT are web services that are composited to achieve a business outcome.

2- Transactional: the modeling of LRT exhibits transactional semantics.

3- A workflow: the LRT is represented as a workflow schema with arbitrary levels of nested tasks.

4- A Reactive Management Model: it is executed in an environment where changes are detected as events and automatically responded to by applying appropriate execution logic through management policies.

From a **web-based business process** perspective, LRT modeling imposes the following requirements (Aguilar-Saven, 2004; Papazoglou, 2003; Peltz, 2003):

REQ. 1 Transactional support to guarantee consistent outcome for participating parties.

REQ. 2 Flexible representation of complex web service compositions that allows nesting and concurrency which naturally occurs in business processes.

REQ. 3 Recovery protocols to undo completed tasks and to choose another acceptable execution path (Dalal et al., 2003).

REQ. 4 Composition of web services must be supported with failure handling mechanisms that allow some failures to be tolerable and/or compensable while others could fail the business process from successfully completing depending on the crucially of the task to the overall outcome of the business process.

One of the main differences between a traditional transactional model and a loosely coupled LRT is that the former is data-centric while the latter is activity-centric or more generally, process-aware (Reichert & Weber, 2012). Therefore, from the **transactional** perspective, LRT modeling imposes the following requirements (Colombo & Pace, 2012; Dalal et al., 2003; Dayal, Hsu, & Ladin, 1991):

REQ. 5 Transactions support nesting and concurrent execution, but they are not flexible enough to capture the highly collaborative and concurrent nature of real B2B processes and hence more flexibility in representation is required as such to allow for selective and alternative choices.

REQ. 6 Transactions' recovery is based on failure handling mechanisms that are inherent in their semantics which delimits flexibility of expressing compensation logic of the business process. Therefore, the failure handling mechanisms should be separated from execution mechanisms, and both should operate in tandem to achieve correct recovery mechanisms.

From the **workflow** perspective, LRT modeling imposes the following requirements:

REQ. 7 Control flow of workflows should be supported by transactional semantics in order to achieve a reliable control flow (Bhiri et al., 2006a).

REQ. 8 Synchronization of concurrent tasks should be formally defined to resolve potential operational ambiguities and potential deadlock situations (Russell et al., 2006).

REQ. 9 Extensibility of a model's operational semantics is an important requirement of modeling which facilitates additions of new control flow constructs to comply with web service composition requirements and to confirm the practicability of the model.

REQ. 10 Transaction support for workflows requires well-formed infrastructure and well-formed relationships between the correlated tasks and hence transactional workflows require well-formed structure applied to the workflow schemas (Kiepuszewski, Hofstede, & Bussler, 2000).

From the **compensation** perspective, LRT modeling imposes the following requirements (Colombo & Pace, 2012; Greenfield et al., 2003):

REQ. 11 Separation of failure handling and compensation handling semantics.

REQ. 12 A mechanism for applying partial compensations that is integrated with the failure handling semantics as part of the failure recovery process. If failure handling requires compensation applied to completed tasks, this can be done without interrupting the execution of the transaction, i.e. tasks that are not interrupted with failures, will continue their executions.

REQ. 13 Flexibility in incorporating compensation logic into business logic. Compensation semantics should not be enforced only by the structure of the

REQ. 14 business process composition; instead, the LRT designer should be assisted by a correct mechanism for freely expressing the customized compensation relationship between transactions tasks without violating the integrity of the overall process. These should apply in the case of global failure of the LRT.

From the **correctness and reliability** of execution perspective, LRT modeling imposes the following requirements (Chrysanthis & Ramamritham, 1990; Colombo & Pace, 2012):

REQ. 15 Providing the means of validating the correctness of execution semantics.

REQ. 16 The transactional relationships between interrelated tasks are best being formalized in a way to make reasoning about their correctness a straight forward task, i.e. using the same formalism for both, modeling and correctness. This will also increase the extensibility of the model.

From the **reactive management** perspective, LRT modeling imposes the following requirements (Papamarkos et al., 2006; Wieringa, 2003):

REQ. 17 Execution behavior of LRT components need to be observed as events, such that when a component completes, fails or compensates, an event is fired.

REQ. 18 Execution semantics of the LRT need to be implemented as rules (policies) to automatically execute business logic.

1.4 Modeling Objectives

Our modeling objectives are driven by the modeling requirements discussed in the previous section, and motivated towards the following aims:

1- Correct control flow of a long running transaction, both in its normal processing path and its compensation processing path.

2- Flexibility in representation of execution semantics.

3- Flexibility in compensation composition.

4- Reliability of execution by correct handling of failures and compensations.

5- Automation of management mechanism as step towards a self-healing transaction model.

6- Formal modeling of execution behavior that provides the means of reasoning about correctness of the behavior.

1.5 Research Questions and Statement

The modeling requirements and objectives listed in the previous sections raise the following research questions:

Q1- How can the structure of the business process be specified with complex and arbitrary levels of nesting?

Q2- How can the control flow semantics of transactions with complex and nested structure be formally captured? How can the same formalism be used to capture failure and compensation semantics?

Q3- What failure mechanism best reflects the propagation of failures through nested structures?

Q4- What formalism is ideal for the flexible incorporation of compensation logic into business logic?

Q5- What management mechanism would be ideal for automating the control flow process?

Q6- How can the control flow formalism be used for reasoning about correctness of control flow, concurrency, failure handling and compensation semantics?

Thesis Statement

In this research, we focus on flexible control flow of web-based workflow modeling with long running transaction support to deliver reliable execution behavior of business processes. Reliability is guaranteed through flexible and autonomous failure handling and comprehensive compensation handling mechanisms.

1.6 Contribution

Our contribution is a fourfold:

Contribution 1: Fine-grained specification model for arbitrary nested transactions.

We specify our LRT model as a hierarchical tree structure that provides a recursive nature for propagating execution events across and along hierarchy levels. Essential to

this fine-grained structure, we explicitly capture the semantics of execution paths and specify them as autonomous components of the LRT. By providing this, we are able to enrich the operational semantics of concurrency with flexibility and extensibility.

Contribution 2: Autonomous Failure Handling Mechanism

An essential propagation policy states that "Failure of a vital atomic node, fails its superior". Based on this policy, we build an autonomous failure handling mechanism that propagates failures recursively through vital ancestors, if the failure event reaches the root of the hierarchy, the transaction fails. Basic to the failure handling mechanism, a downwards propagation of failures is applied to a failed concurrent scope, in order to cancel all its activated components. The failure handling mechanism is integrated with a partial compensation mechanism to apply partial recovery in the case of tolerable failures.

Contribution 3: Compensation Composition Mechanism

We regard compensation composition as being as important as service composition. Therefore, we provide business process designers with the underlying framework to freely specify the order in which compensation of tasks are required to be executed. This functionality is provided through the specification of compensation patterns that are mapped onto the workflow schema. The designer is allowed to specify *compensation patterns* on subsets of component services of an LRT. A compensation pattern then decides the order by which the specified services are compensated. Any 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. We support reliable compensation compositions by validating such compositions, to avoid consistency violations. This implemented through the comprehensive compensation mechanism of COMPMOD.

Contribution 4: Specification Extensibility

One important feature of COMPMOD is its highly flexible extensibility, in the sense that the underpinning representation structure can be enriched with further concurrency, execution and compensation semantics.

1.7 Model Overview

The work presented proposes a reliable control flow management mechanism for sequencing and concurrency in web-based workflow transactions, such that tolerable failures are handled. A tolerable failure is a failure of a task to complete successfully but the failure is acceptable in the sense that it would not cause an interruption of the LRT's execution nor cause a global failure of the transaction. Handling tolerable failures would typically involve partial compensation activities applied to subsets of tasks, but will not stop the transaction from completing its normal execution. In the case of intolerable failures, and when a consensus is reached about the failure of the LRT, a comprehensive compensation is applied to all previously succeeded tasks. The order of compensations can be customized on a subset or subsets of tasks. Tasks that are not part of a customized order can be compensated concurrently. Customized compensations mainly reflect the business and the compensation logic of the transaction.

Our approach for managing LRTs is based on a reactive system controller in event based architecture. Policies define the rules by which the controller acts. In general, an execution event is raised for a component to signal its readiness to perform an execution (activation or compensation), or to signal that an execution of a component has finished (completion or failure). The raised event is then assessed by management policies to reach a consensus as to the current state of the component and the next state of its correlated components.

An LRT in COMPMOD is represented as an arbitrary nested WF transaction. The WF representation of the model imposes a hierarchical tree structure, where the root of the hierarchy represents the main execution path. The respective levels of the tree represent an alternating levels of nodes and execution paths, such that the superior of a path is its enclosing scope node and the superior of a node is its enclosing execution path. This results in atomic nodes being the leaf nodes of the hierarchy tree. Each component in the hierarchy is directly correlated with its superior, inferior, and siblings in an encapsulated manner, such that a component can be indirectly correlated with another component if their superiors are correlated. As an example, nodes on concurrent paths are correlated, since their superiors are siblings.

The encapsulated behavioral interrelationship between components is modeled by dependencies, and automated by policies. Behavioral dependencies and management policies both reflect the execution semantics of the model and complement each other.

The model allows for a separation between vital and non-vital components where a failure of a vital component has an impact on the cancellation of its correlated components, while failure of non-vital components is tolerated. Cancellations will invoke partial compensations to return to a place where an alternative (if one exists) can be attempted without lasting side effects, and the failure of the LRT will lead to comprehensive compensation being applied to all composited nodes in the transaction.

1.8 Thesis Structure

The thesis is organized as follows:

- Chapter 2: discusses the literature background of the thesis and the related work in the field.
- Chapter 3: discusses the two modeling paradigms that we adopt in our workflow semantics; *workflow patterns* and *transactional patterns* and explains how we extend these models.
- Chapter 4: describes the representation structure of the COMPMOD model, and introduces the concepts of execution events, reactive management and management policies. We will also state our model assumptions in this chapter.
- Chapter 5: describes the execution semantics of the model and its formalism, and shows the management mechanism and the failure handling mechanism.
- Chapter 6: describes the logic and formalism of compensation events and policies, and illustrates the partial and the comprehensive compensation mechanisms of COMPMOD.

- Chapter 7: provides a verification of COMPMOD in terms of the correctness of the proposed model and its extensibility feature.
- Chapter 8: concludes the thesis and provides details of future work.
- Bibliography
- Appendix A and B lists a table for all dependencies and management policies of COMPMOD for easier referencing, through related discussions.
- Appendix C: provides an assessment of the COMPMOD model based on the Workflow Patterns Initiative.

Chapter 2

Background

2.1 Introduction

There is a large body of work in the area of business process modeling: transactions, workflows, and long running transactions. In this chapter, we provide a literature review of some of the well-known modeling approaches and we focus on the parts relevant to the respective compensation mechanism. We provide a critique on the limitations of compensation mechanism in WS-BPEL and *Compensation spheres*. The critique is exemplified by a case study from E-supply chain systems. Finally we show how our COMPMOD model fills the gap in the current compensation mechanism limitations.

2.2 Data-Base Centric Transactions

Database centric transactional models provide a strong theoretical foundation for transactions. Failure recovery and concurrency control are inherent within the models.

The first of these models is the **ACID** flat transaction where a strict notion of "all-or-nothing" is applied. Recovery is mainly based on the roll-back mechanism to restore the state of the system to the state before the failure has happened. The ACID transactional model is very restrictive, and is not appropriate when transactions are long lived and complex and may span multiple local database systems. For this reason, a number of extended and relaxed transactional models have been proposed, which relax some of the ACID requirements.

Advanced transactional models have been proposed to introduce:

1- Multi-leveled and nested transactions such as in **Nested Transactions** (Moss, 1985).

2- The compensation concept in **Saga** transactional model (Garcia-Molina & Salem, 1987).

3- Nesting with compensation mechanism in **Open Nested Transactions** (Weikum & Schek, 1992), **Nested Sagas** (Garcia-Molina et al., 1991), and **Flexible transactions** (Elmagarmid, 1992; Zhang et al., 1994; Mehrotra et al., 1992), and **ConTracts** (Reuter, 1989; Reuter, Schneider & Schwenkreis, 1997).

Nested models allow transactions to be nested within transactions to form a tree transaction. The nesting structure is reflected on the commitment, abort, and compensation of its constituent sub-transactions where different models provide different protocols with varying flexibilities.

However, transactional models have the following limitations in business process modeling:

- They are developed from the point of view of database management systems and thus business related semantics such as activity automation are ignored.
- Coordination support for multi-tasking and collaborative activities across organizations is limited, and thus they are not applicable to heterogeneous and loosely coupled systems.
- Compensation mechanisms are strictly in reverse order of the sub-transactions' commitment order and are hidden from transaction designers.

2.3 Transactional Workflows

Business Processes are usually defined by business analysts to capture the activities and their respective orders to achieve some larger business goal. Workflows 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).

Workflows provide a key functionality in integrating heterogeneous and distributed applications into a coherent business process and provide process automation.

Modeling of such workflows is usually conducted in some graphical notation such as BPMN (White, 2004), UML activity diagrams, or YAWL (Van Der Aalst & Hofstede, 2005) which are graphical and textual and have formally defined semantics.

A *structured* work flow 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*". The work in (Kiepuszewski, Hofstede & Bussler, 2000) shows that every structured workflow is well behaved.

Workflow patterns in (Van Der Aalst at al., 2000; Van Der Aalst at al., 2003; Russell, Hofstede, & Mulyar, 2006) present standard defintions of workflow patterns found in practical workflow strucures. This is a good standard for workflow developers, and we provide a detailed description of the approach in Chapter 3.

Workflows lack a clear theoretical basis for correctness criteria and support for reliability in presence of failures. Hence, transactional workflow is supported with transactional semantics such as failure recovery mechanisms and reliable executions.

Failure recovery in transactional workflows can be supported in many ways:

1- Direct compensation semantics such as *compensation spheres* as discussed in section 2.5.3.
2- Indirect compensation support such as YAWL where it is possible to model compensation behavior by using YAWL constructs (Brogi & Popescu, 2006).
3- Dynamic and ad-hoc workflow adaptations in case of failure events such as $ADEPT_{flex}$ in (Reichert & Dadam, 1997) (Reichert & Dadam, 1998) and (Müller, Greiner, & Rahm, 2004).

Transactional patterns have been introduced first in (Bhiri, Perrin, & Godart, 2005) to propose a transactional approach to ensure the failure atomicity of composite web service workflows. Further work in (Bhiri, Godart, & Perrin, 2006) and (Bhiri, Perrin, & Godart, 2006) used the concept of trasnactional patterns to ensure reliable composite services accrording to designers' specific needs. 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 the transactional behaviour of web service compositions. We have drawn inspiration from this work, and we provide a detailed descripiton of the approach in Chapter 3.

2.4 LRT Modeling Approaches in Web Service Settings

Web services are coordinated through coordination protocols, and orchestrated through orchestration languages at a high level of abstraction and where failures are dealt with as exceptions. Coordination protocols describe coordination through transaction messages.

Such as: Tentative Hold Protocol (Roberts & Srinivasan, 2001), Business Transaction Protocol (Ceponkus et al., 2002), and WS-Transaction (Cabrera et al., 2002).

Web services are composited through *orchestration* and flow composition languages. The body of work in this area has been focused in two directions: Formal modeling and orchestration languages.

2.4.1 Formal Modeling

The semantics of flow or interaction based compositions of web services are achieved through proposing extensions of well-known calculi or process algebra. In brief, control flow of compensations is achieved through primitives to *install* and *activate* required compensation activities within compensable processes (processes that are paired with compensation activities). The mechanism for installing and activating compensations is similar to exception handling primitives (*throw* and *try-catch)* of high level languages such as C++ or Java. Common to all models, compensation handlers are called from fault handlers. What differentiates these models is the way compensations are composed and executed. In (Bruni et al., 2005) , these were classified as:

(1) Compensable flow composition where the way compositions are orchestrated is similar to WS-BPEL and where process algebras are designed from scratch to describe the flow of control among services, such as (Bruni, Melgratti, & Montanari, 2005; Butler & Ferreira, 2004; Butler, Hoare, & Ferreira, 2005).

(2) Interaction based compensations as extensions of well-known calculi where modeling dynamic compensations is addressed, such as π-calculus (Bocchi, 2004)

(3) based on BTP, webπ (Laneve & Zavattaro, 2005), and webπ infinity (Mazzara & Lanese, 2006).

In these models, semantic definitions are somewhat complicated. Hence, they are not practical to use to model real time business scenarios.

2.4.2 Orchestration Languages

Orchestration languages build business workflows by developing graphical or XML-based languages such as XLANG (Thatte, 2001) and WS-BPEL (Andrews et al., 2003) (OASIS, 2007). In this section we discuss the general structure and mechanism in WS-BPEL and in section 2.5.2 we discuss by example some limitations of its compensation mechanism.

WS-BPEL is an industrial standard and language for process modeling based on XML and for connecting process activities with web services. WS-BPEL has rich functionality and provides fault and compensation handling capabilities for business process designers.

Scopes in WS-BPEL are used to group activities in the business process based on functionality or shared variables and events. Scopes can be nested, that is scopes can be defined within scopes. Fault, compensation, and termination handlers are process fragments that run if a fault is raised or in case of compensation, to reverse the effect of a set of successfully completed activities. Each scope is attached with its own fault and compensation handlers as well as a termination handler (to terminate the processing of the scope if its parent scope is terminating or exiting).

These handlers can either be specified explicitly or can follow a default specification as provided by WS-BPEL standard. The control flow of activities is defined by two schemes: (1) structured activities controlled by "sequence" or "flow" to impose control logic on activities nested within them, and (2) explicit control links between source and target activities such that a target activity can only start executing after a source activity has completed. A compensation handler can only be invoked by a fault handler which is triggered by a fault in the executing process. Furthermore, compensation handlers can only be attached to scopes and not to activities

One major drawback of orchestration languages is that they do not support formal definitions for their operational semantics. As a consequence, there has been research directed towards formalizing their operational semantics such as BPEL (Qiu et al., 2005) based on WS-BPEL and c-join (based on XLANG) (Bruni, Melgratti, & Montanari, 2004).

2.5 Limitations of Selected Approaches

After having discussed different modeling approaches of business processes in the previous sections, we dedicate this section to highlight these limitations by examining a running example. We choose an example from an e-supply chain management system and we focus on the compensation mechanism of two widely used modeling approaches: (1) the modeling language WS-BPEL, and (2) the conceptual modeling approach of *compensation spheres*.

2.5.1 E-supply chain case study

Internet based supply-chain systems are achieved through integration of information systems of all supply chain partners (customers, suppliers, and manufacturers). E-Supply Chain may be sourced from several countries, assembled in other countries, and delivered to customers all around the world. In service oriented environment, the integration between business parties is represented by business process activities (e.g. a workflow) which are achieved through web services. A typical customer order represented by a long running business transaction, triggers several B2B web services provided by a network of independent companies to provide a streamlined material flow between all partners.

In this thesis, we use examples from E-Supply Chain to illustrate and justify our proposed model.

The example in (figure 2.1) illustrates an inter-enterprise business process occurring in the supply chain: how the supplier does business with one of its trusted manufacturing partners.

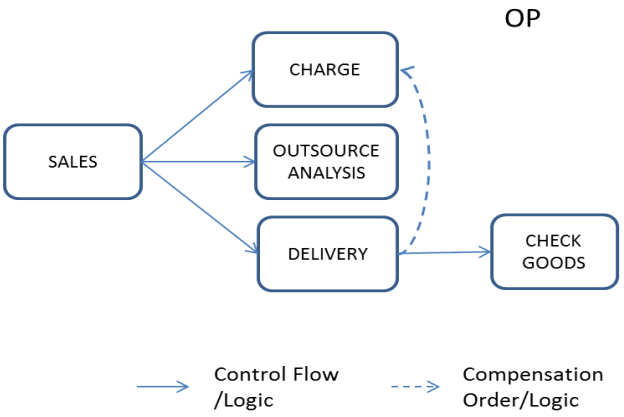


Figure 2.1 Supplier-Manufacturer outsourcing business process *OP*

The sub processes in OP has the following functionalities:

- SALES: performs activities such as receive order from manufacturer, Audit order, and send order acceptance to manufacturer.
- CHARGE: performs payment activities for the outsourced goods.
- OUTSOURCE_ANALYSIS: is a routine activity that is performed with each order transaction to conduct metrics that are used later in determining company's strategies, marketing goals...etc.
- DELIVERY: delivers the goods to the manufacturer.
- CHECK_GOODS: the manufacturer checks the goods. If goods are acceptable then the outsourcing process is completed, otherwise an exception will occur.

The OP process has the following logic:

Once the sales activity is completed, three activities are run in parallel; CHARGE, OUTSOURCE_ANALYSIS, AND DELIVERY. This special outsourcing scenario is conducted with trusted manufacturers. That is why the delivery of goods is performed in parallel with payment. However, the process logic requests that if the goods were to be returned for any reason such as "not meeting the required specifications", the goods must be returned to the supplier warehouses before the payment is refunded to the manufacturer.

2.5.2 Compensation Mechanism in WS-BPEL

Due to lack of compensation semantic formalism, the compensation mechanism in WS-BPEL may show anomalies in certain execution scenarios such as neglecting compensation control links that cross scope boundaries as discussed in (Khalaf, Roller, & Leymann, 2009). In other words, WS-BPEL does not provide guarantee on compensation order.

In WS-BPEL, the compensation order of activities within scopes is strictly in reverse order of their completion and this order is carried out by default compensation handlers. Although explicit control links are allowed between activities/scopes and they are obliged during the normal execution flow, the reverse order of control links during default compensation processing is not straightforward and hence could be violated (König, 2006) and (Thatte & Roller, 2003). In addition, modeling compensation logic in WS-BPEL exhibits high complexity behavior in the presence of scope nesting together with control links that cross scope boundaries.

We will show next in a step by step fashion the compensation mechanism in WS-BPEL by discussing a running example and we show how inconsistencies could occur in the compensation behavior. In (figure 2.2), we show a high-level graphical illustration for the business process of (figure 2.1). The visual cues in (figure 2.2) are borrowed from (Khalaf et al., 2009).

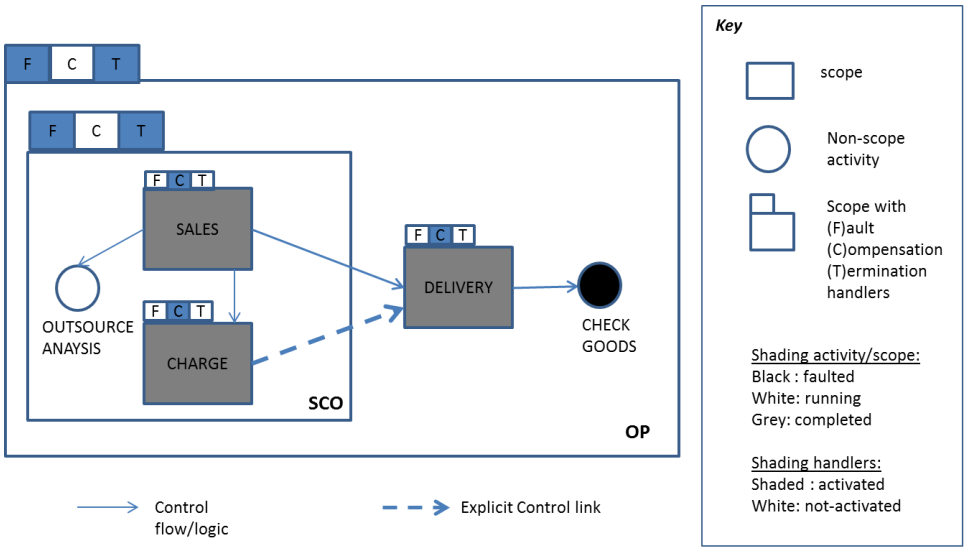


Figure 2.2 WS-BPEL process for supplier-manufacturer outsourcing example

Scope OP represents the outer most scope that groups scope DELIVERY, SCO, and activity CHECK-GOODS. We assume that scope SCO groups SALES, CHARGE, and OUTSOURCE-ANALYSIS activities based on some shared order, customer, and payment variables. The solid arrows represent the control logic of business process activities and the dashed bold arrow represents an explicit control link to represent the compensation logic of the process as explained in section 2.5.1. Hence stating that if the scope OP is compensated, then the goods must be returned first (compensator of DELIVERY) before payment is refunded to the manufacturer (compensator of CHARGE).

In WS-BPEL, when a scope is activated and running then its fault and termination handlers are installed and its compensation handler is not installed. When a scope is completed successfully then its fault and termination handlers are de-installed and its compensation handler is installed.

The illustration in (figure 2.2) assumes an execution instance of OP and hence the execution states of OP components are as follows: OP, SCO and OUTSOURCE_ANALYSIS are activated where SALES, CHARGE, and DELIVERY have been completed. If we assume that CHECK_GOODS has failed, the compensation mechanism of WS-BPEL will perform the following:

1- The failure of CHECK_GOODS will raise a fault exception to fault handler of OP and the termination handler of OP will initiate the termination of immediately nested activated components starting with non-scope components then scope components. In this scenario there are no activated non-scope components and only SCO is activated.

2- The fault handler of SCO is deactivated and the termination handler of SCO terminates the activated OUTSOURCE-ANALYSIS.

3- The termination handler of SCO then invokes the compensation handler of SCO in default compensation order. Since SALES and CHARGE are not linked through explicit control dependency therefore their compensation is performed in any order.

4- When the compensation handlers of both SALES and CHARGE have finished, the control goes to the default fault handler of OP.

5- The fault handler of OP then invokes the compensation handler of OP which invokes the compensation handler of DELIVERY and the scope is compensated.

The compensation mechanism of WS-BPEL on this specific scenario exhibits violation of the explicit control link between CHARGE and DELIVERY and that the payment has been refunded to the manufacturer before the actual goods have been returned to supplier. Hence, the default handlers in WS-BPEL in some execution settings may over rule explicit control links.

There has been research directed to overcome such non-deterministic compensation behavior in WS-BPEL. For example, in (Khalaf et al., 2009) the authors proposed a deterministic model for handling compensations by altering the behavior of handlers and relaxing restrictions on control links. In (Coleman, 2005), the authors request a richer capability of compensation handlers. However, the default compensation of activities within scopes remains the same: reverse order of their completion.

One could argue that the business process could be modeled in a different way but this would necessitate that the business designer should comprehend all possible execution states of the process which is not a feasible solution. Furthermore, as the complexity of the business process increases, modeling compensation behavior becomes cumbersome.

In COMPMOD, the compensation behavior is clearly determined at design time and during compensation mode, the explicit compensation links over rule any other control dependencies. In table in 2.1, we summarize some of the differences between the COMPMOD and WS-BPEL.

	WS-BPEL 2.0	COMPMOD
Model	Executable modeling language	Conceptual model
Control flow	Structured nested activities + explicit control links	Control dependencies derived from workflow structure + explicit compensation dependencies
Scopes	Explicitly assigned to group activities based on shared variables or functionality.	Implicitly formed by the model to group activities nested within workflow structures.
Compensation order	Determined and calculated during runtime depending on execution state of scopes	Determined and calculated at design time
	Compensation links could be over ruled by default handlers behavior	Compensation dependencies have priority over control flow dependencies
	Reverse order	Based on designer tailored compensation dependencies
Compensation design flexibility	*"Default handler behavior causes high complexity in the default compensation order making it difficult for a designer to anticipate the resulting behaviors when making process design decisions" (Khalaf et al., 2009)*	Compensation dependencies can be assigned in any order independent of control flow of activities
Compensation behavior	Possible un-deterministic behaviors	Deterministic

Table 2.1 Compensation behavior WS-BPEL vs. COMPMOD

2.5.3 Compensation Mechanism in Compensation Spheres

Atomic and compensation spheres in (Leymann, 1995) and (Leymann & Roller, 2000) propose a conceptual model for workflow management systems to allow for transactional

properties such as "all-or-nothing" and compensation mechanism to be applied to workflow business processes. We discuss in this section the compensation spheres. A compensation sphere is an arbitrary collection of activities that are tightly related and share a common fate. Each activity in the compensation sphere is coupled with a compensating activity. If an activity in the compensation sphere has failed and aborted, then all completed activities within the sphere are compensated in reverse order. We discuss by example (figure 2.3) the compensation sphere mechanism. The workflow of a business process P is detected as a directed graph (figure 2.3 (a)) where a designer can arbitrarily select a compensation sphere S. Based on this selection, the mechanism induces a compensating graph or map S* (figure 2.3 (c)) by deriving P^{-1} from P where P^{-1} represents the reversed edges of P. When a compensation sphere commences its compensation, the execution starts by compensating activity L and cascades compensation of activities following the control edges in S*.

One advantage of this approach is offering flexibility by involving some degree of arbitrary assignments of compensation orders within a sphere- as opposed to strictly reverse order. For example, indirectly connected activities in P such as B and I but where I is reachable from B in P can be grouped in S. Furthermore, non-connected activities such as B and G in P but where B is reachable from G in P^{-1} can also be grouped in S.

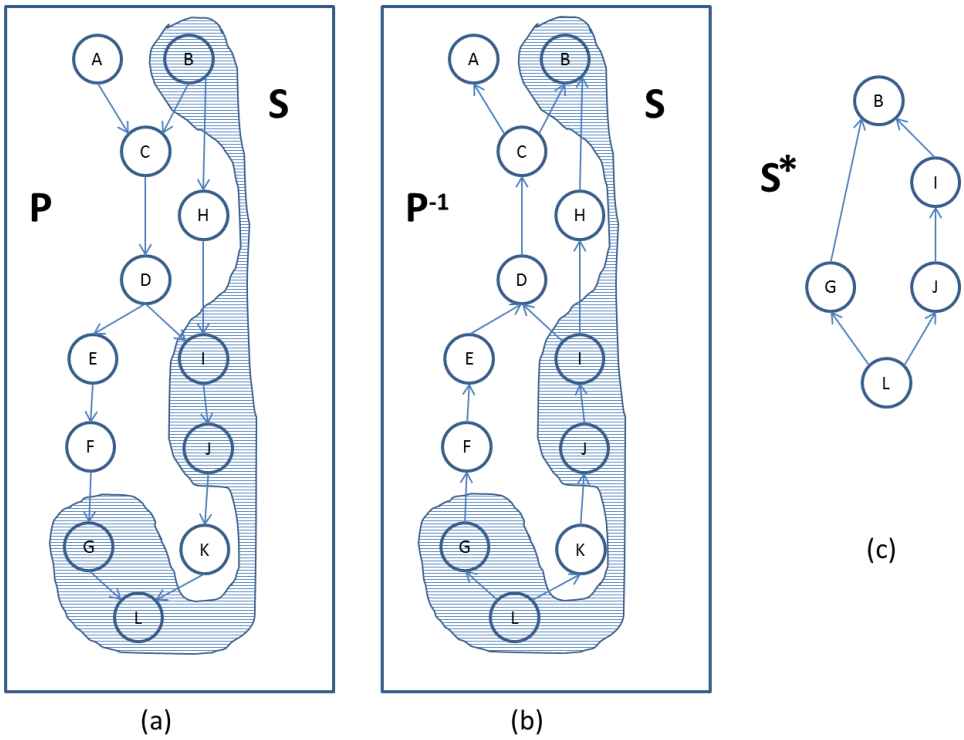


Figure 2.3 Compensation Spheres borrowed from (Leymann & Roller, 2000) p. 271

However, *compensation spheres* have two restrictions:

R1. Any two activities that are non-connected in both P and P^{-1} cannot possibly be grouped alone in a single compensation sphere such as (A and B) or (E and I).

R2. Compensation spheres approach does not provide the process designer the ability to assign extra compensation control flow edges such as to explicitly connect the non-connected activities in the process graph.

We apply the compensating graph algorithm in (figure 2.3) on our outsourcing example as depicted in (figure 2.4). Note that CHARGE and DELIVERY are not connected in both P and P^{-1} (restriction R1) and hence grouping them in a sphere leads to un-connected

graph S* (figure 2.4 (c)). And because of restriction R2, it is not possible to apply the required compensation dependency between CHARGE and DELIVERY.

One could argue that the designer can change the design of process such as to be able to force the required compensation orders if they cannot be systematically applied. However, in COMPMOD model we strongly avoid restricting the making of the design decisions of the business process because of compensation mechanism limitations.

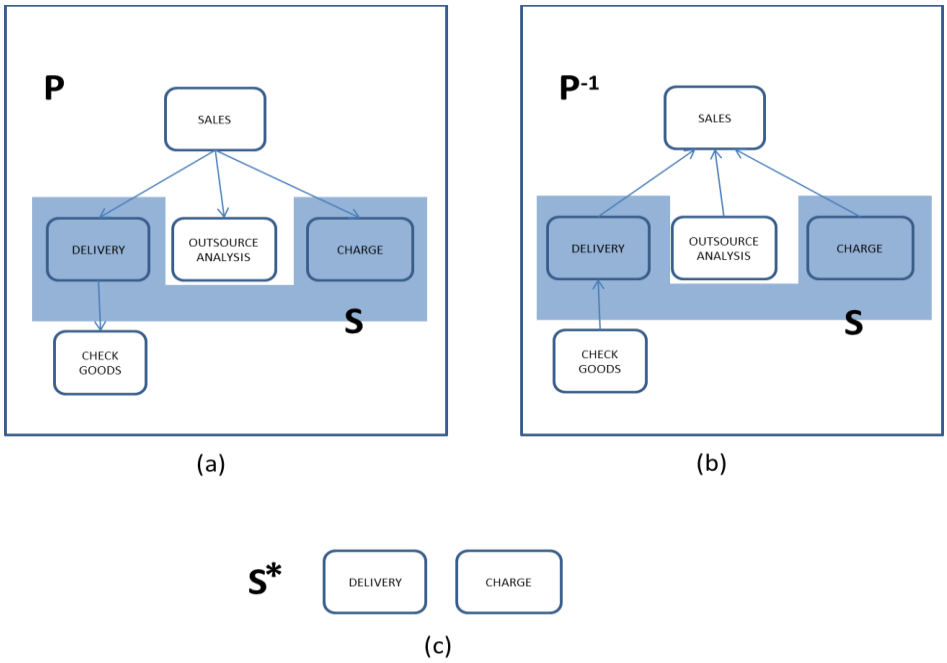


Figure 2.4 applying compensation spheres on outsourcing busing process

2.6 Conclusion

One of the aims of our COMPMOD approach is to simplify the design of business processes. We do so by performing compensations when explicitly requested by the designer and in the order required by the business process logic regardless of how the LRT is structured or how activities are scoped. Hence, designers can easily view and

reason about the customized compensation order to decide how best to design their processes. Scopes in COMPMOD are implicitly defined over nested structures. This structure is totally ignored during compensation and the priority is given to the explicitly defined dependencies

We have shown that purely transactional models force a strict compensation mechanism and that the business designers are not provided with the capability to alter compensation orders and that reverse compensation order is automatically executed.

While workflow models show a high degree of process automation, they fall short in showing transactional properties including compensations.

The complexity of compensation in WS-BPEL is a problem. It is hard for process designers to comprehend all possible behaviors a process will have, due to compensations, as they always must keep in mind all current states in all different scopes and their control link dependencies.

We have also shown by example how there are cases in compensation spheres where un-connected activities in the process cannot be grouped in a sphere which imposes restrictions on process designers.

We strongly claim the importance of freely assigning compensation patterns over process activities without putting into consideration the restriction imposed by the process structure.

Chapter 3

Fundamentals

3.1 Introduction

Our modeling approach adopts and extends two main approaches: *Workflow Patterns* and *Transactional Patterns*. In this chapter, we discuss each of the adopted approaches, and provide an informal description of the operational semantics of our extensions. Definitions are illustrated with examples from the E-supply Chain management systems. We also discuss some reliability and integrity issues related to COMPMOD patterns.

3.2 Workflow patterns

A Long Running Transaction in COMPOD model is represented as a workflow schema (LRT-WF). A workflow systems' functionality depends on task sequencing, split parallelism, synchronization and iteration constructs as means of automating the business process. Different workflow management systems provide different semantics for the same construct. We adopt the semantics from "Workflow Patterns" (Russell et al., 2006) as a paradigm for the informal descriptions of our model constructs. The workflow patterns approach proposes an imperative definition of work flow patterns and provides the patterns as a standard to be employed by business process designers and workflow system developers.

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 for the definition of the 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, twenty patterns were introduced in (Van Der Aalst et al., 2003) and a total of forty three control patterns were revised/proposed in (Russell et al., 2006).

The patterns range from simple constructs that are supported by most of the workflow management systems to complex routing primitives that are not yet supported by today's commercial workflow management systems or business process modeling languages. The work supports each pattern with an informal description and context assumptions, formal descriptions using Colored Petri Nets (Jensen, 1997) implementation related issues, and provides evaluation criteria for workflow developers to assess their offerings of full, partial, or no support of a given pattern.

Workflow Patterns are classified in (Russell et al., 2006) as (a) five basic control-flow patterns, (b) four advanced branching and synchronization patterns, (c) two structural patterns, (d) four multiple instance patterns, (e) three state-based patterns, (f) two cancellation patterns., and (f) twenty three new control flow patterns which add to the above classifications in addition to loops and multiple instances patterns. The COMPMOD model assumes only a single instance of activities for a given process instance and therefore multiple instances, loops and interleaved patterns are not yet supported by the model. However, their applicability is a practical extension of the model and is discussed as a future work in this research in (Chapter 8).

Workflows embrace branches of execution that are split, synchronized, merged, or discriminated at different points in the workflow process. A split pattern splits a branch of execution into two or more branches and the type of split construct determines the mode of branch routing. There are three basic split patterns, namely; Parallel Spilt (AND-split), Multi-Choice (OR-split), and Exclusive Choice (XOR-split). Parallel Spilt and

Multi-Choice create concurrent routing of execution branches, while Exclusive Choice creates exclusive routing, where only one of the split branches is enabled at runtime depending on distinct choice conditions associated with each branch.

Two or more branches of executions can be synchronized, merged without synchronization, discriminated (only 1 out of M paths is chosen), or partially joined (N-out-of-M) by using a join construct that reflects the required semantics of the join.

The LRT-WF schema of COMPMOD is modeled as a structured workflow. Structured in this context can be viewed as a notion of well-formedness (Kiepuszewski, Hofstede, & Bussler, 2000), where concurrent and exclusive branches are encapsulated within *scope patterns*. Scope patterns, our contribution to the workflow patterns initiative, start with a split pattern and end with a join pattern. The type of split and join patterns reflect the required operational semantics of the scope.

Scope patterns in COMPMOD can encapsulate further scopes, thus allowing for the modeling of multi nested transactions. The number of splits and joins within a nested scope are balanced, and not interleaved.

The structured nature and the operational semantics of our scope patterns are emphasized at both; the split type and the join type of the scope pattern. Due to the diversity of join constructs, we apply further classification to the patterns proposed in (Russell et al.,

2006), based on the operational semantics of join patterns and utilize this classification in many different ways throughout the discussions in this thesis, including:

1- Informal and formal description of proposed scope patterns
2- Evaluation of partially supported join patterns
3- Evaluation of potentially applicable new scope patterns given the underpinning structure semantics of the model.
4- Discussions and Conclusions.

We classify join patterns[1] as follows:

1- Synchronization (AND-join): *the convergence of two or more branches into a subsequent branch such that the thread of control is passed to the subsequent branch when all input branches have been enabled*. The context of the pattern assumes that (a) the incoming branches are parallel and result from an earlier AND-split, (b) each incoming branch executes only once, and (c) the construct is enabled once all incoming threads are completed. The (Generalized AND-join) is a variation of AND-join where multiple instances of incoming branches are allowed.
2- Merge: *the convergence of two or more branches into a single subsequent branch. Each enablement of an incoming branch results in the thread of control being passed to the subsequent branch.* There are two variations of this construct, the Simple-Merge (XOR-join), which allows only one incoming thread to be active at any time, while in the (Multiple-Merge) construct, it is possible for more than one

[1] Descriptions in italics are borrowed from RUSSELL, N., TER HOFSTEDE, A. H. M. & MULYAR, N. 2006. Workflow controlflow patterns: A revised view.

3- incoming branch to be active simultaneously. Note that incoming branches are assumed to be distinct, and do not necessarily diverge from an earlier split pattern, and need not to be synchronized.

4- Partial join (N-out-of-M): *the convergence of M branches into a single subsequent branch following a corresponding divergence earlier in the process. The thread of control is passed to the subsequent branch when N of the incoming branches have been enabled.* Variations of this join pattern are: (a) Structured Partial Join, where s*ubsequent ennoblements of incoming branches do not result in the thread of control being passed on. The join construct resets when all active incoming branches have been enabled.* (b) Blocking Partial Join where *the join construct resets when all active incoming branches have been enabled once for the same process instance and subsequent enablement of incoming branches are blocked until the join has reset* – ideal for scopes within loops, and (c) Cancelling Partial Join where *triggering the join also cancels the execution of all of the other incoming branches and resets the construct.*

5- Discriminator (1-out-of-M): *the convergence of two or more branches into a single subsequent branch following a corresponding divergence* (in case of the Structured Discriminator), *or following one or more corresponding divergences* (in case of the Unstructured Discriminator) *earlier in the process model. The thread of control is passed to the subsequent branch when the first incoming branch has been enabled.* Variations of this join pattern are: (a) Structured Discriminator where *subsequent enablement of incoming branches do not result in the thread of control being passed on and the construct is reset when all*

6- *incoming branches have been enabled,* (b) Blocking Discriminator where *the discriminator construct resets when all active incoming branches have been enabled once for the same process instance. Subsequent ennoblements for of incoming branches are blocked until the discriminator has reset* – ideal for constructs within loops, and (c) Cancelling Discriminator where *triggering the discriminator also cancels the execution of all the other incoming branches and resets the construct*.

7- Synchronization Merge: *the convergence of two or more branches into a single subsequent branch. The thread of control is passed to the subsequent branch when each active incoming branch has been enabled.* Variations of this pattern are (a) Structured Synchronization Merge where the *converged branches are diverged earlier in the process at a uniquely identifiable point* -ideal for synchronizing branches resulting from an OR-Split, (b) Acyclic Synchronization Merge where *the converged branches are diverged earlier in the process and determination of how many branches require synchronization is made on the basis of information locally available to the merge construct*-ideal for non-structured workflows, and (c) General Synchronization merge where *the converged branches are diverged earlier in the process and the thread of control is passed to the subsequent branch when each active incoming branch has been enabled or it is not possible that the branch will be enabled at any future time*-ideal for *non-structured and highly concurrent workflows that include looping structures.*

8- Other join patterns: there are a few more join constructs that deal with multiple instances of activities within a given process instance, and with multiple execution thread instances in a single branch. These patterns are not discussed in this work.

In COMPMOD we provide:

1- New operational semantics of exclusive split and join patterns: XOR*-split and XOR*-join that allows for alternative exclusive choices such that only one alternative can be tried at any time.
2- Explicit support for sequence, AND-split, OR-split, and XOR*-split.
3- Implicit support for AND-join (synchronization), OR-Join (Structured Synchronization Merge), and XOR*-join.
4- Explicit support for the operational and transactional semantics of three new scope patterns: AND-scope, OR-scope, and XOR-scope.

Further patterns (other than sequence, split, and join patterns) are also either fully supported by the model, as in "implicit termination" or partially supported, as in "cancel region". In Appendix C, we provide an evaluation for COMPMOD in terms of the extent of support of each pattern.

3.3 Informal description of COMPMOD patterns

In the following subsections we discuss the informal descriptions of the main workflow patterns in COMPMOD that explicitly outlines the three basic execution routing modes: sequence, concurrent, and exclusive execution of branches.

3.3.1 Sequence Pattern

The sequence pattern is the main building block of the WF process. It allows connecting tasks[2] in sequential order. The pattern is informally described as:

> ***Def. 3.1 (Sequence Pattern)*** (Russell et al., 2006)***:*** *An activity in a workflow process is enabled after the completion of a preceding activity in the same process.*

For example, in a supplier's sales department, after the order has been received from a manufacturer, an auditor activity will check the order to decide whether to accept it or not (Figure 3.1).

Figure 3.1 Sequence pattern in supplier sales process

In our model, a task or a set of interrelated tasks (scope pattern) can be appended to another task or scope in sequential order on the same execution branch.

3.3.2 Scope Patterns

Informally, a scope pattern is defined as follows:

> ***Def. 3.2*** (scope pattern): A scope-pattern is a composite pattern that couples a split pattern with a join pattern to ensure a symmetrical structure of the scope. The

[2] Throughout the discussions, tasks, activities, web services, and atomic nodes (Chapter 4 onward) are all used to refer to an atomic unit of work.

scope starts at the split point and ends at the join point. The scope is enabled when the incoming branch to the scope is enabled. The split construct of the scope diverges the incoming branch into two or more branches which are converged later by the joint construct. Enabling diverged branches and the join construct depends merely on the semantics of the split and join patterns respectively.

In Figure 3.2, we illustrate a generic representation of a scope pattern that scopes three activities A1, A2, and A3.

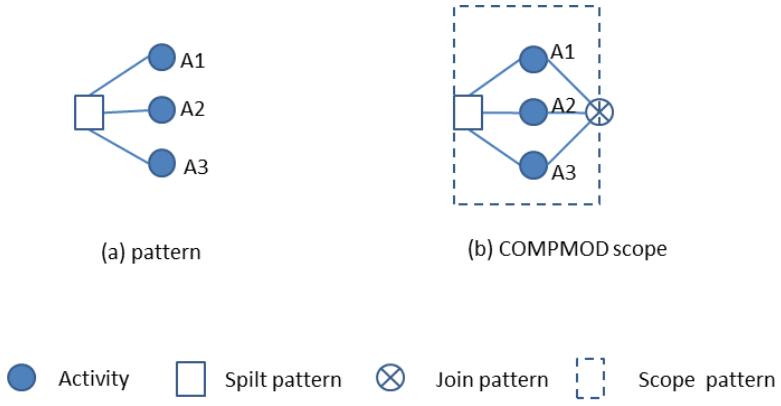


Figure 3.2 A generic scope pattern representation

A diverged branch within a scope may entail one or more tasks that are connected through sequence patterns. A task can be an individual task or a scope pattern, thus allowing the construction of nested scope patterns that contains a balanced number of splits and joins and thus is symmetrical by construction.

3.3.2.1 Concurrent Scopes

A concurrent scope creates two or more parallel branches. Once the scope is enabled, all concurrent branches are enabled simultaneously. Concurrent branches are synchronized via a synchronizer join construct. The synchronizer is enabled when all parallel branches are completed. We introduce two concurrent scope patterns, AND-scope and OR-scope. An AND-scope starts with an AND-split (parallel split) pattern and is coupled with a synchronizer (AND-Join).

We provide an informal description of the AND and OR scope patterns based on both, the semantics of the individual patterns involved as described in (Russell et al., 2006) and the general definition of scope patterns (Def. 3.2).

> ***Def. 3.3 (AND-scope):*** the divergence of a branch at the split point of the scope into two or more parallel branches that are executed concurrently when the scope is enabled. Concurrent branches are synchronized at the join end of the scope and execution control can be passed to the task immediately following the synchronizer once all of the concurrent branches have completed their executions.

As an example, in Supplier-Manufacturer outsourcing business process OP (Figure 2.1), after the SALES activity is completed, three activities (CHARGE, OUTSOURCE_ANALYSIS, and DELIVERY) are instantiated in parallel. This control flow represents an AND-join pattern. In COMPMOD, this structure is represented by an AND-scope pattern as illustrated in Figure 3.3. Note that in the original process logic of

OP, the DELIVERY activity is followed by the CHECK_GOODS activity representing a sequence pattern between them. Therefore, CHECK-GOODS is enclosed within the AND-scope pattern.

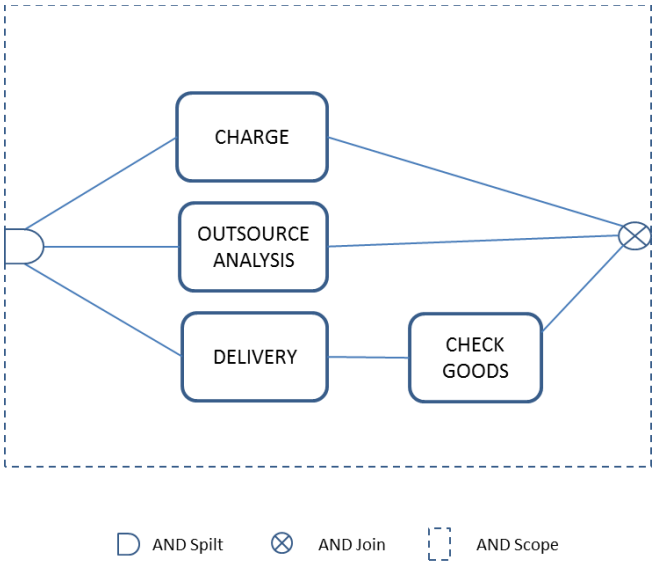


Figure 3.3 AND-scope pattern in OP process

A variant of the concurrent scope is the conditional concurrent scope where only a subset of the parallel branches are enabled based on logical conditions paired with each parallel branch. The synchronizer is enabled when all enabled parallel branches are completed. An OR-scope starts with an OR-split (Multi-Choice) pattern, and is coupled with a Structured Synchronizer Merge.

Def. 3.4 (OR-scope): the divergence of a branch at the split point of the scope into two or more parallel branches where only a subset of the branches are executed concurrently when the scope is enabled. The selection is based on the outcome of logical expressions associated with each parallel branch. The selected concurrent branches are synchronized at the join end of the scope and execution control can be passed to the task immediately following the synchronizer once all of the selected concurrent branches have completed their executions.

As an example, in E-Supply Chain systems, after an order has been received by a company and the payment has been received from the customer, an inventory check is performed to investigate the availability of goods in the company's warehouses. If the ordered goods are available, the goods are delivered to the customer. If the ordered goods are not available, a manufacture plan process is instantiated to provide the customer with the ordered goods from different supplier(s)/manufacturer(s). In COMPMOD, this process logic is represented by the OR-scope pattern illustrated in Figure 3.4.

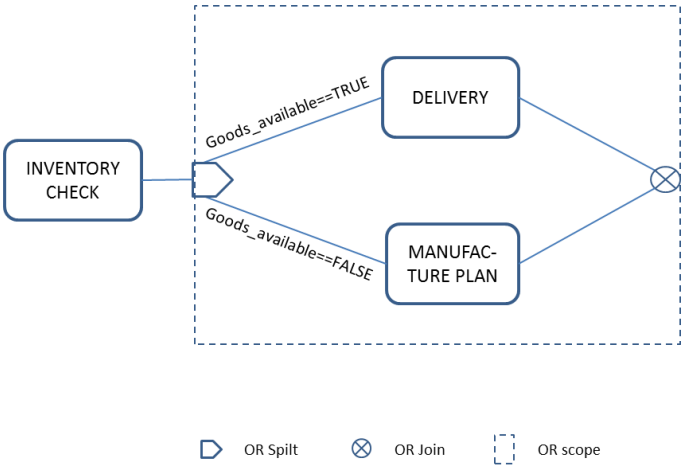


Figure 3.4 OR-scope pattern in Supply Chain process

3.3.2.1 Exclusive Scopes

An exclusive scope creates two or more exclusive branches. Exclusive branches alternate with each other, but only one exclusive branch is enabled, based on some distinct criteria. If an enabled branch fails to complete, an alternative branch is enabled. The scope starts with an exclusive split pattern, and ends with an exclusive join pattern. The join pattern is enabled when exactly one of the incoming exclusive branches has completed. In (Russell et al., 2006), diverged branches in XOR-split pattern are enabled, based on distinct logical values associated with each branch and does not provide alternative enablement of branches. The XOR-join (Simple Merge) allows only one incoming branch to be enabled at a time, but allows all incoming branches to be enabled. Therefore we extend *Workflow Patterns* with two individual patterns as a variation of both the XOR-split and XOR-join, namely the XOR*-split[3] and XOR*-join.

We extend the semantic of the XOR-split as follows:

> ***Def. 3.5 (XOR*-split):*** The divergence of a branch into two or more branches. When the incoming branch is enabled, the thread of control is immediately passed to precisely one of the outgoing branches based on the *highest priority* criteria, where the first branch has the highest priority. If a branch fails to complete, an alternative branch is enabled (if any). The alternative branch is the one with the next highest priority.

[3] Similar to preference relation in ZHANG, A., NODINE, M., BHARGAVA, B. & BUKHRES, O. Ensuring relaxed atomicity for flexible transactions in multidatabase systems. 1994. ACM, 67-78.

Def. 3.6 (XOR*-join): the convergence of two or more branches that had diverged from an XOR*-split at some point earlier in the WF process. The construct is enabled when exactly one of the incoming branches has been completed.

Accordingly, we provide an informal description of the XOR-scope pattern based on definitions (Def. 3.2, 3.5, and 3.6).

Def. 3.7 (XOR-scope): the divergence of a branch at an XOR*-split point of the scope into two or more exclusive branches that are converged later at an XOR*-join point. When the scope is enabled, execution control is immediately passed to precisely one of the outgoing branches, based on *highest priority* criteria where the first branch has the highest priority. If an exclusive branch fails to complete, an alternative branch (if any) is enabled. The XOR*-join construct is enabled when exactly one branch is completed.

The extension of XOR-scope is motivated by two aspects:

(a) Business process aspect: often a number of alternative tasks are proposed in the workflow, but there is a clear preference for one over the other. For example, an e-booking scenario could be searching for an outbound journey to a destination where the priority is given to flights. If no flights are available for the required

(b) date then trains may be tried. The last priority could be travelling by bus if no trains are available.

(c) Long-Running transactional aspect: when a sequence of tasks is required to be executed by a business process that executes over a long period of time and the risk of failing this sequence is not affordable, then the sequence of tasks could be alternated by an alternative sequence of tasks from the business point of view. In case of the failure of the first priority scenario, an alternative scenario is tried. E.g. in an e-supply-chain business scenario, a contract with one of two or more suppliers (prioritized according to their quotes, location, or quality) should be guaranteed for a specific product where the contract process might include many interrelated tasks. If a contract process fails to complete for a specific supplier, an alternative supplier can be tried.

To illustrate the XOR-pattern by example, we consider a delivery process in a typical supply chain system. Usually, different delivery methods are provided depending on the company's delivery policies or customer location. Let us assume that in a specific delivery scenario, a company offers two methods of delivery: deliver by car or deliver by plane where priority is given to car delivery. If car delivery is not possible, then delivery by plane is attempted. In COMPMOD, this process logic is represented by an XOR-scope pattern as illustrated in figure 3.5.

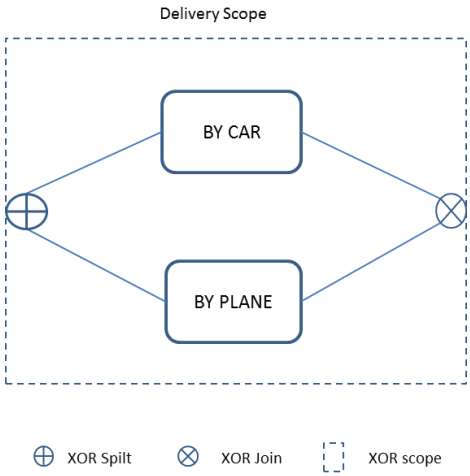


Figure 3.5 Delivery XOR-scope pattern

3.4 Reliability and Integrity Issues

Informal descriptions of workflow patterns clarify the operational semantics of the constructs in an abstract way and from the control point of view of their intended functionality. The descriptions designate when a construct is enabled with respect to the enablement of its incoming or outgoing branches. Our workflow model is a transactional workflow model where the control flow mechanism is influenced by transactional properties such as completions, failures, or cancellations of workflow activities or activity scopes. The transactional behavior of a certain activity has an impact on other interrelated activities. For example, in Def3.1, an activity is executed when the preceding activity has completed. The definition does not state what happens when the preceding activity fails.

Given the nested structure nature of the workflow schema, a failure or cancellation of an activity has an impact on the transactional behavior of other interrelated activities or

encapsulated scopes. An additional challenge is that the transactional nature of our workflow model implies that the behavior of the workflow must be reliable and the overall system should always be guaranteed to be in a consistent state.

A major concern in reliability assurance is on the failure handling mechanism supported by the management model of the workflow. Analogous to failure handling support, and equivalent to it in importance is the compensation handling mechanism. The informal semantics of the exclusive scope Def3.7 states that when an exclusive branch is enabled but fails to complete, then an alternative branch is enabled. However, it does not state what happens to the partially completed activities in the failed branch. Transactional integrity assurance requires the partially completed activities to be compensated before the alternative branch is executed, due to the potential assumption that alternative branches attain the same overall task from the business point of view.

Deadlocks may arise from the ambiguous behavior of join constructs. For example, a synchronizer with *m* incoming branches assumes *m* enablement of branches for the construct to be fired. If one or more of the branches fails, the construct goes into a deadlock state. Hence, the synchronizer should be supported with transactional semantics to constantly ensure the consistent behavior of the construct even in case of failures.

To address the issues mentioned, we augment the operational semantics of workflow patterns with transactional semantics to formally define the implemented patterns in COMPMOD. Thus, each workflow activity, branch, and scope is defined with a set of

transactional dependencies: activation, completion, failure, cancellation (force-fail), and compensation (when necessary). Dependencies are employed to model a reliable interrelated behavior of workflow components which consequently guarantees a reliable overall behavior of the model. The formal transactional semantics of the model are defined through (a) Transactional Dependencies, and (b) Management and Compensation policies. Formal descriptions are detailed in chapters 5 and 6.

3.5 Transactional Patterns

In our model, workflow tasks are web services. Orchestration deals with how different services are composed into a coherent whole (LRT). It specifies the order in which services are invoked, and the conditions under which a certain service may or may not be invoked (Alonso, 2004). Our orchestration mechanism is inspired by the "Transactional Patterns" approach (Bhiri et al., 2006a) (Bhiri et al., 2006b). 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, 1991), and thus they combine the flexibility of work flow control patterns with the reliability of transactional models to ensure the transactional consistency of service compositions.

Web services emphasize transactional properties for their characterization and correct usage. In (Bhiri et al., 2006a), these properties are assumed to be *retriable, compensable*, and *pivot*. A service *s* is said to be *retriable* if it is sure to complete after a finite number of activations, while *s* is *compensable* if it offers compensational policies to semantically

undo its effects, and *s* is said to be *pivot* if once it successfully completes then its effects cannot be undone.

Each service has a set of operations, depending on the transactional property of the service. A pivot service has a minimal set of *abort(), activate(), cancel(), fail()*, and *complete()* to allow its abortion, activation, cancellation, failure, and successful completion. A compensable service has in addition a *compensate()* operation to allow for its compensation. A retriable service has a *retry()* operation to allow for its activation after failure.

The 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 others. Dependencies are used to express the relationships that exist between services such as sequence, alternative, compensation, activation or cancellation dependencies. They also associate preconditions with service operations. The general definition of a dependency is:

> ***Def.3.8 (Bhiri et al., 2006a):*** **A *dependency* from service s_1 to service s_2 exists if a *transition* of s_1 can fire an *external transition* of s_2.**

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()*).

The transactional patterns paradigm discusses simple patterns such as AND-split or XOR-split, where a single service exists on each split branch. In addition, the way in which the dependencies are defined does not allow for nesting in the composite service. The failure handling and recovery mechanism are implemented through compensation and alternative dependencies.

We have drawn inspiration from transactional patterns, but provide solutions for multiple nested transactions. We extend the notion of transactional patterns to model multi-nested transactions by introducing the following concepts (detailed discussion in Chapter 4):

- Atomic nodes, scopes, nested scopes, and execution paths and their transactional dependencies and 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;
- Management and compensation policies to support an underpinning framework for imposing and automating the control flow of events.

Chapter 4

Model Architecture

4.1 Introduction

In this chapter, we discuss the underlying structure of the COMPMOD model. We discuss features of the model, the representation of our workflow model, our model assumptions, and formal definitions of the workflow patterns and the generic formal definitions of transactional dependencies and management policies. This chapter forms the basis for Chapters 5 and 6.

4.2 Features of COMPMOD

COMPMOD is a conceptual management framework for WF Long Running Transactions, focusing on the control flow perspective of management. Transactions are designed based on structured workflow schemas, where WF constructs are supported

with well-defined operational and transactional semantics. On the one hand, the model aims at ensuring the reliability and integrity of transaction execution in the context of long duration executed through autonomous and loosely coupled web services. On the other hand, and given the business oriented nature of LRT's, the model is aimed at providing flexibility in incorporating business and compensation logic into the design of transactions in a clear and user friendly way.

Transactional semantics of WF constructs are defined through behavioral dependencies and management control policies. Dependencies are defined as predicate logic formulas over component states and/or attribute values. Satisfying a dependency fires an execution event, such that when an LRT or one of its components activates, completes, fails, force-fails or compensates, an execution event is fired. A management policy assesses the fired event and performs an action based on the operational semantics of the WF model. The applied event-control-action mechanism is built on top of a recursive hierarchical structure of the WF schema, and is facilitated through automated propagation mechanisms that are merely influenced by the recursive hierarchical nature of the WF schema.

The management of LRTs must proceed in two parallel directions:

(a) The management of the LRT during its normal execution mode, which must embrace a reliable and efficient fault-handling and partial compensation mechanism.

(b) The management of the LRT during the execution of its compensation mode comprehensive compensation, in case the LRT has failed to successfully 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 for the separation of the actual process and handling of execution and exceptions in a vibrant and flexible way; and 5-8 are requirements that ensure 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 the business logic of long running transactions through transactional dependencies;
4- Rule-based Policies for managing execution and compensation control flow;
5- Automated method for propagating activation and successful completion events through the hierarchy structure as a management mechanism.
6- Automated method for propagating failure events through the hierarchy structure as a failure handling mechanism.
7- Automated method for performing compensation actions while the LRT execution is in progress, through backward and forward order compensations.
8- Flexibility in extending the model through new WF patterns.

Aspects 1-7 have been addressed in the proposed model and discussed in this thesis. The flexibility of the model is expressed through the extensibility property of COMPMOD, and is discussed in Chapter 8.

4.3 Representations of Nested LRTs

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

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. A scope node encapsulated by an execution path is interpreted the same as an atomic node. In other words, scope nodes on an execution path are like black boxes that encapsulates execution paths and other nodes.

4.3.1 Workflow Model

An LRT, at its highest level, is executed as a flat transaction, i.e. a sequence of nodes that are executed sequentially (Figure 4.1). The main execution path is denoted as p_0. A node can be an atomic node or a scope node. Each scope creates two or more execution paths that start from the split point and end at the join point 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 allowing arbitrary levels of nesting. Through the rest of the discussion, we will use the term component to refer to both nodes (atomic/scope) and execution paths.

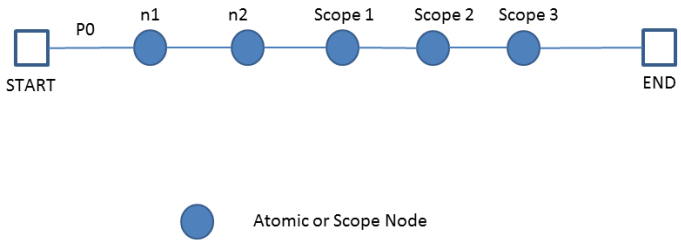


Figure 4.1 A WF showing level 0 of a sample LRT

The modelling 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. The main execution path of a transaction is regarded as level 0 in the workflow. Figure 4.2(a), demonstrates an expanded two level-nesting of the sample LRT in Figure 4.1 and Figure 4.2(b) demonstrates the LRT with level 2 of the WF collapsed.

Considering the execution path 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. We provide a *nodeList* attribute on path objects to express this: for example $p_1.nodeList$=[n_6,$scope_{2.1}$]. 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 (figure 4.1).

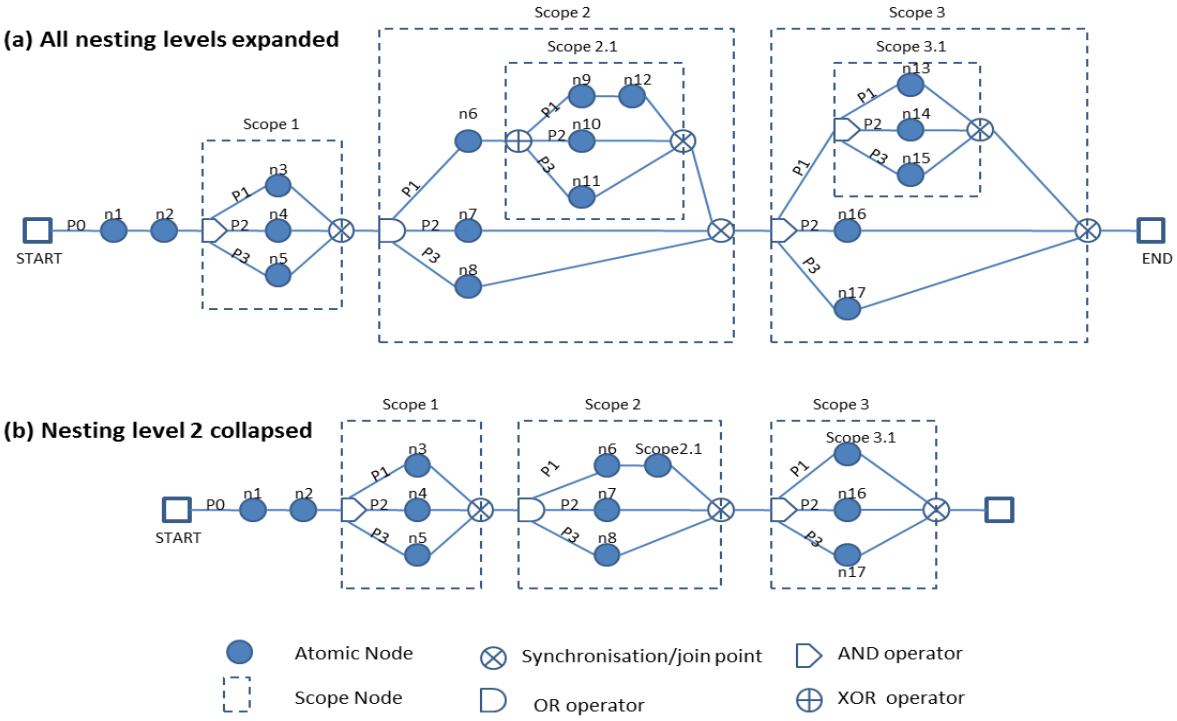


Figure 4.2 A WF showing multi levels of a sample LRT

4.3.2 Hierarchical Structure Model

Transaction components –nodes and execution paths- are linked together in a hierarchical structure. Each component has a single superior, and an ordered set of one or more inferiors. More 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 an LRT has no superior. Each execution path has one or more inferiors. The 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. Figure 4.3 illustrates the hierarchy structure of the sample workflow in Figure 4.2(a).

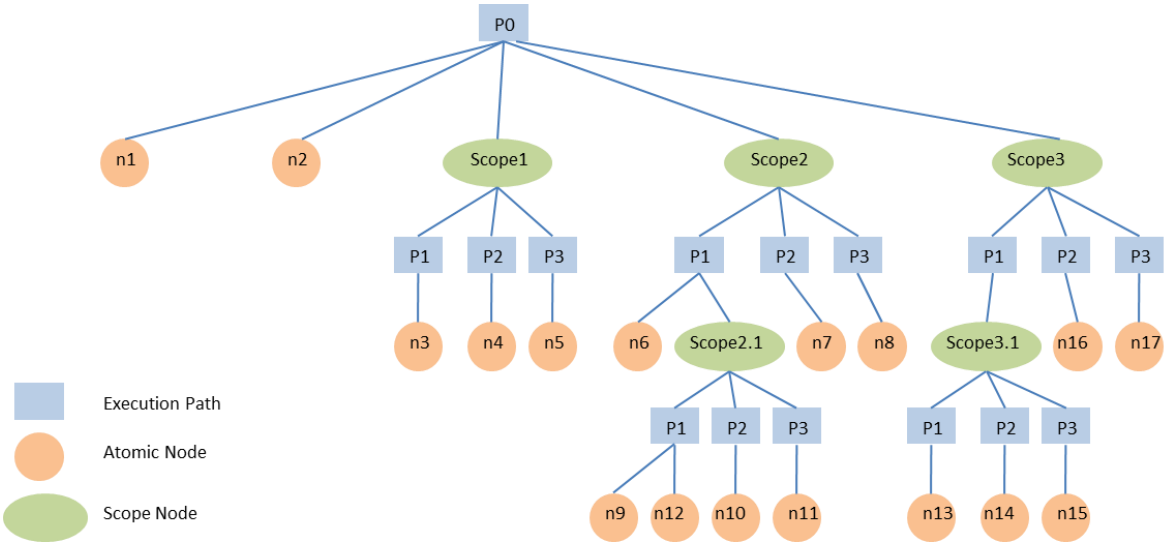


Figure 4.3 Hierarchal Structure of WF schemas

4.3.3 Transactional operators and scopes

COMPMOD's WF schema is formed as a structured workflow that supports the design of arbitrary nested levels of transactions. The well-formed structure of the LRT is forced by the model, meaning that the burden of maintaining the balanced structure of (*split* and *join*) patterns is imposed by the model.

A scope node starts with a split operator (OR, AND, or XOR) that is explicitly assigned while constructing a scope. The syntax of the scope is defined as:

$$\boldsymbol{Scope} = (\boldsymbol{operator}, [\boldsymbol{splitNode_1}..\boldsymbol{splitNode_m}])$$

When a scope is initially defined, a split operator and a list of split nodes are specified. A split node can be an atomic node or a scope node, which facilitates the construction of nested scopes.

The AND-split pattern in (Figure 4.4 (a)) is defined as (AND,[n_1,n_2,n_3]) and is implemented in COMPMOD as depicted in (Figure 4.4 (b)) where the split pattern is coupled with a synchronisation point representing the implicit AND-Join. The number of split nodes corresponds to the number of execution paths encapsulated within the scope. Therefore, the scope in (Figure 4.4(b)) creates three execution paths namely p_1, p_2, and p_3 which are represented by the order list *PathList* of the defined scope node.

A scope in COMPMOD is formally defined as:

Def.4.1: **(Scope Definition)**
A *scope* is defined as follows:
$\forall \boldsymbol{i}$=**1**..$\boldsymbol{m}$ $\boldsymbol{p_i.nodeList}$=$\boldsymbol{splitNode_i}$ ***and***
$\forall \boldsymbol{i}$=**1**..$\boldsymbol{m}$ $\boldsymbol{nodeList_i.type}$={$\boldsymbol{ATOMIC}$,$\boldsymbol{SCOPE}$}**:**
$\boldsymbol{scope}$=($\boldsymbol{operator}$,[$\boldsymbol{splitNode_1}$..$\boldsymbol{splitNode_m}$]) →
$\boldsymbol{scope.pathList}$=[$\boldsymbol{p_1}$..$\boldsymbol{p_m}$]
where operator ∈{AND,OR,XOR}

As mentioned earlier, each execution path creates an ordered list of one or more nodes, denoted by *nodeList*. When a node is appended to an existing execution path pi, the node is appended to $pi.nodeList$. The main building block construct of the WF is the sequence construct. A sequence pattern connects two nodes in a sequential order. The sequence pattern is formally defined as:

Def.4.2: **(Sequence pattern)**
A sequence pattern is defined as follows:
$node_1.type=\{ATOMIC,SCOPE\}$ ***and*** $node_2.type=\{ATOMIC,SCOPE\}$***:***
$SEQPattern=(SEQ,node_1,node_2) \rightarrow$
$p_i.nodeList=p_i.nodeList+[node_2]$ **,** $node_2.superior=p_i$
where $p_i=node_1.superior$

Accordingly, the two level nested scope of (Figure 4,4 (c)) can be denoted by the following constructs:

$Scope_1=(OR,[(SEQ,n_1,scope_{1.1}),n_2,n_3])$ where $scope_{1.1}=(XOR,[(SEQ,n_4,n_7),n_5,n_6])$.

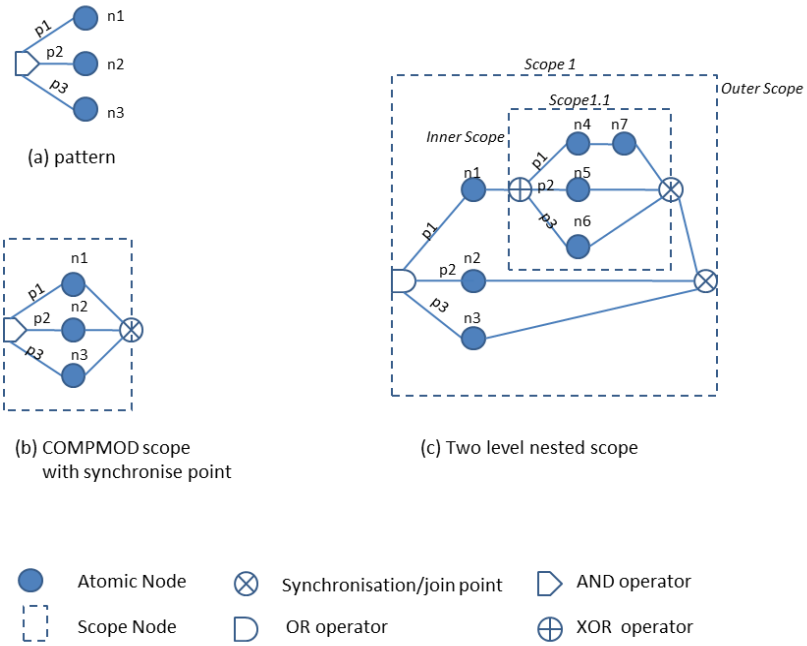


Figure 4.4 Scope Structure

4.3.4 Execution paths

The type of scope pattern determines the routing mode of its encapsulated paths. An AND-scope creates two or more concurrent execution paths, while an OR-scope creates a two or more concurrent paths where only a subset of these paths are executed during runtime, the executed paths are those whose enabling condition are satisfied. An XOR-scope creates two or more exclusive paths: the first path has the highest priority and therefore execution starts with the path with the highest priority. If an exclusive path that has an alternative path with lower priority fails to complete, the path is compensated in backward order, until the split point of the scope is reached (this is done as part of the

forward compensation of the LRT), and then the alternative path is executed. Therefore, execution paths are assigned with the following transactional attributes:

1. An execution path *IsConcurrent* if it is encapsulated within an immediate OR-scope or AND-scope superior.

   path.superior={AND,OR} → path.IsConcurrent=TRUE

2. An execution path *IsExclusive* if it is encapsulated within an immediate XOR-scope superior.

   path.superior=XOR → path.IsExclusive=TRUE

3. An execution path *hasAlternative*, if it *IsExclusive* and has a path with lower priority in the same scope.

   path.IsExclusive ∧ successor(path)≠NULL → path.hasAlternative=TRUE

4. A concurrent path does not have an alternative.

   path.IsConcurrent=TRUE → path.hasAlternative=FALSE

5. Apart from the main execution path, a path must either be concurrent or exclusive.

   path.IsConcurrent=TRUE → path.IsExclusive=FALSE

   path.IsExclusive=TRUE→ path.IsConcurrent=FALSE

6. An execution path is *IsEnabled* if and only if it *IsConcurrent* path within an OR immediate scope and its branching condition is satisfied at runtime.

7. The main execution path is a special case where:

   path=p_0 → path.IsExclusive=FALSE ∧ path.IsConcurrent=FALSE ∧ path.IsEnabled=FALSE

4.3.5 Vitality of components

Each LRT component has a *vitality* attribute that allows it to specify whether a component is vital or non-vital. A vitality value *IsVital*={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 the failure of the specific service or scope of services 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.

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.

Vitality of components is utilised in the control propagation mechanism proposed in the model. The transactional implication of the vitality measure of a component expresses the

impact of successful completion or failure of a component on its immediate superior and on its successor in case of node components.

A **vital node's** successful completion is *necessary* for

1- The successful completion of its superior path

2- The activation of its successor node(if any)

The failure of a **vital node** leads to the failure of it superior path (by propagation), and consequently the execution of the path, ends.

Successful completion of **non-vital nodes** is *desirable* for the successful completion of its enclosing path, but is not necessary. In other words, the failure of a **non-vital node** will not fail its enclosing path unless it was a non-vital path and all its nodes have failed. The same applies to the activation of a non-vital node's successor, if one exists. The successful completion of a non-vital node is *desirable* for the activation of its successor, but not necessary. Hence, the failure of a non-vital node will still trigger the activation of its successor (if any).

Execution paths are either concurrent or exclusive. The effect of the successful completion or failure of paths, with respect to their vitality measure, is most evident for concurrent paths.

The successful completion of a vital concurrent path is necessary for the successful completion of its immediate superior scope. The failure of a vital concurrent path will fail

its superior scope, and consequently force-fail all the concurrent paths within the same immediate superior scope.

The vitality of an exclusive path does not have a direct impact on the successful completion or failure of its enclosing scope. An exclusive scope succeeds if one of its exclusive paths successfully completes, and fails if all its exclusive paths fail to succeed regardless of their vitality measure. Therefore, we consider only concurrent scopes when discussing the assignment of vitality measure to scope nodes.

We classify concurrent scopes with regard to the assignment of vitality to the scope and its encapsulated paths into three cases:

Case 1: a vital scope with at least one vital path.
Case 2: a non-vital scope with any combination of encapsulated vital/non-vital paths.
Case 3: a vital scope with all paths as non-vital.

Case 3 does not seem useful from the business point of view. However, while case 3 could be designed, it is not desirable, and hence, will exclude it through vitality assumptions 2 below.

We justify our exemption of Case 3 as follows: vitality is a way of stating the necessity of success of a specific component. If we assume that a scope is vital and is necessary to succeed, then we implicitly assume that at least one of its paths is guaranteed to succeed.

In case 3, where not all paths are vital, they are all desirable but not necessary to succeed, which seems to contradict with the vital assignment of the enclosing scope. However, it may be argued that in some senses, a vital scope with only non-vital nodes would succeed if only one of the nodes succeeded; thus we wished to leave the option to the business process designer.

However, to ensure that processes are generally sensible, we have assumed logical restrictions by the model with respect to the design of LRTs as listed in Section 4.7.

4.4 Workflow of OP Case Study in COMPMOD

We represent the OP business process in (Chapter 2, Figure 2.1) using COMPMOD architecture. First, in Figure 4.5, we depict the workflow representation of OP in COMPMOD. At this stage, we ignore transactional and compensation dependencies but we will refer back to the OP workflow case study in Chapters 5 and 6. We assume that the process logic of OP defines the OUTSOURCE_ANALYSIS activity as a non-vital activity and hence its failure during runtime will not interrupt the execution of OP.

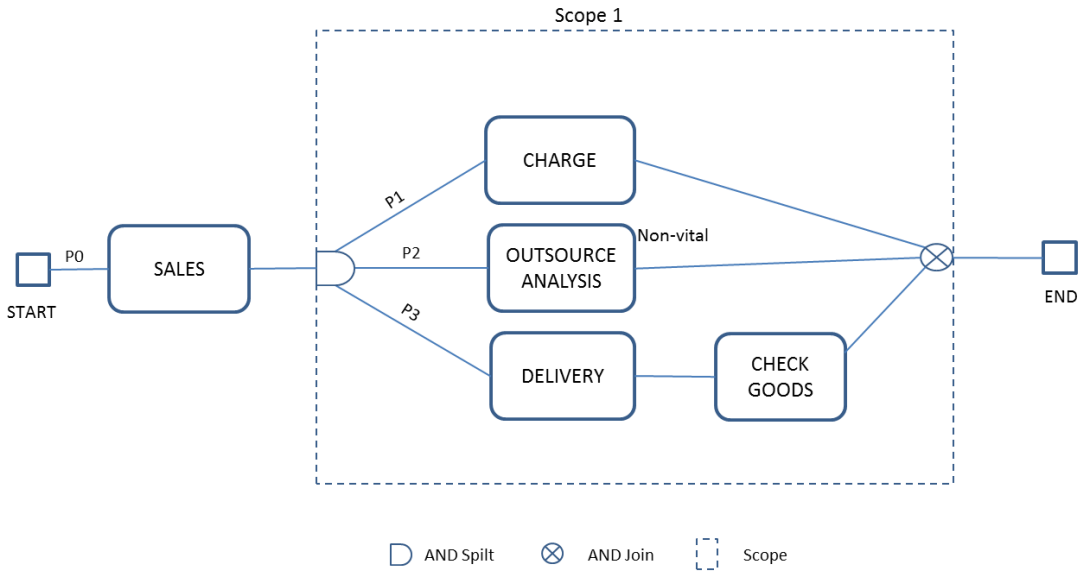


Figure 4.5 OP workflow in COMPMOD

Syntactically, the OP workflow is defined as:

OP=(SEQ,SALES,(AND,CHARGE,OUTSOURCE_ANALYSIS,(SEQ,DELIVERY, CHECK_GOODS)))

In the following, we list the transactional attributes of OP according to COMPMOD model.

(1) Node Types:

Note that *nodeType* is a transactional attribute which is assigned for nodes and hence the following values apply:

SALES.nodeType=ATOMIC ***$Scope_1$.nodeType=SCOPE***

CHARGE.nodeType=ATOMIC ***OUTSOURCE_ANALYSIS.nodeType=ATOMIC***

DELIVERY.nodeType=ATOMIC ***CHECK_GOODS.nodeType=ATOMIC***

(2) Node Lists:

Note that *nodeList* is a transactional attribute which is assigned for execution paths and hence the following values apply:

p_0.nodeList=[SALES,Scope$_1$]
scope$_1$.p_1.nodeList=[CHARGE]
scope$_1$.p_2.nodeList=[OUTSOURCE_ANALYSIS]
scope$_1$.p_3.nodeList=[DELIVERY, CHECK_GOODS]

(3) Path Lists:

Note that *pathList* is a transactional attribute which is assigned for scope nodes and hence the following values apply:

scope$_1$.pathList=[p1,p2,p3]

(4) Vitality attributes:

Note that *IsVital* is a transactional attribute which is assigned for all workflow components and hence the following values apply (Table 4.1):

Component	IsVital
p0	***TRUE***
SALES	***TRUE***
CHARGE	***TRUE***
OUTSOURCE_ANALYSIS	***FALSE***
DELIVERY	***TRUE***
CHECK_GOODS	***TRUE***
scope$_1$.p_1	***TRUE***
scope$_1$.p_2	***FALSE***
scope$_1$.p_3	***TRUE***

Table4.1 Vitality attributes of OP components

(5) Path Routing Attributes:

Note that routing attributes are transactional attributes which are assigned for execution paths and hence the following values apply:

Path	IsConcurrent	IsExclusive	hasAlternative
po	***FALSE***	***FALSE***	***FALSE***
$scope_1.p_1$	***TRUE***	***FALSE***	***FALSE***
$scope_1.p_2$	***TRUE***	***FALSE***	***FALSE***
$scope_1.p_3$	***TRUE***	***FALSE***	***FALSE***

Table 4.2 Path attributes of OP case study

4.5 Reactive Management and Execution states

The management system of transactions (COMPMOD) is implemented as a reactive system controller (Wieringa, 2003) where system components change their execution states and actions in response to stimuli/events. In our model, an event is fired as a result of a behavioral dependency satisfaction. A stimulus is triggered as a result of a transition in the execution state of a transaction component or as a result of the application of a rule (policy), leading to the firing of an event. 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.

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. The state transition diagrams are depicted in Figures 4.6-4.9. Initially, the LRT and all its

components are marked as NOT-ACTIVATED. State transitions are triggered by execution events, and they are marked by the transition actions deployed in the management policies. For example, when an activation event is fired for the LRT, commencing its execution, the activation event is assessed by an activation policy and the action activate(LRT) is performed, which transforms the state of the LRT from NOT-ACTIVTAED to ACTIVATED. Activation of the LRT fires the activation event of the main execution path, and subsequently an activate(p_0) action is performed which transforms the state of p_0 from NOT-ACTIVTAED to ACTIVATED. The effects of events and actions in our model obligate a chain of state transformations that continuously change the state of the LRT and its components, in accordance with the management and compensation policies. The chain of transformations is controlled by the propagation of an events/actions mechanism implemented by the COMPMOD model. If we abstract from the propagation mechanism, then the events and actions have identical effects in our mode, and therefore, the two terms may be used alternatively to refer to state transformation of components.

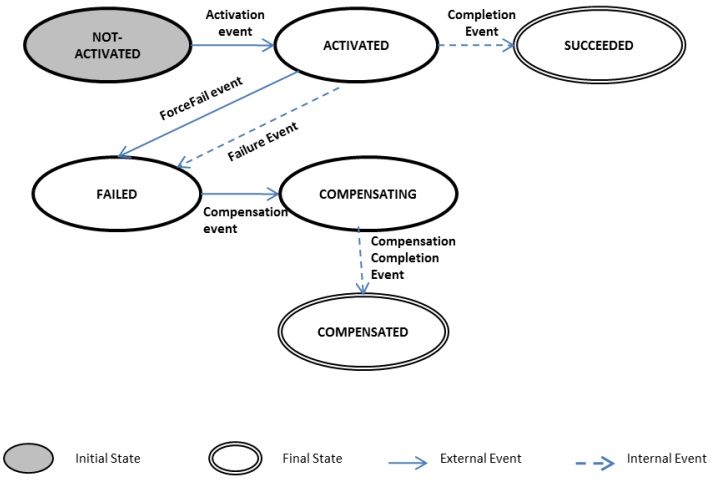


Figure 4.6 STD for LRT

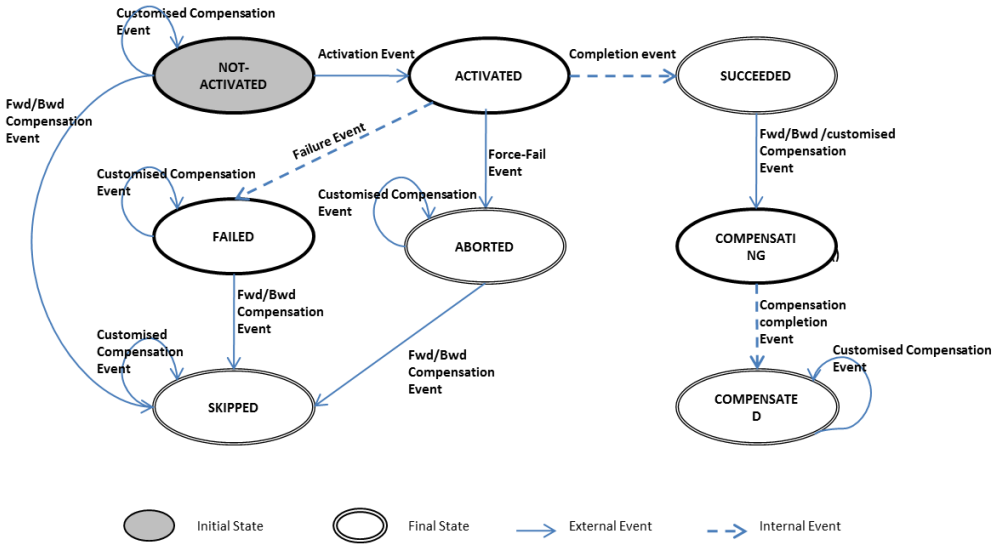


Figure 4.7 STD for atomic nodes

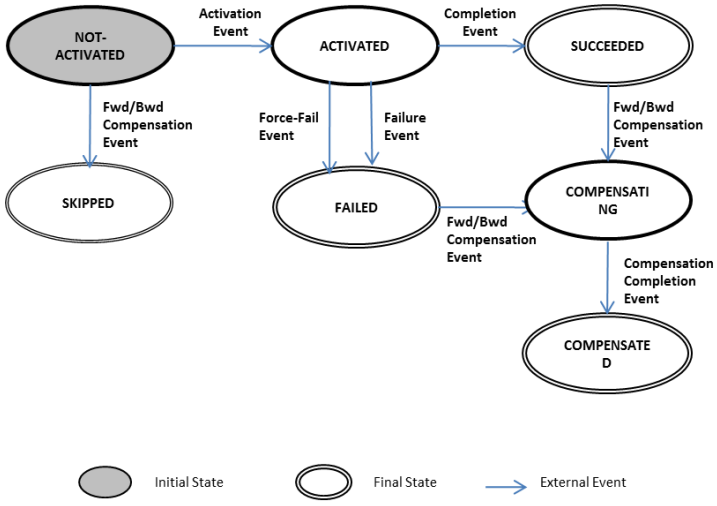


Figure 4.8 STD for scope nodes

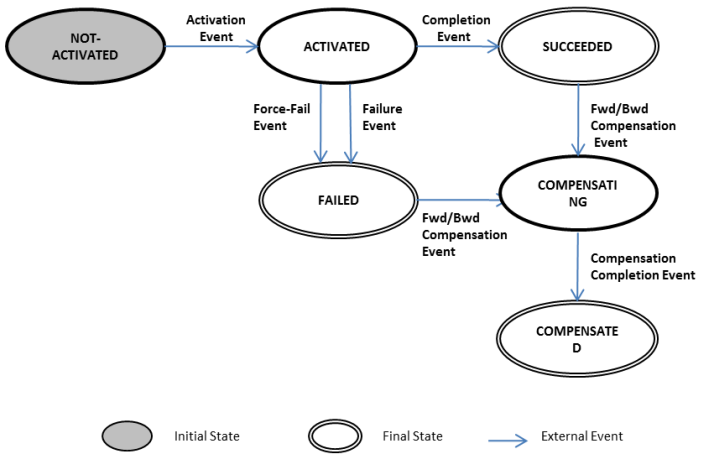


Figure 4.9 STD for execution paths

The state transition in Table 4.3, lists the state transition actions as implemented in the management policies. Note that these actions do not exhibit the propagation of state transformations across LRT components. The propagation mechanism and the semantics of execution states are discussed in detail in Chapters 5 and 6.

Action	Current State	Next State
activate(LRT/component)	NOT-ACTIVATED	ACTIVATED
succeed(LRT/component)	ACTIVATED	SUCCEEDED
fail(LRT/component)	ACTIVATED	FAILED
forcefail(LRT/component)	ACTIVATED	FAILED
compensate(atomicNode)	SUCCEEDED	COMPENSATING
compensate(scopeNode/path)	{SUCCEEDED ,FAILED}	COMPENSATING
skip(atomicNode)	{NOT-ACTIVATED, FAILED,ABORTED}	SKIPPED
skip(scopeNode)	NOT-ACTIVATED	SKIPPED
compensated(component)	COMPENSATING	COMPENSATED

Table 4.3 STT of actions

4.6 Hierarchical Transactional Dependencies and Policies

In the formal definition of the semantics of COMPMOD, we make a clear distinction between the transactional behaviour of a single component and the transactional

behaviour of interrelated components. The behaviour of a single component in relation to its environment is formalised by the transactional dependencies that are defined for each component, while interrelated behaviour between LRT components is formalised by management policies.

In the following subsections, we show how this is realised in COMPMOD.

4.6.1 Dependencies

The LRT[4] and all of its components (atomic nodes, scope nodes, and execution paths) are defined with transactional dependencies that set the execution conditions under which each component may raise one of the execution events (failure, completion, force-fail, compensation or compensation completion). A component can raise an execution event depending on a single change of state of another component. A component can also raise an execution event depending on the single change of state of two or more other components.

Dependencies are defined in *encapsulated* style which is purely driven by the hierarchy structure of the WF schema. Encapsulation means that interrelated components[5] can only interact with each other through their immediate superior component but can interact directly with their superior or their immediate sibling components, e.g. a successor and a predecessor of a node on the same execution path. A component can also interact

[4] In this case, it is considered as a component.
[5] As an example: nodes running on distinct concurrent paths of the same scope.

directly with its immediate inferiors, e.g. a scope with its immediate encapsulated execution paths.

Behavioural[6] dependencies 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.

The general definition for a behavioural dependency is:

> ***Def. 4.3*** A *behavioural dependency* exists from $component_j$ *to* $component_i$ iff a state transition in $component_i$ can fire an execution event for $component_j$:
>
> $$\boldsymbol{Dep}(\boldsymbol{component}_j) := \boldsymbol{preCond}(\boldsymbol{component}_i.\boldsymbol{state})$$

Behavioural dependencies can also be defined between a set of sibling components and their immediate superior component, or between a component and its superior, essentially extending *Def. 4.3* to allow for:

1- A number of sibling components to fire an execution event for the superior component:

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

An example: a failure event fires for an execution path when all its encapsulated nodes fail.

[6] The terms behavioral, transactional and executional dependencies are used interchangeably in the thesis, depending on the context it is used in, but they refer to the same concept.

2- A component to fire an execution event for an inferior component:

Dep(component):=PreCond(componentSuperior.State)

An example: an activation event is fired for the first node in an execution path when the path is activated.

3- A component to fire an execution event for its superior component:

Dep(component):=PreCond(componentInferior.State)

An example: a failure event is fired for an execution path if a vital node on the path has failed to complete.

The way dependencies are defined imposes a hierarchical relationship between components and facilitates the hierarchical propagation of events through management policies.

4.6.2 Policies

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. ECA rules in COMPMOD are implemented to model the expected execution behavior 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:

Def.4.4 (Generic Policy 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 stimulus: a result of executing 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 (stimuli) that need to be handled. Basic to the set of management policies is a well-defined mechanism for marking the execution states of components, based on the transactional semantics of the model.

The set of state transition actions that are implemented in the policies are listed in Table 4.1. Note that actions lead to raising an event (e.g fail()), but also have a side effect on the state of the respective component, as follows:

if component.state=ACTIVATED

then component.state:=FAILED

Analogously, the event raised by an action (e.g. succeed()) may also have a side effect on the state of its immediate neighbor components. For example, *succeed(exclusivePath)* leads to *succeed(exclusivePath.superior).*

COMPMOD policies reflect the following transactional aspects:

1- business logic of the LRT (e.g. a fail policy states that if a node is vital and failed, its superior path fails);
2- semantics of a COMPMOD model (e.g. a force fail policy states that if a force-fail event is fired for an activated atomic node, the node is aborted);
3- semantics of WF patterns (e.g. a completion policy states that the successful completion of an exclusive path signals the successful completion of the scope).

Based on their transactional implications, we split policies in COMPMOD into three categories:

1- Management Policies: automate the control flow of activation and completion events;
2- Propagation Policies: automate the propagation of events through the hierarchy structure of the WF schema. Propagation of failure and force-fail events defines the Failure-Handling mechanism of the model;
3- Compensation Policies: automate control flow of compensation events.

Transactional dependencies, Policies, and Failure-Handling mechanisms are discussed in Chapters 5 and 6.

4.7 Model Assumptions

The model adopts semantic assumptions that are essentially implemented in the proposed formalism. We operate our mechanisms based on these assumptions, but provide an underpinning structure that allows them to be relaxed or extended without affecting the formalism of the model.

4.7.1 Vitality Assumptions

Assumption 1: If the successful completion of a node is necessary for the successful completion of the LRT, the node must be defined as a vital node, and must be preceded with a hierarchy of vital superiors; that is, from the node upwards to the root of the hierarchy (the main execution path).

Assumption 2: The main execution path of the LRT should be vital, and must encapsulate at least one vital node.

Assumption 3: If all paths in a scope are non-vital, their encapsulating scope should be non-vital by specification.

These assumptions are reasonable in practical business processes, and will be discussed in more detail in Chapter 4, section 4.3.5.

4.7.2 Failure Assumptions

It is assumed by the model that a failure event for an atomic node (web service) intuitively means that the node cannot successfully complete its required task. This

assumption allows for a broader meaning of failures than to restrict it to hardware or communication failures. We assume that an atomic node signals an internal failure event for the following reasons:

Assumption 1: Network or remote server failure, where the node cannot be retried.

Assumption 2: Network or remote server failure, where the node had been retried a specific number of times without success.

Assumption 3: The execution time of a node has exceeded its timeout constraints.

Assumption 4: The node gave a FALSE feedback when a TRUE feedback was expected. For example, if a flight booking task tries to book a flight for a specific date on a specific airlines but returns with no available booking, the node is considered as failed.

The failures of the nodes listed above are internal, in the sense they are out of the control of the management system. However, a node may also be forced to abort as a result of an external fail event enforced by a force-fail policy. This is the case when a running node is aborted, due to the failure of it enclosing scope.

4.7.3 Cancellation assumptions

Assumption 1: The LRT can be cancelled by the end-user by raising an external cancellation event at any time during its normal execution.

Assumption 2: A web service can be cancelled by its provider by raising an internal cancellation event and its cancellation is regarded as failure of the node.

Note that the cancellation, failure or force fail events of a component in COMPMOD all result in the component being failed; except for running atomic nodes (web services), they are aborted.

Although the model does not support external cancellation events for components, it provides the necessary infrastructure to extend the model, such that external cancellations of components by the user are possible. The force-fail policies deal with force-fail events that are externally fired, due to the propagation of a failure event from a superior. The policies can also be extended to consider external cancellation events.

4.7.4 Compensation assumptions

Our compensation mechanism is based on traversing compensable nodes on a compensating path, according to a predefined order which is specified at the design time and depends on mode of compensation being applied. This order is sequential reverse order of activation in case of partial compensations and is customized (designer defined order) in case of comprehensive compensations and we apply the following assumptions:

Assumption 1: Each node (web service) is paired with compensating actions[7].

The assumption can be relaxed by adding an attribute to denote pivot nodes (Mehrotra et al., 1992) (i.e. nodes that once they are succeeded, their effect cannot be undone). Consequently, customized compensation dependencies can only be restricted to non-pivot

[7] Similar to "compensators" of SAGA

nodes. If a pivot node is traversed in a compensating path, then it can be skipped (in partial compensation mode) or marked visited (in comprehensive compensation mode). The assumption can be further relaxed by assuming null compensators for tasks that may not be logically undone, e.g. calculating an order price.

Assumption 2: The compensation of an atomic node (web service) is guaranteed to succeed.

It is possible to relax this assumption, and consider failures of compensating actions for atomic nodes. In this case, we expect that feedback is generated to the user to take further action regarding the unperformed compensation of the node. Compensation failure policies may be added to assess such failure events and mark the node as failed. Consequently, a failed node is skipped or visited while traversing a compensating path.

Chapter 5

Management Mechanism

5.1 Introduction

In this chapter, we show how the model's operational semantics are formalized through transactional dependencies and management policies. Formal descriptions of the LRT and its components incorporate three interrelated management mechanisms in the formalism:

1- Autonomous Control Management Mechanism which is realized through activation and completion dependencies and policies.

2- Autonomous Failure Handling Mechanism which is realized through failure and force-fail dependencies and policies, and is merely automated by the propagation mechanism embedded in failure and force-fail semantics.

3- Autonomous Compensation Mechanism, which is realized through compensation dependencies and policies.

The control flow and failure handling mechanisms are defined through control charts, and are linked to dependencies and management policies. The control charts are interrelated in the flow of events and actions, and accordingly, some control flow actions appear in the charts before referencing them in the discussion, or vice versa. In addition, the flow of events and actions necessitates navigation through the control charts. Compensation management, mechanisms and their related control charts are discussed in Chapter 6.

The discussion in this chapter starts by showing how sequence and concurrency control is handled by COMPMOD semantics with respect to vitality measures. A discussion of management and failure handling mechanisms follows. As an illustration, we apply the defined dependencies in this chapter on the OP workflow case study.

5.2 Path and Scope Execution

In the following subsections, we show how the execution semantics of execution paths and scopes are captured in COMPMOD and how the consensus of their successful completion or failure –with respect to vitality measures- is resolved. It is important to note that in the formal definition of the completion conditions of the LRT and its components, the model distinguishes between completion and successful completion events for execution paths and concurrent scopes[8]. In general, a completion dependency

[8] Exclusive scopes share semantics of execution paths

fires a completion event, indicating that a component has finished its execution while a completion policy evaluates a successful completion event and subsequently marks a component as SUCCEEDED. As for atomic nodes, a completion event is concluded as successful completion.

5.2.1 Sequence Control

A sequence of nodes on a path commences its execution by the first node in the path, and ends its execution when the last node in the path ends its execution. Execution paths with single nodes are the case where the node is considered as the first node and last node simultaneously. An execution path is assessed as vital or non-vital by evaluation (Chapter 4, section 4.3.5), and hence the vitality of the path reflects the vitality of its encapsulated nodes and vice-versa. A vital path encapsulates at least one vital node while non-vital paths encapsulate non-vital nodes only. The first node of a sequence is of importance to the activation of the sequence of nodes (if any) on the same path. The last node of the path is of importance to the successful completion or failure of its superior path. We refer our discussions in this section to (Figure 5.1) where we depict three different execution paths with different combinations of vital and non-vital nodes.

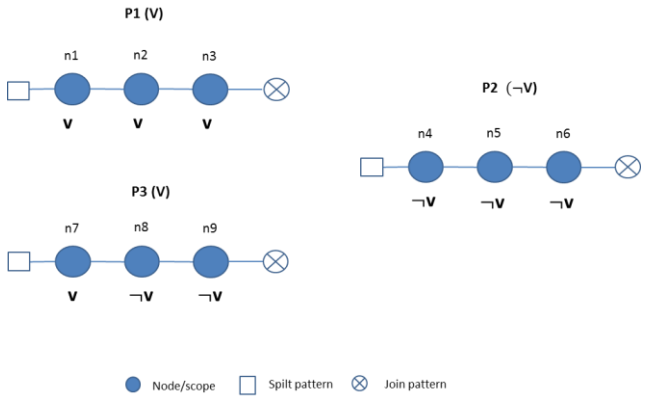


Figure 5.1 Execution path scenarios with respect to vitality

In a non-vital path, execution flows from start node to end node of a path and failures of non-vital nodes will not stop the path from completing its execution (e.g. failure of nodes in p_2). As for vital paths, execution flows from start node to end node of a path as long as no failure of a vital node is triggered, i.e. execution control can only reach the last node iff all preceding vital nodes (if any) have succeeded (e.g. execution will reach n_9 in p_3 as long as n_7 has succeeded).

Two transactional dependencies signal the end of execution of a path; they are completion *CompLDep()* and failure *FailDep()* dependencies.

The semantics of completion and failure of execution paths are defined as follows:

1- A path completes ***CompLDep(path)=TRUE*** when the last node in the path ends its execution either by SUCCEED or FAIL (e.g. execution of p_1 completes when execution of n_3 completes).

2- A non-vital path fails ***FailDep(path)=TRUE*** if all nodes on the path FAIL (e.g. *FailDep*(p_2)=*TRUE* if n_4, n_5, and n_6 all fail).

3- A non-vital path succeeds if it completes ***CompLDep(path)=TRUE*** and no failure event is fired for the path ***FailDep(path)=FALSE*** (e.g. p_2 succeeds if *CompLDep(p_2)=TRUE* and *FailDep(p_2)=FALSE*).

4- It is a failure propagation policy of the model that failure of a vital node will fail its enclosing path.

Def. 5.1: (propagation of vital-node failure)
$$\boldsymbol{node.IsVital} \wedge \boldsymbol{fail}(\boldsymbol{node}) \longrightarrow \boldsymbol{superiorPath.state} = \boldsymbol{FAILED}$$

For example, if n_2 in p_1 fails, then p_1 fails.

The successful completion of vital and non-vital paths is assessed after a completion event of the path is fired, and depends on the final execution state of the last node on the path. When the last node ends its execution either by success or failure, a completion event is fired for its superior path:

Def. 5.2: (completion of path)
$$\boldsymbol{succeed}(\boldsymbol{lastNode}) \vee \boldsymbol{fail}(\boldsymbol{lastNode}) \longrightarrow \boldsymbol{CompLDep}(\boldsymbol{superiorPath}) = \boldsymbol{TRUE}$$

For example, $succeed(p_1.n_1) \vee fail(p_1.n_1) \longrightarrow CompLDep(p_1) = TRUE$

Successful completion of the last node in a path imposes the following semantics:

1- In case of a non-vital path, successful completion of the last node in the path indicates that a failure event could not possibly fire for its superior path since a failure event fires only when all nodes on a path fail to succeed, thus:

Def. 5.3:
$$\boldsymbol{succeed}(\boldsymbol{lastNode}) \wedge \neg \boldsymbol{superioPath.IsVital} \longrightarrow \boldsymbol{FailDep}(\boldsymbol{superiorPath}) = \boldsymbol{FALSE}$$

For example, if both n_4 and n_5 in p_2 fail but n_6 succeeds, then $FailDep(p_2)=FALSE$.

2- In case of a vital path, if the execution control reaches the last node on a vital path, this means that the path has not failed due to failure of a preceding vital node on the path (from ***Def.5.1***), and thus:

Def. 5.4:

$$\boldsymbol{succeed(lastNode) \wedge superiorPath.IsVital \longrightarrow superiorPath.state \neq FAILED}$$

For example, if execution reaches n_9 in p_3, it is not possible that n_7 has failed and has consequently caused the failure p_3 by propagation rule.

Therefore, we can conclude that (a) from ***Def.5.2***, if the successful completion of a node triggers a completion event of its superior path then the node is the last in the path, and (b) from ***Def.5.3*** and ***Def. 5.4***, if the last node in a path succeeds then the path cannot possibly have failed. Hence, the successful completion of a node that triggers a completion event for the path (i.e. it is the last node in path), and also triggers the successful completion of its enclosing path:

Def. 5.5: (successful completion of a vital and non-vital path)$_1$

$$\boldsymbol{succeed(node) \wedge CompLDep(superiorPath) = TRUE \longrightarrow succeed(superiorPath)}$$

Failure of last node in a vital path can lead to either a successful completion or a failure of its superior path depending on the vitality of the node as follows:

1- Vital node: from ***Def.5.1*** we conclude that the failure of vital last node will fail its superior (e.g. failure on n_3 in p_1).

2- Non-vital node: failure of a non-vital last node on a vital path will trigger a completion event of its superior path (from ***Def. 5.2***), and implicitly indicates that no vital node has failed on the path (from ***Def.5.1***). Hence, we conclude that a failure of the last non-vital node on a vital path will succeed its superior path (e.g. failure of n_9 on p_3 will still result in succeeding p_3).

Def. 5.6: (successful completion of a vital path)$_2$

$$\neg\, \boldsymbol{node.IsVital} \wedge \boldsymbol{fail(node)} \wedge \boldsymbol{superiorPath.IsVital} \wedge \boldsymbol{CompLDep(SuperiorPath) = TRUE} \longrightarrow \boldsymbol{succeed(superiorPath)}$$

Failure of last node in a non-vital path can lead to either a successful completion or a failure of its superior path, depending on the failure dependency evaluation of the path as follows:

1- If all the nodes on the path fail, then the path fails. Hence:

Def. 5.6: (failure of a non-vital node)

$$\boldsymbol{FailDep(path) = TRUE} \wedge \neg \boldsymbol{path.IsVital} \longrightarrow \boldsymbol{fail(path)}$$

For example, when n_6 in p_2 fails where n_4 and n_5 has failed as well, then p_2 fails.

2- If no failure event is fired for the path then the path succeeds. Hence:

Def. 5.7: (successful completion of a vital path)$_3$

$$\boldsymbol{Fail(node)} \wedge \neg \boldsymbol{path.IsVital} \wedge \boldsymbol{CompLDep(superiorPath) = TRUE} \wedge \boldsymbol{FailDep(superiorPath) = FALSE} \longrightarrow \boldsymbol{succeed(superiorPath)}$$

For example, when n_6 in p_2 fails but $FailDep(p_2)=FALSE$, then p_2 succeeds because some nodes in p_2 have succeeded.

As a result, failure events raised by failure dependencies are only obvious for non-vital paths and hence are only defined for non-vital paths. However, completion dependencies are defined for both, vital and non-vital paths.

Exclusive scopes in COMPMOD are a broader form of sequencing where only one out of two or more exclusive paths is required to succeed. The successful completion of an exclusive scope is triggered by the successful completion of one of its exclusive paths, and therefore exclusive scopes are not defined with completion dependencies.

Def. 5.8: (successful completion of exclusive scope)

$$\boldsymbol{succeed}(\boldsymbol{path}) \wedge \boldsymbol{path.IsExclusive} \longrightarrow \boldsymbol{succeed}(\boldsymbol{superiorScope})$$

The failure of an exclusive scope is triggered by the failure of its last execution path which intuitively indicates that all paths within the scope have failed. Note that the last exclusive path in the scope has no alternative.

Def. 5.9: (failure of exclusive scope)

$$\boldsymbol{fail}(\boldsymbol{path}) \wedge \boldsymbol{path.IsExclusive} \wedge \neg \boldsymbol{path.hasAlternative} \longrightarrow \boldsymbol{fail}(\boldsymbol{superiorScope})$$

5.2.2 Concurrency Control

Unlike execution paths, the vitality of scope nodes in COMPMOD is assigned by specification. In section 4.3.5, we classified concurrent scopes into three cases with respect to vitality of the scope vs. the vitality of its encapsulated paths. The vitality of a scope does not reflect the vitality of its encapsulated components and vice-versa; hence, we define the operational semantics of concurrent scopes with respect to the vitality of their concurrent paths and irrespectively of the vitality of the enclosing scope.

A concurrent scope encapsulates execution paths that are executed concurrently, and their execution is synchronized at the join end of the scope. Activation dependencies formally define the split point of the scope whereas the synchronizer of the scope is formally defined through completion and failure dependencies.

A concurrent scope starts its execution at the split point of the scope by activating all its concurrent paths[9]. All paths are executed concurrently and each path follows the sequence semantics discussed in Section 5.2.1 regarding their successful completions and failure semantics.

A completion dependency ***ComplDep()*** fires a completion event when all concurrent paths have finished their executions either by failing or succeeding. A failure dependency ***FailDep()*** fires a failure event when all concurrent paths fail to complete (in the case of OR scope this means all concurrently enabled paths). Before we introduce our failure and

[9] Enabled parallel paths in case of OR-scopes

successful completion semantics, two important issues regarding concurrent semantics are discussed, namely, deadlocks and propagation policy.

Deadlocks: in the context of concurrent executions, there are some conditions under which the behavior of concurrent paths could lead to deadlock situations because the behavior of the synchronizer becomes undefined. We list these conditions and show how they are prevented thorough our proposed operational semantics. A synchronizer will deadlock if:

1- In a concurrent (AND/OR) scope, one or more than one of the activated concurrent paths do not respond with failure or completion events caused by a latency in response from a node on the path. However, failure assumption 3 (Chapter 4, section 4.7.2) states that a failure is triggered for a node when the execution time of a node exceeds its timeout constraints. Therefore a path is always guaranteed to finish its execution and this type of deadlock is relieved.

2- In an OR scope, a scope activates but none of the conditions associated with its encapsulated paths evaluates to TRUE. In this case, the behavior of the synchronizer will deadlock. Therefore, we assume that an activated OR scope with no enabled paths is a failed scope and define its failure dependency accordingly.

Propagation policy: it is a propagation policy of COMPMOD that failure of a vital concurrent path will fail its superior scope:

Def. 5.10: (propagation of vital path failure)
$$path.IsVital \wedge path.Isconcurrent \wedge fail(path) \longrightarrow superiorScope.state = FAILED$$

In order to define successful completion and failure criteria for concurrent scopes, we must consider the three possible amalgamations of encapsulated paths (see Figure 5.2 for illustrations):

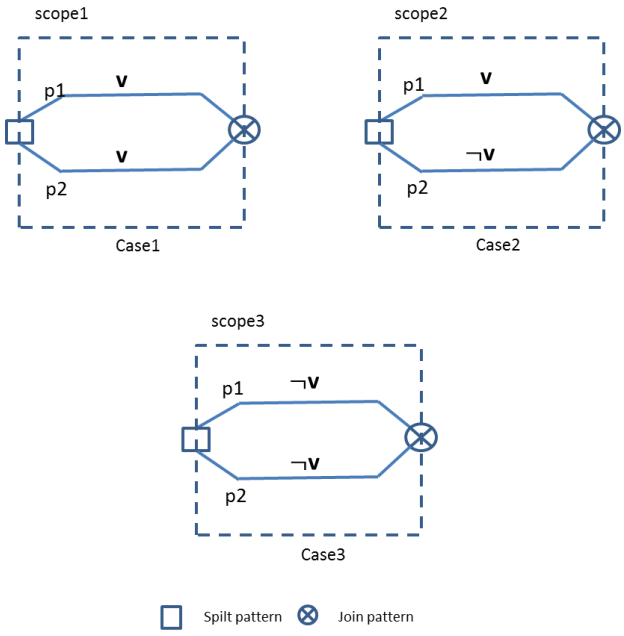


Figure 5.2 Concurrent scope cases with respect to vitality of encapsulated paths

Case 1: The scope encapsulates only vital paths. In this case, the scope successfully completes if (a) the scope completes its execution (i.e. ***ComplDep(scope)=TRUE***) and (b) all paths succeed. From ***Def. 5.1***, we conclude that the successful completion of all concurrent paths implies that ***scope.state≠FAILED***. For illustration, $scope_1$ succeeds only if both p_1 and p_2 complete their executions (i.e. $CompLDep(scope_1)=TRUE$) and they

both succeed. Since p_1 and p_2 are both vital, the possibility of $scope_1$ failing by propagation is not the case, hence it is guaranteed that $scope_1.state \neq FAILED$.

Case 2: The scope encapsulates at least one vital path and at least one non-vital path. In this case, the scope succeeds if it completes (i.e. ***CompLDep(scope)=TRUE***) and all vital paths complete (i.e. ***scope.state≠FAILED***) and all non-vital paths complete either by FAIL or SUCCEED (i.e. ***CompLDep(scope)=TRUE***). As an example, in $scope_2$, when p_1 and p_2 complete their execution, then $CompLDep(scope_2)=TRUE$. If p_1 succeeds then $scope_2$ cannot possibly fail by propagation (i.e. $scope_2.state \neq FAILED$). The final execution state of p_2 (since it is non-vital) does not affect the succeeding of $scope_2$.

Hence:

Def. 5.11: (successful completion of synchronized concurrent paths, cases 1,2)
$$\boldsymbol{CompLDep}(\boldsymbol{scope}) = \boldsymbol{TRUE} \wedge \boldsymbol{scope}.\boldsymbol{state} \neq \boldsymbol{FAILED} \rightarrow \boldsymbol{succeed}(\boldsymbol{scope})$$

Note that in case 1 and case 2, the successful completion of the scope ensures the successful completion of its vital paths and thus the following implication holds:

$$\boldsymbol{succeed}(\boldsymbol{scope}) \rightarrow \boldsymbol{FailDep}(\boldsymbol{scope}) = \boldsymbol{FALSE}$$

Case 3: The scope encapsulates only non-vital paths. In this case, the scope succeeds if it is completed (i.e. ***CompLDep(scope)=TRUE***) and no failure event is fired for the scope and (i.e. ***FailDep(scope)=FALSE***). For example, $scope_3$ succeeds when both p_1 and p_2 complete (i.e. $CompLDep(scope_3)=TRUE$) but at least p_1 or p_2 succeeds (i.e. $FailDep(scope_3)=FALSE$).

Hence:

Def. 5.12: (successful completion of concurrent paths, case3)
$$\boldsymbol{FailDep(scope) = FALSE \ \wedge CompLDep(scope) = TRUE \longrightarrow succeed(scope)}$$

Note that in case 3, inexistence of vital paths within the scope ensures that the scope could not fail through the propagation of a failure event, and thus s$\boldsymbol{cope.state \neq FAILED}$ holds.

Therefore, when a completion event is fired for a concurrent scope, if the scope has not been failed by a propagation rule (explicitly covers case 1, 2, and implicitly 3) and there is no failure event fired for the scope (explicitly covers case 3, and implicitly 1 and 2), then the scope succeeds. Hence:

Def. 5.13: (successful completion of concurrent scope)
$$\boldsymbol{ComplDep(scope) = TRUE \ \wedge FailDep(scope) FALSE \wedge scope.state \neq FAILED \longrightarrow succeed(scope)}$$

Failure of a concurrent scope is triggered in two distinct ways:

1- Prompted by the assessment of the propagation policy (***Def. 5.10***): cases 1,2. For example, failure of p_1 in $scope_1$ will fail $scope_1$ by propagation.

2- Failure event raised by a failure dependency (i.e. ***FailDep(scope)=TRUE)***: case3.

It could be the case that a scope of type case 2 encapsulates one vital path and one or more non-vital paths where all non-vital paths fail, and the vital-path fails but was the last path to synchronize, in this case two failure events are triggered for the scope, one by propagation and one by satisfaction of the failure dependency of the scope. Therefore,

when a failure event is fired for a concurrent scope, and if the state of the scope is not failed by a propagation of failure event, the scope is failed. Hence:

Def. 5.13: (failure of a concurrent scope)
$$\boldsymbol{FailDep(scope) = TRUE \wedge scope.state \neq FAILED \longrightarrow fail(scope)}$$

To illustrate Def. 5.13 we assume the following completion scenarios for p_1 and p_2 in $scope_2$. We assume that *p2* fails first. p_2's failure will not affect the state of $scope_2$ since p_2 is not vital. We assume that the vital path p_1 fails next. Two failure actions will take place as a consequence of p_1's failure. The state of $scope_2$ will change to FAILED by the propagation rule. At the same time *FailDep(*$scope_2$*)* will signal *TRUE* since all encapsulated paths of $scope_2$ have failed. Hence, the general definition of failure semantics *Def. 5.13*, checks first that scope state is not already *FAILED* to avoid failing the scope twice.

Note that a similar argument based on enabled paths concurrent paths can be made for completion, successful completion, and failure semantics.

5.3 Control Management Mechanism

The control flow of the model is automated through a series of activation and completion events/actions as depicted in Control Charts 1 through 7. We discuss the activation and completion semantics in the following sub sections and relate to the relevant dependencies and policies.

5.3.1 Activation Semantics

Activation dependencies are defined for execution paths and nodes (Table 5.1). A component activates according to the following hierarchical control structure:

1- Activation of LRT triggers an activation event for the main execution path (ActD.1).

2- Activation of a path triggers an activation event for the first node in the path (ActD.2).

3- Successful Completion[10] of a node triggers an activation event for the succeeding node (if any) (ActD.3).

4- Failure[11] of a non-vital node triggers an activation event for the succeeding node (if any) (ActD.3).

5- Activation of an AND-scope triggers activation events for all concurrent paths within the scope (ActD.4).

6- Activation of an OR-scope triggers activation events for all enabled concurrent paths within the scope (ActD.5).

7- Activation of an XOR-scope triggers an activation event for the first path in the scope (ActD.4).

8- Compensation completion[12] of an exclusive path, that has an alternative, triggers the activation of the next exclusive path (ActD.6).

[10] Successful completion dependencies and policies are discussed in section 5.3.2.
[11] Failure semantics are discussed in section 5.4.1
[12] A compensation completion event signals that the compensation of a path has completed (Chapter 6, section 6.2.2.2)

In order to automate activation events, we apply activation policies, listed in Table 4.2. Activation Policies assess activation events and mark the execution state for a component as ACTIVATED. ActR.3 ensures that components are only activated if their superior is activated. ActR.1 is triggered by a system generated internal activation event of the LRT and ActR.2 activates the main execution path.

Dep #	Dependency	Component
ActD.1	$\boldsymbol{ActDep(p_0) := LRT.state = ACTIVATED}$	Main path p_0
ActD.2	$\boldsymbol{ActDep(firstNode) := path.State = ACTIVATED}$	First node in a path
ActD.3	$\boldsymbol{ActDep(node_2) :=}$ $\boldsymbol{(node_1.State = SUCCEEDED) \vee (\neg\, node_1.IsVital \wedge node_1.State = FAILED)}$	Sequential Nodes : $node_2$ successor of $node_1$
ActD.4	$\boldsymbol{ActDep(path) := superior.State = ACTIVATED}$	1- Concurrent paths of AND scope 2- First path in XOR scope
ActD.5	$\boldsymbol{ActDep(path) := superior.State = ACTIVATED \wedge path.IsEnabled}$	Concurrent paths of OR scope
ActD.6	$\boldsymbol{ActDep(path_i) :=}$ $\boldsymbol{path_{i-1}.State = COMPENSATED}$	Paths 2..m in XOR scope where m>=2

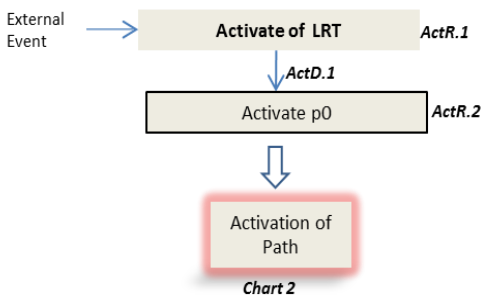
Table 5.1 Activation Dependencies

Rule#	Policy	Component
ActR.1	***ON*** *"activation event of LRT"* ***IF*** *LRT.state=NOT-ACTIVATED* ***DO*** *activate(LRT)*	LRT
ActR.2	***ON*** *ActDep(p_0)* ***DO*** *activate(p_0)*	Main execution path P_0
ActR.3	***ON*** *ActDep(component)* ***IF*** *component.superior.state=ACTIVATED and component≠p_0* ***DO*** *activate(component)*	atomic node, scope, and path≠p_0

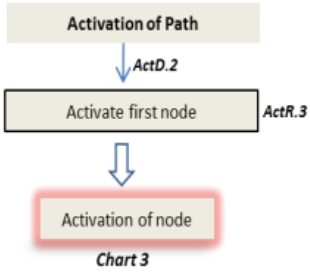
Table 5.2 Activation Policies

The activation mechanism is depicted in Control Charts 1-3. Note that activation of a succeeding node in a sequence depends on the completion state of its predecessor node as

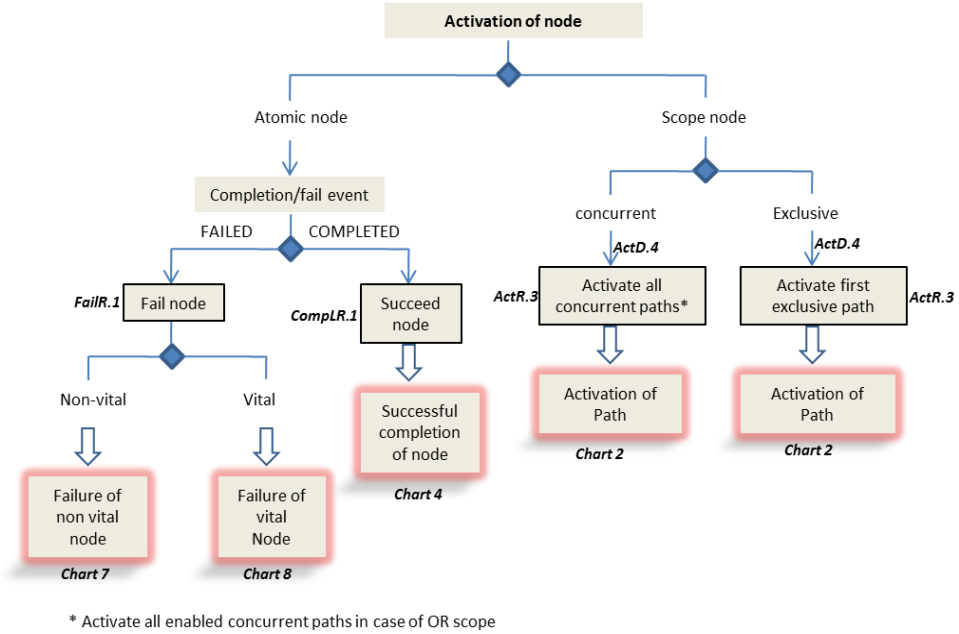
is illustrated in charts 4 and 7 (successful completion of node and failure of non- vital node respectively).



Control Chart 1. Activation of LRT



Control Chart 2. Activation of Path



Control Chart 3. Activation of Node

5.3.2 Completion Semantics

In this section, we relate our discussion to the completion dependencies and policies listed in (Tables 5.3 and 5.4) and control charts (4-7).

Atomic nodes and exclusive scopes are not defined with completion dependencies; instead, their successful completion event is assessed by completion policies. Completion dependencies are explicitly defined for **execution paths** and **concurrent scopes**. A completion event signals that the path/scope has ended its execution. Subsequently, a completion policy concludes if the path/scope has successfully completed and hence marks the path/scope as SUCCEEDED.

Dep #	Dependency	Component
CompLD.1	$\boldsymbol{CompLDep}(\boldsymbol{path_i}) \coloneqq$ $\boldsymbol{lastNode.State = SUCCEEDED} \vee \boldsymbol{lastNode.State = FAILED}$)	Path
CompLD.2	$\boldsymbol{CompLDep}(\boldsymbol{scope}) \coloneqq$ $\bigwedge_{1 \le i \le m} (\boldsymbol{path_i.State = SUCCEEDED} \vee \boldsymbol{path_i.State = FAILED})$	AND scope with $\boldsymbol{m}$ concurrent paths
CompLD.3	$\boldsymbol{CompLDep}(\boldsymbol{scope}) \coloneqq$ $\bigwedge_{1 \le i \le m} (\boldsymbol{path_i.IsEnabled} \wedge (\boldsymbol{path_i.State = SUCCEEDED} \vee \boldsymbol{path_i.State = FAILED}))$	OR scope with $\boldsymbol{m}$ concurrent paths

Table 5.3 Completion Dependencies

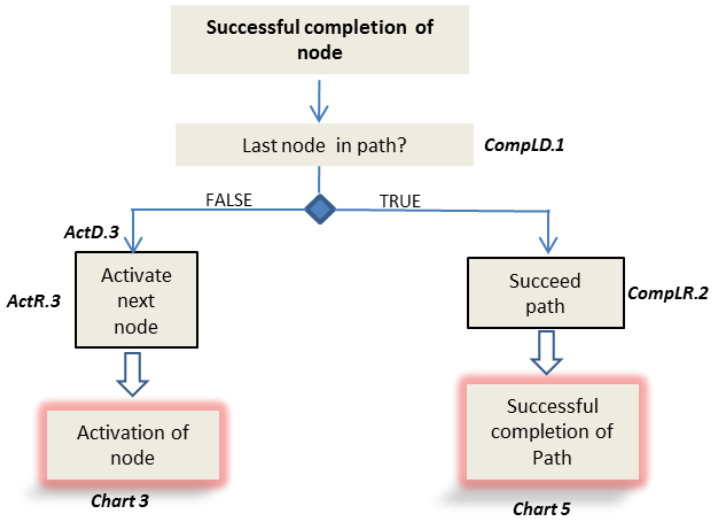
Rule#	Policy	Component
CompLR.1	***ON*** *"successful completion event of atomic node"* ***DO*** *succeed(node)*	Atomic node
CompLR.2	***ON*** *succeed(node)* ***IF*** *CompLDep(node.superior)=TRUE* ***DO*** *succeed(node.superior)*	Vital and non-vital path with succeeded last node
CompLR.3	***ON*** *succeed(path)* ***IF*** *path=p_0* ***DO*** *succeed(LRT)*	LRT
CompLR.4	***ON*** *succeed(path)* ***IF*** *path.IsExclusive=TRUE* ***DO*** *succeed(path.superior)*	Exclusive scope
CompLR.5	***ON*** *CompLDep(scope)* ***IF*** *scope.state≠failed and FailDep(scope)=FALSE* ***DO*** *succeed(scope)*	Concurrent scope
CompLR.6	***ON*** *fail(node)* ***IF*** *¬node.IsVital and node.superior.IsVital and CompLDep(node.superior)=TRUE* ***DO*** *succeed(node.superior)*	vital path with failed non-vital last node
CompLR.7	***ON*** *fail(node)* ***IF*** *¬node.IsVital and ¬node.superior.IsVital and CompLDep(node.superior)=TRUE and FailDep(node.superior)=FALSE* ***DO*** *succeed(node.superior)*	non-vital path with failed non-vital last node

Table 5.4 Completion Policies

The completion and successful completion events of a component have a transactional impact on its interconnected components and are modelled as follows:

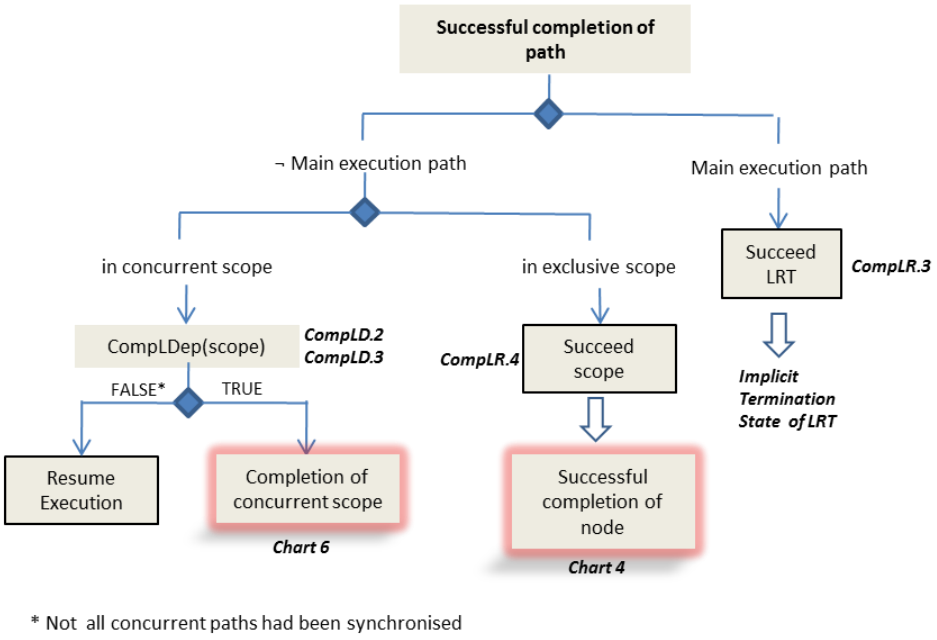
1. When an atomic node is activated, the system awaits for an internal event indicating its failure or completion (chart 3). An internal completion event for an **atomic node** signals the successful completion of the node. The completion event is assessed by the completion policy (CompLR.1), and the node is marked as SUCCEEDED.
2. Successful completion of a **node** (chart 4) has an impact on (a) the activation of a succeeding node on the same path if any, or (b) on the completion of its superior path if the node was the last node on the path.

3. Successful completion of a **last node** (chart 4) in a path has an impact on the completion of its superior path. When the last node finishes its execution, a completion event is fired for the path (CompLD.1) and the path is marked succeeded by policy (CompLR.2).



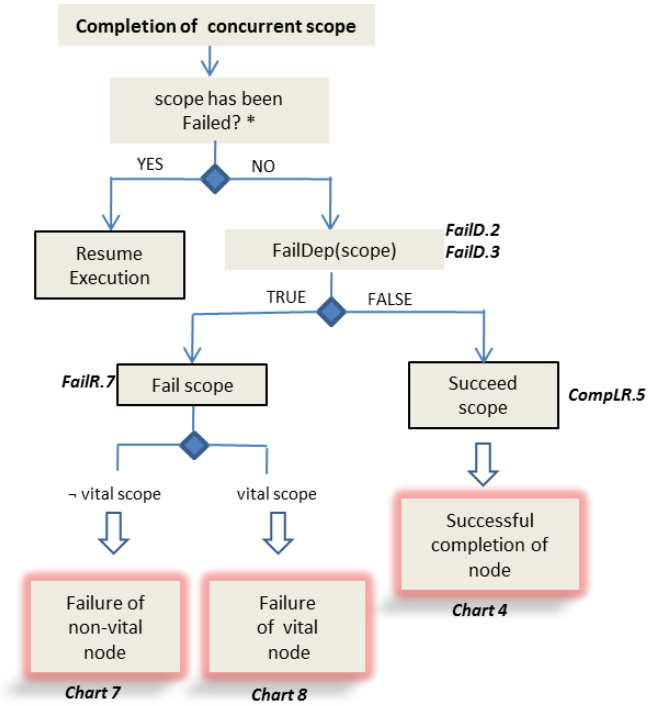
Control Chart 4. Successful Completion of Node

4. Successful completion of the **main execution path** (chart 5) triggers the successful completion of the LRT (CompLR.3).
5. An **exclusive scope** succeeds (chart 5) when one of its exclusive paths succeed (CompLR.4).
6. Concurrent paths are synchronised (chart 5) through a completion event that is fired for an AND-scope (ComplD.2) or OR-scope (CompLD.3).



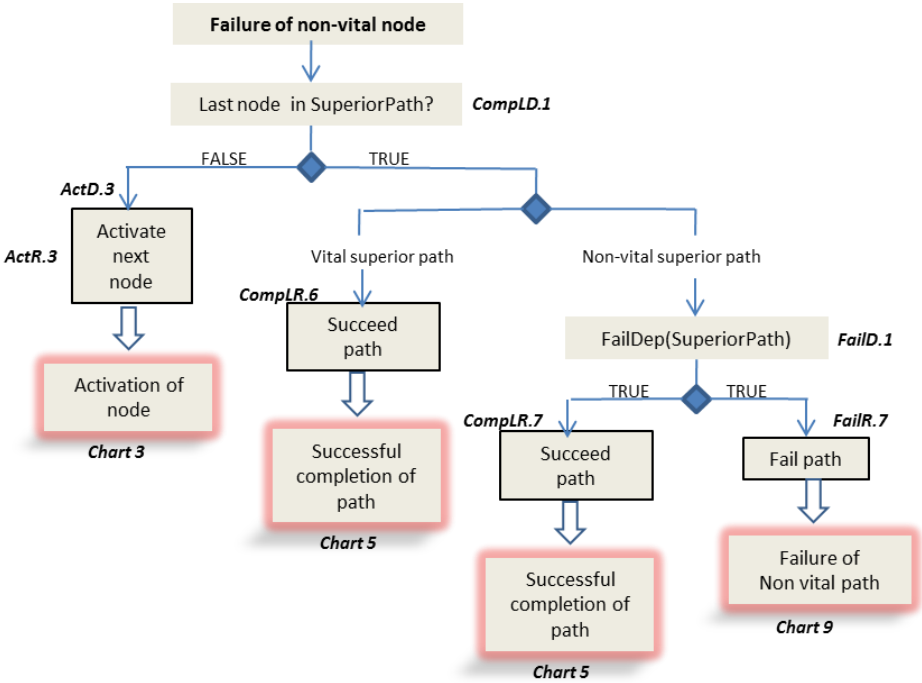
Control Chart 5. Successful Completion of Path

7. A **concurrent scope** successfully completes (chart 6) when a completion event has been fired for the scope and no failure event is raised for the scope (CompLR.5).
8. The failure of a **non-vital last node** (Chart 7) leads to successful completion of its superior path in two cases: (1) if it is encapsulated by a vital path (CompLR.6), or (2) if it is encapsulated by a non-vital path and no failure event was fired for the path (CompLR.7).



- The scope could have been failed due to bottom-up propagation of vital failure event within the scope

Control Chart 6. Completion of a Concurrent Scope



Control Chart 7. Failure of non-Vital Node

5.4 Failure-Handling Mechanism

COMPMOD applies a recursive method for propagating vital failure events through the recursive hierarchical structure of LRT components. Propagation is applied in parallel with policy-based actions in order to reach a consensus about the execution state of LRT components and the LRT itself.

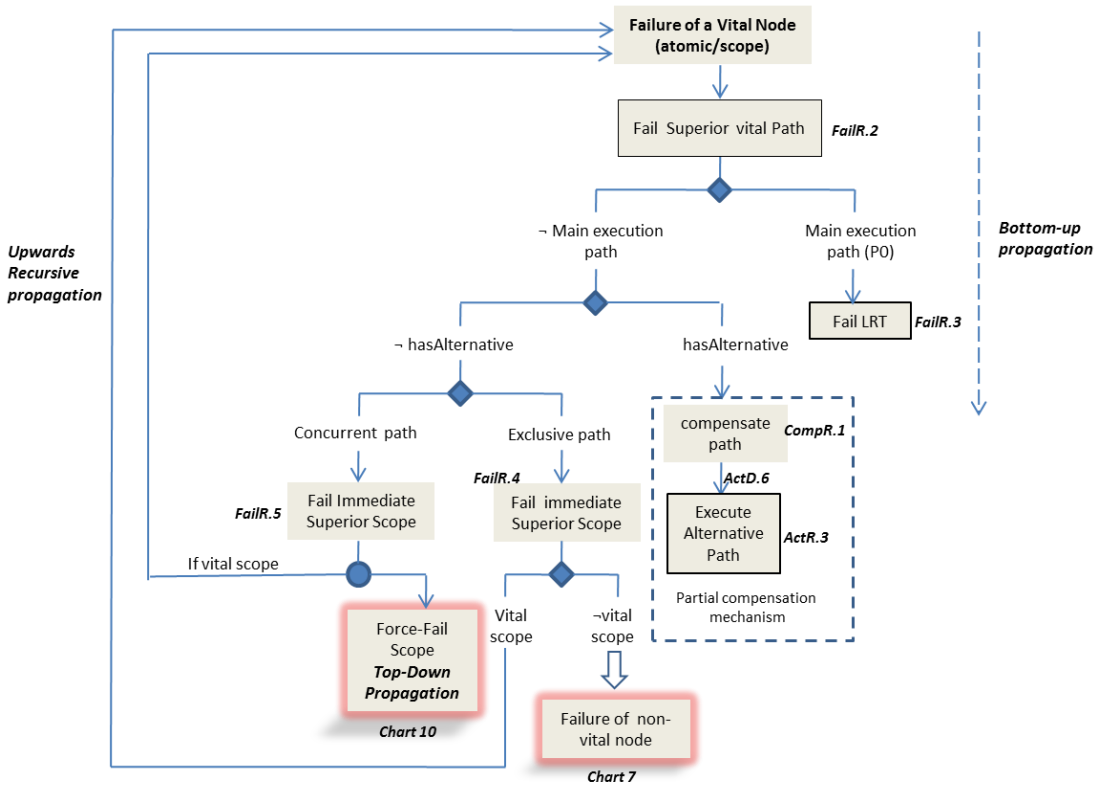
Within the context of the proposed hierarchical structure, the recursive failure propagation mechanism entails a combination of three types of propagation method:

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

4. 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 and force-fail control charts (7-11) illustrate the failure and force-fail mechanism linked to failure and dependencies and policies (Tables 4.5-4.8). In particular, charts 8 and 10 illustrate the main propagation mechanism implemented by COMPMOD. Control charts include compensation mechanisms that are discussed and illustrated in chapter 6.

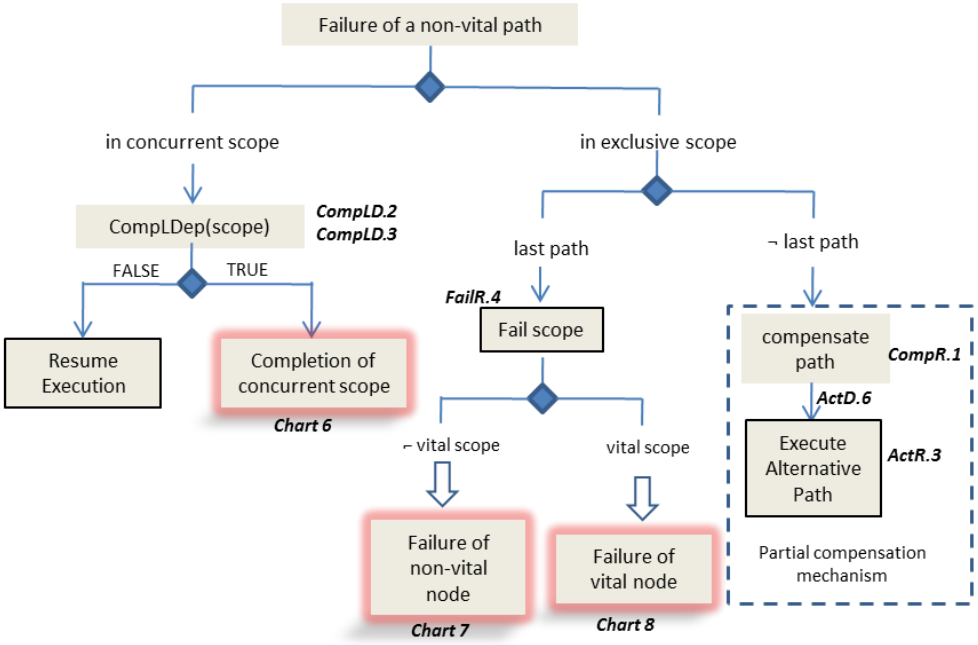
Failure propagation is always initiated by the failure of a vital (chart 8) atomic node, and propagates recursively through vital component ancestors in the hierarchy structure to stop when a non-vital ancestor component is reached or when the root of the hierarchy is reached. As for Top-down propagation 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.



Control Chart 8. Failure of a Vital Node

The failure mechanism also handles failures of non-vital components. Failure of a non-vital atomic node (chart 7, section 5.3.2) could fail its enclosing path if the enclosing path was a non-vital path, and the node is the last node in the path, and all other nodes in the path (if any[13]) have failed. Failure of a non-vital path (Chart 9) 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.

[13] If the path is atomic (i.e. contains one node), then the node is considered as the first node and last node.



Control Chart 9. Failure of non-vital Path

In the following subsections, the semantics of failure and force fail semantics are discussed.

5.4.1 Failure Semantics

Failure semantics are formalized through failure dependencies (Table 5.6) and failure policies (Table 5.7) and they are conceded as follows:

1- Failure events for atomic nodes are fired internally by the system. When a failure event is fired for an atomic node, the event is assessed by failure policy (FailR.1) and the node is marked as FAILED.

2- Failure of a vital node triggers the propagation of the failure event to its superior path and subsequently the superior path fails (FailR.2)

3- A failure event is fired for a non-vital path iff all its encapsulated nodes fail (FailD.1), the failure event is assessed by (FailR.6) and the state of the path is marked FAILED.

4- Failure of the main execution path triggers the failure of the LRT (FailR.3).

5- Failure of an exclusive path that has alternative initiates a compensation process for the path. This event is assessed by the compensation policy (CompR.1) and is explained in (Chapter 6, section 6.2.3).

6- Failure of an exclusive path that has no alternative triggers the failure of its encapsulation scope (FailR.4).

7- Failure of a vital concurrent path propagates the failure event to its enclosing scope (FailR.5).

8- A failure event is fired for a concurrent AND scope (FailD.2) if all its enclosing concurrent paths fail to complete.

9- A failure event is fired for a concurrent OR scope (FailD.3) if all its enclosing enabled concurrent paths fail to complete or the scope has been activated but none of its enclosing paths has been enabled.

10- When a failure event is fired for a concurrent scope, if the scope has not been failed as a result of bottom-up failure propagation from within the scope, the scope is marked failed (FailR.7).

Dep #	Dependency	Component
FailD.1	$\boldsymbol{FailDep(path)} := \left(\bigwedge_{1 \le i \le m} \boldsymbol{nodeList_i.nodeState = FAILED} \right)$	Non-Vital path
FailD.2	$\boldsymbol{FailDep(ANDscope)} := \bigwedge_{1 \le i \le m} (\boldsymbol{path_i.State = FAILED})$	AND scope with m paths
FailD.3	$\boldsymbol{FailDep(ORscope)} := (\bigwedge_{1 \le i \le m} (\boldsymbol{path_i.IsEnabled} \wedge \boldsymbol{path_i.State = FAILED})) \vee (\boldsymbol{ORscope.state = ACTIVATED} \wedge (\bigwedge_{1 \le i \le m} \boldsymbol{path_i.IsEnabled = FALSE}))$	OR scope with m paths

Table 5.6 Failure Dependencies

Rule#	Policy	Component
FailR.1	***ON*** *"failure/cancellation event for atomic node"* ***DO*** *fail(node)*	Atomic node
FailR.2	***ON*** *fail(node)* ***IF*** *node.superior.state=ACTIVATED and node.IsVital* ***DO*** *fail(node.superior)*	vital path (bottom-up propagation)
FailR.3	***ON*** *fail(path)* ***IF*** *path=p_0* ***DO*** *fail(LRT)*	LRT
FailR.4	***ON*** *fail(path)* ***IF*** *¬path.HasAlternative and path.IsExclusive* ***DO*** *fail(path.superior)*	Exclusive scope
FailR.5	***ON*** *fail(path)* ***IF*** *path.IsVital=TRUE and Path.IsConcurrent* ***DO*** *fail(path.superior)*	concurrent scope (bottom-up propagation)
FailR.6	***ON*** *FailDep(path)* ***DO*** *fail(path)*	non-vital path
FailR.7	***ON*** *FailDep(scope)* ***IF*** *scope.state≠failed* ***DO*** *fail(scope)*	Concurrent scope

Table 5.7 Failure Policies

5.4.2 Force-Fail semantics

Force-fail is a counterpart for cancellation. As illustrated in chart 8, when a vital concurrent path fails, its immediate outer scope fails. Force-fail dependencies and policies 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 (Table 5.8) are defined between components and their immediate superiors, such that failure of an activated component's superior will force the component to fail.

Dep #	Dependency	Component
FFailD.1	$\boldsymbol{ForceFailDep(node) := node.state = ACTIVATED \wedge path.state = FAILED}$ Where path=node.superior	Atomic node/scope
FFailD.2	$\boldsymbol{ForceFailDep(path) := path.state = ACTIVATED \wedge scope.state = FAILED}$ Where scope=path.superior	$path \neq p_0$
FFailD.3	$\boldsymbol{ForceFailDep(p_0) := LRT.state = FAILED}$	Main path p_0

Table 5.8 Force-fail Dependencies

Force-fail policies (Table 5.9) automate the propagation of Force-fail events (chart 10) in a downwards recursive fashion through the hierarchy structure of the WF.

Rule#	Policy	Component
FFailR.1	***ON*** *FFailDep(node)* ***IF*** *node.type=ATOMIC* ***DO*** *abort(node)*	Atomic node-Propagation
FFailR.2	***ON*** *FFailDep(node)* ***IF*** *node.type=SCOPE* ***DO*** *fail(node)*	scope -Propagation
FFailR.3	***ON*** *FFailDep(path)* ***DO*** *fail(path)*	Path- Propagation
FFailR.4	***ON*** *"cancellation event of LRT"* ***IF*** *LRT.State=ACTIVATED* ***DO*** *fail(LRT)*	LRT

Table 5.9 Force-fail Policies

Force-fail propagation originates from a failure of concurrent scope, and triggers the failure of all its encapsulated active components in the following recursive mechanism:

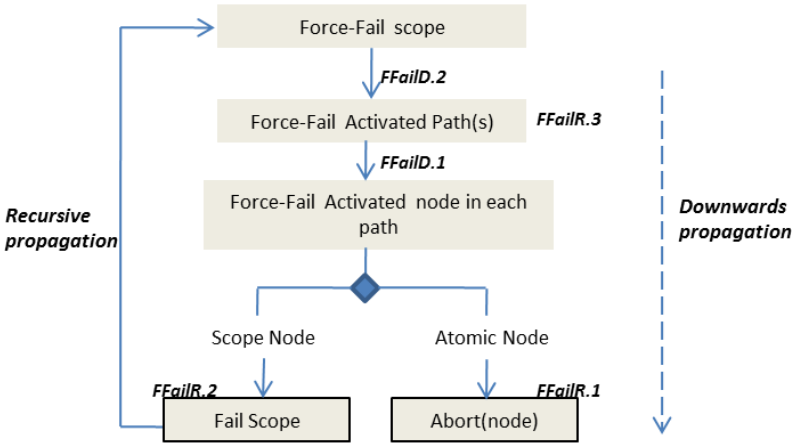
1- A force-fail event is fired for all activated paths within a scope if the superior scope has failed (FFailD.2).

2- A force-fail event of a path is assessed by (FFailR.3) policy and the path is marked FAILED.

3- A force-fail event is fired for an activated node (atomic/scope) if its superior path has failed (FFailD.1).

4- A fired Force-fail event of an atomic node is assessed by (FFail.R.1) policy and the node is aborted.

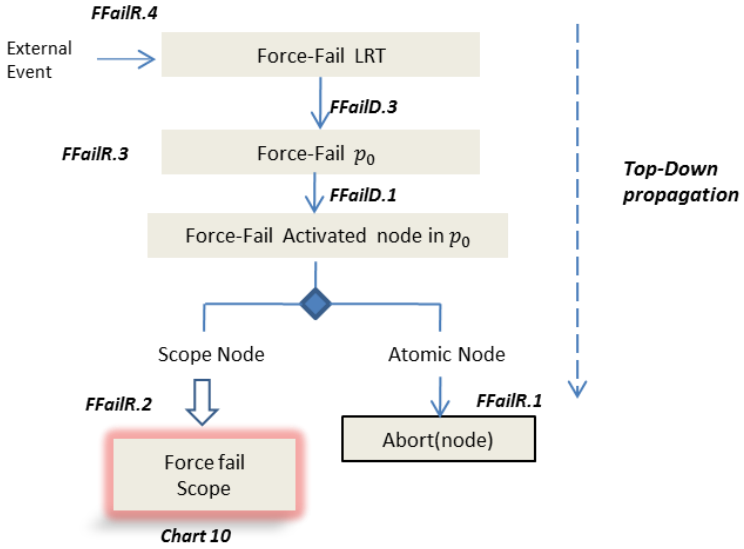
5- A fired Force-fail event of a scope node is assessed by (FFailR.2) policy and scope is failed.



Control Chart 10. Force-Fail Scope

Cancellation of the LRT by the end-user is supported by COMPMOD (Chart 11). When a cancellation event is fired for the LRT, the event is assessed by the (FFailR.4) policy, and the LRT is marked as failed. Failing the LRT triggers a force-fail event for the main execution path (FFailD.3) which consequently leads to a force-fail event fired for the

activated node on $p0$ at the time the cancellation of LRT occurred. Following the force-fail mechanism mentioned above, if the activated node is atomic, the node is aborted. If the activated node is a scope, the activated components within the scope are force-failed in a downwards recursive propagation fashion, following the mechanism illustrated in chart 10.



Control Chart 11. Force-Fail LRT

5.5 Examples

5.5.1 Control Flow Dependencies of OP Case Study

In section 4.4, we depicted the workflow representation and the transitional attributes of our OP case study (Figure 4.5). In (Table 5.10), we define the control flow dependencies for OP with respect to workflow semantics and transactional dependencies of COMPMOD which have been discussed in the previous sections.

	ActDep	ComLDep	FailDep	ForceFailDep
P_0	*OP.state= ACTIVATED*	*$scope_1$.State=SUCCEEDED ∨ $scope_1$.State=FAILED*	*By propagation*	*OP.state=FAILED*
SALES	*p_o.State= ACTIVATED*	Internal event	Internal event	*SALES.state=ACTIVATED ∧ p_o.state=FAILED*
$Scope_1$	*SALES.State= SUCCEEDED*	*(p_1.state=SUCCEEDED ∨ p_1.state=FAILED) ∧ (p_2.state=SUCCEEDED ∨ p_2..state=FAILED) ∧ (p_3.state=SUCCEEDED ∨ p_3.state=FAILED)*	*p_1.state=FAILED ∧ p_2..state=FAILED ∧ p_3.state=FAILED*	*$scope_1$.state=ACTIVATED ∧ p_o.state=FAILED*
p_1	*$scope_1$.State= ACTIVATED*	*CHARGE.State= SUCCEEDED ∨ CHARGE.State= FAILED*	By propagation	*p_1.state=ACTIVATED ∧ $scope_1$.state=FAILED*
p_2	*$scope_1$.State= ACTIVATED*	*OUTSOURCE _ANALYSIS .State=SUCCEEDED ∨ OUTSOURCE _ANALYSIS.State= FAILED*	*OUTSOURCE _ANALYSIS.State= FAILED*	*p_2..state=ACTIVATED ∧ $scope_1$.state=FAILED*
p_3	*$scope_1$.State= ACTIVATED*	*CHECK_GOODS.State= SUCCEEDED ∨ CHECK_GOODS.State= FAILED*	By propagation	*p_3.state=ACTIVATED ∧ $scope_1$.state=FAILED*
CHARGE	*p_1.State= ACTIVATED*	Internal event	Internal event	*CHARGE.state= ACTIVATED ∧ p_1.state=FAILED*
OUTSOURCE _ANALYSIS	*p_2.State= ACTIVATED*	Internal event	Internal event	*OUTSOURCE _ANALYSIS .state=ACTIVATED ∧ p_2..state=FAILED*
DELIVERY	*p_2.State= ACTIVATED*	Internal event	Internal event	*DELIVERY.state= ACTIVATED ∧ p_2..state=FAILED*
CHECK_GOODS	*DELIVERY.State = SUCCEEDED*	Internal event	Internal event	*CHECK_GOODS.state= ACTIVATED ∧ p_3.state=FAILED*

Table 5.10 Control flow dependencies of OP case study

5.5.2 E-Booking Example

We demonstrate our management and failure handling mechanism on an E-booking example as depicted in Figure 5.3. This example illustrates how an LRT can succeed in the case of non-vital node failures. In this scenario, there is a need to book a flight, a hotel room and a car for a specific period as received by the *BookingOrder*. It is necessary to find a flight booking and a hotel room for the requested dates, and thus the nodes *Flight* and *Hotel* are assigned as vital nodes. It is desirable for the *MakeBookings* scope that a car rental is booked for the same dates, but not necessary. In other words, if

car rental is not available, the *MakeBooking* is considered to have succeeded from a business point of view, and hence it is assigned as a non-vital node. The successful completion of the nodes *BookingOrder*, *MakeBookings*, and *Payment* are necessary for the successful completion of the E-booking LRT and thus they are all assigned as vital by specification. Note that p_1 and p_2 in *MakeBookings* scope are vital, and p_3 is non-vital by evaluation. The main execution path encloses three nodes:

p_0.nodeList=[BookingOrder,MakeBookings,Payment].

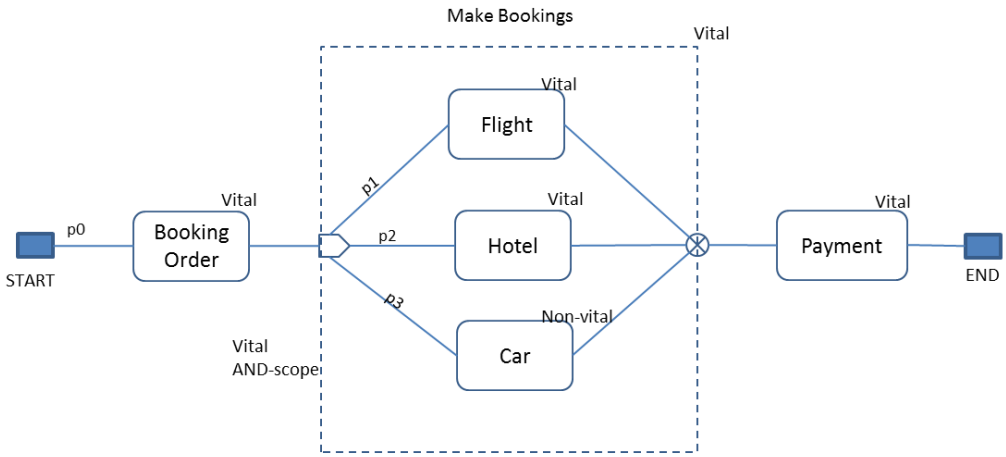


Figure 5.3 E-booking Example

Activation of the LRT (ActR.1) triggers an activation event for p_0 (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 (Control Chart 3). We assume that a completion event has been fired for the node and the *BookingOrder is* marked SUCCEEDED (CompLR.1). Successful completion of *BookingOrder* (Chart 4) activates the *MakeBooking* scope (ActD.3) since *BookingOrder* is not the last node on p_0.

Activation of *MakeBooking*, fires activation events (ActD.4) for p_1,p_2, and p_3 encapsulated by *MakeBooking* and they are all activated by (ActR.3). Subsequently and in the same manner illustrated above, the first nodes on the concurrent paths are activated; *Flight*, *Hotel* and *Car* are executed concurrently. Assume that *Flight* succeeded and *Hotel* succeeded and the system is waiting for the *Car* node to finish its execution. Note that p_1 and p_2 has succeeded by (CompLR.2). To demonstrate how the completion and successful completion of concurrent scopes are dealt with in the case of non-vital failures, we assume that *Car* node fails to complete. Following Chart 7, failure of the non-vital *Car* node fires a failure event for p_3 (FailD.1) and thus p_3 fails (FailR.7). Following chart 9, failure of p_3 fires a completion event for *MakeBookings* since it is the last path to complete, and hence *CompLDep(MakeBookings)=True*. Following chart 6, *MakeBookings* has not failed, since all its vital components have succeeded and there is no failure event fired for the path since p_1 and p_2 has succeeded; hence the *MakingBookings* is succeeded by the completion policy (CompLR.5). Successful completion of *MakeBookings* activates *Payment*. If we assume that *Payment* succeeds, then a completion event is fired for p_0 (CompLD.1) and policy (CompLR.2) succeeds p_0. Following chart 5, successful completion of the main execution path succeeds the LRT by (CompLR.3).

5.5.3 Nested LRT Sample

To further illustrate the propagation mechanism, we will consider the sample LRT_1 presented in Figure 5.4.

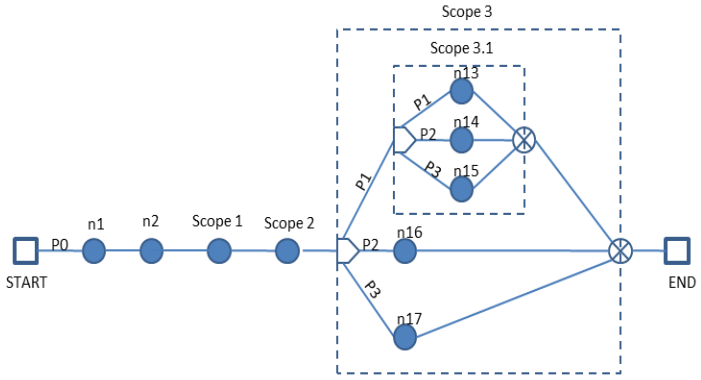


Figure 5.4 An execution instance of LRT_1 in Figure 4.2

Assume an execution instance with the following states of its components: n_1, n_2, $scope_1$ and $scope_2$ have succeeded, and $scope_3$ is activated. n_{17} is a vital node and has failed to complete. Table 5.11 shows $scope_3$'s sub hierarchy tree attribute values and execution states when the node n_{17} failure event has been fired, and we show how the failure propagation algorithm is employed.

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	✕	$scope_3$	✕	succeeded
$scope_3.p_3$	path	✓	$scope_3$	✕	activated
n_{16}	node	✕	$scope_3.p_2$	-	succeeded
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	✓	$scope_{3.1}$	✕	succeeded
n_{13}	node	✓	$scope_{3.1}.p_1$	-	activated
n_{14}	node	✓	$scope_{3.1}.p_2$	-	activated
n_{15}	node	✓	$scope_{3.1}.p_3$	-	succeeded

Table 5.11 Execution Instances of $scope_3$

Following the propagation mechanism in (Charts 8 and 10) and applying the propagation mechanism on scope$_3$'s sub hierarchy tree (Figure 5.5), 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, since it is a concurrent. $scope_3.p_3$ 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 LRT$_1$ (FailR.3). Failure of $scope_3$ will force fail all its activated paths. At this point of execution, $scope_3.p_3$ has already failed and $scope_3.p_1$ has succeeded while $scope_3.p_2$ is still activated and therefore is forced to fail. Force failing a path fails the activated node in that path. Therefore, activated $scope_{3.1}$ is 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.

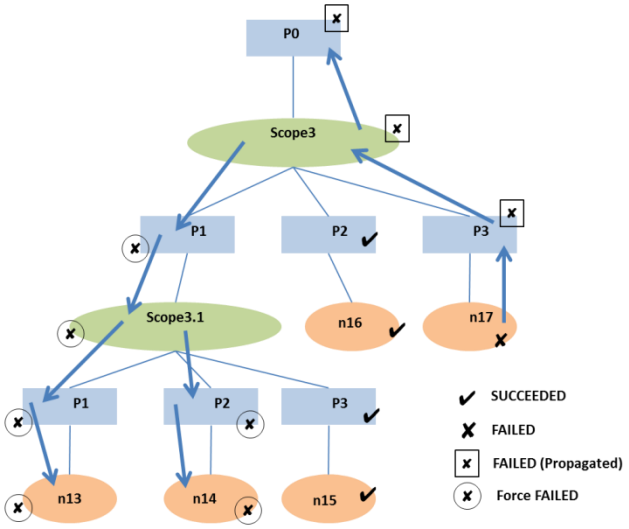


Figure 5.5 Scope$_3$'s Sub-Hierarchy Tree

In the above example, failure of a vital node $scope_3$ on p_0 caused LRT$_1$ to fail. Our management/compensation model applies a reliable mechanism that controls failure of the LRT in a designer specified order that reflects the business logic of the transaction. In case of force failing a scope that has un-activated components, these components can never activate, since their enclosing scope state is failed, ensuring the correctness of the model and avoiding activation of paths in failed scopes.

Chapter 6

Compensation Mechanism

6.1 Introduction

The COMPMOD model supports two types of compensation modes:

Partial compensation: where some compensation actions take place while the LRT is executing in its normal mode, i.e. the LRT state is activated. Partial compensation is applied to nodes, paths, and scopes in tolerance with failures and it primarily reflects WF semantics.

Comprehensive Compensation: when an explicit consensus is reached about the failure of the LRT, the LRT starts its global compensation applied to all successfully completed atomic nodes in a customized-order that is defined by the business process designer at

design time. Comprehensive compensation mainly reflects the compensation logic of the business process.

In this chapter, we demonstrate our compensation mechanisms and show how they are automated through formal definitions of dependencies and compensation policies.

6.2 Partial Compensation

Partial compensation is triggered by failure of an exclusive path that has an alternative. To illustrate our partial compensation semantics, we consider sample LRT_2 in (Figure 6.1).

Exclusive scopes encapsulate paths that alternate each other in execution such that only one path is allowed to succeed (e.g. paths p_1, p_2, and p_3 in $scope_2$). 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 its succeeded nodes (if any) are compensated. For example, assume that p_1 in $scope_2$ is activated and n_3 and $scope_{2.1}$ have succeeded. We further assume that n_{11} is vital but has failed, this will fail and compensate p_1 and consequently $scope_{2.1}$ and n_3 are compensated. Only when the failed path has completed its compensation actions is an activation event fired for its alternative path (ActD.6 and Charts 8 and 9 in Chapter 5). E.g. only when p_1 in $scope_2$ has finished its compensation, p_2 can be activated.

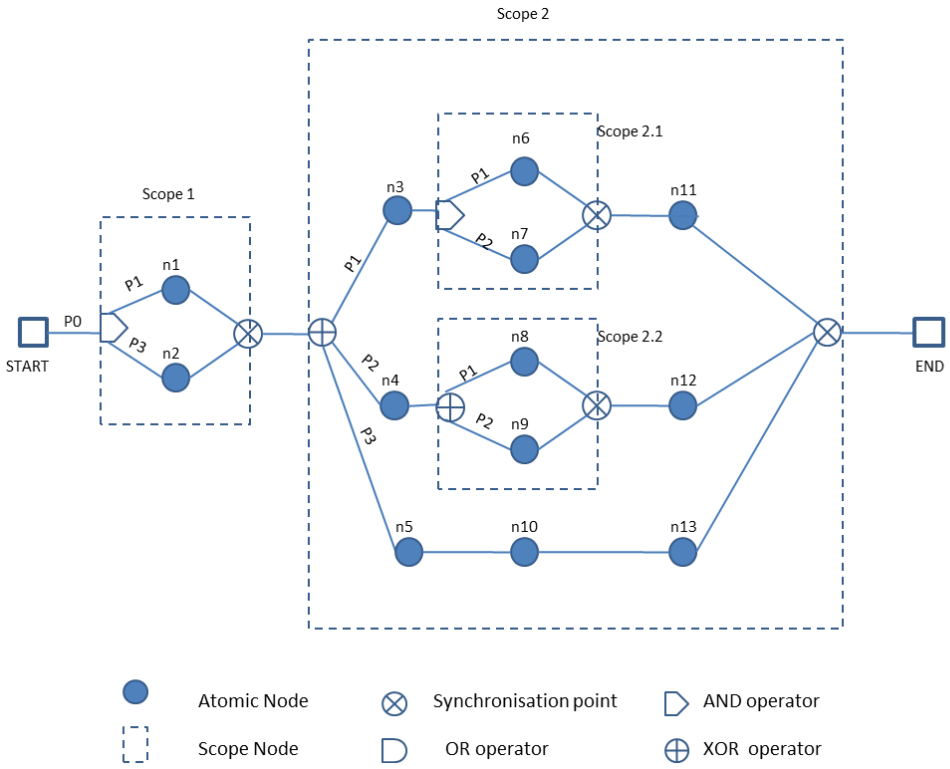


Figure 6.1 Sample LRT_2

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

1. The failed exclusive path could have nodes on the path that were succeeded, failed or not-activated (i.e. the failure occurred before the node has been activated).
2. Nodes could be scopes, and hence, if the scope was activated and some tasks had been succeeded within the scope, then all succeeded work has to be compensated.

3. The path is an atomic path that encapsulates a single node, and its failure has caused failure of the path where the node may be atomic or scope.

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

Forward compensation[14] is 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. For example, in case p_1 in $scope_2$ has failed, then its compensation is performed in forward order.

Backward compensation is used to refer to the compensation process of a scope node that has been previously succeeded or failed (i.e. some partial work could have been succeeded within the scope) and is formally defined 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. For example, if

[14] We adopt the term from the sagas.

we assume that $scope_{2.1}$ has succeeded but its enclosing path p_1 is compensated, then compensation of $scope_{2.1}$ is performed in backward order.

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 (e.g. p_1 in $scope_2$) and a backward compensable path (a path within a backward compensable scope like p_1 in $scope_{2.1}$) are both potentially compensable paths. Analogously, a node is potentially compensable if it is encapsulated with a potentially compensable path. Thus, in COMPMOD, 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.

Based on the above discussion, we provide in the next sections a detailed description of partial compensation semantics.

6.2.1 Compensational Attributes

In order to define compensation dependencies, we provide an essential compensational attribute for LRT components, ***IsCompensable,*** and its value is assessed as follows:

1- The main execution path is not compensable since if it fails then its compensation is performed in customized order.

$$\boldsymbol{path = p_0 \longrightarrow path.IsCompensable = FALSE}$$

2- A path *IsCompensable* iff the path has an alternative.

$$\boldsymbol{path.hasAletrnative = TRUE \longrightarrow path.IsCompensable = TRUE}$$

3- A scope node *IsCompensable* iff its superior path *IsCompensable*

$$\boldsymbol{scopeNode.superior.IsCompensable = TRUE \longrightarrow scopeNode.IsCompensable}$$

4- A path that has no alternative *IsCompensable* iff its superior scope *IsCompensable.* This is the case of concurrent paths, and the last exclusive path in an exclusive scope.

$$\boldsymbol{scope.IsCompensable = TRUE \ \wedge \neg scopeInferior.hasAlternative \longrightarrow scopeInferior.IsCompensable = TRUE}$$

To further illustrate the *compensability* attribute, we consider the sample LRT_2 in figure 6.1. In LRT_2, $scope_1$ and its encapsulated path are not compensable, since the scope is not encapsulated by a compensable path *(p_0.IsCompensable=FALSE).* Paths p_1 and p_2 in $scope_2$ are compensable since they both have alternatives. $Scope_{2.1}$ and $scope_{2.2}$ and their encapsulated paths are compensable since both scopes are encapsulated by compensable paths. Note that p_1 in $scope_{2.2}$ is compensable for another reason: it is a path with an alternative. Path p_3 in $scope_2$ is not compensable hence its failure will not trigger compensation activity.

6.2.2 Dependencies Semantics

The general behavioral dependency in **Def. 4.3** is used for defining compensation dependencies, $\boldsymbol{Dep}(\boldsymbol{component}_j):=\boldsymbol{preCond}(\boldsymbol{component}_i.\boldsymbol{state})$, to indicate that there exists a compensation dependency $\boldsymbol{component}_i \leftarrow_{\boldsymbol{Comp}} \boldsymbol{Component}_j$ such that a state transition in $component_i$ can fire a compensation event for $component_j$.

Compensation intuitively means performing compensating activities that undo the effects of a succeeded atomic node. A succeeded atomic node could only exist on a succeeded or failed[15] path and hence not-activated paths or compensated[16] paths could not possibly have succeeded nodes. A succeeded and failed execution path may only be encapsulated by succeeded or failed scopes since a not-activated scope encapsulates only not-activated paths.

Therefore, a component is a compensation candidate iff it is:

1- Succeeded atomic node

2- Succeeded or failed compensation path

3- Succeeded or failed scope node.

In our compensation model, compensation dependencies are defined for all compensable components (i.e. *component.isCompensable=TRUE*) and when satisfied, compensation events are fired but only components that are candidate for compensation actions are compensated.

[15] A failed path could be partially succeeded
[16] Within an exclusive scope

Compensation semantics for execution **paths** states that when a path commences its compensation, a compensation event is fired for the last node on the path, and compensation events and actions are cascaded in reverse order until the first node is compensated such that the preceding node in a sequence can commence its compensation only after the succeeding node on the same path has finished its compensation.

To enforce the reverse order of compensations along compensating paths, the model explores all nodes existing on the path (i.e. all compensation events of nodes are assessed) but only compensates candidate nodes. Non-candidate nodes are explored but skipped. So mainly exploring a non-candidate is required just to enforce the reverse sequence of compensations, but without performing any compensation actions in the node.

Compensation semantics for **scope** nodes states that when a scope commences its compensation, a compensation event is fired for all its enclosed paths but only candidate paths are compensated. Non-candidate paths are ignored.

Based on the above argument, when a compensation event is fired for a component that is not a candidate for compensation action, the model corresponds according to the following rules:

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

Def. 6.1: (skipping a node on a compensating path)

$$\boldsymbol{CompDep(atomicNode) = TRUE \wedge atomicNode.state \neq SUCCEEDED \longrightarrow skip(atomicNode)}$$

$$\boldsymbol{CompDep(scopeNode) = TRUE \wedge scopeNode.state = NOT-ACTIVATED \longrightarrow skip(scopeNode)}$$

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 remains as its current marked state.

6.2.2.1 Compensation Dependencies

A partial compensation dependency is defined as follows:

1- Between a node and its successor (if any) iff its superior path *IsCompensable.*

2- Between a path and its superior scope iff and only iff the scope *IsCompensable.*

3- Between the last node and its encapsulating path iff the path *IsCompensable.*

Therefore, compensation dependencies are defined for nodes encapsulated by a compensable path and for execution paths that are encapsulated by a compensable scope (Table 6.1) such that when a compensation event is fired for a component, the event is assessed by compensation policies, and if there are possible compensation actions to be performed on the component, then the component commence compensation and its state is marked as COMPENSATING. As mentioned earlier, partial compensation is always

triggered by the failure of an exclusive path with an alternative and its compensation is mainly dependent on its own failure. Hence, a path that has an alternative is not defined with compensation dependency; instead, its compensation is triggered by a compensation policy.

Dep #	Dependency	Component
CompD.1	$\boldsymbol{CompDep(lasNode) := superiorPath.State = COMPENSATING}$	Last node on a compensable path
CompD.2	$\boldsymbol{CompDep(node_1) :=}$ $\boldsymbol{superior.State = Compensating}$ $\boldsymbol{\wedge\ (node_2.State = COMPENSATED\ \vee\ node_2.State = SKIPPED)}$	A node that has a successor on a compensable path $node_2$=successor($node_1$)
CompD.3	$\boldsymbol{CompDep(\ path) := superiorScope.State = COMPENSATING}$	Path within a compensable scope

Table 6.1 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.

6.2.2.2 Compensation Completion Dependencies

Compensation completion dependencies are defined for compensable paths and scopes to signal the end of their compensation process (Table 6.2) such that when fired, they are marked by a completion policy as COMPENSATED.

For **atomic nodes**, compensation completion is triggered by an internal event for the node. It is an assumption of the model that compensation completion of the node is

guaranteed to succeed, and thus, when a compensation completion event is fired for an atomic node, it is marked as COMPENSATED.

A **compensating path** ends its compensation process when the first node in the path has either compensated or skipped (CpCompLD.1).

To reach a consensus about the 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.

From Chapter 5, an **exclusive scope** succeeds if an exclusive path succeeds, thus the scope completes with one succeeded path in addition to one or more compensated and/or not-activated paths depending on rank of the succeeded path in the scope. And an exclusive scope fails when the last path fails; hence, the scope completes with one failed path and one or more compensated paths, depending on the number of paths within the scope. From CR2, we may conclude that a compensable exclusive scope always has only one compensation candidate path, and hence the scope is treated as a single path, and consequently the compensation completion of its candidate path signals the compensation completion of the scope.

Compensation of concurrent paths is executed concurrently but in reverse order of activations of their nodes. Hence, their compensation completion has to be synchronized and this is formalized through compensation completion dependency. A failed or succeeded concurrent scope may encapsulate only succeeded and/or failed paths in case of AND-scope, and succeeded and/or failed and/or not-activated paths in case of OR-

scope. We say that a concurrent path completes its compensation when all paths are either compensated or not-activated (CpCompLD.2) and we define the same dependency for both AND and OR scopes.

Dep #	Dependency	Component
CpCompLD.1	$\boldsymbol{CpCompLDep(path)} :=$ $(\boldsymbol{firstNode.State = COMPENSATED}) \vee \boldsymbol{firstNode.State = SKIPPED})$	Compensable Path
CpCompLD.2	$\boldsymbol{CpCompLDep(scope)} := \boldsymbol{scope.state = compensating} \wedge$ $(\bigwedge_{\boldsymbol{i=1..m}} (\boldsymbol{pathList_i.state = COMPENSATED} \vee \boldsymbol{pathList_i.state = NOT-ACTIVATED}))$	Compensable concurrent scope with m paths

Table 6.2 Compensation Completion Dependencies

6.2.3 Partial Compensation Mechanism

Partial compensation is automated through compensation policies (Table 6.3) and compensation completion policies (Table 6.4). The automation process is illustrated through Control charts (12-15). Note that we add a consistency condition (LRT.state=ACTIVATED) to all compensation policies to differentiate between the partial compensation mode and comprehensive compensation mode (LRT.state=COMPENSATING), such that compensation events are handled reliably and in context with the correct mode of compensation.

Rule#	Compensation Policies	Component
CompR.1	***ON*** *fail(path)* ***IF*** *path.hasAlternative and node.superior.state=ACTIVATED* ***DO*** *compensate(path)*	Exclusive path with alternative
CompR.2	***ON*** *CompDep(path)* ***IF*** *LRT.State=ACTIVATED and (*path.state=SUCCEEDED or path.state=FAILED) ***DO*** *compensate(path)*	Compensable path previously succeeded or failed
CompR.3	***ON*** *CompDep(node)* ***IF*** *LRT.State=ACTIVATED and node.Type=ATOMIC and node.State=SUCCEEDED* ***DO*** *compensate(node)*	succeeded Atomic node
CompR.4	***ON*** *CompDep(node)* ***IF*** *LRT.State=ACTIVATED and node.Type=ATOMIC and (node.State=FAILED or node.state=NOT-ACTIVATED Or nodeState=ABORTED)* ***DO*** *skip(node)*	Non succeeded atomic node
CompR.5	***ON*** *CompDep(node)* ***IF*** *LRT.State=ACTIVATED and node.Type=SCOPE and (node.State=SUCCEEDED or node.state=FAILED)* ***DO*** *compensate(node)*	Succeeded or failed scope
CompR.6	***ON*** *CompDep(node)* ***IF*** *LRT.State=ACTIVATED and node.Type=SCOPE and node.state=NOT-ACTIVATED* ***DO*** *skip(node)*	Not activated scope

Table 6.3 Compensation Policies

Rule#	Compensation Policies	Component
CpCompLR.1	***ON*** *"internal compensation completion event of atomic node"* ***IF*** *LRT.State=ACTIVATED* ***DO*** *compensated(node)*	Atomic node
CpCompLR.2	***ON*** *CpCompLDep(path)* ***DO*** *compensated(path)*	Path
CpCompLR.3	***ON*** *CpCompLDep(node)* ***IF*** *node.Type=SCOPE and node.state≠SKIPPED* ***DO*** *compensated(node)*	Concurrent Scope
CpCompLR.4	***ON*** *compensated(path)* ***IF*** *path.IsExclusive and Path.superior.state=compensating* ***DO*** *compensated(path.superior)*	Exclusive path

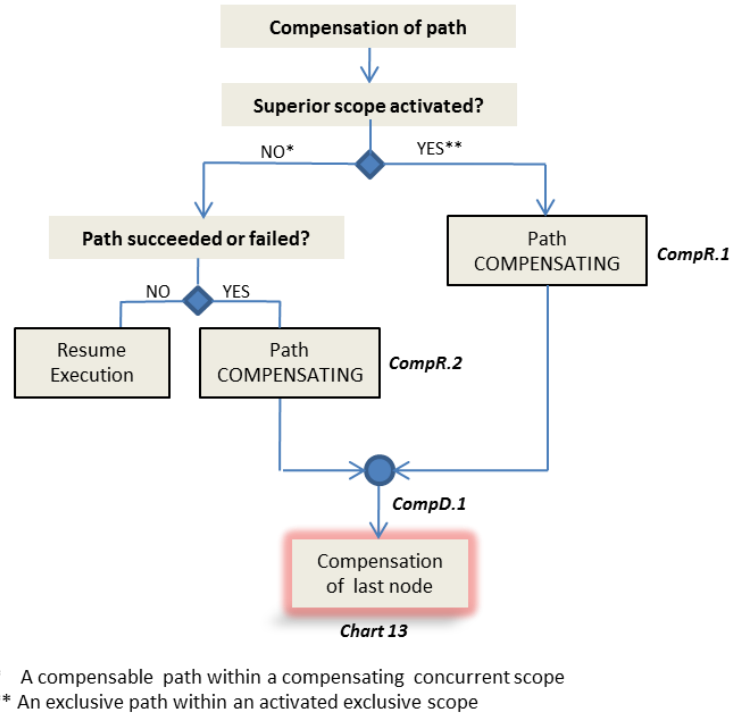
Table 6.4 Compensation Completion Policies

The partial compensation mechanism is automated as follows:

1- When a failure event is fired for an exclusive path with an alternative, the event is assessed by (CompR.1) and the path is marked COMPENSATING.

2- A compensation event fired for a compensable path is assessed by (CompR.2) and the path is marked COMEPSNATING.

3- When a path commences its compensation, a compensation event is fired for the last node in the path (CompD.1).



Control Chart 12. Compensation of Path

4- When a compensation event is fired for an atomic node, if the node has succeeded, the event is assessed by (CompR.3) and the nodes start COMPENSATING.

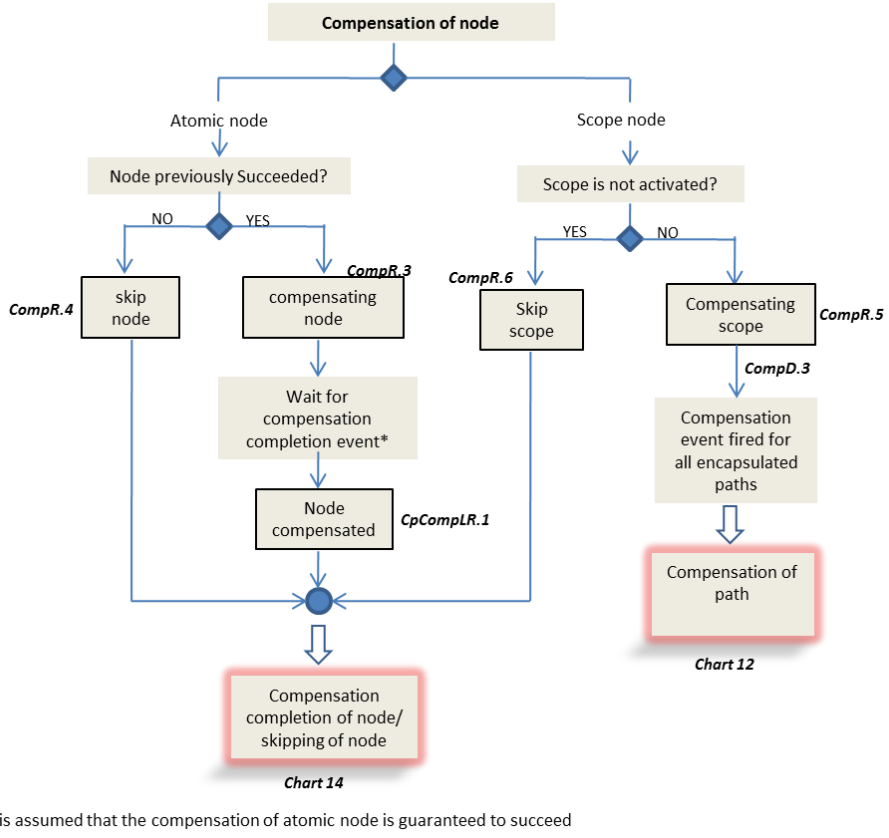
5- When a compensation event is fired for an atomic node, if the node that has not been succeeded, the event is assessed by (CompR.4) and the node is SKIPPED.

6- When an internal compensation completion event is fired for an atomic node, the node is marked as COMPENSTAED (CpCompLR.1).

7- A compensation event fired for a non-activated scope is assessed by (CompR.6) and the scope is SKIPPED.

8- A compensation event fired for a SUCCEEDED or FAILED scope is assessed by (CompR.5) and the scope starts compensating.

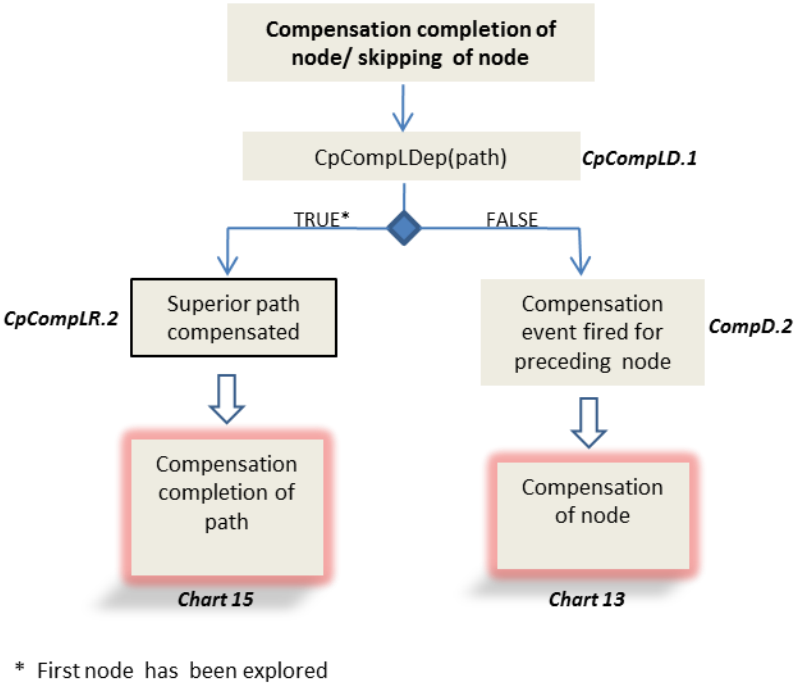
9- When a scope commences its compensation, a compensation event is fired for all its encapsulated paths (CompD.3) and control goes to step 2.



Control Chart 13. Compensation of node

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) 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) and control goes to step 4 or 5.



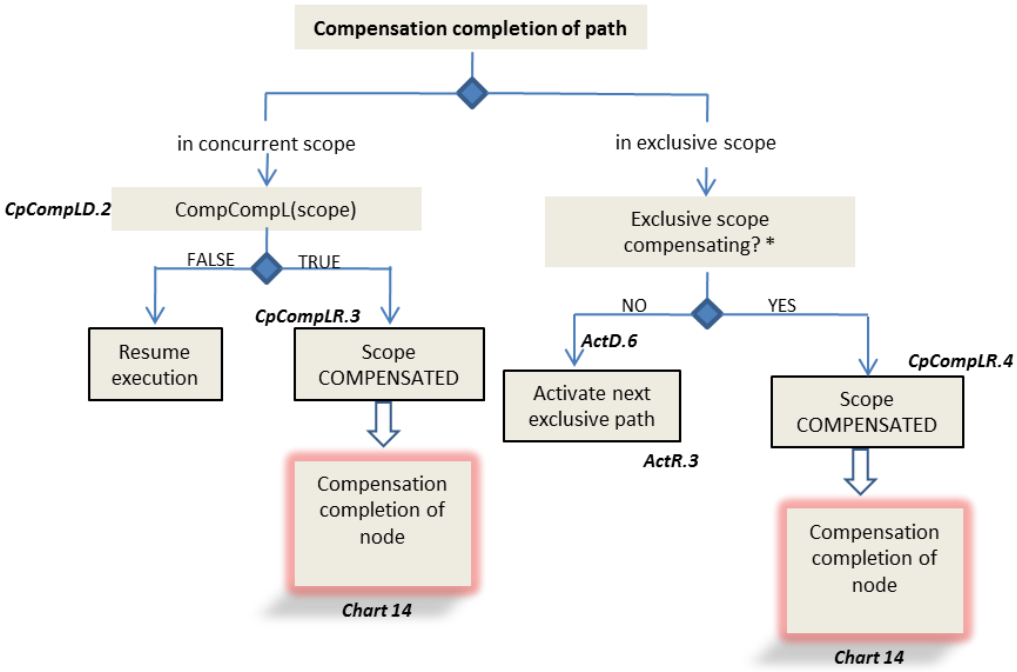
Control Chart 14. Compensation Completion or Skipping of a node

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)[17] and the node is activated by activation policy (ActR.3)[18].

[17] Table 5.1, Chapter 5.
[18] Table 5.1, Chapter 5.

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).

14- When a compensation completion event is fired for a concurrent scope (CpCompLD.2), the scope is marked compensated by policy (CpCompLR.3).



Control Chart 15. Compensation Completion of a Path

6.2.4 Example

For further illustration of the mechanism, we assume different failure scenarios for path p_1 in $scope_2$ (Figure 6.1), and we apply partial compensation semantics on each scenario and demonstrate the state of all components before and after compensation is applied.

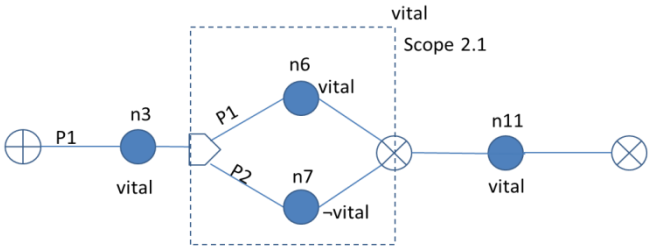


Figure 6.2 A caption of $scope_2.p_1$ in sample LRT_2

We assume the following failure scenarios:

Scenario1: n_3 fails and hence p_1 fails by propagation, and hence all other components are not activated.

Scenario2: n_3 succeeds, n_7 is activated but n_6 fails and thus $scope_{2.1}.p_1$ fails by propagation. n_6 fails $scope_{2.1}.p_1$ and $scope_{2.1}$ by propagation. Failure $scope_{2.1}$ force fails $scope_{2.1}.p_2$ and hence n_7 if forced to fail (aborted). Failure of vital $scope_{2.1}$ fails p_1 by propagation and n_{11} remains not activated

Scenario3: n_3, n_6 succeeds, and non-vital n_7 fails which fails $scope_{2.1}.p_2$ by FailR.7 but $scope_{2.1}$ succeeds by CompLR.5. n_{11} vital but fails and thus p_1 fails by propagation.

The current execution states (w.r.t different scenarios) of $scope_2.p_1$'s encapsulated components at the time p_1 fails are represented in (Table 6.5).

S	n_3	$Scop_{2.1}$ p_1	n_6	$Scop_{2.1}$ p_2	n_7	$scope_{2.1}$	n_{11}
1	FAILED	NOT-ACTIVATED	NOT-ACTIVATED	NOT-ACTIVATED	NOT-ACTIVATED	NOT-ACTIVATED	NOT-ACTIVATED
2	SUCCEEDED	FAILED	FAILED	FAILED	ABORTED	FAILD	NOT-ACTIVATED
3	SUCCEEDED	SUCCEEDED	SUCCEEDED	FAILED	FAILED	SUCCEEDS	FAILED

Table 6.5 Current Execution State Instances of failed $scope_2.p_1$ in figure 6.2

In (Table 6.6), we show the current execution states of the components after applying partial compensation to $scope_2.p_1$ w.r.t different failure scenarios. Note that failure of $scope_2.p_1$ triggers a compensation of the path by policy CompR.1.

In scenario 1: n_{11} is not-activated, and thus is SKIPPED by policy CompR.4. $Scope_{2.1}$ is not-activated and thus skipped by CompR.6. p_1 and p_2 are paths within a skipped scope hence no compensation dependency is fired for them, and hence all $scope_{2.1}$'s encapsulated components states remain unchanged. Note that $scope_{2.1}$ can never be fired with compensation completion dependency, since it can only be evaluated for compensating scopes (CpCompLD.2). n_3 has failed and thus skipped after compensation.

In scenario 2: $scope_{2.1}$ has failed hence it is explored by CompR.5. Compensating $scope_{2.1}$ fires compensation events for p_1 and p_2 which both have failed and hence explored by policy CompR.2. n_7 has been aborted and n_6 has been failed and hence they are both skipped which consequently triggers a compensation completion event for both p_1 and p_2. Marking p_1 and p_2 as compensated triggers a compensation completion event for $scope_{2.1}$ (CpCompLD.2) and hence the scope is COMPENSATED.

In scenario 3, $Scope_{2.1}$ has succeeded and thus it is explored in the following manner: both its paths p_1 and p_2 start compensating, n_6 and n_7 are explored because they are the last nodes on the paths, n_6 is compensated (CompR.3) and n_7 is skipped. Subsequently p_1, p_2, and their $scope_{2.1}$ are all marked COMPENSATED by compensation completion events and policies. .

S	n_3	$Scop_{2.1}$ p_1	n_6	$Scop_{2.1}$ p_2	n_7	$scope_{2.1}$	n_{11}
1	SKIPPED	NOT-ACTIVATED	NOT-ACTIVATED	NOT-ACTIVATED	NOT-ACTIVATED	SKIPPED	SKIPPED
2	COMPENSATED	COMPENSATED	SKIPPED	COMPENSATED	SKIPPED	COMPENSATED	SKEPPED
3	COMPENSATED	COMPENSATED	COMPENSATED	COMPENSATED	SKIPPED	COMPENSATED	SKIPPED

Table 6.6 Current Execution State Instances of compensated *$scope_2.p_1$* in figure 6.2

6.3 Comprehensive Compensation

We have demonstrated in section 6.3 the mechanism for partial compensations which takes place while the LRT is still activated and for which failures are tolerable and does not lead to a global failure of the transaction. Some failure events lead to global failure of the transactions, as discussed in the failure handling mechanism in Chapter 5. These failures are mainly triggered by a failure of a vital node that is preceded by a hierarchy of vital ancestor components towards the top of the hierarchy, such that the failure propagates up the hierarchy structure and reaches the main execution path and consequently the transaction globally fails.

From a business point of view, a failed transaction means that it has failed to achieve its expected outcome, however some tasks could have succeeded but their effects are no longer required, since the transaction has failed. From the reliability and consistency of transaction's point of view, these successful tasks should be compensated and their effects should be undone to guarantee the consistency of all systems involved in the transaction. The question arising is how to apply compensations and in which order. As

we have discussed in Chapter 1, strict global backward recovery restricts business process designers from freely expressing compensation logic in an arbitrary manner and as required by the business logic of the process.

The COMPMOD model supports a ***customized-compensation*** method that provides transaction designers with the flexibility of expressing their business process logic. Compensation logic can then be 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. A compensation pattern decides the order by which the specified services are compensated. 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:

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

Customized compensations are formalized through customized compensation dependencies, and are automated through customized compensation policies. The formalism embeds a traversing mechanism that ensures navigation of all atomic nodes in the transaction. In the following subsections we demonstrate the method.

6.3.1 Customized Compensation Dependency Graph

The customized compensation control flow is represented in COMPMOD as a directed acyclic dependency graph, where vertices in the graph represent atomic nodes, and edges in the graph represent compensation dependencies. In order to achieve an acyclic dependency graph, we provide a method for defining customized dependency patterns that are cycle free, such that only valid dependencies are allowed. The motivation for the graph to be acyclic is that this ensures that the mechanism is deadlock free by design.

A customized compensation dependency is denoted as $\boldsymbol{node_i \leftarrow_{CCDep} node_j}$ for any two atomic nodes $node_i, node_j \in NODES$, where *NODES* is the set of all nodes in LRT and is read as "there exists a customized compensation dependency from $node_j$ to $node_i$ such that $node_j$ can be compensated only after $node_i$ has been visited in the traversing graph". Therefore, it is convenient to think of $node_i$ as the source of the dependency and $node_j$ as the target of the dependency. To define a compensation dependency, a ***source node*** and a ***target node*** are assigned from the set of LRT atomic nodes, provided that the dependency is valid.

A compensation pattern is formed when two or more nodes are associated through customized compensation dependencies. Figure 6.3 shows an LRT-WF schema for a sample LRT_3 with allocated compensation dependencies. For example, the dotted arrow from n_3 to n_1 states that n_3 is the source node and n_1 is the target node. Note that customized compensations in Figure 6.3 are assigned arbitrarily in the sense that it can be assigned between nodes on different paths, e.g. n_5 and n_3 or between nodes in different scopes, e.g. n_8 and n_6. The motivation behind these arbitrary assignments is principally, the business logic of the process. E.g. update inventory should be compensated after shipping goods is compensated where both activities may exist in different scopes.

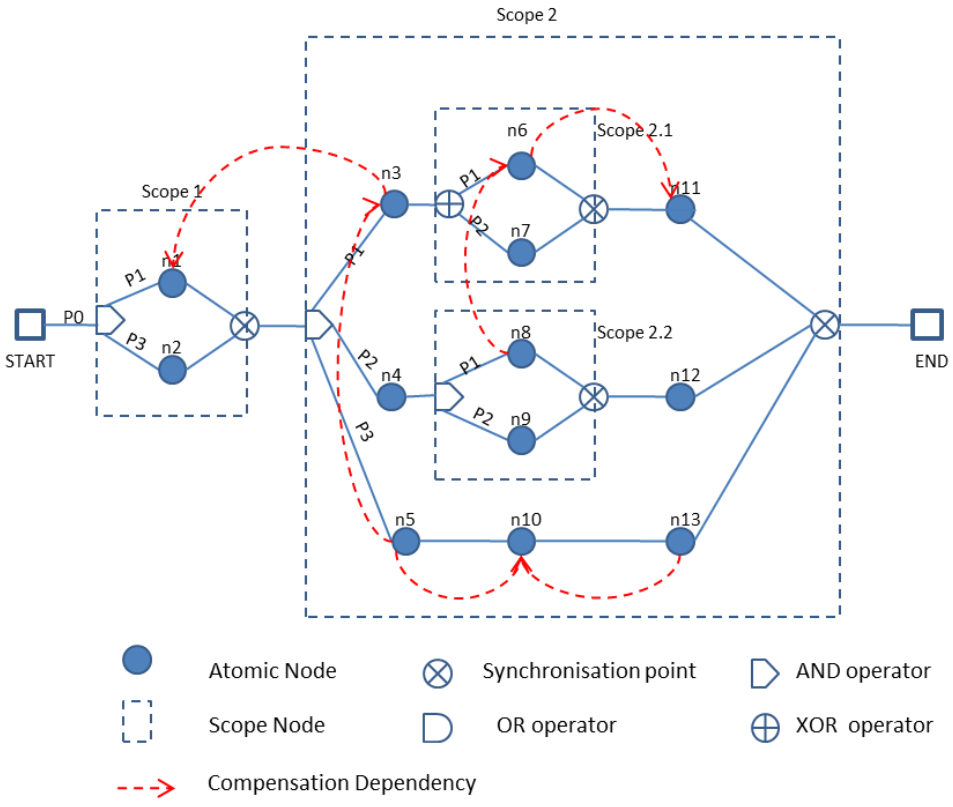


Figure 6.3: A Sample LRT_3 with customized compensation dependencies

The dependency graph is represented by the pair $\boldsymbol{G = (V, E)}$ where

$\boldsymbol{V = \{ node | node \epsilon NODES \text{ and } node.type = ATOMIC\}}$ is the set of all atomic nodes in the LRT, and

$\boldsymbol{E = \{(node_i, node_j) | node_i, node_j \epsilon V \text{ and } \exists\ node_i \leftarrow_{CCDep} node_j \text{ where } i \neq j\}}$

is a binary relation on $\boldsymbol{V}$ representing customized compensation dependencies, where a pair $\boldsymbol{(node_i, node_j)}$ designates that there is a directed edge in the dependency graph from $node_i$ to $node_j$ and that $node_j$ is *compensationally* dependent on $node_i$.

A node in G is ***isolated*** if it has no edges incident from it or incident to it. In the dependency graph, a node is isolated if it is not part of any compensation dependency. We define $\boldsymbol{D = \{ node | node \in V \text{ and } \nexists (node_i . node_j) \in E \text{ where } node_i = node \text{ or } node_j = node\}}$ to be the set of all isolated nodes in **G**.

A node in the dependency graph is a ***Root*** if there are no edges incident to it. In other words, a node is a root if it is not a target in a customized compensation dependency. We define $\boldsymbol{RT = \{node | node \in V \text{ and } \nexists (node_i, node_j) \in E \text{ where } node_j = node\}}$ to be the set of all root nodes. Note that isolated nodes are also root nodes, since an isolated node has no edges incident to it, that is $\boldsymbol{D \subset RT.}$

A node in G is a source root node if it has at least one edge incident from it and has zero edges incident to it. A source root node can be a source node of one or more compensation dependencies, but it is not a target node in any compensation dependency. Any source root node is in RT since it has no edges incident to it, but it is not in D since

it has at least one edge incident from it. We define $\boldsymbol{SR} = \{\boldsymbol{node}|\boldsymbol{node} \in \boldsymbol{R\ and\ node} \notin \boldsymbol{D}\}$ to be the set of all source root nodes in G.

A node in G is a target node if it has at least one edge incident to it. We define $\boldsymbol{TG} = \{\boldsymbol{node}|\boldsymbol{node} \in \boldsymbol{V\ and} \exists (\boldsymbol{node_i}, \boldsymbol{node_j}) \in \boldsymbol{E\ where\ node_j} = \boldsymbol{node}\}$ to be the set of all target nodes in V.

A node in G is a terminal node if it has no edges incident from it and has at least one edge incident to it. A terminal node can be a target node in one or more compensation dependencies, but not a source in any compensation dependency. We define

$\boldsymbol{T} = \{\boldsymbol{node}|\boldsymbol{node} \in \boldsymbol{V\ and} \exists (\boldsymbol{node_i}, \boldsymbol{node_j}) \in \boldsymbol{E\ where\ node_j} =$

$\boldsymbol{node\ and} \nexists (\boldsymbol{node_k}, \boldsymbol{node_l}) \in \boldsymbol{E\ where\ node_k} = \boldsymbol{node}\}$ to be the set of all terminal nodes in G.

The dependency graph of the LRT_3 in Figure 6.3 is depicted in Figure 6.49(a) and in figure 6.4(b) we show the set mapping of the dependency graph.

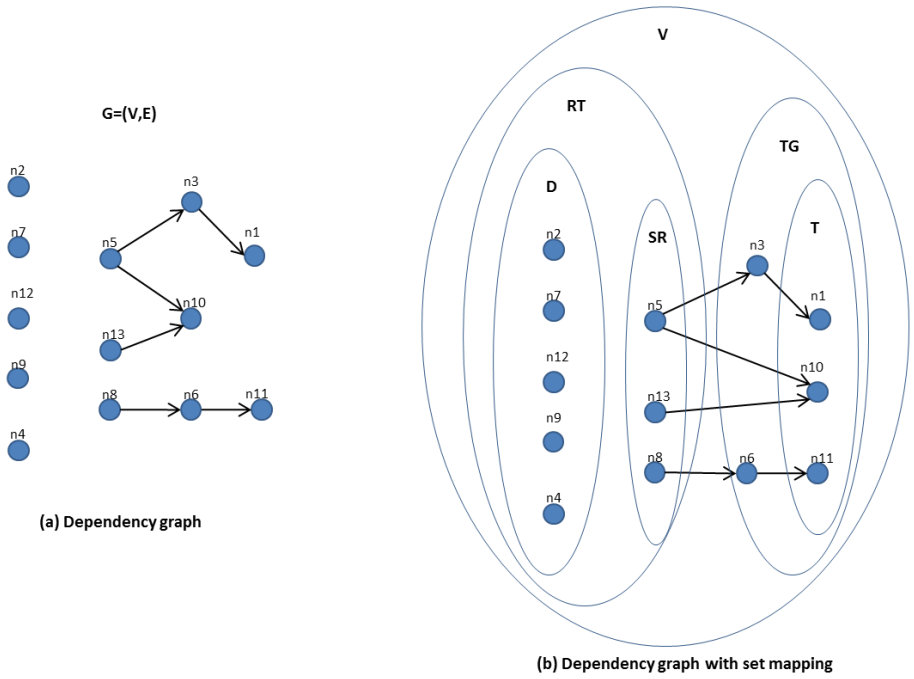


Figure 6.4 Dependency Graph for sample LRT3

6.3.2 Compensational Attributes

A customized compensation dependency entails the following compensational attributes:

- A target node $node_i$ can be a target node for one or more source nodes. Each target node $node_i \in TG$ is associated with a list of its source nodes as $\boldsymbol{node_i.SourceList} = [\boldsymbol{source_1}..\boldsymbol{source_n}]$ where n≥1.
- Every $node_i \in V$ is associated with an ***IsVisited*** Boolean attribute to indicate whether the node has been traversed or not in the comprehensive compensation process. We say that $node_i$ has been visited if ($\boldsymbol{node_i.IsVisted} = \boldsymbol{TRUE}$) otherwise the node has not been traversed and ($\boldsymbol{node_i.IsVisted} = \boldsymbol{FALSE}$).

6.3.3 Validity of Compensation Dependencies

In this section, we describe the construction process of the dependency graph, and show that this process will avoid inclusion of cycles.

Given an existing valid dependency graph G, appending a new dependency between any two nodes $n_i, n_j \in V$ will add an edge to G. The new edge will be tested first for its validity. If the edge does not lead to a cycle in G, then the edge is considered valid and appended to G. If the added edge forms a cycle in G, then the edge is not valid and will not be appended to G.

We say that an empty graph, a graph with no edges in E (G.E=ϕ), is a valid graph. Adding a new edge $(n_i, n_j) for\ i \neq j$, to an existing valid graph G have the following possibilities:

1- If **G.E= ϕ** then adding a single edge to the graph will not form a cycle, therefore it is valid.
2- If $\boldsymbol{n_i} \in \boldsymbol{RT}$ then the added edge is valid since all root nodes have no edges incident to it and therefore there is no possibility of cycle formation in G.
3- If $\boldsymbol{n_j} \in \boldsymbol{T}$ or $\boldsymbol{n_j} \in \boldsymbol{D}$ then the added edge is valid since all terminal and isolated nodes have no edges incident from it and therefore there is no possibility of cycle formation in G.
4- A new edge ($\boldsymbol{n_i}, \boldsymbol{n_j}$) will form a cycle in the existing dependency graph if (a) it does not satisfy any of the above conditions, and (b) there exists a path from $\boldsymbol{n_j}$ to

5- $\boldsymbol{n_i}$ in the graph ($\boldsymbol{n_j} \rightsquigarrow \boldsymbol{n_i}$). If there is no path from $\boldsymbol{n_j}$ to $\boldsymbol{n_i}$ then the compensation dependency $\boldsymbol{n_i} \leftarrow_{\boldsymbol{CCDep}} \boldsymbol{n_j}$ is a valid dependency. We define ***vaildG*** algorithm to validate case (b).

The following algorithms are implemented to initialize, construct, update and validate dependency graphs.

The dependency graph G is initialized as follows:

InitG(G)

1. $E=\phi$
2. $V = \{ node | node \in NODES \text{ and } node.state = ATOMIC\}$
3. $RT=D=V$
4. $T=SR=TG=\phi$
5. $For\ each\ node \in V$
6. $node.IsVisited = FALSE$
7. $initialise\ node.SourceList$

Given a valid dependency graph G and a valid new edge (s,t), the following algorithm updates G and the associated graph sets.

UpdateG(G,(s,t))

1. $E=E+\{(s,t)\}$
2. *append s to t.SourceList*
3. *if s in D then D=D-{s}*
4. $SR=SR+\{s\}$
5. *if s in T then T=T-{s}*
6. *if t in D then D=D-{t}*
7. $T=T+\{t\}$
8. $RT=D \cup SR$
9. $TG=TG \uplus T$

Suppose that v denotes node vertices in G, then we define a compensating path $< v_1,..,v_k >$ as a path in the dependency graph that starts with a source root vertex

$v_1 \in SR$ and ends with a terminal vertex $v_k \in T$ such that $k \geq 2$ and $(v_i, v_{i+1}) \in E \; for \; i = 1..(k-1)$. Note that isolated nodes do not belong to any compensation path. We say that a new added edge (s,t) forms a cycle in G if and only if there exists a path from t to s in G, that is, $\exists < v_1, .. v_k >$ where $k \geq 2$ such that $v_1 = t$ and $v_k = s$ and $(v_i, v_{i+1}) \in E$ for i=1..(k-1).

Given a valid dependency graph G and a new edge (s,t), ***validG*** algorithm returns TRUE if the added edge does not form a cycle in the dependency graph and returns FALSE otherwise.

The algorithm uses a breadth first search algorithm adapted from (Cormen et al., 2009), but it was amended to find the shortest path (if one exists) from t to s.

validG(G,(s,t))

1. ***Q=Ø***
2. ***ENQUEUE(Q,s)***
3. ***While Q≠Ø***
4. ***u=DEQUEUE(Q,s)***
5. ***For each*** $v \in u.SourceList$
6. ***if*** $v = t$ ***then RETURN FALSE***
7. ***else ENQUEUE(Q,v)***
8. ***RETURN TRUE***

6.3.4 Compensational Behavior

Given a valid dependency graph of an LRT, COMPMOD applies a graph traversing mechanism to ensure that every $node_i \in V$ is visited and that the specified order of customised compensations is enforced.

A node in the graph can either be waiting, explored or visited. A node is said to be waiting if there is no compensation event fired for the node. A node is said to be explored when a compensation event ***(CCDep())*** is fired for the node and some actions are taking place with regard to the explored node. When the actions have been completed, execution leaves the node and marks it as visited.

The traversing mechanism operates as follows:

Initially, all nodes are in the waiting phase, where they are all marked as not visited and no compensation event is fired for them:
$(\forall\, \boldsymbol{node_i} \in \boldsymbol{V}, \boldsymbol{node_i.IsVisited = FALSE\ and\ CCDep(node_i) = FALSE})$. If the LRT commences its compensation ***(LRT.State=COMPENSATING)***, a compensation event is fired for all root nodes in RT and they are all explored concurrently:

$$(\forall\, \boldsymbol{node_i} \in \boldsymbol{RT}, \boldsymbol{node_i.IsVisited = FALSE\ and\ CCDep(node_i) = TRUE}).$$

When a node is being explored, the current state of the node is checked. If the node has been previously succeeded, the node is called for compensation and its state is marked as COMPENSATING. When an internal compensation completion event is fired for an

explored compensating node, its state is marked as COMPENSATED, and it is marked visited:

$$(\boldsymbol{compensated}(\boldsymbol{node}) \longrightarrow \boldsymbol{node.IsVisited = TRUE}).$$

If the node being explored has not been previously succeeded, that is, it is in one of the following states {NOT-ACTIVATED, COMPENSATED, SKIPPED, FAILED, ABORTED} and therefore no compensation action is taking place on the explored node, then the node is marked visited:

$$(\boldsymbol{CCDep}(\boldsymbol{node}) = \boldsymbol{TRUE} \wedge \boldsymbol{node.state \neq SUCCEEDED} \longrightarrow \boldsymbol{node.isVisited = true})$$

A target node remains waiting until all its source nodes have been visited. Once all the source nodes are visited, the target node is explored.

Once all the atomic nodes in the LRT are visited, a customized compensation completion event is fired for the LRT and is marked as COMPENSATED.

6.3.5 Customized Compensation Dependencies

Customized compensation dependency (Table 6.7) is defined for target nodes and root nodes.

A customized compensation dependency is fired for all root nodes when the LRT commences its compensation (CCD.1).

A target node can be a target node to one or more source nodes. Hence, a customized compensation dependency is fired for a target node iff all its source nodes have been

visited (CCD.2). A compensation completion dependency is fired for the LRT when all atomic nodes in the LRT have been visited (CCCD.1).

Dep #	Dependency	Component
Customized Compensation		
CCD.1	$\boldsymbol{CCDep(RootNode) :=}$ $\boldsymbol{LRT.State = COMPENSATING}$	Atomic Root node
CCD.2	$\boldsymbol{CCDep(TargetNode) := \bigwedge_{i=1..n} SourcList_i.IsVisited}$	Atomic Target node to n source nodes where n≥1
Customized Compensation Completion		
CCCD.1	$\boldsymbol{CCCDep(LRT) := \bigwedge_{i=1..n} node_i.IsVisited}$	LRT where n=number of atomic nodes

Table 6.7 Customized compensation dependencies

6.3.6 Customized compensation mechanism

The compensation mechanism is automated by compensation and completion policies presented in Table 6.8. We support our discussion with control charts 16 and 17.

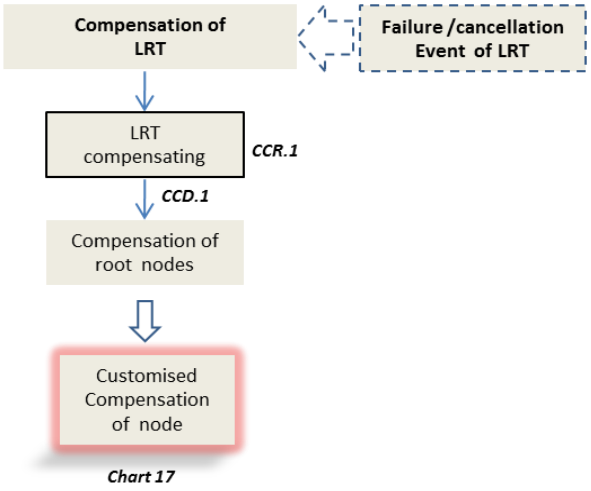
Rule#	Policy	Component
Customized Compensations		
CCR.1	***ON*** *fail(LRT)* ***IF*** *"no nodes executing(activated/compensating)"* ***DO*** *compensate(LRT)*	LRT
CCR.2	***ON*** *CCDep(node)* ***IF*** *LRT.State=COMPENSATING and node.state=SUCCEEDED* ***DO*** *compensate(node)*	Succeeded atomic node
CCR.3	***ON*** *CCDep(node)* ***IF*** *LRT.State=COMPENSATING and node.state≠SUCCEEDED* ***DO*** *node.Visited=TRUE;*	Non succeeded Atomic node
Customized Compensation Completion		
CCCR.1	***ON*** *"compensation completion event of an atomic node"* ***IF*** *LRT.state=COMPENSATING* ***DO*** *compensated(node);* *node.IsVisited=TRUE;*	Atomic compensating node
CCCR.2	***ON*** *CCCDep(LRT)* ***DO*** *compensated(LRT)*	LRT

Table 6.8 Customized compensation policies

The mechanism is automated as follows:

1- When the LRT fails, the LRT commences its comprehensive compensation process (CCR.1).

2- When the LRT starts COMPENSATING, a compensation event is fired for all root nodes (CCD.1).



Control Chart 16. Comprehensive Compensation

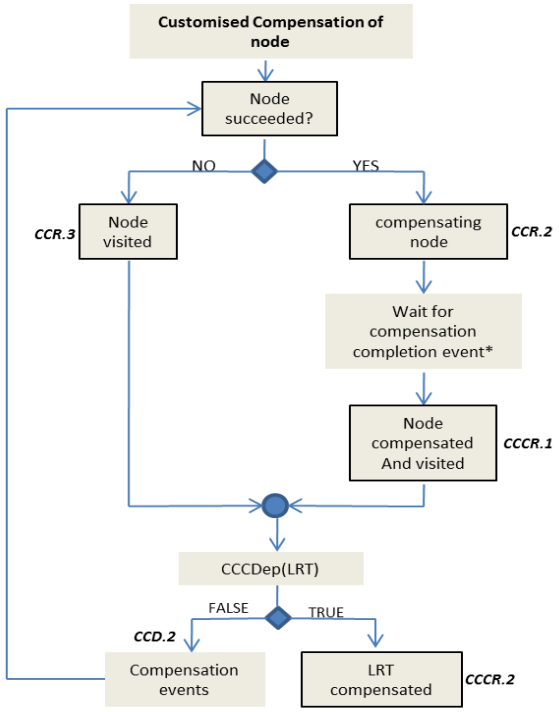
3- If a compensation event is fired for a non-succeeded node, the node is marked as visited (CCR.3) and compensation action is not performed.

4- If a compensation event is fired for a succeeded node, the node starts compensating (CCR.2).

5- When a compensation completion event is fired for a compensating node, the nodes is marked compensated and visited (CCCR.1).

6- A compensation event is fired for a target node when all its source nodes have been visited (CCD.2) and execution control goes back to step 3.

7- When all nodes have been visited, a compensation completion event is fired for the LRT (CCCD.1) and the LRT is marked as COMPENSATED (CCCR.2).



Control Chart 17. Customized compensation of atomic nodes

6.3.7 Examples

6.3.7.1 Customized Compensation Dependencies for OP Case Study

The OP workflow (Figure 4.5) does not represent any XOR patterns and therefore it is not defined with partial compensation dependencies. We refer back to the original specification of OP (Figure 2.1, Chapter 2), where the business logic requires that in case of compensation, the compensation of DELIVERY must be performed before the compensation of CHARGE. Hence there exists a compensation dependency from

DELIVERY to CHARGE. In Figure 6.5 (a), we expand the OP workflow with customized compensation dependencies. Note that by (1) allowing the business process designer to freely assign customized dependencies, and (2) providing a deadlock free dependency graph algorithm to define the comprehensive compensation order, the designer implicitly defines a deterministic compensating process schema (Figure 6.5 (b)).

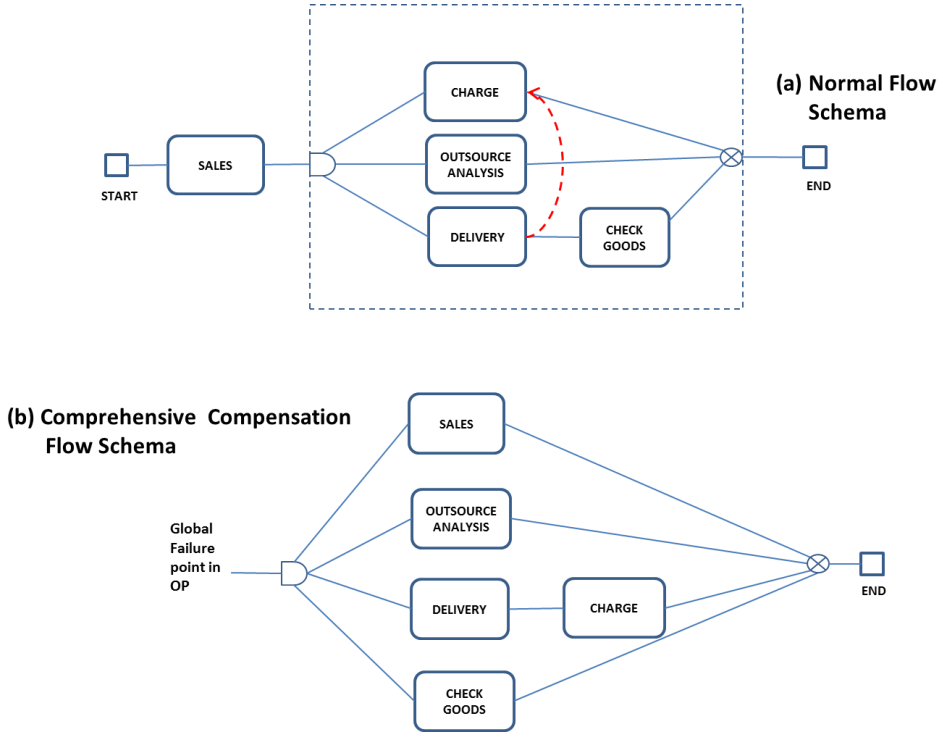


Figure 6.5 Final OP workflow schemas

When a failure declares OP failed and hence it starts compensating, the comprehensive compensation mechanism starts by compensation all activities that are not a target in any compensation dependency (the root nodes). The compensation of root activities are initiated in parallel, hence they are best represented by an AND-split like pattern in the compensating schema (Figure 6.5 (b)). Compensation of CHARGE will wait for

compensation of DELIVERY to be completed. The compensation of OP is completed once all activities are traversed and compensated/skipped depending on their execution states. The synchronization of all compensating activities fires a completion event for OP (*CCCDep(OP)=TRUE*) and it is represented in (Figure 6.5 (b)) by a synchronizer point.

In COMPMOD, comprehensive and customized compensation dependencies are only defined for atomic nodes, and therefore they are only defined for OP's activities as listed in Table 6.9. Compensation completion for atomic nodes is an internal event and thus it is not defined for atomic nodes. A compensation completion dependency is defined for OP to signal the end of compensation execution for all activities in OP.

Note that by defining transactional attributes of OP (Section 4.4), Control flow dependencies for OP (Section 5.5.), and the compensation dependencies in this section, the modeling of OP in COMPMOD has been completed.

	Customized Compensation Dependency	Customized Compensation Completion Dependency
OP	Not Defined	$SALES.IsVisited=TRUE \wedge$ $CHARGE.IsVisited=TRUE \wedge$ $OUTSOURCE\_ANALYSIS.IsVisited=TRUE \wedge$ $DELIVERY.IsVisited=TRUE \wedge$ $CHECK\_GOODS.IsVisited=TRUE$
SALES	*OP.State= COMPENSATING*	Internal Event
CHARGE	*DELIVERY.IsVisited= TRUE*	Internal Event
OUTSOURCE _ANALYSIS	*OP.State= COMPENSATING*	Internal Event
DELIVERY	*OP.State= COMPENSATING*	Internal Event
CHECK _GOODS	*OP.State= COMPENSATING*	Internal Event

Table 6.9 Customized compensation dependencies for OP activities

6.3.7.2 Comprehensive Compensation Mechanism for Sample LRT_3

We illustrate our comprehensive compensation mechanism on sample LRT_3. First, we show how nodes' customized compensation dependencies are defined, then we assume an execution scenario that fails the transaction, and finally we demonstrate how the transaction is compensated. In section 6.3.5, we have shown how the dependency graph splits the atomic nodes into root nodes and target nodes, and accordingly, decides their customized compensation dependencies. Note that ***UpdateG*** updates the source list of all target nodes such that: n_1.sourceList=[n_3], n_3.sourceList=[n_5], n_6.sourceList=[n_8], n_{10}.sourceList=[n_5,n_{13}], and n_{11}.sourceList=[n_6] where RT={n_2,n_4,n_5,n_7,n_9,n_{12},n_{13}} and TG={n_1,n_3,n_6,n_{10},n_{11}}.

Following the definitions in (Table 6.3) and mapping them onto the dependency graph in (Figure 6.4 (b)), the set of atomic nodes in LRT_3 are defined with the following dependencies:

1- $\forall\, node \in RT, CCDep(node) \coloneqq LRT.state = COMPENSATING$ (CCD.1). For example, $CCDep(n_2) \coloneqq LRT.state = COMPENSATING$.

2- $\forall\, node \in TG, CCDep(node) = \bigwedge_{i=1..n} SourcList_i.IsVisited$ (CCD.2). For example, $CCDep(n_1) = n_2.IsVisted$ and $CCDep(n_{10}) = n_5.IsVisited \wedge n_{13}.IsVisited$

In Table 6.10, we assume an execution scenario for LRT_3 components that causes the failure of the transaction and illustrate through execution state transitions and change of compensation attribute values how the compensation mechanism is applied. We assume that n_{13} was a vital node and failed and we assume the $scope_2$ is vital. Failure of n_{13} will

propagate to p_0 through scope$_2$ causing the LRT to fail. Failure of and $scope_2$ will force fail activated nodes n_9 and n_{12} and hence they are both aborted. Failure of LRT commences its comprehensive compensation. Compensating the LRT fires compensation events for root nodes where succeeded nodes start compensating while non-succeeded nodes are marked visited. Nodes that are in waiting phase wait for their source nodes to be visited. For example, n_1 waits for its source node n_3 to be compensated and hence visited and n_3 waits for n_5. Note that n_{10} waits for both n_{13} and n_5, n_{13} is visited while n_5 is still compensating. Once n_5 is compensated and visited, n_{10} can commence its compensation. Although n_{11} is aborted and no compensation action will be performed for it, but it still waits for its source node n_6 to be visited to oblige the order of customized compensations. We assume an arbitrary compensation completion events scenario for compensating nodes and show how attribute values change. The LRT is compensated when all nodes are visited.

node	LRT activated	LRT failed	LRT compensating	LRT compensating
n1	SUCCEEDED	SUCCEEDED	Waiting	Waiting
n2	SUCCEEDED	SUCCEEDED	COMPENSATING	Visited
n3	SUCCEEDED	SUCCEEDED	Waiting	COMPENSATING
n4	SUCCEEDED	SUCCEEDED	COMPENSATING	visited
n5	SUCCEEDED	SUCCEEDED	COMPENSATING	visited
n6	SUCCEEDED	SUCCEEDED	Waiting	COMPENSATING
n7	NOT-ACTIVATED	NOT-ACTIVATED	visited	visited
n8	SUCCEEDED	SUCCEEDED	COMPENSATING	visited
n9	ACTIVATED	ABORTED	visited	visited
n10	SUCCEEDED	SUCCEEDED	waiting	COMPENSATING
n11	ACTIVATED	ABORTED	waiting	waiting
n12	NOT-ACTIVATED	NOT-ACTIVATED	visited	visited
n13	ACTIVATED	FAILED	visited	visited

Table 6.10 execution instances of LRT_3

Chapter 7

Verification and Extensibility of COMPMOD

7.1 Introduction

In this chapter, we reason about the soundness of our approach; whether or not our semantics deal with executions and compensations correctly. We adopt a rule-based approach in our verification to negotiate the correctness of the model. We also dedicate a section to highlight the extensibility of our approach; that is, to show how variant semantics of workflow patterns can be modeled using the underlying infrastructure of COMPMOD.

7.2 Verification Approach

In COMPMOD, WF semantics exhibits two control flow types:

1- Forward control flow which defines the normal flow logic according to WF patterns' execution semantics such as control flow in AND scopes or sequence patterns.

2- Compensation control flow which defines two different control flows:
(a) Reverse control flow of forward flow in case of partial compensation mode. For example, compensating paths within concurrent scopes in reverse order.
(b) Customized control flow in case of comprehensive compensation mode.

To reason about the correctness of our proposed model, we apply the following verification approach:

First we verify correctness of forward flow and reverse flow using soundness properties of workflows proposed in (Van Der Aalst, 1997, 1998; Van Der Aalst et al., 2011). The works discuss the soundness properties in terms of WF-nets (type of Petri-nets) and does not capture compensation semantics. We adopt the soundness properties of (Van Der Aalst, 1997, 1998; Van Der Aalst et al., 2011) but we define them in the context of COMPMOD rule-based semantics, or more specifically, we define soundness properties based on analysis of *rule invocation graph*. In other words, we negotiate the correctness of COMPMOD rule invocations.

Soundness properties in brief are: (1) *option to complete (reachability)*: the end point of the WF is reachable for each WF execution, (2) *proper completion (consistent execution)*: if the end point is reached, all executions of tasks must have terminated in consistent final

state, and (3) *no dead transitions*: all possible WF executions must not lead to dead-lock points in the flow.

To verify properties (1) and (2), we adopt rule correctness approaches from literature such as *triggering graphs* (Aiken et al., 1992, 1995) and we set our own definitions and demonstrations. To verify property (3), we demonstrate the property by the proposed model's *construction* and *formalism*.

Second we verify the behavior of customized compensation flow by a rule invocation graph. We show how the customized order of compensations is enforced and executed correctly. As a proof-of-concept, we demonstrate this property using the OP case study in Figure 6.5

The general definition of a correct and reliable execution of control flow is:

> ***Def. 7.1: (Correct and reliable control flow execution)***
> A control flow is correct and reliable iff it satisfies the following properties:
> 1- It guarantees a consistent change of states for LRT and its constituent components.
> 2- When execution starts, a final accepted state of the LRT is reached either by successful completion or successful compensation.
> 3- If the control flow is deadlock free.

In Figure 7.1, we depict a chart to illustrate our verification approach. The discussions follow in the next sections.

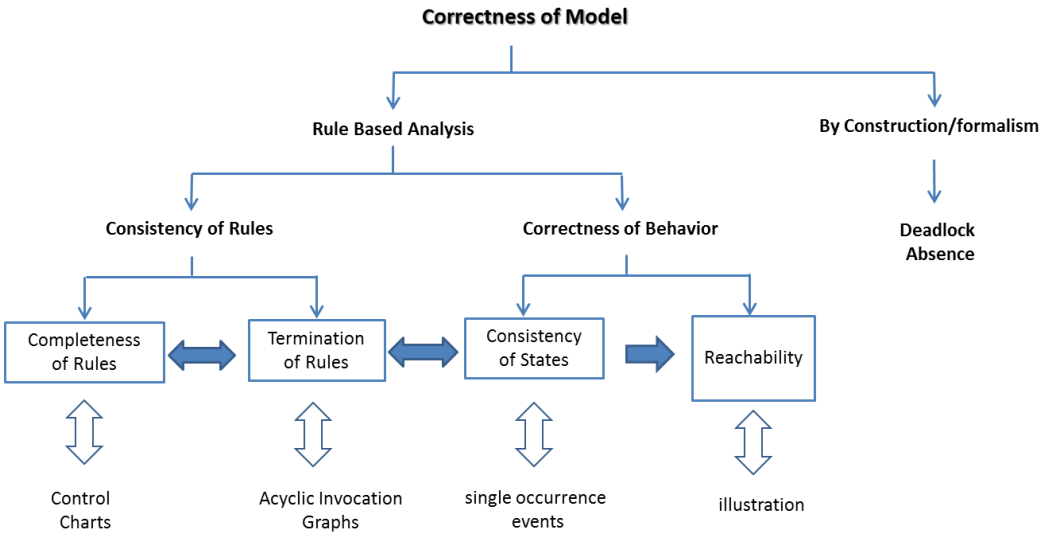


Figure 7.1 Verification Approach Chart

7.3 Soundness of WF model

As we have discussed in previous section, verifying soundness properties of our WF model implicitly verifies correctness of both forward and reverse control flows.

We represent the control flow in our workflow schema by a rule invocation graph which is defined as:

> ***Def. 7.2: (rule invocation graph)***
> A rule invocation graph is a chain of rule invocations. A rule r_i may invoke r_j if the action of r_i fires an event assessed by r_j. An invocation graph is a directed graph where each vertex is a rule and an edge exists from r_i to r_j iff r_i invokes r_j.

To illustrate, we depict in Figure 7.2 the rule invocation graph for a running sample LRT_4 that consists of two parallel nodes n_1 and n_2 in a concurrent AND scope. We assume that

both nodes are vital and n_2 fails after n_1 has succeeded. The first vertex ActR.1 is triggered by an internal event for activating the LRT. ActR.1 changes the state of LRT from NOT-ACTIVATED to ACTIVATED. Consequently, ActDep(p_0) is set to TRUE which is the event assessed by ActR.2 and therefore ActR.2 is invoked by ActR.1. Analogously, the same method applies to the rest of transitions in the graph.

Highlighted vertices denote rules that are invoked by raising internal events. However, they appear in the graph as being invoked by another rule. Note that internal events are: activation of LRT and successful completion, failure, or successful compensation completion of atomic nodes. Internal events (except for LRT activation) are immediate subsequent events for an activated atomic node. Hence, ActR.3 for an atomic node is either followed by CompLR.1 or FailR.1 and CCR.2 is always followed by CCCR.1.

Another important note is the *root rule* and *terminal rules* in the invocation graph (bold vertices). A root rule is the first rule in any invocation graph which cannot be invoked by any other rule. In our set of management rules, ActR.1 is the root rule which activates the LRT and initiates the invocation graph. On the other hand, terminal rules are rules that cannot invoke any further rules, that is, the action part of the rule does not trigger any further events in the system. In our set of rules, CompLR.3 and CCCR.2 are terminal rules which declare respectively that the LRT has either successfully completed or successfully compensated.

We chose to verify soundness based on rule analysis for the following reasons:

1- Rules relate to components and to the method by which components change their execution states. A correct and consistent rule invocation graph implies consistent change of component states.

2- Rules relate to the ordering of events and enforce constraints on their firings. Note that components are structured with arbitrary nesting, but their interrelationships are encapsulated such that components can only interact directly with their immediate neighbors or indirectly through their immediate superiors in a recursive manner. Encapsulation of execution semantics enforce a well-defined structure, and ease stepwise correctness verification, i.e. along and across LRT structure. For example, in the rule invocation graph of LRT_4, we note how ordering of activations are enforced by rules. Activation of p_0 can only happen after activation of LRT_4 and activation of p_1 and p_2 can only happen after activation of $scope_1$.

In the following subsections, we discuss soundness properties.

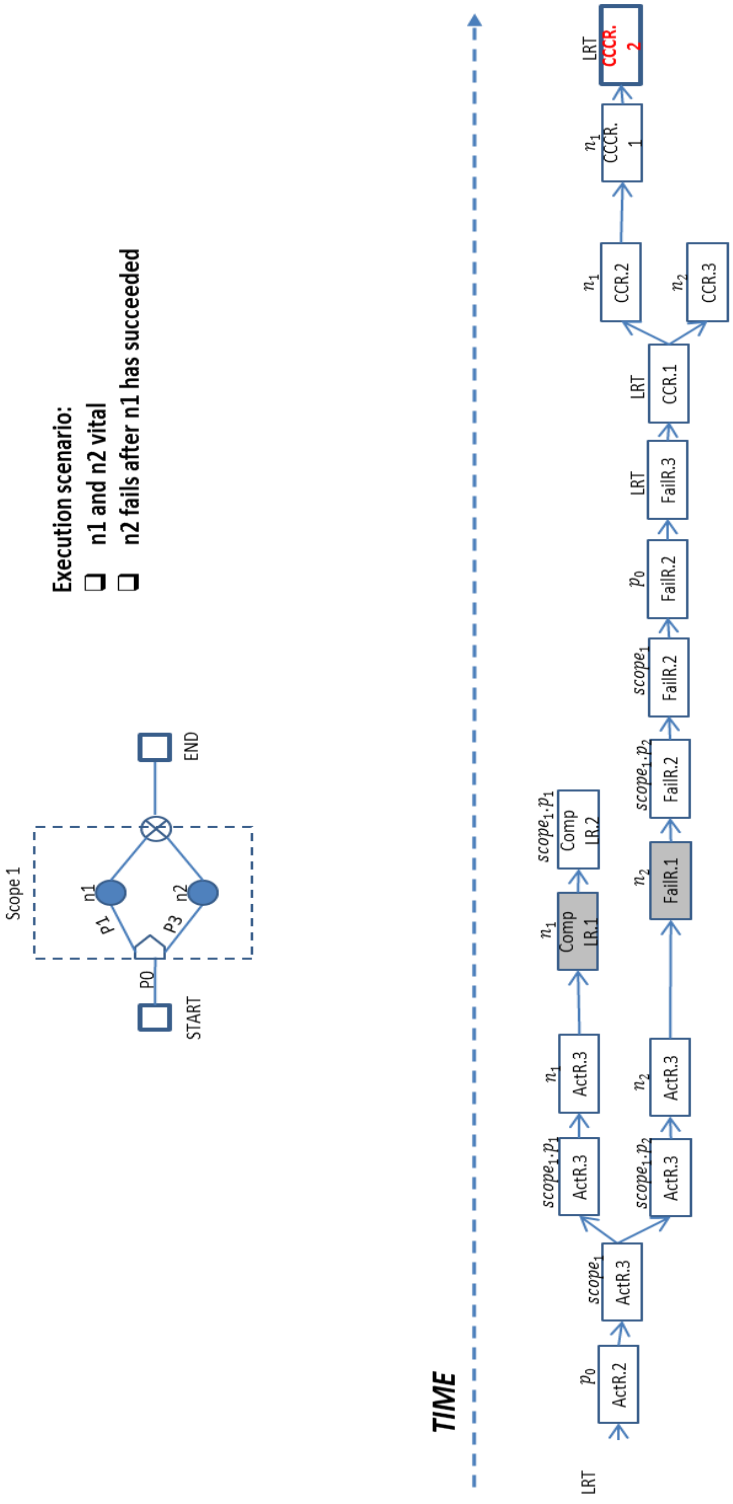


Figure 7.2 Rule Invocation Graph for sample LRT_4

7.3.1 Consistent Rule Invocation

A consistent execution of control flow depends on a consistent rule invocation graph.

> ***Def. 7.3: (Consistent rule invocation graph)***
> A consistent rule invocation graph is a terminating graph and where each rule invocation moves the state of LRT and its components from a consistent state to another consistent state.

In order to verify consistency of execution, we show two properties: *completeness* of rules and rule *termination*.

Completeness implies that every possible execution event fired by the running LRT triggers a rule, or as proposed in (Suwa et al., 1982), that there are no *missing rules* to refer to a situation that may exist in which a particular event is fired but no rule can be triggered to handle the event.

To demonstrate completeness property, we make use of the control charts listed in Chapters 5 and 6, which demonstrate the automated execution behavior of an LRT and its components throughout the LRT's execution life cycle. The complete set of rules (thirty six rules) is mapped onto the charts as a means to illustrate its completeness criteria: that is, for every possible event fired by the LRT and its constituent components, there exists a rule to handle the event. For example, (Charts 5, page 111) shows which rules may be invoked when an execution path fires a successful completion event. A path could be main path p_0, exclusive, or concurrent. For each type of path there exists a rule that handles the event. If the path is p_0, then CompLR.3 succeeds the LRT. If the path was exclusive path, then CompLR.4 succeeds its superior exclusive scope. If it was a

concurrent path, then depending on the completion and failure dependencies of its enclosing scope, the enclosing scope could either succeed by CompLR.5 or fail by FailR.7 (chart 6, page 112). Similar argument applies to the rest of control charts.

Rule termination correctness criteria guarantee that rules cannot activate each other indefinitely, or as defined in (Papamarkos et al., 2006) "A set of ECA rules is said to be terminating if for every initial event and any initial database state, the rule execution terminates". The *Triggering graphs* approach proposed in (Aiken et al., 1995) and (Aiken et al., 1992) detects non-terminating rules. We adopt their general definition of triggering graphs and implement our own verification method.

The definition of the triggering graph is same as the rule invocation graph in (Def. 7.2). A triggering graph is non-terminating if a cycle exists in the graph. A cycle exists if a rule r_i triggers/invokes rule r_j where r_j precedes r_i it in the triggering graph.

In our rule invocation graph, a cycle may occur iff an event *e* could fire more than once for the same component *c*. For example, if after n_1 in LRT4 has been succeeded; an activation event fires again for n_1 to move the state of n_1 to activated once again. Therefore, we conclude the following definition for terminating rule invocation graphs:

Def. 7.4: (Terminating Rule Invocation Graph)

A rule invocation graph is Acyclic and terminating iff an event *e* may fire exactly once for a component *c*.

Note that actions in rules change the state of components. For example, ActR.1 changes state of LRT from NOT-ACTIVATED to ACTIVATED. In order to verify the termination property, we demonstrate that "actions" in our management rules which change the execution state of components are *single occurrence* events.

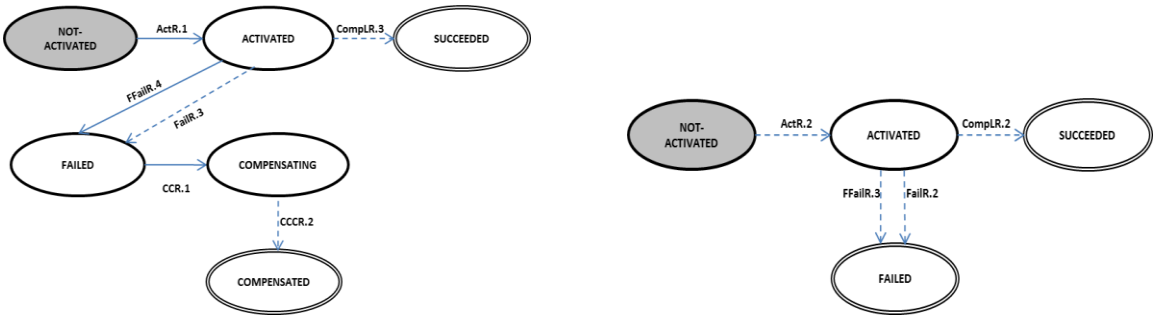
We introduce the term *single-occurrence* to refer to events that can only fire ones for each component in its execution life cycle. We show that all events in our model are single-occurrence events.

There two ways by which LRT and its components change their states. Either by *intra-event* raised for the component that causes the transition of its state (e.g. FailR.1 changes state of an atomic node from ACTIVATED to FAILED upon receiving "*failure/ cancelation event of atomic node*"). A change of state of a component may also be triggered by *Inter-events* raised by a change of state of an interrelated component (e.g. FFailR.1 changes the state of an atomic node from ACTIVATED to FAILED upon forced failure of its enclosing path). Therefore, we further classify the rules into two categories: (1) *intra-rules*: rules triggered by intra-events, and (2) *inter-rules*: rules triggered by inter-events.

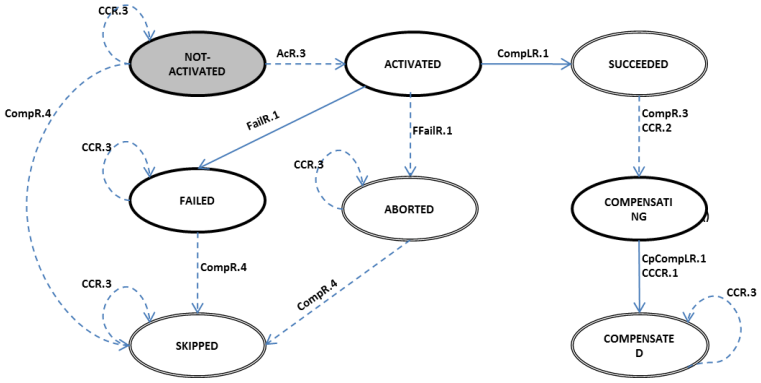
One way of demonstrating that events are single occurrence events, is to map the set of thirty six management rules on the state transition diagrams of the LRT and its components (Figure 7.3). We show that both inter-rules and intra-rules always maintain a consistent change in the state of components and interrelated components during their

execution life cycle. That is, no rule invocation moves a component to a previous state in the STDs. Therefore, the events of activations, completions, failures, compensations, compensation completions are all single occurrence events. We thus conclude that, our rules invocation graph is acyclic and terminating.

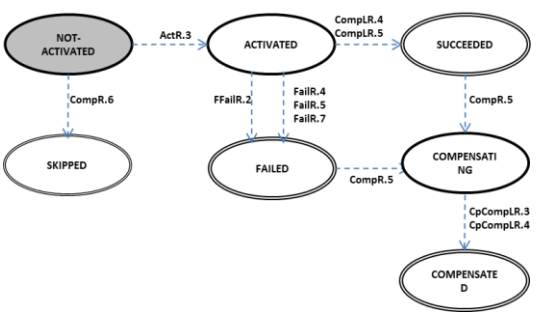
Note that in the STD diagrams, some arcs are labeled with more than one rule. However, these rules are mutually exclusive, that is only one rule may be invoked. For example in Figure 7.3 (c), two rules namely CpCompLR.1 and CCCR.1 label the transition of an atomic node from COMPENSATING state to COMPENSATED upon receiving an internal "compensation completion event" for the atomic node. The constraints in these rules are disjoint and only one will be invoked depending on state of the LRT. If the LRT was activated, then the compensation of the node is performed during partial compensation mode and CpCompLR.1 will be invoked. If the LRT state was COMPENSATING, then the compensation of the atomic node is performed during comprehensive compensation mode and hence CCCR.1 will be invoked.



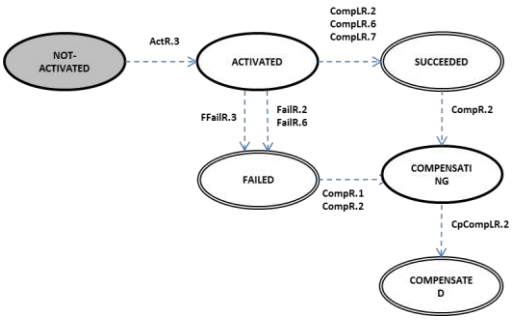
(a) LRT **(b) P_0**



(c) Atomic Nodes



(d) Scope Nodes



(e) Execution Paths

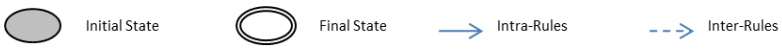


Figure 7.3 STDs with Management Rules

7.3.2 Deadlock Absence

In COMPMOD, deadlock behaviors are avoided in two ways:

1- By construction: undefined behavior of a join pattern could occur if more than one failure or completion event is triggered for the same concurrent path relating to multiple instances of the process. The structured construction of the workflow schema ensures that all path triggered events relate to the same process instance.

2- By formalism:

(a) Through deadlock free semantics of synchronizer patterns. In case of concurrent scopes, failure assumptions listed in (Chapter 4, section 4.7.2) guarantee that an execution path reaches a termination state within a finite time, no matter what. We have also shown in (Chapter 5, section 5.2.2) that in the case of OR scope, if none of the paths is enabled, then a failure event is fired for that scope.

(b) Compensation completion dependency in (Chapter 6, section 6.2.2.2) which is defined for compensable concurrent scopes, implicitly defines a synchronizer for the compensating paths. The behavioral context of such synchronization cannot possibly lead to deadlock, because a compensating path is guaranteed to complete by compensation assumption 2, in (Chapter 4, section 4.7.4). Furthermore, the compensation completion of a compensating scope does not exhibit ambiguous behavior, since each compensating path is guaranteed to complete.

(c) Customized compensation dependencies are only allowed to be defined for valid assignments that do not lead to a cycle in the customized compensation graph (Chapter 6, section 6.3.3) and thus, our comprehensive compensation mechanism provides a deadlock free compensation traversing algorithm.

7.3.3 Reachability

Reachability property guarantees that execution of LRT reaches a final accepted state. In our model, the final accepted states for the LRT are SUCCEEDED or COMPENSATED. In the context of rule invocation graphs, we can define reachability as follows:

Def. 7.5: (reachability of rule invocation graphs)

A rule invocation graph for a running instance of an LRT satisfies reachability property iff and only if it creates a connected graph where the graph starts by the root rule ActR.1 and terminates with a terminal rule CompLR.3 or CCCR.2.

By showing in the previous discussions that the set of rules are complete, and that the rule invocation graph is terminating, consistent, and deadlock free, we conclude that reachability property is satisfied.

For illustration, we refer to the rule invocation graph of LRT_4 in Figure 7.2. We have demonstrated how rule invocations have terminated with CCCR.2.

7.3.4 Proof-of-concept by Example

In this section, we verify by example a specific behavioral property of our COMPMOD semantics. We show using the rule invocation path how the customized compensation order is preserved, that is, we verify the correctness of the customized compensation control flow. We refer to our motivating case study OP in (Figure 6.5) and its compensation dependencies listed in Table 6.9.

We have assumed a failure scenario in Chapter 2 for OP process and we use the same scenario here. We assumed that vital activity CHECK_GOODS failed to succeed during which SALES, DELIVERY, and CHARGE have already succeeded and OUTSOURCE_ANALYSIS was still activated. According to the failure handling mechanism of COMPMOD, the failure of CHECK_GOODS will propagate upwards to fail OP. At the same time, activated activity OUTSOURCE_ANALYSIS will be aborted by downwards propagation.

In Figure 7.4, we depict a caption of the rule invocation graph for the running OP process. The graph starts at time t_1[19]when rule CCR1 is invoked due to the global failure event of OP and the process starts its comprehensive compensation mode. Note that in OP (Figure 6.5 (a)), there was only one customized compensation dependency from DELIVERY to CHARGE to enforce compensation of CHARGE to be performed only after DELIVERY has successfully compensated. When OP.State=COMPENSATING,

[19] Note that time labeling does not reflect real time clock. It is used for referencing arbitrary time intervals where $t_1<t_2<..<t_7$

the customized compensation dependency of SALES, OUTSOURCING_ANALYSIS, DELIVERY, and CHECK_GOODS evaluate to true (Table 6.9). Hence, CCR.2 is invoked for succeeded activities SALES and DELIVERY and they start compensating. However, for non-succeeded activities OUTSOURCE_ANALYSIS and CHECK_GOODS, rule CCR.3 is invoked which marks the two activities IsVisited and no compensation actions takes place. We arbitrarily assumed that compensation completion of SALES happens at time t_3 at which rule CCCR.1 is invoked and marks SALES.State=COMPENSATED and SALES.IsVisited=TRUE. Note that the customized compensation dependency for CHARGE still evaluates to FALSE since it waits for DELIVERY to finish its compensation. Assuming that DELIVERY completes its compensation at t_4 at which CCCR.1 is invoked and marks DELIVERY.State=COMPENSATED and DELIVERY.IsVisited=TRUE. At this point, the customized compensation dependency for CHARGE evaluates to TRUE and CCR.2 is invoked to start compensating CHARGE. CCCR.2 will only be invoked when all activities are visited and the compensation completion dependency for OP (Table 6.9) evaluates to TRUE. Therefore, we have illustrated using the rule invocation graph of OP that the customized compensation order is preserved.

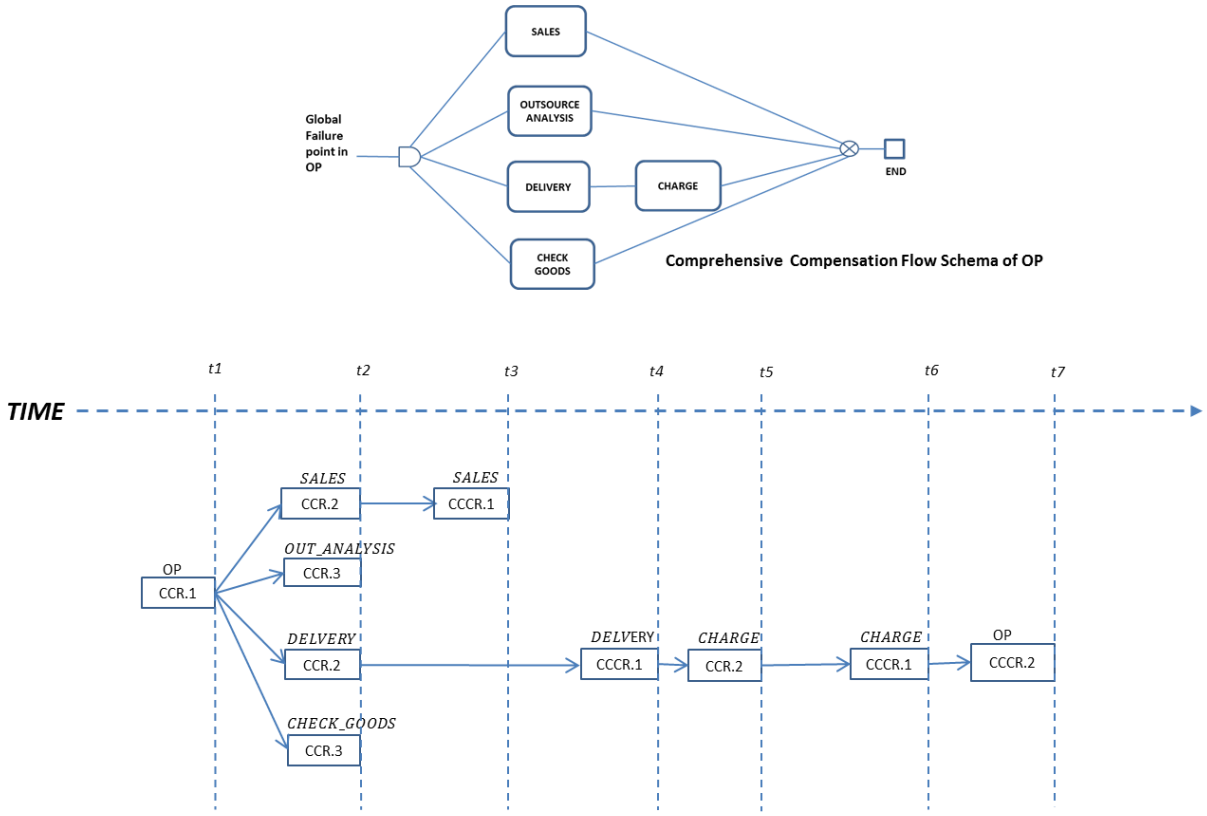


Figure 7.4 Rule invocation graph for OP

7.4 Extensibility

The model supports two types of concurrent executions through AND-scopes and OR-scopes. We provide the essential infrastructure such that the formalism of concurrency can be relaxed or extended in many different ways in order to add additional flexibility to concurrent executions or to add new concurrency semantics.

As the approach is based on a declarative (rule based) method, it is easy to implement and extend its operational semantics. For example, we can implement "the successful completion of an exclusive scope triggers a successful completion event of its enclosing scope" directly into a completion policy. Declarative formalisms add two advantages: (1)

expressing consistency conditions such as "a component can only activate if its superior is activated" and (2) provides flexibility for extending the model by adding new scope pattern semantics.

In General, in order to introduce a new scope pattern, the following steps are required:

1- A split pattern is combined with a join pattern. The informal semantics of the new scope must be stated and context issues related to possible deadlocks or undefined behavior of its join end must be analyzed.
2- The impact of vitality of constituent components on the desired operational semantics of the new scope patterns must be stated. Adding more flexibility and practicality on concurrent executions may lead to complex concurrency semantics, and would necessitate imposing additional vitality assumptions to achieve a correct consensus about successful completion and failure conditions of the new scope.
3- Formally implement the new scope's operational and transactional semantics by defining its completion and failure dependencies and defining its successful completion criteria. This may require adding new transactional attributes and possibly new management policies if the existing modeled infrastructure requires extension.
4- Some of the join patterns in workflows describe high flexibility in the semantics of concurrent executions in order to progress execution as quickly as possible. For example, such patterns may allow succeeding the pattern upon completion of a

5- specified number of execution paths while a decision is made about the remaining in-progress paths such as cancelling them or allowing them to complete without affecting the state of the join pattern. In case the pattern is allowed to succeed while there is remaining work in progress, it is important to analyze the effect of possible failures of the remaining paths and make decisions about their compensations in order to retain the overall reliable behavior of the pattern.

We illustrate extensibility criteria in the following section by showing three examples of how extensions of the model can be achieved. In section 7.4, we provide an evaluation of COMPMOD based on workflow patterns of (Russell et al., 2006).

7.4.1 Examples

In the following examples, we outline a preliminary analysis of extending COMPMOD with three variant semantics for AND scopes. As stated above, more thorough analysis is required to maintain a reliable and consistent operational semantics.

Example 1: it is possible to relax the concurrent execution of an AND-scope that encapsulates vital paths, so that the scope successfully completes when all vital paths successfully complete. Subsequent completions or failures of non-vital paths will not be assessed, and will not affect the state of its enclosing scope. In this case, the completion dependency of the scope will be defined as follows (AND^R denotes a relaxed AND-scope):

$$\boldsymbol{CompLDep}(AND^{R}) \coloneqq \bigwedge_{\boldsymbol{1 \le i \le m}} (\boldsymbol{path.isVital} \wedge \boldsymbol{path_i.State} = \boldsymbol{SUCCEEDED})$$

Accordingly, the following new policy can be added:

ON *CompLDep(scope)*
IF *scope=AND^R*
DO *succeed(scope)*

Example 2: Extend the model by a Cancelling Discriminator AND Scope (AND^{CD}). This would typically introduce a new scope pattern where the split point is an AND-split and the join end is a cancelling discriminator (Russell et al., 2006). An AND^{CD} scope successfully completes when 1 out of m paths succeeds, other activated paths are cancelled. This can be implemented by adding a discriminable attribute to execution paths where paths within an AND^{CD} have this attribute set to TRUE. The semantics of scope can be informally defined as follows:

1- When a discriminable path succeeds the scope succeeds (new policy):

ON *CompLDep(path)*
IF *path.IsDiscriminable=TRUE*
DO *succeed(path.superior)*

2- When a discriminator scope succeeds, fire a cancellation event for all its activated paths:

$$\boldsymbol{FFailDep}(\boldsymbol{DiscriminablePath}) := \boldsymbol{path.state} = \boldsymbol{ACTIVATED} \wedge \boldsymbol{PathSuperior.state} = \boldsymbol{SUCCEEDED}$$

The force fail event will be assessed by FFailR.3.

3- To prevent a cancelled vital discriminator path from failing its superior, it should be ensured as a consistency constraint that a path can either be discriminable or concurrent. Hence FailR.5 will not propagate the force failure of the path upwards to its superior.

6- Decisions have to be made about cancelled paths. For integrity of process, cancelled paths must be compensated unless the business logic requires otherwise. To compensate a discriminable path, a policy "on force-fail event of discriminable path, compensate the path" must be implemented.

Example 3: Extend the model by a Structured Partial Join AND Scope (AND^{SPJ}). This would typically introduce a new scope pattern where the split point is an AND-split and the join end is a structured partial join (Russell et al., 2006). An AND^{SPJ} scope is successfully completed when n out of m paths succeeds, other activated paths are allowed to complete but their completion will not trigger any further events for the enclosing scope. Partial join scopes may be extended with two local attributes, one attribute to specify n and one as a counter that is set to zero and incremented by one each time a successful completion of a path is triggered. The scope succeeds by applying a rule that specifies "On successful completion of a concurrent path, and if the path's superior is an AND^{SPJ} and the number of succeed paths within the scope is equal to n, then succeed path's superior".

Chapter 8

Conclusions and Future Work

8.1 General Remarks

The COMPMOD model is not anticipated to be "yet another LRT model". Our aim is to provide an underlying infrastructure that supports compensation composition, as well as service composition, in a way that is flexible in structure and in representation. In general, strict execution semantics greatly simplify the task of ensuring reliability; however, this is usually at the expense of reduced flexibility. We have attempted to balance reliability and flexibility requirements. Our aim is "what the designer wants" and not "what the formalism obliges", and in keeping a running process alive despite of recoverable failures.

In this work, we have addressed the failures and cancellations of web services, the cancellation of the LRT by a user, and cancellation of the LRT by irrecoverable failures. We have dealt with cancellation events as failures, since the cancellation of a web service or an LRT semantically means "its failure to complete its required task". Furthermore, the user is permitted to cancel a transaction at any point during its execution. However, it is possible to add extra constraints to prohibit LRT cancelation after a certain point during its execution, e.g. after goods have been shipped or after payment have been received, to reflect real world business practice.

We have shown that capturing the semantics of execution paths adds great flexibility in applying different semantics for concurrent executions, as well as for representing arbitrary nesting. A further advantage would be a relatively simple representation and formalism of the control flow. For example, some approaches supporting control flow modeling depend on using a control flow token, passing either (true tokens) as in petri-nets or (true /false tokens) as in (Weske, 2001). A true token triggers the next activity and false token for skipped activities. As for petri-nets, to manage concurrent threads of execution, "places" in the graph are used to manage token passing to join patterns. As may be seen from the examples demonstrated in (Russell et al., 2006), the graphs show complexity in representation and modeling, even for a small number of connected nodes.

There is a plethora of LRT models that address the similar problems that we address in this research, but with different emphasis, and the overlying domains and solutions we provide are comparatively different. One of the main differences between the

COMPMOD model and other LRT models is that nodes in COMPMOD are simply web services or tasks, and are not sub-transactions. Moreover, we have developed the model in a generic way that applies to any technology other than web services for representing tasks. The choice of web services was to highlight on the most loosely coupled environment where the probability of failure is high and can happen at any time during the execution of the business process.

The limitations of COMPMOD are those which are related to the assumptions made – lifting the assumptions could be investigated in the future. For example, compensation assumption 2 which guarantees a successful completion of a compensating atomic node could be lifted. In this case, further investigation and analysis must be carried out to decide what actions must be done when a compensating node fails or when a compensating transaction fails. Another limitation is the absence of loop patterns that could be realized as a necessary practice in workflow scenarios. Our proposed infrastructure provides the necessary mechanisms to apply variant semantics for concurrency and compensations as well as to loop structures and we have considered modeling loop semantics as future work.

With respect to scalability, policy driven management systems have been used in many time critical large scale systems successfully, but only industrial scale case studies would bring these to the forefront.

Lastly, we emphasize the importance of making a clear separation between failure handling and compensation handling mechanisms, to allow for better management of the overall process.

8.2 Thesis Conclusions

How to compensate a long running transaction is a challenging problem in designing infrastructures to support B2B integration. Compensation of long running activities requires correct recovery mechanisms to guarantee reliable execution.

We presented an approach for modelling 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 - which 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 throughout examples.

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, & Hofstede, 2006; Lerner, Christov, & 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 it. The presented work lays a foundation for abstracting away from specific errors and considering how failure and compensation should be handled in the situations which 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.

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 as 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.

To conclude, our approach investigates the reliability of long running transactions in a conceptual, rather than an implementation dependent way, and as consumers of these models are generally business developers, we believe that formalisms should be relatively simple to understand, express and reason about.

8.3 Future Work

For future research, we intend the following works:

1- Direct future work includes implementation of an operational system reflecting this approach and its use in some larger case studies

2- Providing more expressiveness of the compensation semantics by adding more flexibility in compensation composition by:

   (a) Enriching the semantics of partial compensations by allowing customized compensations in concurrent scopes.

   (b) Enriching the semantics of comprehensive compensations by allowing the source or a target node in a compensation pattern to be a scope node, rather than an atomic node.

   (c) Allowing scopes to be atomic by specification, such that if an atomic scope fails, it is compensated without interrupting the execution of the LRT. A necessary constraint in assigning a scope with the atomic transactional property is that the scope cannot trigger the global failure of the LRT, because otherwise it will be compensated through comprehensive compensation.

3- To aim to “keep a running process alive despite of irrecoverable failures” by supporting dynamic adaptations of web services/workflows during run time.

4- Extending the model by adding loop structures, as well as enriching concurrency semantics.

5- As the number of rules may expand by extending the model with new concurrency semantics, a rule-base model checker for the correctness criteria discussed in the evaluation section may be implemented.

The research conducted in this thesis raises interesting questions for future investigation:

1- Is it possible to design a reliable transaction management model for unstructured workflows, and how far might this benefit from the work done in transforming unstructured workflows into structured ones?
2- Given the formal semantics of individual split and join patterns, can we develop a technique to create new scopes patterns by joining splits and joins, and automatically define the semantics of the new scope and raise deadlock issues and context problems?

APPENDIX A – TABLE OF DEPENDENCIES

Dep #	Dependency	Component
Activations		
ActD.1	$\boldsymbol{ActDep(p_0) := LRT.state = ACTIVATED}$	Main path p_0
ActD.2	$\boldsymbol{ActDep(firstNode) := path.State = ACTIVATED}$	First node in a path
ActD.3	$\boldsymbol{ActDep(node_2) :=}$ $\boldsymbol{(node_1.State = SUCCEEDED) \vee (\neg node_1.IsVital \wedge node_1.State = FAILED)}$	Sequential Nodes : $node_2$ successor of $node_1$
ActD.4	$\boldsymbol{ActDep(path) := superior.State = ACTIVATED}$	1- Concurrent paths of AND scope 2- First path in XOR scope
ActD.5	$\boldsymbol{ActDep(path) := superior.State = ACTIVATED \wedge path.IsEnabled}$	Concurrent paths of OR scope
ActD.6	$\boldsymbol{ActDep(path_i) :=}$ $\boldsymbol{path_{i-1}.State = COMPENSATED}$	Paths 2..m in XOR scope where m>=2
Completions		
CompLD.1	$\boldsymbol{CompLDep(path_i) :=}$ $\boldsymbol{lastNode.State = SUCCEEDED \vee lastNode.State = FAILED)}$	Path
CompLD.2	$\boldsymbol{CompLDep(scope) :=}$ $\boldsymbol{\bigwedge_{1 \le i \le m} (path_i.State = SUCCEEDED \vee path_i.State = FAILED)}$	AND scope with ***m*** concurrent paths
CompLD.3	$\boldsymbol{CompLDep(scope) :=}$ $\boldsymbol{\bigwedge_{1 \le i \le m} (path_i.IsEnabled \wedge (path_i.State = SUCCEEDED \vee path_i.State = FAILED))}$	OR scope with ***m*** concurrent paths
Failures		
FailD.1	$\boldsymbol{FailDep(path) :=}$ $\boldsymbol{\left(\bigwedge_{1 \le i \le m} nodeList_i.nodeState = FAILED\right)}$	Non-Vital path
FailD.2	$\boldsymbol{FailDep(ANDscope) :=}$ $\boldsymbol{\bigwedge_{1 \le i \le m}(path_i.State = FAILED)}$	AND scope with m paths
FailD.3	$\boldsymbol{FailDep(ORscope) :=}$ $\boldsymbol{(\bigwedge_{1 \le i \le m}(path_i.IsEnabled \wedge path_i.State = FAILED)) \vee (ORscope.state = ACTIVATED \wedge (\bigwedge_{1 \le i \le m} path_i.IsEnabled = FALSE))}$	OR scope with m paths
Force-Fails		
FFailD.1	$\boldsymbol{ForceFailDep(node) := node.state = ACTIVATED \wedge path.state = FAILED}$ Where path=node.superior	Atomic node/scope
FFailD.2	$\boldsymbol{ForceFailDep(path) := path.state = ACTIVATED \wedge scope.state = FAILED}$ Where scope=path.superior	$path \neq p_0$
FFailD.3	$\boldsymbol{ForceFailDep(p_0) := LRT.state = FAILED}$	Main path p_0

Dep #	Dependency	Component
Forward\Backward Compensations		
CompD.1	$\boldsymbol{CompDep(lasNode) := superiorPath.State = COMPENSATING}$	Last node on a compensable path
CompD.2	$\boldsymbol{CompDep(node_1) :=}$ $\boldsymbol{superior.State = Compensating}$ $\boldsymbol{\wedge\ (node_2.State = COMPENSATED\ \vee\ node_2.State = SKIPPED)}$	A node that has a successor on a compensable path $node_2$=successor($node_1$)
CompD.3	$\boldsymbol{CompDep(\ path) :=\ superiorScope.State =\ COMPENSATING}$	Path within a compensable scope
Compensation Completion		
CpCompLD.1	$\boldsymbol{CpCompLDep(path) :=}$ $\boldsymbol{(firstNode.State = COMPENSATED)\ \vee\ firstNode.State = SKIPPED)}$	Compensable Path
CpCompLD.2	$\boldsymbol{CpCompLDep(scope) := scope.state = compensating\ \wedge}$ $\boldsymbol{(\bigwedge_{i=1..m}(pathList_i.state = COMPENSATED\ \vee\ pathList_i.state = NOT-ACTIVATED))}$	Compensable concurrent scope with m paths
Customised Compensations		
CCD.1	$\boldsymbol{CCDep(RootNode) :=}$ $\boldsymbol{LRT.State = COMPENSATING}$	Atomic Root node
CCD.2	$\boldsymbol{CCDep(TargetNode) := \bigwedge_{i=1..n} SourcList_i.IsVisited}$	Atomic Target node to n source nodes where n≥1
Customised Compensation Completion		
CCCD.1	$\boldsymbol{CCCDep(LRT) := \bigwedge_{i=1..n} node_i.IsVisited}$	LRT where n=number of atomic nodes

APPENDIX B – TABLE OF POLICIES

Rule#	Policy	Component
	Activations	
ActR.1	***ON*** *"activation event of LRT"* ***IF*** *LRT.state=NOT-ACTIVATED* ***DO*** *activate(LRT)*	LRT
ActR.2	***ON*** *ActDep(p_0)* ***DO*** *activate(p_0)*	Main execution path P_0
ActR.3	***ON*** *ActDep(component)* ***IF*** *component.superior.state=ACTIVATED and component≠p_0* ***DO*** *activate(component)*	atomic node, scope, and path≠p_0
	Completions	
CompLR.1	***ON*** *"successful completion event of atomic node"* ***DO*** *succeed(node)*	Atomic node
CompLR.2	***ON*** *succeed(node)* ***IF*** *CompLDep(node.superior)=TRUE* ***DO*** *succeed(node.superior)*	Vital and non-vital path with succeeded last node
CompLR.3	***ON*** *succeed(path)* ***IF*** *path=p_0* ***DO*** *succeed(LRT)*	LRT
CompLR.4	***ON*** *succeed(path)* ***IF*** *path.IsExclusive=TRUE* ***DO*** *succeed(path.superior)*	Exclusive scope
CompLR.5	***ON*** *CompLDep(scope)* ***IF*** *scope.state≠failed and FailDep(scope)=FALSE* ***DO*** *succeed(scope)*	Concurrent scope
CompLR.6	***ON*** *fail(node)* ***IF*** *¬node.IsVital and node.superior.IsVital and CompLDep(node.superior)=TRUE* ***DO*** *succeed(node.superior)*	vital path with failed non-vital last node
CompLR.7	***ON*** *fail(node)* ***IF*** *¬node.IsVital and ¬node.superior.IsVital and CompLDep(node.superior)=TRUE and FailDep(node.superior)=FALSE* ***DO*** *succeed(node.superior)*	non-vital path with failed non-vital last node

Rule#	Policy	Component
	Failures	
FailR.1	***ON*** *"failure/cancellation event for atomic node"* ***DO*** *fail(node)*	Atomic node
FailR.2	***ON*** *fail(node)* ***IF*** *node.superior.state=ACTIVATED and node.IsVital* ***DO*** *fail(node.superior)*	vital path (bottom-up propagation)
FailR.3	***ON*** *fail(path)* ***IF*** *path=p_0* ***DO*** *fail(LRT)*	LRT
FailR.4	***ON*** *fail(path)* ***IF*** *¬path.HasAlternative and path.IsExclusive* ***DO*** *fail(path.superior)*	Exclusive scope
FailR.5	***ON*** *fail(path)* ***IF*** *path.IsVital=TRUE and Path.IsConcurrent* ***DO*** *fail(path.superior)*	concurrent scope (bottom-up propagation)
FailR.6	***ON*** *FailDep(path)* ***DO*** *fail(path)*	non-vital path
FailR.7	***ON*** *FailDep(scope)* ***IF*** *scope.state≠failed* ***DO*** *fail(scope)*	Concurrent scope
	Force-Fails	
FFailR.1	***ON*** *FFailDep(node)* ***IF*** *node.type=ATOMIC* ***DO*** *abort(node)*	Atomic node-Propagation
FFailR.2	***ON*** *FFailDep(node)* ***IF*** *node.type=SCOPE* ***DO*** *fail(node)*	scope -Propagation
FFailR.3	***ON*** *FFailDep(path)* ***DO*** *fail(path)*	Path- Propagation
FFailR.4	***ON*** *"cancellation event of LRT"* ***IF*** *LRT.State=ACTIVATED* ***DO*** *fail(LRT)*	LRT
	Forward\Backward Compensations	
CompR.1	***ON*** *fail(path)* ***IF*** *path.hasAlternative and node.superior.state=ACTIVATED* ***DO*** *compensate(path)*	Exclusive path with alternative
CompR.2	***ON*** *CompDep(path)* ***IF*** *LRT.State=ACTIVATED and (*path.state=SUCCEEDED or path.state=FAILED) ***DO*** *compensate(path)*	Compensable path previously succeeded or failed
CompR.3	***ON*** *CompDep(node)* ***IF*** *LRT.State=ACTIVATED and node.Type=ATOMIC and node.State=SUCCEEDED* ***DO*** *compensate(node)*	succeeded Atomic node
CompR.4	***ON*** *CompDep(node)* ***IF*** *LRT.State=ACTIVATED and node.Type=ATOMIC and (node.State=FAILED or node.state=NOT-ACTIVATED Or nodeState=ABORTED)* ***DO*** *skip(node)*	Non succeeded atomic node
CompR.5	***ON*** *CompDep(node)* ***IF*** *LRT.State=ACTIVATED and node.Type=SCOPE and (node.State=SUCCEEDED or node.state=FAILED)* ***DO*** *compensate(node)*	Succeeded or failed scope
CompR.6	***ON*** *CompDep(node)* ***IF*** *LRT.State=ACTIVATED and node.Type=SCOPE and node.state=NOT-ACTIVATED* ***DO*** *skip(node)*	Not activated scope

Rule#	Policy	Component
Compensation Completions		
CpCompLR.1	***ON*** *"internal compensation completion event of atomic node"* ***IF*** *LRT.State=ACTIVATED* ***DO*** *compensated(node)*	Atomic node
CpCompLR.2	***ON*** *CpCompLDep(path)* ***DO*** *compensated(path)*	Path
CpCompLR.3	***ON*** *CpCompLDep(node)* ***IF*** *node.Type=SCOPE and node.state≠SKIPPED* ***DO*** *compensated(node)*	Concurrent Scope
CpCompLR.4	***ON*** *compensated(path)* ***IF*** *path.IsExclusive and Path.superior.state=compensating* ***DO*** *compensated(path.superior)*	Exclusive path
Customised Compensations		
CCR.1	***ON*** *fail(LRT)* ***IF*** *"no nodes executing(activated/compensating)"* ***DO*** *compensate(LRT)*	LRT
CCR.2	***ON*** *CCDep(node)* ***IF*** *LRT.State=COMPENSATING and node.state=SUCCEEDED* ***DO*** *compensate(node)*	Succeeded atomic node
CCR.3	***ON*** *CCDep(node)* ***IF*** *LRT.State=COMPENSATING and node.state≠SUCCEEDED* ***DO*** *node.Visited=TRUE;*	Non succeeded Atomic node
Customised Compensation Completion		
CCCR.1	***ON*** *"compensation completion event of an atomic node"* ***IF*** *LRT.state=COMPENSATING* ***DO*** *compensated(node);* *node.IsVisited=TRUE;*	Atomic compensating node
CCCR.2	***ON*** *CCCDep(LRT)* ***DO*** *compensated(LRT)*	LRT

APPENDIX C – An assessment of COMPMOD model based on Workflow Patterns Initiative

A workflow pattern is (1) *Fully Supported (FS)* if COMPMOD provides an explicit operational semantics for the pattern, (2) *Implicitly Supported (IS)* if the pattern's semantics are embedded in the COMPMOD's management mechanism, (3) *Partially Supported (PS)* if partial operational semantics of the pattern is supported, (4) *Achievable (A)* if the operational semantics of the pattern are applicable given the current COMPMOD's infrastructure, or (5) *Not Supported (NS)* if the current semantics in COMPMOD does not support the pattern. Sequence, split and join patterns descriptions have been listed in Chapter 3. The full description of the rest of the patterns can be found (Russell et al., 2006).

Pattern	Score	Motivation
1 (sequence)	FS	Through activation and completion semantics of execution paths
2 (Parallel Split)	FS	Through activation semantics of AND scopes
3 (Synchronization)	FS	Through completion and failure semantics of AND scopes
4 (Exclusive Choice)	FS	Through activation semantics of XOR scopes with richer semantics that allows for branches to be executes alternatively
5 (Simple Merge) 8 (Multiple Merge)	NS	The patterns are applicable on unstructured workflows to merge distinct threads of executions. However, a structured counterpart for these patterns is the XOR-join, and it is supported through completion, failure and compensation semantics of XOR scopes.
6 (Multi-Choice)	FS	Through activation semantics of OR scopes.
7 (Structured Synchronization Merge)	FS	Through completion and failure semantics of OR scopes
9 (Structured Discriminator)	A	Through similar analysis as shown in Example 2, Section 7.4.1, except that subsequent completions of discriminable paths are allowed. The decision as to how to respond to failures of subsequent completions must be made explicit.
10 (Arbitrary Cycles)	NS	The pattern is applicable to unstructured workflows, to allow for unstructured loops and iterations in the process.
11 (Implicit Termination)	FS	Through implicit termination state of components. This pattern indicates the ability to specify when a process or sub process terminates its execution, and no remaining work is expected either now or at any time in the future.
12 (Multiple Instances without Synchronization) 13 (Multiple Instances with *a priori* Design-Time Knowledge) 14 (Multiple Instances with *a priori* Design-	NS	These patterns support multiple instances of executions of the same activity (sequential or concurrent instances). The patterns describe different ways of creating multiple instances with the option of synchronizing them upon completion or not synchronizing them. These patterns are applicable in loop structures. Loop structures are planned as the future work of this research.

Time Knowledge) 15 (Multiple instances without *a priori* run-time knowledge)		
16 (Differed Choice)	A	This pattern is similar to exclusive choice but the activation of the chosen path depends on human or operating system interaction. It could be achievable in COMPMOD by allowing the activation of the chosen path to be triggered by an internal activation event instead of the path being activated by an activation dependency.
17 (Interleaved Parallel Routing)	A	This pattern allows mutual exclusion of activation of nodes on parallel paths, such that only one node on the interleaved rout can be activated at any time. This is achievable in COMPMOD by defining activation dependencies between interleaved nodes. Further analysis is required to study the impact of the pattern on partial compensations.
18 (Milestone)	NS	
19 (Cancel Activity)	PS	Through FailR.1 policy. The pattern allows for an enabled activity to be withdrawn before execution or disabled after commencing execution. COMPMOD only supports the cancelation of activated atomic nodes where cancelations are treated as failures and follow failure semantics.
20 (Cancel Case)	PS	Through force-failing mechanism of scopes. The pattern allows for cancelling a complete process with possibly executing sub processes based on user interaction. The pattern is achievable through allowing internal cancellations of activated scopes; however its full support requires further analysis.
21 (Structured Loop)	NS	This pattern describes the ability of executing an activity or sub activities a repeated number of times. Loop structures are planned as the future work of this research.
22 (Recursion)	NS	The pattern describes the ability of an activity to invoke itself during its execution or invoking an ancestor.
23 (Transient Trigger)	FS	The pattern describes the ability for an activity to be triggered by a signal from another part of the process. Triggers constitute the main concept on which COMPMOD is based on. They are referred to as transient to defer them from the next pattern, and they are transient according to (Russell et al., 2006) in the sense that they are lost if not acted on immediately by the receiving activity.
24 (Persistent Triggers)	PS	Through internal cancellation events of LRT or atomic nodes. The pattern describes the ability for an activity to be triggered by a signal from another part of the process or from the external environment. They are persistent by being retained by the workflow until they can be acted on by the receiving activity.
25 (Cancel Region)	NS	The pattern describes the ability of disabling a set of activities that are not a connected subset of the overall process model.
26 (Cancel Multiple Instance Activity) 27 (Complete Multiple Instance Activity) 28 (Blocking Discriminator)	NS	Loop structures and multiple instance activities are planned as the future work of this research.
29 (Cancelling Discriminator)	A	As discussed in Example 2, Section 7.4.1

30 (Structured Partial Join)	A	AS discussed in Example 3, Section 7.4.1
31 (Blocking Partial Join)	NS	This partial join pattern is intended for loop structures
32 (Cancelling partial Join)	A	Through similar analysis of partial join in Example 3 and cancellation analysis of undesired activated paths in Example 2 in section 7.4.1
33 (Generalized AND-Join) 34 (Static Partial Join for Multiple Instances) 35 (Cancelling Partial Join for Multiple Instances) 36 (Dynamic Partial Join for Multiple Instances)	NS	These patterns describe variations of join semantics for multiple instances. Loop structures are planned as the future work of this research.
37 (Acyclic synchronization Merge) 38 (General Synchronization Merge)	NS	These two patterns are variations of the OR-join and they are proposed for unstructured workflows. COMPMOD supports only structured workflows.
39 (Critical Section)	NS	The pattern allows for two or more connected sub graphs to be identified as critical sections, such that only one critical section can be active at any time during runtime.
40 (Interleaved Routing)	NS	The pattern allows for a set of activities to be executed once such that no two activities can be active at the same time. Execution progresses to the next step once all interleaved activities completed their executions.
41 (Thread Merge) 42 (Thread Split)	NS	Loop structures and multiple instance activities are planned as the future work of this research.
43 (Explicit Termination)	NS	As described in the reference, this pattern allows the termination of a process when execution reaches and end node even if there is remaining work in the process instance, it is assumed that the remaining work must be cancelled. The description is not clear, and we assume it is proposed for unstructured workflows.

BIBLIOGRAPHY

AIKEN, A., HELLERSTEIN, J. M. & WIDOM, J. 1995. *Static analysis techniques for predicting the behavior of active database rules*. ACM Transactions on Database Systems (ACM TODS), vol. 20, n. 1, pp. 3-41.

AIKEN, A., WIDOM, J. & HELLERSTEIN, J. M. 1992. *Behavior of database production rules: Termination, confluence, and observable determinism*. In Proceedings of the ACM SIGMOD International Conference on Management of Data.

AGUILAR-SAVEN, RUTH SARA. 2004. *Business process modeling: Review and framework*. International Journal of production economics, 90(2), 129-149.

ALI, M. S. & REIFF-MARGANIEC, S. 2012. *Autonomous Failure-Handling Mechanism for WF Long Running Transactions*. In Proceedings of SCC 2012, IEEE, pp. 562-569.

ALONSO, G., CASATI, F., H., KUNO, H & MACHIRAJU, V. 2004. *Web Services: Concepts, Architectures, Applications*. Springer.

ANDREWS, T., CURBERA, F., DHOLAKIA, H., GOLAND, Y., KLEIN, J., LEYMANN, F., LIU, K., ROLLER, D., SMITH, D. & THATTE, S. 2003. *Business process execution language for web services*. version 1.1.

BHIRI, S., GODART, C. & PERRIN, O. 2006a. *Transactional patterns for reliable web services compositions.* Proceedings of ICWE06. pp 137-144. ACM.

BHIRI, S., PERRIN, O. & GODART, C. 2005. *Ensuring required failure atomicity of composite Web services*. Proceedings of WWW05, pp 138-147, ACM.

BHIRI, S., PERRIN, O. & GODART, C. 2006b. *Extending workflow patterns with transactional dependencies to define reliable composite Web services*. Proceedings of AICT-ICIW '06; *pp.* 145-145, IEEE.

BOCCHI, L. 2004. *Compositional nested long running transactions*. Fundamental Approaches to Software Engineering, 7th International Conference, FASE 2004, Lecture Notes in Computer Sci., vol. 2984, Springer. 194–208

BROGI, A., & POPESCU, R. 2006. *From BPEL processes to YAWL workflows*. In Proceedings of the 3rd International Workshop on Web Services and Formal Methods (WS-FM'2006), volume 4184 of Lecture Notes in Computer Science, pages 107–122. Springer-Verlag, 2006.

BRUNI, R., BUTLER, M., FERREIRA, C., HOARE, T., MELGRATTI, H. & MONTANARI, U. 2005a. *Comparing two approaches to compensable flow composition.* In CONCUR, LNCS 3653, pp. 383–397.

BRUNI, R., MELGRATTI, H. & MONTANARI, U. 2004. *Nested commits for mobile calculi: extending Join*. In proc. Of IFIP-TCS'04, pp. 563-576, Kluwer.

BRUNI, R., MELGRATTI, H. & MONTANARI, U. 2005b. *Theoretical foundations for compensations in flow composition languages*. In *POPL*, pp. 209–220. ACM.

BUTLER, M. & FERREIRA, C. 2004. *An operational semantics for StAC, a language for modelling long-running business transactions*. In COORDINATION, LNCS 2949, pp. 87–104, Springer.

BUTLER, M., HOARE, T. & FERREIRA, C. 2005. *A trace semantics for long-running transactions.* In 25 Years of CSP, LNCS 3525, pp. 707-711, Springer.

CABRERA, F., COPELAND, G., COX, B., FREUND, T., KLEIN, J., STOREY, T. & THATTE, S. 2002. *Web services transaction (WS-transaction).* Technical Report, IBM developerWorks Report.

CASADO, R., TUYA, J. & YOUNAS, M. 2012. *Testing the reliability of web services transactions in cooperative applications*. In 27th ACM Symposium on Applied Computing (SAC), Trento, Italy.

CEPONKUS, A., FURNISS, P., GREEN, A., DALAL, S. & LITTLE, M. 2002. *Business ransaction protocol*. Available from http://www.oasisopen.org/committees/download.php/1184/2002-0603.BTP_cttee_spec_1.0.pdf

CHRYSANTHIS, P. & RAMAMRITHAM, K. 1990. *ACTA: A framework for specifying and reasoning about transaction structure and behavior.* In Proc. ACM SIGMOD Symp. on the Management of Data, pp. 194-203.

COLEMAN, J. 2005. *Examining BPEL's compensation construct*. In: REFT Proceedings of the Workshop on Rigorous Engineering of Fault-Tolerant Systems, Newcastle upon Tyne, UK, pp. 122–128

COLOMBO, C. & PACE G. 2013. *Recovery within long running transactions*. ACM Computing Surveys, 45(3).

CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L. & STEIN, C. 2009. *Introduction to algorithms*, MIT press.

DALAL, S., TEMEL, S., LITTLE, M., POTTS, M. & WEBBER, J. 2003. *Coordinating business transactions on the web*. Internet Computing, IEEE, 7(1)**,** pp. 30-39.

DAYAL, U., HSU, M., & LADIN, R. 1991. *A Transaction Model for Long-Running Activities*. Proc. of the 17th Int'l Conference on Very large Databases, Barcelona, Spain. pp 113-122.

ELMAGARID, A. K. 1991. *Transaction models for advanced database applications*. Morgan Kaufmann.

GARCIA-MOLINA, H., GAWLICK, D., KLEIN, J., KLEISSNER, K. & SALEM, K. 1991. *Modeling long-running activities as nested sagas*. Data Engineering, 14**,** pp. 14-18.

GARCIA-MOLINA, H. & SALEM, K. 1987. *SAGAS*. ACM International Conference on Management of Data (SIGMOD)**,** pp. 249-259.

GORTON, S., MONTANGERO, C., REIFF-MARGANIEC, S. AND SEMINI, L. 2009. *StPowla: SOA, Policies and Workflows.* In ICSOC 2009 Workshops, LNCS 4907, pp 351-362. Springer.

GRAY, J. 1978. *Notes on data base operating systems*. In Advanced Course: Operating Systems, pp. 393–481.

GRAY, J. 1981. *The Transaction Concept: Virtues and Limitations*. In Proceedings of the 7th Interna- tional Con[erence on Very Large Database Sys- tems (Cannes, France, Sept. 9-11). ACM, New York, pp. 144-154.

GREENFIELD, P., FEKETE, A., JANG, J., & KUO, D. 2003. *Compensation is not enough.* In 7th IEEE International Enterprise Distributed Object ComputingConference (EDOC), Brisbane, Australia.

HAERDER, T. & REUTER, A. 1983. *Principles of transaction-oriented database recovery*. ACM Computing Surveys (CSUR), 15**,** pp. 287-317.

JENSEN, K. 1997. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*. Volume 1: Basic Concepts. Monographs in Theoretical Computer Science, Springer-Verlag.

KHALAF, R., ROLLER, D., & LEYMANN, F. 2009. *Revisiting the behavior of fault and compensation handlers in WS-BPEL*. On the Move to Meaningful Internet Systems: OTM 2009 (pp. 286-303): Springer.

KIEPUSZEWSKI, B., TER HOFSTEDE, A. & BUSSLER, C. 2000. *On structured workflow modelling.* . Proc. Int'l Conf. Advanced Information Systems Eng. (CAiSE), volume 1789, pp. 431–445.

KOTZ, A., DITTRICH, K. & MULLE, J. 1988. *Supporting semantic rules by a generalized event/trigger mechanism.* Advances in Database Technology—EDBT'88**,** pp 76-91.

KÖNIG, D. (2006). *R26: Default Compensation Order Conflict*. from http://www.oasis-open.org/committees/download.php/21303/WS_BPEL_review_issues_list.html#IssueR26, http://www.oasis-open.org/committees/download.php/21199/Issue%20R26.ppt

LANEVE, C. & ZAVATTARO, G. 2005. *Foundations of web transactions.* In Foundations of Software Science and Computational Structures LNCS 3441**,** pp. 282-298.

LERNER, B. S., CHRISTOV, S., OSTERWEIL, L. J. , BENDRAOU, R., KANNENGIESSER, U., & WISE, A. 2010. *Exception Handling Patterns for Process Modelling.* IEEE Transactions on Software Engineering, vol. 183, pp. 162-183, IEEE.

LEYMANN, F. 1995. *Supporting business transactions via partial backward recovery in workflow management systems*. In Proceedings of BTW '95, Springer-Verlag, Berlin.

LEYMANN, F., & ROLLER, D. 2000. *Production workflow: concepts and techniques.* Prentice Hall PTR Upper Saddle River.

MAZZARA, M. & LANESE, I. 2006. *Towards a unifying theory for web services composition.* In WS-FM, LNCS 4184, pp. 257–272

MEHROTRA, S., RASTOGI, R., SILBERSCHATZ, A. & KORTH, H. F. 1992. *A transaction model for multidatabase systems*. In Proceedings of the 12th International Conference on Distributed Computing Systems (ICDCS), pp. 56–63, IEEE.

MONTANGERO, C., REIFF-MARGANIEC, S., & SEMINI, L. 2011. *Model-driven development of adaptable service-oriented business processes.* In Rigorous Software Engineering for Service-Oriented Systems. LNCS Vol 6582.Springer.

MOSS, J. E. B. 1982. N*ested Transactions and Reliable Distributed Computing*. The MIT Press, Cambridge, MA, USA.

MOSS, J. E. B. 1985. *An Approach to Reliable Distributed Computing*. The MIT Press Series in Information Systems.

MÜLLER, R., GREINER, U. & RAHM, E. 2004. *Agentwork: a workflow system supporting rule-based workflow adaptation.* Data & Knowledge Engineering, 51**,** pp 223-256.

NEWCOMER, E. & LOMOW, G. 2004. *Understanding SOA with web services.* Addison-Wesley Professional.

OASIS. 2007. *Web Services Business Process Execution Language (WS-BPEL),* Version 2.0. From http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html

ÖZSU, M. T. & VALDURIEZ, P. 1991. *Principles of distributed database systems.* Prentice Hall (Englewood Cliffs, NJ).

PAPAMARKOS, G., POULOVASSILIS, A. & WOOD, P. T. 2006. *Event-condition-action rules on RDF metadata in P2P environments.* Computer Networks, 50(10)**,** pp 1513-1532.

PAPAZOGLOU, M. P. & GEORGAKOPOULOS, D. 2003. *Service-oriented computing.* Communications of the ACM, 46**,** pp 25-28.

PATON, N. W. 1999. *Active rules in database systems*, Springer Verlag, New York, NY.

PELTZ, C. 2003. *Web Services Orestrestration and Choreography.* Computer, vol. 36, no. 10, pp. 46-52.

QIU, Z., WANG, S., PU, G. & ZHAO, X. 2005. *Semantics of BPEL4WS-like fault and compensation handling.* FM 2005: Formal Methods**,** pp 632-633.

REICHERT, M., & DADAM, P. 1997. *A Framework for Dynamic Changes in Workflow Management Systems.* In DEXA Workshop 1997, pp. 42-48, IEEE.

REICHERT, M., & DADAM, P. 1998. *ADEPT flex - Supporting Dynamic Changes of Workflows Without Loosing Control*. Journal of Intelligent Information Systems, Vol. 10; pp. 93-129, Springer.

REICHERT, M., & WEBER, B. 2012. *Enabling flexibility in process-aware information systems*. Springer.

ROBERTS, J. & SRINIVASAN, K. 2001. *Tentative hold protocol Part 1: White Paper. W3C Note,* 28 November 2001.

RUSSELL, N., TER HOFSTEDE, A. H. M. & MULYAR, N. 2006. *Workflow controlflow patterns: A revised view.* Technical Report BPM-06- 22; BPM Centre.

RUSSELL, N., VAN DER AALST, W.M.P., & TER HOFSTEDE, A.H.M. 2006. *Exception Handling Patterns in Process-Aware Information Systems.* In Proc. CAiSE'06, pp. 288-302.

SUWA, M., SCOTT, A. C. & SHORTLIFFE, E. H. 1982. *An approach to verifying completeness and consistency in a rule-based expert system.* Technical Report: CS-TR-82-922, pp. 16-21, 1982, Stanford University, Stanford, CA, USA.

THATTE, S. 2001. *XLANG: Web services for business process design*. Microsoft Corporation.

THATTE, S., ROLLER, D. 2003. *Default compensation order*. From http://www.oasis-open.org/committees/download.php/4449/Default%20Compensation%20Order.ppt

VAN DER AALST, WMP. 1997. *Verification of workflow nets*. In: Azema P, Balbo G (eds) Application and theory of Petri nets. Lecture notes in computer science, vol 1248. springer-verlag, Berlin, pp 407-426.

VAN DER AALST, WMP. 1998. *The application of Petri nets to workflow management*. Journal of circuits, systems, and computers, 8(01), pp 21-66.

VAN DER AALST, WMP., BARROS, A., TER HOFSTEDE, A. & KIEPUSZEWSKI, B. 2000. *Advanced workflow patterns.* Proceeding of Fifth IFICIS International conference on Cooperative Information systems (CoopIS'2000), Springer, 18-29.

VAN DER AALST, WMP. & TER HOFSTEDE, AHM. 2005. *YAWL: yet another workflow language.* Information Systems, 30(4)**,** 245-275.

VAN DER AALST, WMP., TER HOFSTEDE, AHM., KIEPUSZEWSKI, B. & BARROS, A. P. 2003. *Workflow patterns.* Distributed and parallel databases, 14**,** 5-51.

VAN DER AALST, WMP., VAN HEE, KM., TER HOFSTEDE, AHM., SIDOROVA, N., VERBEEK, HMW., VOORHOEVE, M., & WYNN, MT. 2011. *Soundness of workflow nets: classification, decidability, and analysis*. Formal Aspects of Computing, 23(3), pp 333-363.

WEIKUM, G. & SCHEK, H.-J. 1992. *Concepts and applications of multilevel transactions and open nested transactions.* Morgan Kaufmann Publishers Inc, pp. 515-553.

WHITE, S. A. 2004. *Process modeling notations and workflow patterns.* In L. Fischer, ed., 'Workflow Handbook 2004', Future Strategies Inc., Lighthouse Point, FL, USA., pp. 265–294.

WESKE, M. 2001. *Formal foundation and conceptual design of dynamic adaptations in a workflow management system.* In Proceedings of the Thirty-Fourth Annual Hawaii International Conference on System Science (HICSS-34). IEEE Computer Society Press, Los Alamitos, California.

WIDOM, J. & CERI, S. 1996. *Active database systems: Triggers and rules for advanced database processing*, Morgan Kaufmann Pub.

WIERINGA, R. 2003. Design methods for reactive systems: Yourdon, statemate, and the UML. San Mateo, CA: Morgan Kaufmann.

YAN, S., LI, Y., DENG, S. & WU, Z. 2005. *A transaction management framework for service-based workflow*. Proceedings of the International Conference on Next Generation Web Services Practices, pp 377-381.

ZHANG, A., NODINE, M., BHARGAVA, B. & BUKHRES, O. 1994. *Ensuring relaxed atomicity for flexible transactions in multidatabase systems*. In Proc. ACM SIGMOD, pp. 67–78.