THE UNIVERSITY OF ADELAIDE

South Australia

Formal Verification of Transactional and Configurable Service-Oriented Processes

A dissertation submitted for the degree

Doctor of Philosophy in the School of Computer Science

by

Scott Bourne

Supervised by A/Prof. Michael Sheng and Dr. Claudia Szabo

2016

ABSTRACT OF THE DISSERTATION

Formal Verification of Transactional and Configurable Service-Oriented Processes

by

Scott Bourne

Doctor of Philosophy in School of Computer Science

The University of Adelaide, *South Australia*, 2016

The industrial rise of Web services and cloud services provides ample opportunities for business processes to be implemented with third-party components in a way that is rapid to develop, low-cost, and with reduced start-up risk. *Service-oriented processes* are business processes implemented by remotely provisioning third-party services: autonomous black box implementations of common software or hardware requirements. However, executing a workflow structure of these distributed and heterogeneous components creates several transactional concerns. These include ensuring an acceptable level of atomicity over long-running executions, handling a diverse range of potential fault types, and considering the various transactional properties of component services.

In this thesis, we present three related approaches towards ensuring well-formed transactional behavior in service-oriented processes. We address the problem of identifying issues in the transactional behavior of service-oriented processes at design-time, to prevent costly issues or redevelopment at later stages. We adapt an expressive service-oriented process modeling approach that allows for developers to specify detailed transactional behavior. A set of rules can be applied to this model in order to identify transactional issues such as deadlock and invalid termination. Furthermore, developers can elicit complex and varied transactional requirements for the process with ease using our set of *temporal logic templates*. Model checking is used to ensure that process designs satisfy these rules and requirements.

Recent innovations in *cloud services* have led to the proposal of *Business Process as a Service* (BPaaS). BPaaS offers common business processes as configurable cloud services, enabling clients to perform complex or resource expensive business operations in a simple pay-be-use manner. Both service providers and clients have concerns to be satisfied during BPaaS configuration. Providers must enforce *domain constraints* to restrict the service to valid configurations, while the client has their own application-dependent requirements for the service to meet. Using *Binary Decision Diagram* (BDD) analysis and model checking as formal methods, we devise a multi-step process that identifies a BPaaS configuration satisfying the requirements of both parties.

These verification and configuration techniques have been implemented in a prototype tool called TL-VIEWS. We include six validation scenarios to demonstrate the effectiveness of our methods, using real Web and cloud services. An extensive performance analysis is performed for each model checking feature used by TL-VIEWS and the results indicate that our state-space reduction measures can decrease verification time for complex models by up to 98.63%.

Originality Statement

Scott Bourne

Selected Publications Generated from This Thesis

1. S. Bourne, C. Szabo, and Q.Z. Sheng. Managing Configurable Business Process as a Service to Satisfy Client Transactional Requirements. In *Proceedings of the 11th International Conference on Services Computing*, IEEE, 2015.

2. S. Bourne, C. Szabo, and Q.Z. Sheng. TL-VIEWS: A Tool for Temporal Logic Verification of Transactional Behavior of Web Service Compositions. In *Proceedings of the 12th International Conference on Service Oriented Computing*, Springer, 2014.

3. S. Bourne, C. Szabo, and Q.Z. Sheng. Verifying Transactional Requirements of Web Service Compositions using Temporal Logic Templates. In *Proceedings of the 14th International Conference on Web Information Systems Engineering*, Springer, 2013.

4. S. Bourne, C. Szabo, and Q.Z. Sheng. Ensuring Well-Formed Conversations Between Control and Operational Behaviors of Web Services. In *Proceedings of the 10th International Conference on Service Oriented Computing*, Springer, 2012.

5. Q, Z. Sheng, Z, Maamar, L, Yao, C, Szabo, and S, Bourne. Behavior Modeling and Automated Verification of Web Services. In *Information Sciences*, Elsevier, 2012.

6. S. Bourne, C. Szabo, and Q.Z. Sheng. Transactional Design Time Verification of Web Service Compositions. *(Under review for Transactions on Services Computing, IEEE).*

7. S. Bourne, C. Szabo, and Q.Z. Sheng. Verification and Assurance of Transactional Requirements in Service-Oriented Processes: A Survey. *(Under review for Computing Surveys, ACM).*

ACKNOWLEDGMENTS

I have been extremely fortunate during my candidature to have the backing of a great support network, both inside and outside my work environment. This thesis simply could not have been completed without all of their presence, input, and encouragement.

I am deeply grateful to my co-supervisor Dr. Claudia Szabo for the time, guidance, effort, input, and knowledge she has dedicated to me over the last few years. Under her encouraging, ambitious, and understanding supervision, I felt I could really produce my best work and grow as a researcher. Her role in improving my ability to *analyze* research work, rather than simply absorb it, cannot be understated.

From day one of my candidature, my principal supervisor Associate Professor Michael Sheng has given me valuable insight into conducting and writing research, while instilling me with the confidence to aim high. I am very appreciative for everything he has done for me over the years, and for the tremendous impact he has had on my professional development.

During my time at the School of Computer Science in The University of Adelaide, there have been many other students that have provided me with support, help, encouragement, and friendship. For all this, I want to thank Yihong, Ali, Javier, Lu, Lachlan, Kewen, Sujith, and Lijuan.

Away from academic life, I am very thankful to my parents for the unwavering support that has been behind every accomplishment I have made. Regular camaraderie from my interstate brother delivered to my inbox has been appreciated, masked in dry humour though it was. I also want to thank my friends Michael, Daniel, Nathan, Diana, Tom, Candice, Angela, and Heather. All the fun we have had together allowed me to keep going.

TABLE OF CONTENTS

LIST OF FIGURES

List of Tables

List of Algorithms

CHAPTER 1

Introduction

1.1 Motivation

Service-oriented computing, an active research area [32, 133] and adopted technology within industry [2, 98] over the last decade, has given tremendous support towards the rapid development of low-risk, distributed, loosely coupled business processes [117]. At the core of the paradigm are *services*: autonomous black-boxes software and/or hardware components deployed on the Internet or a network. Services offer implementations of commonly required functionality to customers or users (hereafter referred to as *clients*), enabling remote use in a pay-per-usage agreement. Through a service's public network-accessible interface, clients can take advantage of the offered software or hardware capabilities remotely, without having to purchase or develop their own local infrastructure.

Web services [116] began to emerge as the standard implementation of service-oriented architectures around the turn of the century. Today, there are thousands of Web services available to potential clients through online directories and dedicated Web sites. Some common usages for Web services include handling secure payments[1], providing access to map tools[2], integration with social media[3], access to search functions[4], and automating transactions with other businesses[5], among many others. Through common interface standards [39], Web services offer software components that are lightweight, easy to integrate, reusable, interchangeable, and easily interoperable. As such, related Web services, even those from different service providers, can be composed to-

[1]http://developer.paypal.com/
[2]http://developers.google.com/places/web-service/
[3]http://dev.twitter.com/overview/documentation
[4]http://api.duckduckgo.com/api
[5]http://www.fedex.com/us/developer/web-services

gether into a single process fulfilling a complex task.

The provisioning of several Web services into a workflow structure to implement a complex business process is known as *Web service composition* [120]. For example, a business process that requires searching several different web resources could be implemented as a Web service composition that guides users through each search step. Web service compositions enable the rapid development of business processes as they are outsourced software components with loose coupling and low cost [154]. Since the individual services in a composition are autonomous and loosely coupled, revising the process or replacing services as needed is simple. Significant research attention has been given to Web service compositions since the inception of service-oriented computing [126, 133], and several standards for their definition and execution have been proposed, such as the *Web Services Business Process Execution Language* (WS-BPEL) [6] and *Yet Another Workflow Language* (YAWL) [139].

In this thesis, the term *service-oriented process* is used to refer to Web service compositions and other similar service-based implementations, such as distributed workflows [112] and business process-structured cloud services [118]. Further background details on service-oriented processes can be found in Chapter 2.

A primary concern with service-oriented processes since their inception has been coordinating their execution such that an acceptable level of reliability is achieved [66]. As services are independent, autonomous, and heterogeneous, applying the appropriate fault prevention and recovery measures to achieve reliable process execution is not a trivial task [18, 138]. Below, we identify three motivational issues towards ensuring reliable and valid execution of service-oriented processes, according to both application-independent correctness criteria and application-dependent requirements drawn from business logic.

1.1.1 Reliable Transactional Service-Oriented Processes

Ensuring the reliable execution of service-oriented processes is an ongoing challenge, as they can be long-running and comprised of distributed and heterogeneous components from third parties [12, 22, 32]. *Transactional behavior*, adopted from the management of advanced distributed database management [4], has been used to coordinate service-oriented process execution in a way that handles faults and ensures an acceptable level of relaxed atomicity [18, 33, 100]. Some examples of transactional behavior applied to service-oriented processes are retrying failed component services [18, 100], partial rollback through compensating the effect of completed or failed activities [12, 53], and using alternative services or operations [12, 33], and redundantly provisioning several services to perform the same task to minimize the effect of faults [96, 103]. Moreover, dependable processes require transactional behavior that avoids undesirable states, such as deadlocking scenarios [22], and ensures execution terminates in an acceptable state [62].

Transactional behavior that is unreliable and is not verified can have serious consequences for owners and users of the process [66, 99]. For example, information gathered by the *International Working Group on Cloud Computing Resiliency* on major service outages show how even small periods of downtime for several major service providers resulted in an estimated total of $285 million loss from 2007-2012 [64]. The massive impact these outages have had on service clients, including those provisioning services to implement business processes, can only be speculated. Suspended execution of service-oriented processes can lead to a disruption of business operations, monetary loss, violation of internal policies, or a damaged consumer confidence. If faults are not correctly handled, processes may terminate in an invalid state, such as leaving inconsistent data across sites.

Ensuring that transactional behavior minimizes or avoids the impact of Web service component faults can be difficult due to complex business process models, unpredictable third-party service behavior, and unexpected outages of communication networks or services [29, 65]. Many existing approaches aim to automatically handle faults at runtime [30, 97, 137], but this removes transactional behavior from the developer's control, which may cause unexpected process behavior or requirement violations. Furthermore, identifying and resolving these issues early in the development cycle, such as at *design-time*, can avoid costly redevelopment or unexpected faults during execution.

1.1.2 Transactional Requirements Compliance in Service-Oriented Processes

Another concern for the developers of service-oriented processes is that their own application-specific requirements over the transactional behavior are not violated [33, 57, 111]. These *transactional requirements* concern the fault-handling behavior and are specific to each process. For example, a service-oriented process for handling online payment requires different responses for faults that occur during credit card payment and faults during basic read operations. Furthermore, the developer may consider some component operations more critical to the success of the process, and thus require more substantial fault handling. An active body of research in service-oriented processes is dedicated to ensuring transactional requirements, specified in ways such as as valid termination states [18, 62], critical and non-critical component services [12, 100], and whether compensation needs to occur [53, 81], or temporal logic properties [87].

Violating transactional requirements can result in serious consequences such as mismanaged funds, or violated internal policies [87]. However, ensuring transactional requirements is challenging. Firstly, transactional requirements must be expressed in a language that allows for formal methods or machine interpretation. This specification can be error prone, because it requires expertise in a formal language, and requirements can be very complex [23, 59]. Secondly, as Web services are heterogeneous, autonomous, and loosely coupled, requirement compliance can be impeded by the unpredictable behavior such as outages or updates [18].

Several approaches have addressed this issue by nominating a specification method for transactional requirements, and automating or verifying the service-oriented process accordingly [32]. Common issues with specification methods have are difficulty scaling to large process models, being unable to expressive complex requirements, or being error-prone for developers to use [23]. For example, *Acceptable Termination States* (ATS) models [83] allow a developer to exhaustively define how a process may terminate, but becomes laborious to implement or revise as the size of a process grows. However, simpler methods such as binary variables [53] or selecting critical components [12, 100] are unable to formalize complex requirements, such as fault-handling for certain operations. Using formal languages like *temporal logic* [56] for transactional requirements [87] allows for complex and varied properties, but requires expert knowledge in the language. In summary, current approaches have so far been unable to support component-level and process-level transactional requirements in a way that greatly reduces the capacity for human error. They also suffer from state space explosion when applying formal verification to large models.

1.1.3 Business Process as a Service Configuration

In recent years, *cloud services* have become a popular paradigm for offering a diverse range of utilities over the Internet, such as software applications, storage, and computing capacity [154, 159]. Cloud services differ from traditional Web services as they offer a wide variety of software, platform, and infrastructure service types, are highly scalable to changes in workload, and make efficient use of pooled virtualized resources [7, 49].

Business Process as a Service (BPaaS) [118, 121, 142] is an emerging type of cloud service that offers business processes as a remote executable service for clients. The aim of BPaaS providers is to target a large potential market by offering business processes that are common, proven, or require complex or expensive resources to execute. Clients benefit from using BPaaS as they can execute business processes in a pay-by-use agreement, rather than obtaining and developing the necessary hardware, software, or other infrastructure. A BPaaS can be developed by mashing up external third-party services to fulfil process tasks or roles, such as other types of cloud services, Web services, and local software and hardware [25, 118]. For example, a BPaaS could handle similar payment operations or document processing tasks for a diverse range of business clients.

A key property of cloud services is *configurability* [129, 146], which allows services to be personalized to meet the requirements of individual client. Configurable service properties can be functional or non-functional, such as *Quality of Service* (QoS) guarantees, service structure, user interface, auxiliary features, and underlying resources [114, 143]. From the service provider perspective, configurability allows services to reach a larger market, and provide financial benefit through economies of scale. Managing the process of configuring cloud service to satisfy client requirements has been an area of research interest in recent years [118, 129, 142].

Managing BPaaS configuration remains a largely open issue in research. Clients should be able to configure BPaaS from numerous perspectives, such as removing activities, reprovisioning resources or services, or selecting data objects vital to the process [94]. Furthermore, as the complexities of BPaaS are hidden from clients, there must be a guarantee that the transactional behavior of the process adheres to the client's requirements. Without establishing this trust, there is great risk for business clients outsourcing entire processes of the internal operations to BPaaS providers. At the same time, if BPaaS providers were able to provide a formal guarantee of transactional requirements provided by prospective clients, it would make their services much more attractive. At present, this challenge hasn't been addressed in BPaaS research.

1.2 Goals

The above issues lead us to target three main objectives:

Uncovering and resolving transactional behavior issues in service-oriented processes at design-time

Firstly, we will aim to develop a design-time verification method for service-oriented process that allows developers to ensure reliable transactional behavior. This will require a formal model for service-oriented processes that includes a detailed view of the transactional behavior. Rules or correctness criteria that express *well-formed* transactional behavior must also be defined. Finally, an appropriate use of formal methods for verification will allow transactional design issues to be identified in the model. The end result will be an easy-to-use verification process and its associated tool that helps developers find and resolve critical problems, such as deadlocking or incomplete transactional behavior, before starting development.

Formalizing complex transactional requirements for the verification of service-oriented processes

Secondly, we aim to create a design-time verification approach that can elicit application-dependent transactional requirements from developers, in order to verify service-oriented processes against them. This method will include a requirement specification method that is easy-to-use, capable of both simple and complex requirements, and scalable to large process models. This will enable developers to verify complex transactional behavior before development, and will provide greater flexibility and scalability over state-of-the-art approaches, through the breadth and depth of behavior coverage offered by our requirements.

A BPaaS configuration method that preserves provider constraints and client transactional requirements

Finally, we will develop a configuration process and associated tool for clients to use when provisioning a BPaaS. This process will ensure the satisfaction of transactional requirements and feature selections from the client, while preserving domain constraints over configuration. The BPaaS will be able to be configured from numerous perspectives, including activities, resources, and data objects. Therefore, it requires an expressive model of configurable and transactional BPaaS, formalization of configuration constraints, and formal methods capable of handling large and complex models in a timeframe reasonable for clients. Ensuring transactional requirements will be able to provide clients with a greater level of trust to outsource business operations to BPaaS. Our model will also be able to offer high configurability, to increase the potential market of the service.

1.3 Contributions

In this thesis, we present work providing the following contributions to research in transactional service-oriented processes:

An expressive design-time model for transactional service-oriented processes

Our service-oriented process model [22, 23] uses statecharts [74] and provides separate views of the transactional and functional views of the process behavior. This allows each perspective to be designed and modified by domain experts, and enabling formal methods to be applied for verification.

Rules for well-formed transactional behavior

In [22], we propose a set of rules that can be applied to our service-oriented process model in order to verify that the transactional behavior is *well-formed*. These rules ensure that the transactional behavior of the process always i) initiates correctly, ii) avoids deadlocking scenarios, and iii) terminates in a valid state. These rules can be expressed in *temporal logic* [56] for formal verification.

Temporal logic templates as a transactional requirement specification method

We develop a set of *temporal logic templates* [52] to enable formalization of simple or complex transactional requirements over service-oriented processes [23]. These templates do not require expertise in any formal languages to use, and can be used in verification. Compared to existing specification methods, the template set is easy-to-use, scalable for large process models, and capable of expressing a diverse range of transactional requirements.

A BPaaS modeling approach that combines transactional behavior, domain constraints, and configurable activities, resources, and data objects

At the core of our BPaaS configuration process [25], we propose a modelling approach that incorporates all these necessary properties in a way that allows formal methods. *Business Process Modelling and Notation* (BPMN) [156] is utilized in conjunction with statecharts, *feature models* [82], and *Binary Decision Diagrams* (BDDs).

A BPaaS configuration process that preserves service provider domain constraints and client transactional requirements

We compose a multi-step configuration process for BPaaS that applies BDD analysis for ensuring domain constraints, and model checking with temporal logic templates for verification against client transactional requirements [25]. This allows clients to configure a BPaaS from activity, resource, and data object perspectives, while ensuring the process satisfies their requirements.

A prototype tool for verification of service-oriented process designs and BPaaS configuration

Our proposed verification and configuration approaches are implemented in a Java-based prototype tool called TL-VIEWS [24]. The prototype integrates a statechart modelling programs with formal verification tools for model checking and BDD analysis. It contains interfaces for process modelling and verification, and for client-driven configuration of BPaaS.

1.4 Thesis Outline

The remaining chapters of this thesis are organized as follows:

- **Chapter 2** contains a summary of the background for transactional service-oriented processes. We discuss how transactional management of service-oriented processes has built on concepts from advanced database transactions, namely, the compensation-enabled SAGAS model, nested transactions, and advance models that apply transactional properties to local managers. Then, we provide background on the transactional requirements and possibilities of Web service compositions, including standards, benefits, and the challenges in ensuring transactional requirements. We also discuss *cloud services*, which is a type of service that has risen in prominence in recent years, offering a wide range of utilities from storage, to development environments, to software. Finally, we introduce BPaaS as a novel form of cloud service and explain why transactional behavior is an important management factor.

- **Chapter 3** details our approach for ensuring well-formed transactional behavior in service-oriented processes at design-time. Our analytical survey of related work identifies a need for design-time verification to identify issues in transactional behavior. We introduce our modelling approach for service-oriented processes, which allows separate views of transactional and functional behavior. These models interact using a series of messages, which allows them to pass instructions and remain consistent with each other's status. The messages represent the transactional behavior of the process, and we propose a set of rules that can be applied to this model to make sure this behavior is well-formed. We apply temporal logic and model checking for verification, and use a state space reduction algorithm to address the state space explosion problem inherent in model checking.

- **Chapter 4** focuses on ensuring application-dependent transactional requirements at design-time. From an in-depth analysis of existing work, we show that transactional requirement specification methods so far applied to service-oriented processes have had issues with expressibility, scalability, and usability. We propose our requirement specification method, namely, using temporal logic templates for formalization and design-time verification. Complex and varied transactional requirements can be specified using our template set by assigning simple variables. Developers are not required to have expertise in temporal logic. Our template set allows the specification of transactional requirements applying to single components or the entire process. Model checking is used for verification, with a state space reduction method to increase verification performance with complex process models and large sets of transactional requirements.

- **Chapter 5** presents our configuration approach for BPaaS. From analyzing the state-of-the-art in both configurable cloud services and configurable business processes, we identify that configuring BPaaS while preserving transactional requirements and a valid structure remains an open issue. For modeling configurable BPaaS, we adapt *Business Process Modeling and Notation* (BPMN) with our separation of behaviors model for service-oriented processes, while including mapping of configurable resources and data objects. Feature models are adapted from the software product line engineering domain for representing domain constraints over valid configurations. Our configuration process combines the temporal logic templates and model checking of our design-time verification approach with *Binary Decision Diagram* (BDD) analysis for constraint checking.

- **Chapter 6** contains the details of our prototype tool, and the results of validation and performance tests. We provide an overview our prototype tool architecture, with detailed descriptions of each module. The implementations of the modeling, verification, and configuration approaches of Chapters 3-5 are explained. A set of six validation tests with real world services are conducted to demonstrate the effectiveness of our approaches. Finally, each function that uses model checking for verification undergoes a thorough performance evaluation with large models and property sets, in order to verify the effectiveness of our state space reduction measures.

- **Chapter 7** contains a concluding summary of the work presented in this thesis, as well as identified directions for future research. We revisit each goal stated above, and discuss how we implement them in the approaches detailed in Chapters 3-5, and the prototype tool in Chapter 6. We several extensions that would allow our work to make further contributions towards transactional service-oriented processes. These are the diagnosis of requirement violations identified by model checking, enabling dynamic BPaaS configuration in a way that preserves client transactional requirements and domain constraints, and a BPaaS configuration framework that would realise our approach with deployed services.

CHAPTER 2

Background

This chapter provides the background and historical perspective of the research areas relevant to this thesis. Our work contributes towards the domains of Web service composition and cloud services, while building on transaction concepts first proposed in database management. This chapter will provide the necessary background in each area. Below, we first provide a historical overview of *advanced transaction models* for complex distributed database transactions, and highlight features relevant to the service-oriented process domain, namely, backwards recovery measures, nested transactions, and transactional properties for heterogeneous resources.

Secondly, we discuss how service-oriented computing has affected the realization of repeatable business processes, by enabling Web services to be composed together to fulfil complex tasks. Web service composition provides new opportunities to transactional management of business processes, such as service replacement and late binding, but also challenges in ensuring transactional requirements over distributed components. We also provide a brief overview of standards that have been proposed for Web service composition definition, execution, and coordination.

Finally, we provide an overview of cloud services, and give particular focus to *Business Process as a Service* (BPaaS), a type of service still in early stages of research. Cloud services have emerged in recent years as delivery method for a wide range of utilities, such as storage, database management, computing capacity, Web application hosting, and software. Distinctive cloud service properties such as configurability, multi-tenancy, and elasticity provide benefit both service providers and clients. While the existing cloud service architecture contains *Infrastructure as a Service* (IaaS), *Platform as a Service* (PaaS), and *Software as a Service* (SaaS), BPaaS has emerged in recent years as a prospective fourth layer. BPaaS providers deliver business processes as cloud services, by potentially mashing up services from numerous third-party providers.

2.1 Advanced Transaction Models

Database transaction management, in general, aims to control read and write access to data resources in a way that maximises their reliability and availability [12]. Many transaction models and protocols that have been proposed to accomplish this for distributed and complex systems have been extended or adapted towards managing distributed business process execution, in order to guarantee or improve their correctness and completeness [4, 47, 72]. *Traditional* transactional management [70], from which more advanced models have been derived, targets the four *ACID* properties, namely, *atomicity*, *consistency*, *isolation*, and *durability*:

Atomicity: A transaction must either complete in full, or not at all. The transaction can never be left in an incomplete state, and failed transactions should leave no impact.

Consistency: Transactions must leave all involved databases or resources in a valid state, according to any local constraints applicable to those databases.

Isolation: This property prevents any concurrently executing transactions from affecting each other. In other words, the same resulting database state would be obtained if transactions upon it were executed concurrently or sequentially.

Durability: The effect of a transaction must be permanent once it has been committed. The result of the transaction is preserved and not compromised by crashes or failures in the system.

ACID transactions are desirable because they have guaranteed failure atomicity. A series of ACID transactions also have serializability [15] in their scheduling, which means they can occur in any order and will leave data resources in the same state. However, the ACID properties are not always practical for complex business operations, and may need to be relaxed in a controlled manner [40, 66, 67]. The reasons for this include:

- Business processes can require a nested structure of transactions, where the process can be seen as a parent transaction whose success is dependent on a set of sub-transactions [55]. In these scenarios, the atomicity of the process is not trivial to ensure. Traditional ACID transactions are not flexible enough to handle partial failures of composite transactions, when some *sub-transactions* have succeeded while others have failed. Partial failures can be acceptable in a business process scenario, as long as faults are handled in a way that satisfies the requirements of stakeholders [18]. For example, the failure of some minor tasks may not require the entire process to be aborted, so long as the failures are properly compensated.

- A transaction with strict isolation requires *locks* to be placed on the resources it uses, which prevent other transactions from reading or writing to them until the first transaction completes. This is not feasible for many long-running transactions, as they create unacceptable delays for other transactions executing concurrently [67]. For example, a booking system in use by one client cannot lock out all others without causing unacceptable outages and delays.
- Business processes can require the coordination of multiple heterogeneous partners or services, such as payment services, database resources, and systems serving different functions within the business. These components can have varying transactional requirements and interaction patterns [59]. This is difficult to handle with traditional transaction models, as resources cannot be handled in a uniform manner [18].

To address these concerns, several *advanced transaction models* were proposed. These models were developed to handle composite transactions in distributed multi-database systems over potentially long-running execution times. We highlight a selection of influential advanced transaction models to discuss the significant innovations they contribute towards service-oriented processes.

Nested transactions [113] were proposed to handle composite transactions or tasks in distributed systems. Indivisible work units are labelled as sub-transactions, which can be nested by arranging into a hierarchical structure expressing dependencies. This allows for a transactional manager to treat sub-transaction failures in ways that prevent the whole process from being aborted, such as in simple single-level transactions. Further extensions into *multilevel* transactions [155] allow for management of sub-transactions composed into more complex tree structures. Multilevel transactions are designed to execute on layered system architectures, such that each level in the transaction tree corresponds to work units in abstraction layers in the system.

The *SAGAS* transaction model [67] addresses the problem of managing long-running processes composed of multiple transactions with some level of atomicity. SAGAS executes nested long-running transactions without locking resources. Instead, completed sub-transactions can be undone or reversed by associating each with a *compensation* operation. Each compensation operation removes the effect of a completed sub-transaction, such as removing a record from a database. When a fault occurs, the associated compensation operations of committed sub-transactions are executed in the reverse order of completion to rollback the progress of the process. The rollback halts upon reachign a pre-defined *save-point*. These save-points in the process can be specified by the designer, to indicate that execution may resume once rollback has reached a certain point. For example, a set of closely-related sub-transactions can be recovered and re-attempted as a group any time a single one fails by specifying a save-point directly before them.

Another notable innovation is the management of nested transactions that utilize heterogeneous resources for sub-transactions. In [107], a *transactional property* for each sub-transaction is obtained from the local manager of each resource. These properties classify sub-transactions as *retriable* (no consequence for failure and may be retried), *compensatable* (has an associated compensation operation, such as SAGAS sub-transactions), or *pivot* (neither retriable or compensatable). This enables forward and backward recovery to be applied to an in-progress nested transaction as appropriate given what is permitted by transactional properties. Nested transactions with transactional properties can also be analyzed for correctness criteria. For example, at most, only *one* pivot operation should be used during execution, otherwise an unrecoverable state may occur when a fault occurs between two pivot operations [149].

Concepts from these advanced transaction models have since been applied to standardization [27] and research [12, 18] efforts for transactional management of service-oriented processes. However, as they have been developed with a focus on complex database systems, their design neglects many features of workflows and service-oriented processes [4], such as using several services to implement a task to increase reliability or provide alternatives, and managing both activity and data flow.

2.2 Web Service Composition

The rise of service-oriented computing [116, 133] over the last 15 years has had a large impact on how businesses conduct common and repeatable software operations. *Web services* are independent and reusable black-box software entities publicly published on the Internet with discoverable interfaces. *Clients*, in the form of applications, users, or other services, can remotely invoke Web services through a public service interface defined by the *provider*. When invoking a Web service, clients include input parameters, which are translated by the provider into inputs for the underlying software. The service returns output to the client once the request has been processed, such as confirmation, error notifications, or more complex data schemas.

Common interface standards help make Web services highly re-usable and interoperable. *Web Service Descriptive Language* (WSDL) [39] descriptions are a common XML-based format for Web service interfaces, which contain the parameters, outputs, and technical details for invoking the service. *Representational State Transfer* (REST) [60] APIs are an alternative frequently used for Web services with simpler communication requirements. RESTful Web services are called by URIs that invoke methods with HTTP verbs such as GET, DELETE, and PUT. These lightweight interfaces can be more appropriate for services with basic inputs and less security requirements.

Web services have been developed and deployed to provide a wide variety of functionalities from accounting operations, to domain traffic analytics, to image, file, or text processing. Even social networks such as *Twitter*[1] and *Facebook*[2] have public Web service APIs to provide software developers with access to their search functions or application platforms. At present, thousands of Web services are accessible to potential clients through service directories such as *ProgrammableWeb*[3].

The benefit of Web services from a client perspective is that they can lower development cost and risk, while meeting some software requirements. The pay-per-use model of provisioned Web services can be more attractive for small or medium businesses than buying or developing private software and infrastructure. Common Web service interfaces also make integration with internal

[1]https://dev.twitter.com/rest/public
[2]https://developers.facebook.com/
[3]http://www.programmableweb.com/apis/directory

systems simple. For providers, deploying Web services makes their software products easy to use and readily available to a large market. Furthermore, Web services' versatility means the same piece of software can be used by applications, human users, and various platforms with no changes required to the service itself.

A *Web service composition* is an aggregation of Web services in a workflow structure that together perform a complex task [120, 132]. The component Web services in a composition can be from several service providers, such that each provides some necessary functionality towards a common goal. For example, a business process for handling payment may contain transaction managment Web services from numerous providers for users to choose from, in addition to other Web services for accounting. Several languages have been developed in order to implement Web service compositions, with *WS-Business Process Execution Language* (WS-BPEL) [6] emerging as the de-facto standard. WS-BPEL supports business processes using basic programming constructs, such as conditional statements, sequences, while loops, and parallel execution. Another key feature is that *late-binding* can be used to automatically decide which concrete version of a WS-BPEL process is to be used until runtime, such as selecting which Web services to invoke.

There are two common perspectives for Web service composition: *Web service orchestration* and *Web service choreography* [120]. An orchestration is an ordered, executable process of provisioned Web services from a *single* perspective. For example, an internal business process managed by a single department, but using external Web services for some tasks, is an orchestration. A choreography is a *multi*-perspective view of how two or more complex parties interact with each other through Web service invocations. A choreography can be seen as two or more related orchestrations or stateful services, each with their own local manager, that require several points of interaction between each other. In our work, we limit our focus to Web service orchestration, as we aim to address transactional issues in single perspective service-oriented processes.

Figure 2.1 shows an example Web service composition for a business to send email or postal mail content to a group of customers, such as promotional details or newsletters. The composition uses Web services from the *Retain.cc*[4] API for customer relationship management and sending

[4]https://retain.cc/api.html

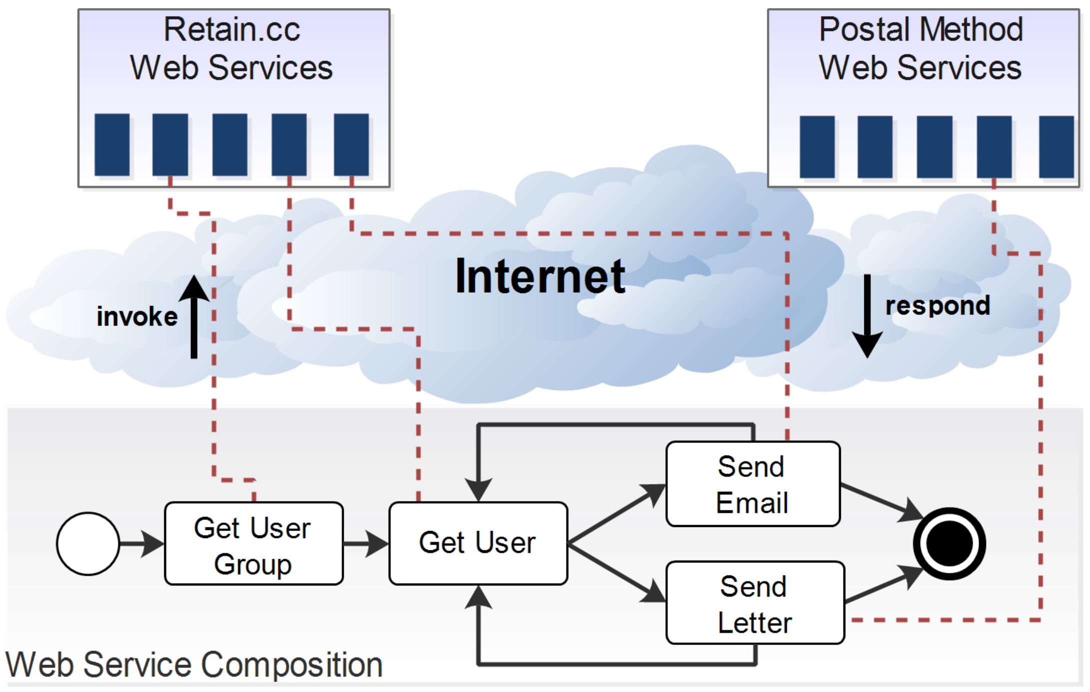


Figure 2.1: A Web service composition for sending email or postal mail to a group of customers

emails, and a Web service from *PostalMethods*[5], which sends postal mail of input documents. The composition retrieves a group of users, and then for each entry, obtains the user's contact details, and sends an email or letter. At each task implemented by a Web service, the provisioned service is invoked by the composition engine sending a request along with input. The response from the Web service contains the output, which could be used in ways such as data input for another service in the process, confirmation of a task complete, or input for a local system.

Using Web service compositions for business process execution has several domain-specific challenges. For instance, as Web services are third-party black box entities, verification or analysis of Web service composition behavior is limited [26, 50, 62]. Furthermore, the execution of Web service compositions is dependent on network, hardware, and software reliability issues outside of the client's control, such as Internet connection integrity, and Web service outages [61, 103]. Ensuring transactional requirements or correctness in Web service compositions is also a challenge, as we discuss below.

[5]http://www.postalmethods.com/postal-api

2.2.1 Transactional Web Service Compositions for Business Processes

Implementing and executing business processes as Web service compositions creates new challenges in ensuring transactional requirements, such as:

- Managing the heterogeneous interaction requirements or transactional protocols of component services [12, 18, 59]. A web service composition may require component services from providers that have interaction requirements or local transaction protocols that are strict, relaxed, or non-existent. Furthermore, as services implement a variety of software operations, they cannot all be considered retriable or recoverable upon failure, and hence cannot be handled in a uniform manner.

- Ensuring or verifying the correctness of the composition with respect to transactional requirements drawn from business rules. These requirements could include component services whose success is considered critical for the composition to commit [100], acceptable failure conditions[18, 111], or formalized behavioral properties [59].

- Handling the several types of faults that Web services are vulnerable to. Behavioral faults occur when Web services return an error notification as output, while outages in the network or Web service infrastructure lead to *communication* and *availability* faults respectively [61, 151]. Invisible faults can also occur, such as Web services returning incorrect data as output [103]. The appropriate recovery or prevention measures to apply to faults can depend on the precise fault type [29].

The nature of service-oriented computing opens many possible means of transactional behavior to address these issues. Transactional behavior that has been applied to Web service compositions to prevent or handle faults includes service replacement [12, 61, 111], semantic service discovery [12, 33], *fault tolerance* strategies [96, 103, 105], backwards recovery through compensating activities [18, 53, 100], dynamic workflow restructuring using planning methods [65], and architectural context-based fault diagnosis [29].

Several standards for managing the transactional aspect of Web service compositions have been developed, such as *WS-AtomicTransaction* (WS-AT) [28], *WS-BusinessActivity* (WS-BA) [27], and

Business Transaction Protocol (BTP) [35]. These standards either implement or relax the classic ACID properties in various ways during Web service composition execution, such as avoiding resource locks to ensure strict isolation in favour of attaching compensating activities to undo operations. WS-BPEL also allows developers to define *scopes* within a process, which are comparable to both save-points in SAGAS and sub-transactions in nested transactions.

In addition, ensuring transactional requirements in Web service compositions has been an active research area since the inception of service-oriented computing [32, 66]. Concepts from advanced transaction model have been applied, such as partial rollback through compensating activities [12, 18, 23], and directing or verifying transactional behavior from the properties of component services [53, 100, 111], eg., *retriable*, *pivot*, and *compensatable*. Other innovations include dynamic re-provisioning [30, 33, 135], whereby component Web services that have caused a fault are replaced at runtime with an alternate service providing the same functionality. Well-proven fault tolerance strategies from software engineering have also been adapted, such as provisioning several Web services to perform the same action, in order to reduce the likelihood and impact of faults [96, 103]. We will provide an in-depth analysis of this research landscape in Chapters 3 and 4.

2.3 Cloud Services

In recent years, *cloud services* have had a dramatic impact on the research [21] and industry [106] landscape of service-oriented computing. Cloud computing has become a popular paradigm for delivering or provisioning a wide range of services, such as software applications, computing capacity, storage, and virtual platforms [49, 159]. Cloud service providers can offer these utilities to clients over the Internet in a pay-by-use manner similar to Web services. Apart from the diversity of service types, cloud services have a number of distinctive properties that make them innovative and appealing for providers and clients:

On-demand access over networks: Cloud services are easily accessed through network connections, most commonly the Internet [159]. Access to these services is on-demand and instantaneous, without the need for human interaction with the service provider. Easy access is commonly provided across heterogeneous platforms, such as smartphones, tablets, and laptops [49].

Pooled resources with elasticity: Clients are provided with access to shared resources, such as servers, applications, CPU time, or storage. The precise capacity and capability of resources provisioned and released by the client are dynamic in response to their workload. This refers to the *elasticity* property of cloud services. Clients only use the resources they need, while the service architecture applies appropriate mechanisms to measure their usage [7, 49, 159]. This benefits both client and provider by lowering service provisioning costs and increasing the number of clients the service handles within a given amount of resources.

Configurability: The potential market of a service is significantly increased if clients are able to adjust the behavior or properties of that service to suit their own requirements or preferences. Configuration can affect such service properties as software structure, features, UI, data, access control, and QoS [92, 101, 128]. Another benefit is that highly configurable services can satisfy the requirements of particularly unique clients, in a way that is overall more cost-effective than deploying separate services for all client types [37].

Multi-tenancy: Services that are *multi-tenant* are able to handle several clients (or tenants) with a single software instance in a way that prevents any interference between clients and their specific configurations [16, 49, 131]. Multi-tenant services allow providers to exploit *economies of scale* and serve more clients with less supporting infrastructure. As clients are efficiently *sharing* infrastructure, the cost of the provider to serve a given number of clients reduces.

Although cloud services share similarities with pre-existing technologies such as *Grid Computing* and Web services, there are distinctions [159]. A computing grid virtualizes physical resources on a large scale to work towards a common goal, and is generally used for computationally expensive tasks [20]. Cloud services also utilize a pool of resources, but these are dynamically provisioned and de-provisioned as required. Furthermore, cloud services expand beyond the infrastructure layer of virtualized physical resources, into platforms and software. Similarly, this

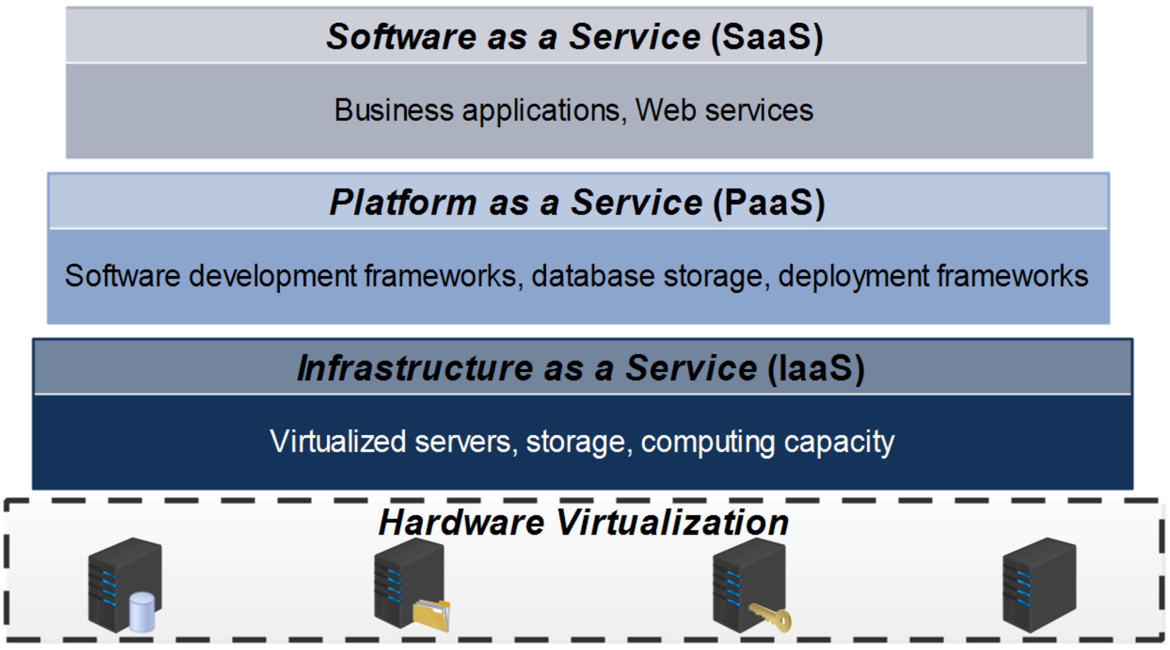

Software as a Service (SaaS)
Business applications, Web services
Platform as a Service (PaaS)
Software development frameworks, database storage, deployment frameworks
Infrastructure as a Service (IaaS)
Virtualized servers, storage, computing capacity
Hardware Virtualization

Figure 2.2: Cloud service hierarchy [159]

elasticity property and the multi-tier architecture of service types distinguish cloud services from Web services.

Figure 2.2 shows the traditional cloud hierarchy, comprised of three layers of cloud services. Each layer is able to provide the *base* (infrastructure or platform) for running services within the layer above [49], although real-world services vary in how they are deployed [38]. *Infrastructure as a Service* (IaaS) is the bottom service layer, providing access to virtualized physical resources, such as storage and computation capacity. Computing capacity offered by Amazon EC2[6] or IBM SmartCloud Enterprise+[7] are examples of IaaS offerings. *Platform as a Service* (PaaS) provides access to utilities such as software development and hosting frameworks. For example, Google App Engine[8] and Microsoft Azure[9] both contain PaaS features for web application development and hosting. Finally, *Software as a Service* (SaaS) are software applications deployed in a way that is Internet accessible, automatically scaling, and multi-tenant. SaaS enables clients to remotely use software complex systems, such as customer relationship management through Salseforce[10],

[6]https://aws.amazon.com/ec2/
[7]http://www-07.ibm.com/au/managed-cloud-hosting/
[8]https://cloud.google.com/appengine/docs
[9]https://azure.microsoft.com/en-gb/
[10]www.salesforce.com/au/

or payroll management with ePayroll[11].

2.3.1 Business Process as a Service

A proposed *fourth* level of the cloud service architecture residing above SaaS has been in the form of *Business Process as a Service* (BPaaS), which has had increasing research interest in recent years [1, 25, 104, 118]. The driving idea behind BPaaS is to mash-up services from numerous providers into a business process structure, which can then be offered to clients as its own service. BPaaS providers will naturally target common or proven business processes that apply to a large potential market, or require management of several complex components. This is appealing to clients as it provides them with an outsource option for integral business operations that is low in cost and risk.

It is important to clarify that the definition of BPaaS we use is not yet universally recognised. *BPaaS* has also been used to describe approaches for enabling *Business Process Management* (BPM) using cloud services [51]. BPM management as a cloud service has also been proposed as *Orchestration as a Service* (OaaS) [75] and *Composition as a Service* (CaaS) [127] in research publications. In our work, we consider BPaaS to be a cloud service offering an executable business process, rather than a BPM environment for business processes provided by clients.

Figure 2.3 shows an abstract example that demonstrates the structure and variety of services and resources that a BPaaS can utilize. In this example, the BPaaS is composed of heterogeneous component services from the service provider and third parties. Two SaaS used by the BPaaS are hosted and managed by the same provider. Private internal software exclusive to the BPaaS is also required. Two of the SaaS services are from external sources - *SaaS 3* is from a third party, while *SaaS 4* is another service of the BPaaS provider, but hosted on an external PaaS.

Like other services in the cloud hierarchy, *configurability* is an important property for BPaaS. Business process configuration can affect several of its properties, such as the workflow structure [109, 145], resources used [71, 91], and variables [94]. This allows clients to bring these services as in-line as possible with their internal business operations and policies, while providers

[11]http://www.epayroll.com.au/epayroll

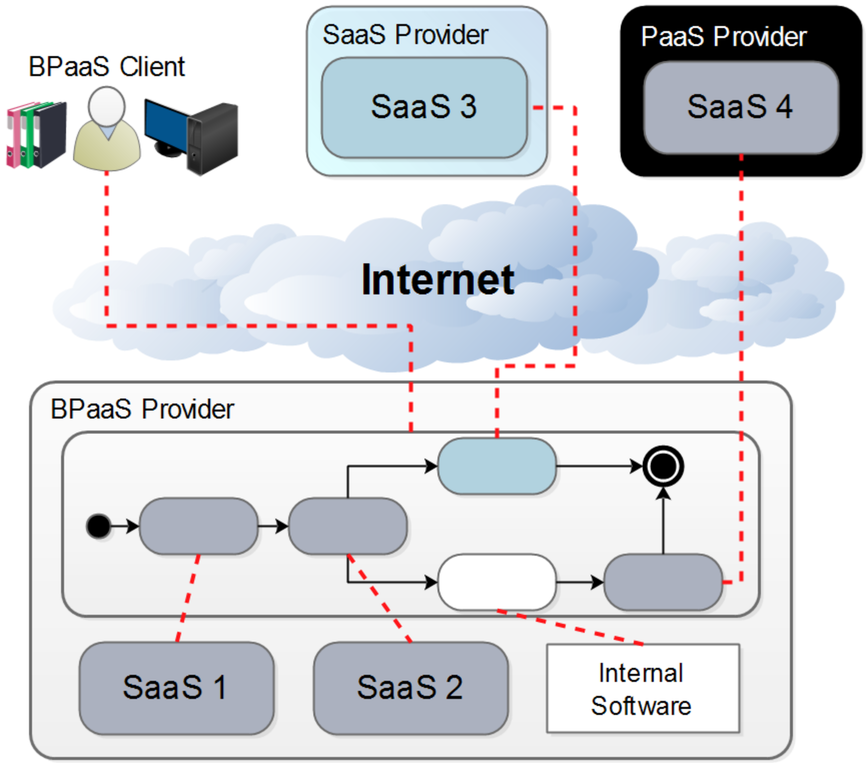


Figure 2.3: An abstract example of a BPaaS

can exploit economies of scale by offering their service to a larger potential market. For example, some business clients may have internal systems already in place to handle certain steps, while smaller businesses may be looking to outsource more features.

At present, research and adoption of BPaaS is still in its infancy. Despite cloud services increasing their presence in industry, with directories such as *Cloudbook*[12] and *Cloud Showplace*[13] listing services from several thousand vendors, services advertised as BPaaS remain scarce. As such, there are many open issues for research in BPaaS to address. These include managing configuration, composing cloud services together, maintaining elasticity, multi-tenancy, and ensuring correctness and transactional behavior [118, 119].

Transactional management is a critical concern for BPaaS. Like Web service composition, BPaaS execution requires the potentially long-running coordination of several heterogeneous black-box resources from numerous parties. However, BPaaS are implemented and managed by service

[12]http://www.cloudbook.net/
[13]http://www.cloudshowplace.com/

providers rather than clients, so the transactional management must be formal and substantial for clients to trust outsourcing sensitive business operations, such as managing finances or customer details. To the best of our knowledge, this is a BPaaS issue that is yet to be addressed in the current state of research.

2.4 Summary

Our work builds on research contributions first proposed in managing complex database transactions. Concepts such as compensation, relaxed ACID properties, and transactional properties of sub-transactions were first proposed in advanced transaction models, and have since been applied to service-oriented computing.

Web services, black-box software units deployed online with public interfaces, bring a host of challenges and opportunities to business processes and their management. By composing Web services together to perform a set of related tasks, businesses can implement their processes in a rapid and low-cost manner. Transactional management of Web service composition has been the attention of several proposed standards and years of research. Advanced transaction model concepts such as compensation and nested transactions have been applied, while the service-oriented computing paradigm allows for behavior such as dynamic Web service re-provisioning and fault tolerance strategies.

A recent innovation in service-oriented computing is cloud services, which offer clients access to a virtualized pool of computing utilities in an on-demand fashion. Cloud services offer storage, computing capacity, software applications, and developments platforms amongst other services. BPaaS is an emerging type of cloud service that allows clients to provision business processes from service providers and configure them to their requirements. Like Web service composition and long-running distributed transactions, BPaaS execution requires transactional management to ensure an acceptable level of atomicity, consistency, isolation, and durability. However, as BPaaS research is still in its infancy, this remains a largely open issue.

CHAPTER 3

Well-Formed Transactional Behavior in Service-Oriented Processes

This chapter presents our method for identifying and resolving issues in the transactional behavior of service-oriented processes. First, we overview the state-of-the-art, which comprises of design tools, composition frameworks, service brokers, transactional protocols, and other approaches related to service-oriented processes. Our analysis finds that design-time approaches have been been under-represented, which may be a concern to developers looking to identify and resolve issues prior to costly development. Furthermore, most approaches conduct automatic transactional behavior, which may not allow developers to ensure that fault-handling conforms to their own requirements or expectations.

To address these issues, we develop a design-time modeling and verification approach for service-oriented processes. We propose a statechart-based model transactional service-oriented processes, and demonstrate it with an example scenario using Web services for handling different payment methods. This model separates the behavior of the process into two perspectives, namely, *control* and *operational* behavior. *Inter-behavior messages* are used to co-ordinate these two models and apply transactional behavior to the process. We define and target *well-formed* inter-behavior conversations as a correctness property for transactional service-oriented processes, and present a set of rules to enforce it.

A verification process formalizes the conversation rules in temporal logic, and applies model checking to exhaustively verify that process models conform to them. Furthermore, we apply state space reduction measures to improve verification performance for large and complex models. An algorithm is defined in order to transform the control and operational behavior models into a minimal Kripke structure. The NuSMV model checker is used for verification as it provides support for temporal logic properties formulated in *Linear Temporal Logic* (LTL) and *Computational Tree Logic* (CTL), which are both required to express our conversation rules. The online payment process is used to demonstrate the effectiveness of our conversation rules in identifying errors in transactional behavior, and the state space reduction achieved by our Kripke structure generating algorithm.

3.1 Related Work

Since the advent of service-oriented computing [116], ensuring a level of transactional correctness for business processes implemented with services has been an active research area [32]. In this section, we discuss the state-of-the-art and apply a set of criteria to compare each approach. Using the comparison results, we identify open issues to address with our work.

3.1.1 Comparison Criteria

We define a set of criteria that we apply to compare all approaches for ensuring transactional behavior correctness in service-oriented processes. The criteria we use is divided into four categories, namely, *Overview*, *Support*, *Method*, and *Transactional Behavior*. as shown in Table 3.1 shows the criteria that fall into each category, with example values from the literature.

3.1.1.1 Overview

The *overview* category contains criteria to describe the approach in a high-level manner, focusing on the *lifecycle phase* and *implementation* of the approach.

Lifecycle phase refers to the stage in the service-oriented process development lifecycle where the approach is applied. As service-oriented processes encompass many domains, such as Web service compositions and distributed workflows, we consider three broadly applicable lifecycle phases:

Table 3.1: Overview of comparison criteria

Category	**Criteria**	**Values**
Overview	Lifecycle Phase	Design, Development, Runtime
	Implementation	Verification tool, Development framework, Broker, Standard extension, etc.
Support	Process Model	Supported workflow patterns, data representation, transactional properties, adopted or extended standards
Method	Transactional Requirements	Relaxed atomicity, fault-tolerance, mandatory operations for commit, composition compensatability, etc.
	Verification or Assurance	Model checking, service provisioning strategy, composition rules, recovery algorithms, etc.
Transactional Behavior	Faults	Behavioral, logical, communication, availability, etc.
	Fault-handling	The fault recovery and prevention operations shown in Figure 3.2.
	Specification	Automated, Semi-automated, Manual

- In the *design* phase, a formal model of the eventual process is produced, and may be checked for correctness. The details of the formal model depend on the approach, but the specification can include the workflow structure of tasks in the process, the interactions between business partners, process variables and the flow of data between tasks or partners, exception handling, transactional behavior, and other properties.

- *Development* encompasses all tasks following the design phase until the process is deployed and ready for use. These tasks may include Web service discovery and selection, necessary coding, and the deployment of the process.

- The *runtime* phase refers to the execution of the process, following a user request. This phase can include invoking operations or services, exception handling, monitoring execution, service replacement, determining optimal recovery to faults, and service coordination according to a transaction protocol.

The implementation criterion is a high-level description of how the approach is realized. This includes design-time formal verification [90], service provisioning strategies [17], runtime coordination frameworks [33], among other implementations.

3.1.1.2 Support

The support category contains the *process model* criterion, which indicates the complexity of processes supported by the approach. Process models may not always be easily comparable, as such formalisms as statecharts [23], workflow graphs [62], and Petri-nets [33] have all been applied. Therefore, we identify the following common set of properties to consider:

- The supported *workflow patterns* [147], such as *parallel* (AND), *exclusive choice* (XOR), *inclusive or* (IOR), and *iteration* (Itr), have a large impact on the complexity of processes that can be supported. The AND pattern allows workflow tasks to branch into two or more paths executing simultaneously. XOR and IOR provide a split of paths of which only one or one or more can be executed respectively. Iteration refers to cyclic patterns in the workflow. Even this basic set of patterns can be broken down into several semantic variations [147].
- *Transactional properties* (TP) may be attached to component operations to indicate basic transaction requirements for their invocation and fault-handling [18, 53, 100]. Commonly considered transactional properties in for component services include *retriable*, *non-retriable*, *compensatable*, and *pivot* (not retriable or compensatable).
- The *data representation* supported by the model, which can include process variables [87] or data flow between components [86].
- The *standards* that are directly adapted, supported, or extended for process modeling or execution, such as WS-BPEL [87] or other workflow formalisms [57].

3.1.1.3 Method

The *method* category identifies the targeted transactional requirements or correctness criteria, and the *verification or assurance* method applied towards their satisfaction. Such requirements can include fault tolerance [103], relaxed atomicity [17], and compensatability [108]. Verification methods, such as model checking [22], apply formal methods to ensure that requirements or properties are met. In contrast, assurance methods manage design, development, or runtime in such

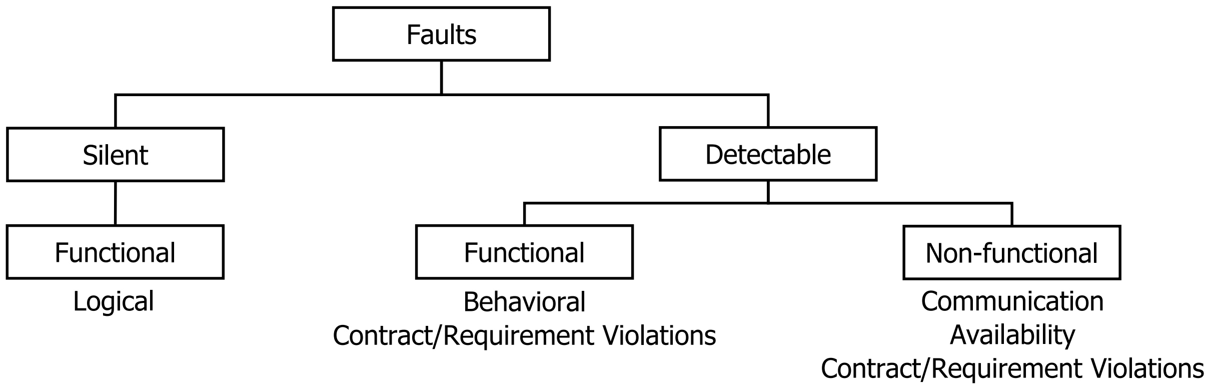


Figure 3.1: Faults of transactional service-oriented processes

a way that preserves requirements. These methods include recovery algorithms [29], automated reprovisioning at runtime [61], and composition rules [17].

3.1.1.4 Transactional Behavior

The transactional behavior category contains three criteria to describe the *faults* it can handle, the *fault-handling* measures it supports, and the *specification* method of this behavior.

The faults criterion lists the fault types that each approach detects and handles. Service-oriented processes are subject to several types of faults [29]. Figure 3.1 shows these faults organized according to *silent*, *detectable*, *functional*, and *non-functional* categories. Logical faults occur when a service produces incorrect output with no notification that a fault has occurred. In contrast, behavioral faults are explicitly flagged by the service. Communication and availability faults are caused by outages in the network or service infrastructure respectively. Contract or requirement violations are when the terms of a pre-approved service-level agreement between client and provider are violated. In cases where the fault types are not specified by the approach, we assume that behavioral faults are the focus.

Fault-handling compares the techniques each approach employs or supports to handle the occurrence of faults. These can be divided into fault-prevention and fault-recovery measures, as shown in Figure 3.2. Fault recovery measures can be further divided into forward recovery, which attempts to resume execution without changes to the process state, and backward recovery, which executes necessary recovery operations before execution can be resumed or aborted.

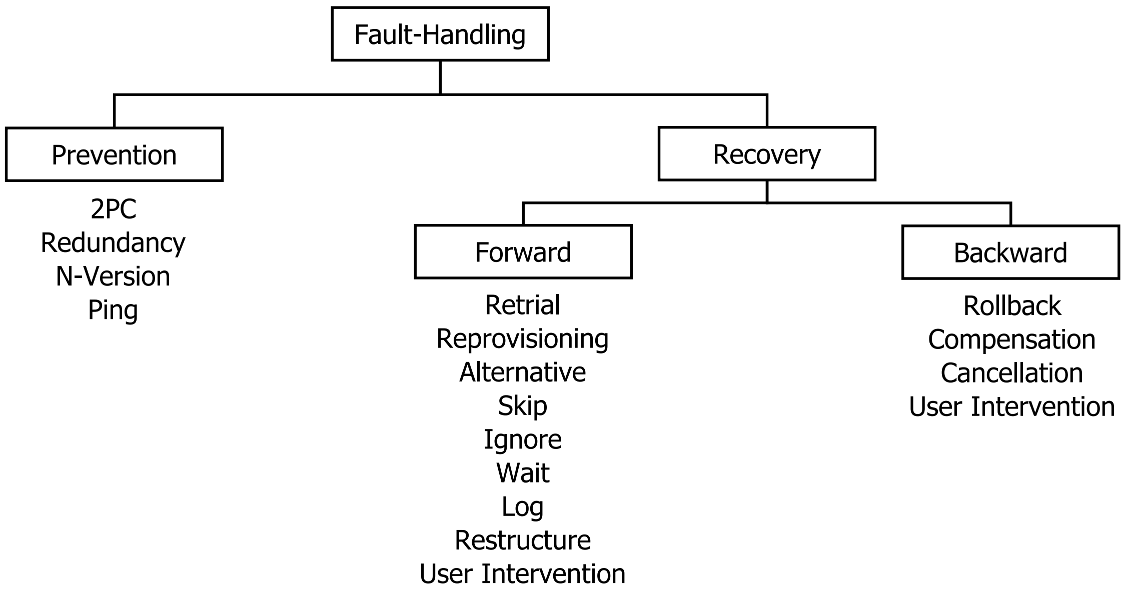


Figure 3.2: Fault-handling of transactional service-oriented processes

Prevention measure include enforcing strict ACID properties through protocols such as *two-phase commit* (2PC) [122, 123], testing component liveness before invocation [22], and fault-tolerance strategies such as redundantly provisioning or implementing several services for the same task [96, 103]. Commonly used forward recovery strategies include retrying services [17, 63], re-provisoning a functionally equivalent service as a replacement [33, 137], and executing pre-defined alternative operations [96, 130]. Other options include dynamically restructuring the workflow to handle the fault [65], and lower-level exception handling such as waiting, skipping, ignoring, and logging faults [100]. Backwards recovery applied to service-oriented processes include partial rollback of completed components [10, 105], full compensation of a completed process [97, 149], and cancellation or concurrently executing components [90, 111]. Some approaches also rely on user intervention in some instances to direct recovery operations or select from a list of proposed actions [29, 135].

Finally, the specification criterion indicates the level of developer control over transactional behavior. Approaches are categorised according to three possible values; *manual*, *automated*, and *semi-automated*. In manual approaches, the developer specifies all transactional behavior in the process [22], while automated approaches coordinate fault-handling automatically according to an algorithm or transactional properties of components [17]. Semi-automated approaches enable de-

Table 3.2: *Overview* criteria of application-independent requirements approaches.

Approach	**Lifecycle Phase**	**Implementation**
Kumar et al. [90]	Design	Verification algorithm
Bhiri et al. [17]	Design, development	Design method
Gabrel et al.[63]	Design, development	Composition framework
Gaaloul et al. [62]	All	Design verification and runtime monitoring framework
Vonk et al. [150]	All	Cross-organisational workflow management framework
WS-FTM [103]	Runtime	Broker extending FT-Grid [140]
FTWS [96]	Development, runtime	Broker
Mansour, Dillon [105]	Runtime	Broker
XIP [115]	Runtime	Transaction protocol
Mei et al. [108, 153]	Runtime	Transaction mechanism
Cao et al. [29]	Runtime	Transaction mechanism
Schuldt [130]	Runtime	Execution framework
REL [61]	Runtime	WS-BPEL extension
Gajewske et al. [65]	Runtime	Re-planning mechanism
Hwang et al. [77]	Runtime	Provisioning strategy
Stein et al. [137]	Runtime	Provisioning strategy
Wagner et al. [151]	Runtime	Provisioning strategy
Cardinale et al. [33]	Runtime	Execution framework
Li et al. [97]	Runtime	Execution framework
Cao et al. [30]	Runtime	Execution framework

velopers to partially specify aspects of the fault-handling behavior while automating others [100].

3.1.2 Survey

In order to identify the unique contribution of our work, we apply our comparison criteria to the state-of-the-art of work in ensuring application dependent transactional requirements in service-oriented processes. The results of the overview, support, method, and transactional behavior criteria are shown in Tables 3.2, 3.3, 3.4, and 3.5 respectively. We discuss design-time verification approaches, followed by service brokers, transactional protocols and mechanisms, service selection strategies, and other implementation methods.

Related design-time verification approaches include the work of Gaaloul et al. [62] and Kumar et al. [90]. Gaaloul et al. propose to model Web service composition with a formalism based on event calculus [88], and verify them at design-time against rules to prevent transactional inconsistencies. Kumar et al. present a verification algorithm that identifies *partial synchronization errors*

Table 3.3: *Support* criteria of application-independent requirements approaches
.

Approach	**Process Model**					
	AND	XOR	IOR	Itr	TP	Notes
Kumar et al. [90]	√	√	√			
Bhiri et al. [17]	√	√			√	Transactional dependencies between components
Gabrel et al. [63]	√	√			√	
Gaaloul et al. [62]	√	√			√	Transactional dependencies between components
Vonk et al. [150]	√	√		√		
WS-FTM [103]						Independent of process model
FTWS [96]						Unspecified
Mansour, Dillon [105]						Sequential workflow only
XIP [115]		√			√	
Mei et al. [108, 153]	√	√		√		
Cao et al. [29]		√				
Schuldt [130]		√			√	
REL [61]	√	√	√	√		WS-BPEL equivalent
Gajewske et al. [65]	√	√		√		
Hwang et al. [77]	√	√	√	√		WS-BPEL equivalent
Stein et al. [137]		√				
Wagner et al. [151]	√	√				
Cardinale et al. [33]	√	√			√	
Li et al. [97]	√	√	√	√		WS-BPEL equivalent
Cao et al. [30]	√	√	√	√		WS-BPEL equivalent

in workflows. This algorithm ensures that, during execution, if one or more incoming branches to a *synchronization* pattern is inactive, or more than one incoming branch to a *merge* pattern [147] is active, that rollback is applied successfully to reach a strictly correct state.

A common implementation to ensure fault-tolerant execution is to develop a service broker to implement fault-tolerance strategies. One such broker, the *Web Service-Fault Tolerance Mechanism* (WS-FTM) [103], is an implementation of the n-version [36] fault-tolerance strategy for Web services. It transparently invokes several functionally equivalent services from different sources to perform the same operation. A majority vote is used to select the result from the numerous responses. Implementing component operations in this way addresses the risk of logical faults. *Fault-Tolerant Web Services* (FTWS) [96] is another n-version broker strategy that enables several voting strategies and performance metrics to determine the result. The broker proposed by Mansour and Dillon [105] applies the rockery block fault-tolerance strategy [125]. If the result returned from a service is deemed unacceptable by an internal testing mechanism, rollback and

Table 3.4: *Method* criteria of application-independent requirements approaches.

Approach	Transactional Requirements	Verification or Assurance
Kumar et al. [90]	Process model soundness, compensation of incomplete scopes	Verification algorithm
Bhiri et al. [17]	Relaxed atomicity	Composition rules
Gabrel et al. [63]	Relaxed atomicity	0-1 Linear Programming
Gaaloul et al. [62]	Relaxed atomicity	Composition rules with runtime monitoring
Vonk et al. [150]	Autonomic fault-handling of a partner workflow	Compensation algorithm
WS-FTM [103]	Fault-tolerance	Fault-tolerance strategy
FTWS [96]	Fault-tolerance	Fault-tolerance strategies
Mansour, Dillon [105]	Fault-tolerance	Fault-tolerance strategies
XIP [115]	Deadlock and livelock free	Backwards recovery algorithm
Mei et al. [108, 153]	Compensatable compositions, deadlock and livelock free	Petri-net composition modeling and generation of compensation.
Cao et al. [29]	Optimal fault recovery strategies given precise type of faults	Context-aware recovery algorithm
Schuldt [130]	Relaxed atomicity	Process development framework
REL [61]	Communication fault-tolerance	Runtime provisioning strategy
Gajewske et al. [65]	Fault tolerance	Automated workflow re-planning
Hwang et al. [77]	Higher completion rate	Runtime provisioning strategy
Stein et al. [137]	Higher completion rate	Runtime provisioning strategy
Wagner et al. [151]	Higher completion rate	Runtime provisioning strategy
Cardinale et al. [33]	Fault-tolerance	Colored Petri-net fault-handling algorithm
Li et al. [97]	Fault-tolerance	Case-based reasoning
Cao et al. [30]	Adaptation to context changes, including faults	Context monitoring and policy library

reprovisioning is applied to recover to process to an acceptable state.

The following approaches define transaction protocols or mechanisms to ensure transactional requirements at runtime. *XIP* [115] is a transaction protocol for composite Web services that enables both forward and backward recovery. A process-level parent transaction manager cooperates with component-level child transaction managers to coordinate complex distributed transactions free of deadlock and livelock. The protocol considers whether or not component transactions are cancellable as a transactional property. Mei et al. [108, 153] propose transactional mechanisms that use Petri-nets to dynamically construct recovery processes at runtime. The Web service composition transaction mechanism proposed by Cao et al. [29] evaluates the context of faults to diagnose them using a detailed set of fault types, so the optimal recovery strategy can be applied.

A set of approaches aim to reduce the likelihood of faults by dynamically selecting services

Table 3.5: *Transactional behavior* criteria of application-independent requirements approaches.

Approach	**Faults**	**Fault-handling**	**Specification**
Kumar et al. [90]	Partial synchronization	Rollback, cancellation	Manual
Bhiri et al. [17]	Behavioral	Retrial, rollback, cancellation, compensation, reprovisioning	Automated
Gabrel et al. [63]	Behavioral	Retrial, rollback	Automated
Gaaloul et al. [62]	Behavioral	Rollback, compensation, cancellation, reprovisioning	Automated
Vonk et al. [150]	Behavioral	Rollback, compensation	Automated
WS-FTM [103]	Behavioral, communication, logical, availability	N-version strategy	Automated
FTWS [96]	Behavioral, communication, logical, availability	Alternative, N-version strategy	Manual
Mansour, Dillon [105]	Behavioral, communication, availability	Rollback, reprovisioning	Automated
XIP [115]	Behavioral, timeout	Retrial, compensation, cancellation, ping	Automated
Mei et al. [108, 153]	Behavioral	Rollback, compensation	Automated
Cao et al. [29]	Behavioral, contract violations, temporary and permanent network failures, replaceable and irreplaceable physical failures	Retrial, rollback, user intervention, reprovisioning	Semi-automated
Schuldt [130]	Behavioral	Retrial, rollback, alternative	Automated
REL [61]	Communication	Reprovisioning	Automated
Gajewske et al. [65]	Behavioral, contract violations	Cancellation, re-planning	Automated
Hwang et al. [77]	Behavioral, timeout	Reprovisioning, rollback, compensation	Automated
Stein et al. [137]	Behavioral, communication, availability	Reprovisioning, redundancy	Automated
Wagner et al. [151]	Behavioral, communication, availability	Reprovisioning	Semi-automated
Cardinale et al. [33]	Behavioral	Retrial, rollback, reprovisioning	Automated
Li et al. [97]	Behavioral, communication, availability, contract violations	Retrial, reprovisioning, compensation, restructuring	Automated
Cao et al. [30]	Behavioral, communication, availability	Reprovisioning, restructuring	Automated

at runtime based on predicted reliability. Hwang et al. [77] propose two dynamic Web service selection strategies that are directed by an aggregated reliability value indicating the probability of successful execution. This probability is calculated and maintained at each state according to component service reliability values. Similarly, the dynamic selection strategy proposed by Stein et al. [137] proactively deals with communication, availability and behavioral faults. Web services are selected based on predicted performance, and redundancy and reprovisioning are utilized to tolerate faults. Wagner et al. [151] enable some user control over service selection by producing a set of selections to compare. In addition to component service reliability, other QoS attributes such as response time and price are considered in finding optimal selections. Gabrel et al. [63] apply 0-1 linear programming for selecting Web services to maximise the process reliability. Their model extends similar approaches that consider the QoS values of candidate services by also including transactional properties such as retriable, compensatable and pivot.

Implementing approaches as execution frameworks allows transactional behavior to be automatically managed to satisfy certain requirements. The Web service composition execution framework proposed by Cardinale et al. [33] uses colored Petri nets [78] to model and monitor Web service compositions. Colored Petri-nets are used to dynamically determine if forward recovery is possible, and track executed tasks for backward recovery purposes. Li et al. [97] develop a fault-tolerant framework for composed Web services using case-based reasoning. When a fault occurs, the type is determined from symptoms, and a recovery solution is proposed based on the most similar case found in a case history. The framework proposed by Cao et al. [30] uses context-awareness to dynamically adapt to changes in the Web service composition execution environment. An adaptive algorithm is used to determine the optimal solution given the current context and a policy library.

Other approaches do not fit into the implementation categories discussed so far. Bhiri et al. [17] extend a set of workflow patterns with transactional dependencies between components, forming *transactional patterns* for Web service composition. These patterns contain dependencies between components for transactional behavior such as rollback, alternative, and cancellation. The *Robust Execution Layer* (REL) is a SOAP proxy that extends the WS-BPEL execution engine to automate the handling of communication faults by reprovisioning.

In the distributed workflow domain, Vonk et al. [150] presented an architecture for process specification and transactional support that extends the SAGAS model. A transaction model called *X-transactions* enables intra-organizational and inter-organizational rollback by computing an appropriate compensation process in response to a fault. Gajewski et al. [65] utilize *replanning* to reconfigure the remainder of the process structure in the event of unexpected faults. Their method draws on research in automated workflow composition [126]. Schuldt [130] addresses atomicity and isolation for distributed business processes. Component operations of the process are partially ordered, managed as conventional transactions, and are scheduled by a process manager.

3.1.3 Research Direction

Table 3.2 indicates that 70% of approaches are not enabled until runtime. The implication of this is that potential issues in managing transactional behavior are not identified or addressed until late in the process lifecycle. If there are design issues in the process that impede correct transactional behavior, the necessary revisions become more costly than if they were addressed before the process was developed and deployed.

Another trend in the state-of-the-art is the heavy reliance on automated transactional behavior, with only 10% of surveyed approaches enabling manual specification. Automated and semi-automated transactional behavior reduces the burden on the process developer by coordinating fault-handling according to a protocol or algorithm. These are used in such a way that assures application-independent transactional requirements, as shown in Table 3.4. However, a critical trade-off with this approach is that the developer may not be able to ensure that fault-handling conforms to their expectations or business rules. For example, while some payment services may be automatically compensatable, it may violate a company policy to reimburse payments without conducting an internal review process. Manually specified transactional behavior allows developers to retain control over how faults are handled in the process.

The most common targeted transactional requirements, as shown in Table 3.4, include relaxed atomicity, fault-tolerance as enabled by a set of strategies, and higher rates of successful completion by provisioning the most reliable component services. As most approaches are assurance methods, very few attempt verify developer specified transactional behavior for correctness issues. As such, verifying service-oriented process transactional behavior for reliability issues such as deadlock and invalid termination remains a largely open area. These issues can suspended process execution, or leave inconsistent states across resources, which can cause problems such as halted business operations, loss of funds, violation of business policies, and damaged consumer confidence.

From this analysis, we direct our work towards identifying and resolving reliability issues in transactional behavior at design-time. Furthermore, we aim to give the developer control of the transactional behavior through a service-oriented process design model, so that they may ensure all faults are handled in the most suitable manner.

3.2 Transactional Service-Oriented Process Modeling

To address the identified issues, we propose a design-time model for service-oriented processes that enables transactional behavior to be specified and verified against rules for *well-formed* behavior. Our model separates process behavior into two views; control and operational behavior models. This allows transactional and functional behavior to be modeled and revised independently by developers with relevant expertise. The following section outlines our modeling approach by applying it to an example scenario.

3.2.1 Motivating Example

We consider a scenario where online payment transactions are managed using Web services. This process is to be developed by a small business that uses a web store to handle sales. The business manages a customer repository and transaction history internally, but wishes to outsource operations for handling credit card and direct deposit payments. Other related operations such as registering customer details, checking inventory, confirming price and shipping, are handled outside the scope of this process.

We implement this process using the Paylane Web Service API [1]. Figure 3.3 shows an overview

[1] http://devzone.paylane.com

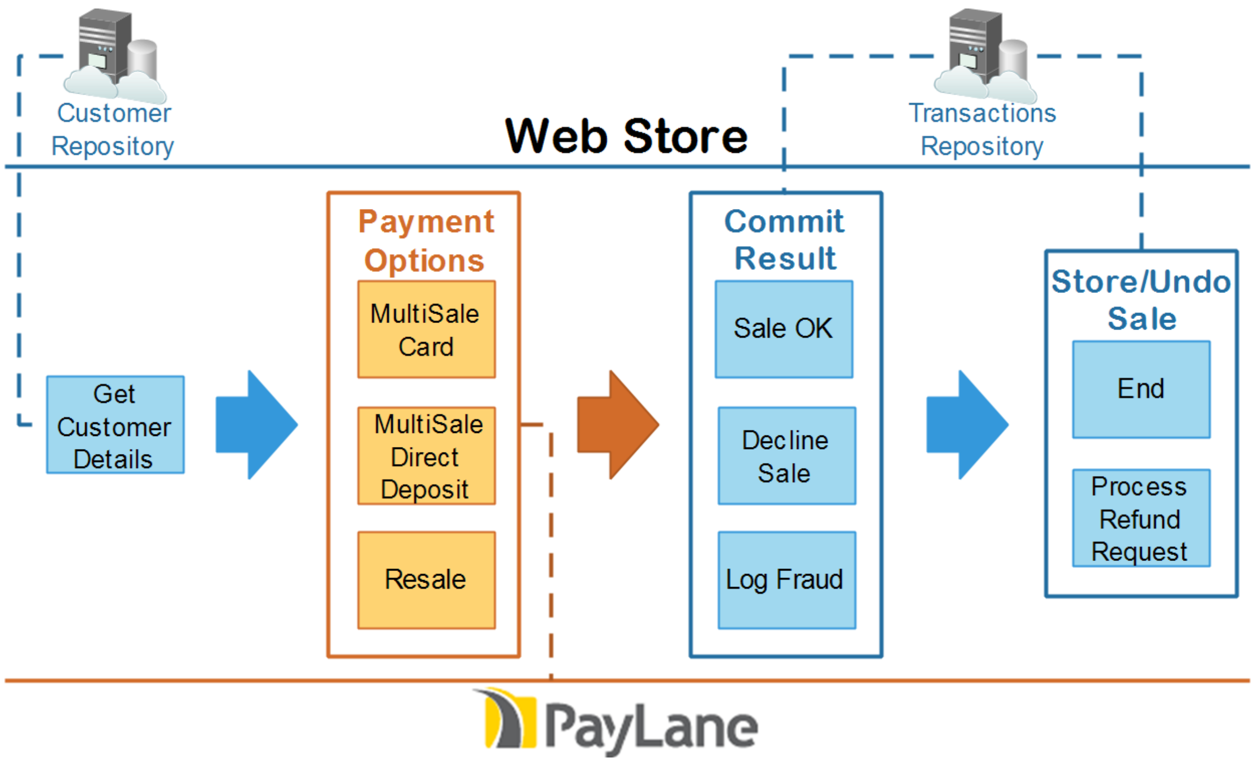


Figure 3.3: Overview of the online payment process

of the operations. After retrieving customer details, various Web services from the API are used to handle payment using credit card information, direct deposit, or *resales* stored payment information from a previous sale. The response from PayLane controls the result of the sale. Following a card payment attempt, payment either succeeds, or is unable to be processed. Furthermore, PayLane responds with a *fraud score* value, which indicates the likelihood of attempted credit card fraud. If this score is above a threshold decided by the Web store, it must be logged. The outcome of direct deposit payments must be checked after they are invoked until they are confirmed by the Web store's financial institution. After storing the payment outcome, the process can also handle refund requests to compensate completed sales. However, due to internal policies of the business composing this process, not all sales will be refundable.

The consequences of unreliable transactional behavior in this process could lead to payments being suspended, incorrectly completed, or unverified. This could lead to serious consequences such as mismanaged funds, loss of consumer confidence, unlogged or undetected instances of credit card fraud, and violation of company policies. Therefore, it is critical that the transactional behavior of the process is ensured and verified to be reliable before it is developed and deployed.

3.2.2 Control and Operational Behaviors

We propose a design-time modeling approach for service-oriented processes that applies the *separation of concerns* principle, to produce transactional and functional views of process behavior. This allows each perspective to be designed and revised independently by relevant domain experts. Our model builds on the approach proposed in [134], which separates process into two models, namely, *control* and *operational* behavior.

The purpose of the control behavior model is to maintain the transactional state of the process, while directing execution and recovery operations. The states of the control behavior include the transactional states a process may encounter from prior to invocation to termination, namely, $NotActivated$, $Activated$, $Suspended$, $Rollback$, $Done$, $Aborted$, and $Compensated$. In contrast, the operational behavior model contains the control flow of component tasks, and reports events such as faults, invocation requests, and process completion. As such, the states and control flow of the operational behavior depend on the application.

These behavior models can be expressed using statecharts [74], as shown in Figure 3.4, which models the online payment scenario. The control behavior contains transactional states and transitions of the process, from $NotActivated$ until termination occurs through $Done$, $Aborted$, or $Compensated$. The operational behavior model contains all component activities as states, such as credit card and direct deposit payment operations, and logging potential fraud incidents. Furthermore a compensatory process is included to undo the effect of a committed payment, by processing a refund request. *Event-condition-action* (ECA) labels are attached to transitions in order to restrict their activation to appropriate conditions, and trigger other operations. For example, the success or failure of a process that is $Activated$ determines whether the $Done$ or $Aborted$ control behavior states are reached.

These two behavior models can be expressed formally as a 5-tuple $\mathcal{B} = \langle \mathcal{S}, \mathcal{L}, \mathcal{T}, \mathcal{F} \rangle$, where:

- $\mathcal{S}$ is a finite set of state names

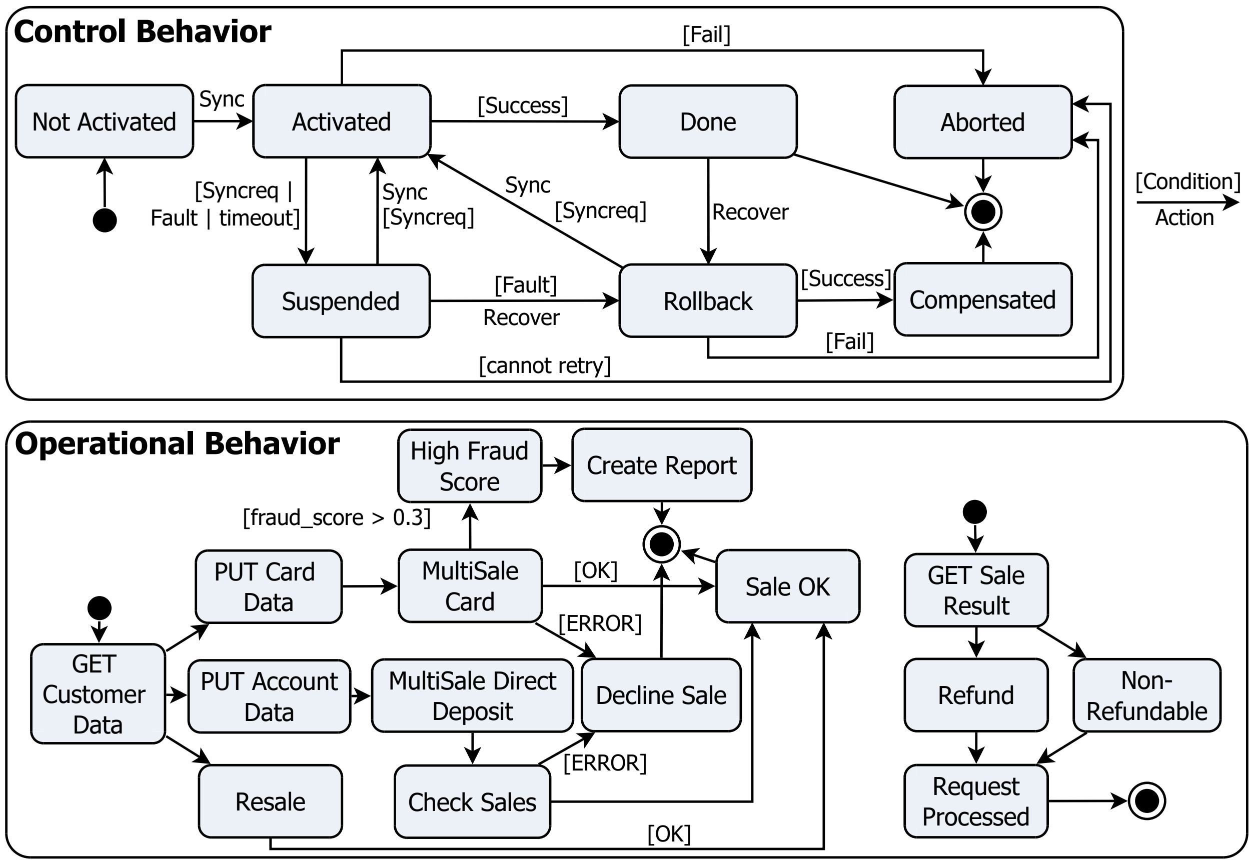


Figure 3.4: Control and operational behavior models of the online payment composition

- $s_0 \subseteq \mathcal{S}$ is the initial state
- $\mathcal{F} \subseteq \mathcal{S}$ is a set of final states
- $\mathcal{L}$ is a set of ECA labels
- $\mathcal{T} \subseteq \mathcal{S} \times \mathcal{L} \times \mathcal{S}$ is the transition relation, where each $t \in \mathcal{T}$ consists of a source and target state, and a transition label

As such, the control behavior model is expressed as $\mathcal{B}^c = \langle \mathcal{S}^c, \mathcal{L}^c, \mathcal{T}^c, \mathcal{F}^c \rangle$, while the operational behavior model is defined as $\mathcal{B}^o = \langle \mathcal{S}^o, \mathcal{L}^o, \mathcal{T}^o, \mathcal{F}^o \rangle$.

3.2.3 Inter-Behavior Messages

Communication between the behavior models is necessary in order for them to pass instructions and remain aware of each other's status. For example, the control behavior requires a message to

Table 3.6: Initiation inter-behavior messages

Message	Source	Role	Effect
Sync	`Activated`	Used to initiate or resume execution of the composition.	Activates an operational behavior state.
Recover	`Rollback`	Triggers backwards recovery actions, whether they be partial rollback or service compensation.	Activates an operational behavior state.
Delay	`Activated`	Forces a response from the operational behavior after an unacceptable delay has occurred.	The control behavior enters the `Suspended` state until a valid outcome message is received.
Ping	`Activated`	Tests the liveness of an operational behavior state.	The control behavior enters the `Suspended` state until an acknowledgement is received.

trigger the execution of operational behavior states once the process becomes *Activated*, while the operational behavior must indicate the nature of termination to trigger the correct control behavior state. We propose a set of *inter-behavior messages* to enable this. These messages can originate from any behavior model state, and target a state from the alternate behavior model.

We divide these messages into two types, namely, *initiation* and *outcome* messages. Initiation messages, shown in Table 3.6, are sent from control behavior states to operational behavior states. Their functions include triggering the execution of process operations, initiating backwards recovery, forcing a response following unacceptable delay, and testing the liveness of services. Outcome messages, shown in Table 3.7, originate from operational behavior states and target the control behavior. These messages can indicate successful or failed completion, the presence of a fault that requires recovery, request to retry or resume execution, or acknowledge service liveness. For each message type, the tables contain the valid source or target control behavior states, a description of its purpose, and the effect it has on the activity of the models once it has been received.

Two examples of inter-behavior messages in action are shown in Figure 3.5. In Figure 3.5a, a *Sync* message is used to initiate the process, which responds with a *Success* message once the payment was successful. Figure 3.5b shows a scenario where checking the result of a direct deposit payment results in a *Fault* message, which triggers *Recover* to decline sale, before the process aborts through a *Fail* message. These examples each demonstrate a *conversation session*; an ordered sequence of inter-behavior messages that are used from the initialization of the process until termination. Formally, a conversation session can be defined as a

Table 3.7: Outcome inter-behavior messages

Message	**Target**	**Role**	**Effect**
Success	`Activated`, `Rollback`	Indicates the successful commit of the service or completion of compensation.	Activates the `Done` or `Compensated` control behavior state and halts operational behavior execution.
Fail	`Activated`, `Rollback`	Sent when the service is to be aborted.	Activates the `Aborted` control behavior state and halts operational behavior execution.
Fault	`Activated`	Indicates the presence of a fault that requires recovery.	Activates the `Suspended` control behavior state and halts operational behavior execution.
Syncreq	`Activated`, `Rollback`	Requests a Sync message to retry a task or following the completion of partial backwards recovery.	Activates the `Suspended` control behavior state and halts operational behavior execution.
Ack	`Activated`	Acknowledges the liveness of an operational behavior state.	Activates the `Activated` control behavior state and resumes operational behavior execution.

sequence of message types of length n, where each message is expressed as $m(t)$ such that $m \in [Sync, Recover, Delay, Ping, Success,\ Fail, Fault,\ Syncreq, Ack]$, and t denotes the order such that $t \in [1, ..., n]$. For example, Figure 3.5b can be expressed as:

```
Sync(1).Fault(2).Recover(3).Fail(4)
```

These conversation sessions represent, at a high-level, the transactional behavior used during an instance of the process. In the following section, we outline our approach for ensuring the reliability of the transactional behavior of a process, by verifying that all possible conversation sessions conduct reliable transactional behavior.

3.3 Well-Formed Inter-Behavior Conversations

Our separated behaviour model allows design issues with transactional behavior to be identified and resolved at design-time, to avoid costly revisions during development. As inter-behavior conversations model the transactional behavior of a process, they can be checked for problems that may suspend the execution or prevent it from terminating correctly. For example:

```
Sync(1).Syncreq(2).Delay(3)
```

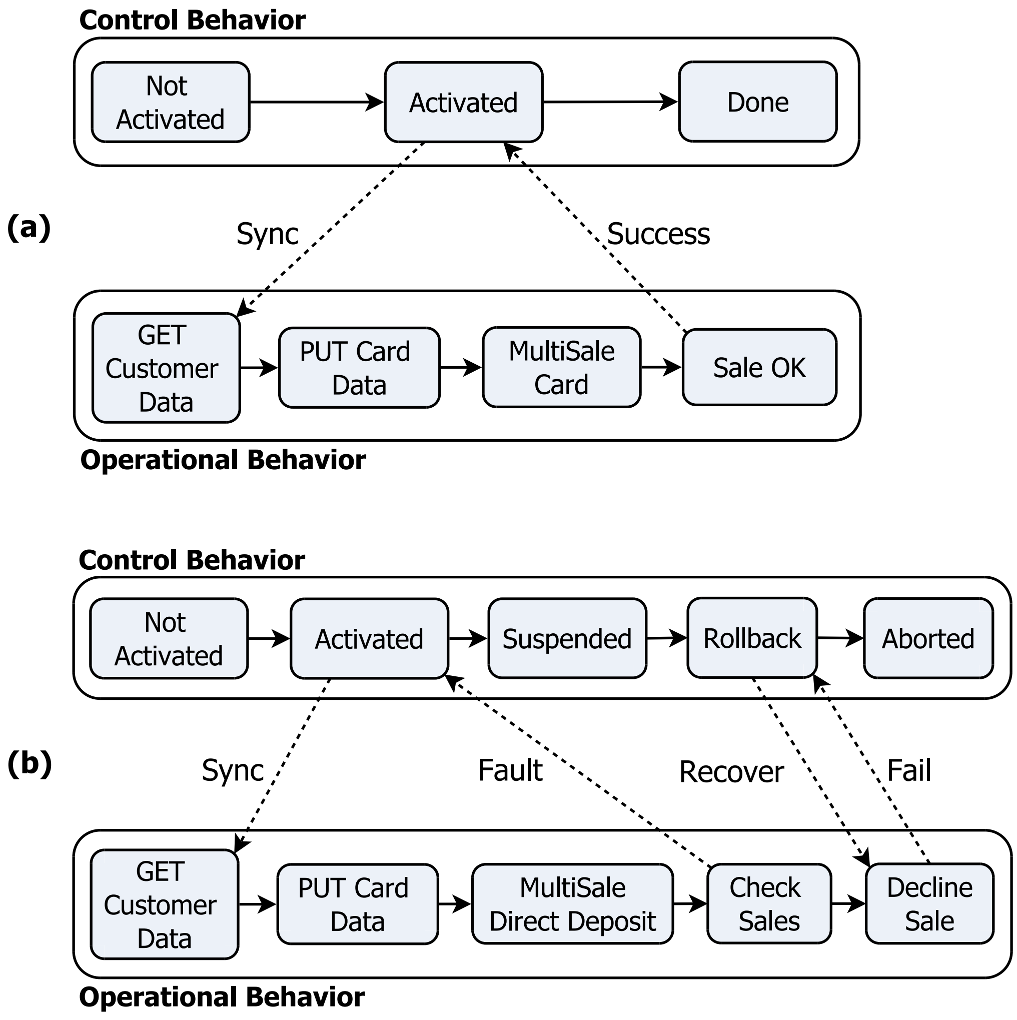


Figure 3.5: Two examples of exchanged inter-behavior messages during successful (a) and failed (b) process execution

is a conversation session suspended by a deadlock. The operational behavior is waiting for a `Sync` message before if can continue, while the control behavior is in the *Suspended* state until it receives a reply to the `Delay` message. Moreover, conversation sessions can also terminate in an invalid manner, such as:

```
Sync(1).Ping(2).Ack(3).Ping(4).Ack(5)
```

This lacks a message to indicate to the control behavior that the process has terminated. As a result, the control behavior permanently remains in the *Activated* state.

To avoid issues such as these, we aim to ensure that inter-behavior conversation sessions are *well-formed*. We refer to inter-behavior conversations as well-formed if they:

Table 3.8: Conversation structure rules

ID	Conversation Rule
CSR1	$Sync(1)$
CSR2	$\exists m \in [Success, Fail, Ping, Syncreq], m(n)$

(i) Always initialize with a valid inter-behavior message type.

(ii) Are free of deadlocking scenarios between the behavior models.

(iii) Are guaranteed to terminate with a message type from the valid set of $\{Success, Fail, Ping, Syncreq\}$.

We propose a set of *conversation rules* to make sure all inter-behavior conversations are well-formed. These rules can be expressed as if-then conditions in the form:

$$\forall m_i \in \mathcal{I}, \exists m_j \in \mathcal{J}, \forall t \in \mathcal{T}, m_i(t) \Rightarrow m_j(t+1)$$

where $\mathcal{I}, \mathcal{J} \subseteq [Sync, Success, Fail, Syncreq, Delay, Ping, Ack]$ and $\mathcal{T} \subseteq [1, ..., n-1]$. The set $\mathcal{I}$ specifies a set of message types, and $\mathcal{J}$ defines those that can immediately follow. Our rule set is divided into *conversation structure rules* and *message sequence rules*, which are both outlined below.

3.3.1 Conversation Structure Rules

Conversation structure rules ensure that inter-behavior conversations initialize and terminate in a valid way. This prevents incomplete inter-behavior conversation sessions that leave the result of the process undetermined. Our two conversation structure rules are shown in Table 3.8.

CSR1 simply specifies that the first message in any conversation must be *Sync*. This is because *Sync* is used to trigger the execution of operational behavior states, which must begin once a process becomes *Activated*.

CSR2 defines the valid termination messages for conversation sessions. *Success* and *Fail* are included as they indicate process should commit or abort respectively. *Ping* can be the final mes-

Table 3.9: Message sequence conversation rules

ID	Conversation Rule
MSR1	$\forall t \in \{2, ..., n-1\}, \forall m_i \in \{Sync, Ack, Recover\}$, $\exists m_j \in \{Success, Fail, Ping, Fault, Syncreq, Delay\}, m_i(t) \Rightarrow m_j(t+1)$
MSR2	$\forall t \in \{2, ..., n-1\}, \exists m_j \in \{Success, Fail, Syncreq, Fault\}$, $Delay(t) \Rightarrow m_j(t+1)$
MSR3	$\forall t \in [2, ..., n-2], Syncreq(t) \Rightarrow Sync(t+1)$
MSR4	$\forall t \in \{2, ..., n-2\}, Fault(t) \Rightarrow Recover(t+1)$
MSR5	$\forall t \in \{2, ..., n-2\}, \exists m_j \in \{Ack, Sync, Recover\}, Ping(t) \Rightarrow m_j(t+1)$
MSR6	$\forall t \in \{2, ..., n-1\}, Success(t) \Rightarrow Recover(t+1)$
MSR7	$\forall t \in \{2, ..., n-1\}, \exists m_j \in \{Sync, Success, Delay, Syncreq, Ping, Ack, Fault, Recover\}, m_i(t)$

sage of a conversation session in the event that a service or resource is not responsive, and i) retrial is not possible, and ii) no alternative operations are specified. Similarly, *Syncreq* can be a terminating message if the request for *Sync* is unable to be fulfilled, leading to the process being aborted. *Fault* is not considered valid, as it indicates a state that requires recovery operations before the process can resume or abort. *Ack*, *Delay*, and *Recover* are also invalid for termination as they leave the control behavior model in the *Activated*, *Suspended*, or *Rollback* state until an outcome message is received.

3.3.2 Message Sequence Rules

Our set of message sequence rules specify what sequences of messages are required to avoid deadlocks and ensure all requests lead to a valid response. The rule set is shown in Table 3.9.

MSR1 defines the valid message types to follow *Sync*, *Ack*, and *Recover*. After these initiation messages are sent, the process is either executing or recovering, and the only valid messages to interrupt these states are the outcome messages *Success*, *Fail*, *Fault*, and *Syncreq*. *Ping* and *Delay* are also valid in the event that the liveness of a service or resource needs to be confirmed, or the process is taking an unacceptable time to respond.

The valid responses to a *Delay* message are specified in rule MSR2. These include *Success* and *Fail*, to respond that the process has completed, *Syncreq* to retry or trigger alternative operations, and *Fault* to indicate that recovery is necessary.

MSR3 specifies that only a *Sync* message may follow *Syncreq*. Similarly, MSR4 specifies that *Recover* is required to recover the process following a *Fault* message. MSR5 defines that following

Ping the process may only respond with *Ack*, otherwise *Sync* and *Recover* are used to retry, execute alternatives, or recover the process state. The timeline for these rules is restricted to $\{2, ..., n - 2\}$ as *Sync*, *Ack*, and *Recover* are not valid termination messages, as specified in `CSR2`.

`MSR6` specifies that whenever Success is used within $\{2, ..., n - 1\}$ (i.e. not as a termination message), then it must be followed by *Recover*. Since Success indicates that the process has committed successfully, the only valid behavior that may follow is compensation, initiated by *Recover*.

Finally, `MSR7` is used to specify that *Fail* may only be used as a termination message, as it indicates that the process is to be aborted. The rule expresses this by specifying that during the timeline $\{2, ..., n - 1\}$, every message except *Fail* may be used.

3.4 Enabling Formal Verification Through Model Checking

We plan to apply model checking [43] to ensure that all possible conversation sessions between the control and operational behaviors satisfy our conversation rule set. The inputs to a model checking tool include:

- A set of properties formalizing ideal system behavior, such as our conversation rule set
- A model representing a system, which in our approach is a service-oriented process

Model checking will exhaustively verify the system against the property set, in order to identify any violations. Therefore, before it is applied, our concerns are i) formalizing our conversation rule set in a property language enabling model checking, and ii) addressing the state space explosion problem [8] inherent in model checking. Then, these may be used as input for a model checking tool verify the control and operational behavior models against our conversation rule set.

3.4.1 Temporal Logic Transformations

A range of model checking tools support properties specified in ways including temporal logic properties [41], assertions [45], and modal logic properties [46]. For our conversation rule set,

Table 3.10: Conversation rules formalized using LTL and CTL

Rule	**Language**	**Temporal Logic Transformation**
`CSR1`	LTL	$initial \cup Sync$
`CSR2`	CTL	$AG(\exists F(final))$
	LTL	$G((Sync \vee Ack \vee Recover \vee Fault) \rightarrow X(\neg final))$
`MSR1`	LTL	$G((Sync \vee Ack \vee Recover) \rightarrow X(Success \vee Fail \vee Ping \vee Fault \vee Syncreq \vee Delay))$
`MSR2`	LTL	$G(Delay \rightarrow X(Success \vee Fail \vee Syncreq \vee Fault))$
`MSR3`	LTL	$G(Syncreq \rightarrow X(Sync \vee final))$
`MSR4`	LTL	$G(Fault \rightarrow XRecover)$
`MSR5`	LTL	$G(Ping \rightarrow X(Ack \vee Sync \vee Recover \vee final))$
`MSR6`	LTL	$G(Success \rightarrow X(final \vee Recover))$
`MSR7`	LTL	$G(Fail \rightarrow Xfinal)$

we use temporal logic [56], as it is a language for formally expressing system behavior over time. This makes it appropriate for specifying if-then conditions over the exchange of messages during execution.

We use two temporal logic languages to represent our rules; *Linear Temporal Logic* (LTL) and *Computational Tree Logic* (CTL). While similar, these languages have different timeline representations (linear and branching timelines respectively), and are non-equivalent in the properties they are able to specify [56]. Both languages are necessary to represent our rules set completely, as we will show.

Temporal logic allows complex properties to be constructed from boolean variables using a set of *temporal operators*. We will briefly outline those used in our work, rather than provide a comprehensive guide. Both LTL and CTL support basic propositional logic operators for *AND* ($\wedge$), *XOR* ($\vee$), *not* ($\neg$), and *implication* ($\rightarrow$). Supported temporal operators include *next* (X) to indicate a property of the next state, *until* ($\cup$) to indicate that one property should hold until another is satisfied, *global* (G) for properties that are true for all states, and *future* (F) for properties that will become true at least once in the future. We also use past operators supported by LTL, which include *once* (O) to indicate a property that has held during at least one previous state, and *historically* (H) to indicate a property that has held for all previous states. Furthermore, the branching timeline used by CTL allows for the specification of *path quantifiers*, such as *'for all paths'* (A), meaning for all possible future execution paths, and *'for some path'* ($\exists$), for properties that will be true during at least one execution path.

The temporal logic transformations of our conversation rule set is shown in Table 3.10. Most transformations use LTL with G and X operators to specify that all messages of certain types must always be followed by other specific types. For example, `MS1` formalizes the behavior that all states with *Sync*, *Ack*, or *Recover*, implicate the next state to contain either *Success*, *Fail*, *Ping*, *Fault*, *Syncreq*, or *Delay*. `CSR2` requires a CTL and LTL property to formalize. A CTL properties specifies that the final state is reachable for all executions, while the LTL property ensures all invalid termination messages never transition to the final state.

3.4.2 State Space Reduction

As model checking is an exhaustive technique, it is subject to the state space explosion problem, in which the state space of the system under verification increases exponentially in relation to the number of processes and variables. This greatly impacts verification time and feasibility for complex models. Therefore, as part of our transformation of the control and operational behavior models for model checking, we aim to reduce the state space as much as possible.

For our state-space reduction measures, we transform the control and operational behavior models into a Kripke structure [89], as it is a concise representation of events occurring within a system, and commonly used for formal verification [132]. A Kripke structure is a finite-state system model with a directed graph structure, where each node represents a system state where one or more properties are satisfied. We formally define a Kripke structure as $\mathcal{K} = \langle \mathcal{S}^k, \mathcal{T}^k, \mathcal{L} \rangle$, where:

- $\mathcal{S}^k$ is a finite set of states
- $s_0 \subseteq \mathcal{S}^k$ is the initial state
- $s_{fin} \subseteq \mathcal{S}^k$ is the final state
- $\mathcal{T}^k \subseteq \mathcal{S}^k \times \mathcal{S}^k$ is the transition function
- $\mathcal{L}$ is the labelling function that assigns *atomic propositions* to each state

Atomic propositions are a set properties that hold at a given state. Mapping atomic propositions

Table 3.11: A set of inter-behavior messages defined over the online payment design

Message	Source	Target	Condition
Sync	*Activated*	*GET Customer Data*	`no message`
	Activated	*PUT Card Data*	`Resale SYNCREQ`
	Activated	*PUT Account Data*	`Resale SYNCREQ`
Syncreq	*Resale*	*Activated*	–
Delay	*Activated*	*MultiSale Card*	–
Fault	*MultiSale Card*	*Activated*	–
	Check Sales	*Activated*	–
Success	*Sale OK*	*Activated*	–
	Request Processed	*Rollback*	–
Fail	*Decline Sale*	*Rollback*	–
	Create Report	*Rollback*	–
Recover	*Rollback*	*GET Sale Result*	`Sale OK SUCCESS`
	Rollback	*High Fault Score*	`MultiSale Card FAULT`
	Rollback	*Decline Sale*	`FAULT`

to each state enables the Kripke structure to store the temporal dependencies between events in a system.As we aim to verify the inter-behavior conversation in the model, the atomic propositions in the Kripke structure we generate will be the inter-behavior messages as they are sent in the model.

In order to reduce the state space, we use an algorithm to generate a Kripke structure that captures the inter-behavior messages sent between the control and operational berhavior models. The algorithm works by traversing the operational behavior model and constructing Kripke structure states and transitions as inter-behavior messages are encountered. To enable this traversal, we first transform the inter-behavior messages and operational behavior to a single model.

3.4.2.1 Flattened Behavior Model

The *flattened behavior model* $\mathcal{B}^f = \langle \mathcal{S}^f, \mathcal{L}^f, \mathcal{T}^f, \mathcal{F}^f \rangle$ enables the operational behavior model and inter-behavior messages to be efficiently analyzed by combining them into a single statechart. The traversal at the core of the state-space reduction algorithm can be simplified by using this single model as input, rather than interacting control and operational berhavior models. This model expands the operational behavior by replacing every inter-behavior message with a new state. For example, in place of a *Sync* message targeting the state *GET Customer Data*, a state is created named *Sync_GET Customer Data* with an outgoing transition to *GET Customer Data*. Therefore, the states of the flattened behavior model $\mathcal{S}^f = S^o \cup S^m$, where S^m are *message states* generated from inter-behavior messages. Considering the control and operational behavior of Figure 3.4 and the inter-behavior messages shown in Table 3.11, the resulting flattened behavior model is shown in Figure 3.6.

Message states are systematically created with transitions to or from their relevant operational behavior state. *Sync* and *Recover* message states transition to the operational behavior states they target. *Ack* message states transition to their source operational behavior state, because as shown in Table 3.7, operational behavior execution resumes from that state once they are sent. Message states of all other types are created with an incoming transition from their operational behavior state, as they are not exchanged before the state is entered, and they do not cause execution to immediately resume.

Next, transitions must be created between message states, in order to capture all possible execution paths through the control and operational behavior models. These transitions are created depending on i) the active control behavior state after sending or receiving each message type, and ii) the ECA labels on the inter-behavior messages that dictate when they can be sent. For example, *Syncreq_Resale* transitions to *Sync_PUT Card Data* and *Sync_PUT Account Data* because it satisfies their labeled condition, and because *Sync* can be sent as the control behavior transitions from *Suspended* to *Activated*.

Finally, an *initial* and *final* state are added. The initial state transitions to all *Sync* message states that contain a valid or empty condition label. Transitions to the final state are created from all states with no outgoing transitions. All *Success* message states also transition to final, even though they may already contain outgoing transitions for compensation, such as *Success_Sale OK* in Figure 3.6.

3.4.2.2 Reduced Kripke Structure Algorithm

Using our algorithm, *Conversation Checking Kripke Structure Reduction* (*CKSR*), we generate a Kripke structure capturing the temporal dependencies between all inter-behavior behavior messages in the flattened behavior model. The result is a model containing the behavior we aim to verify, with a reduction in state space for model checking. Each state in the resulting Kripke struc-

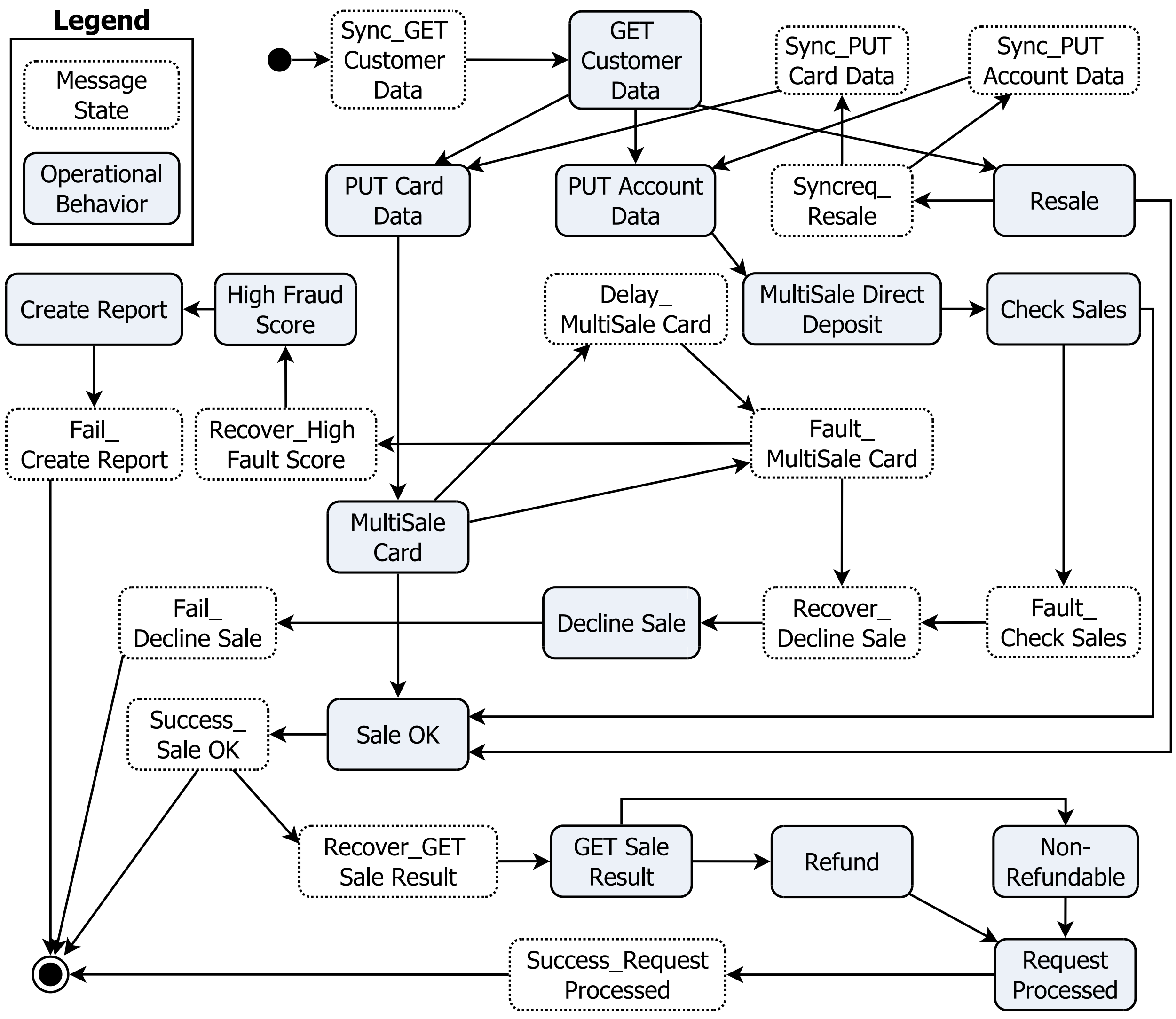


Figure 3.6: Flattened behavior model of the online payment composition

ture will contain an inter-behavior message type and related (target or source) operational behavior state as atomic propositions. The model will be reduced such that only states that send an inter-behavior message are included in the Kripke structure. This provides an ideal representation to verify against our conversation rules set.

CKSR traverses the flattened behavior model in a depth-first order, and constructs Kripke states and transitions as message states are encountered. We formalize *CKSR* in Algorithm 1, which uses the recursive procedure *Conversation Checking Depth-First Traversal* (*CDF*) in Algorithm 2. *CKSR* initializes the Kripke structure with initial and final states before invoking CDF on the initial state of the flattened behavior model.

CDF requires three input parameters. Firstly, a state in the flattened behavior model s_x is required as the active state in the traversal. *CDF* will traverse the outgoing transitions from s_x at each invocation. Secondly, the previously visited message state, in terms of recursive hierarchy, is passed as s_m. This state is required as the source state when transitions are added to the Kripke structure. Finally, a set of flattened behavior states visited since the last message state are passed as S_v. This is required to detect iterative control flow patterns in the model and prevent infinite recursion.

At each invocation, *CDF* visits all states in S^f that are targeted by a satisfiable transition from the input state s_x (lines `1, 2`). The first check for each visited state is whether the final state has been reached. If so, a transition to the final state in the Kripke structure is created (lines `3-5`). Next, if a message state is targeted, then the Kripke structure must be updated (line `6`). If a state corresponding to the message state exists in S^{kc}, then a transition to it is created (lines `7, 8`). Otherwise, a new Kripke state is created and added to S^{kc} as well as the new transition (lines `9-11`). *CDF* is then called recursively, as the flattenend behavior model has not yet been explored beyond this new message state (line `12`). If the targeted state is not a message state, then the algorithm checks if it has been visited since the last Kripke state was added (line `15`). If so, then a loop has been detected and the algorithm does not continue. Otherwise, the targeted state is added to the visited state set (line `16`), and *CDF* recursively calls itself with updated parameters (line `17`).

Algorithm 1 Conversation Checking Kripke Structure Reduction: *CKSR*

```
1: $K^c \leftarrow \langle \mathcal{S}^{kc}, \mathcal{T}^{kc}, \mathcal{L}^c \rangle$
2: Add $s_0$ and $s_{fin}$ to $S^{kc}$
3: CDF($s_0$, $s_0$, $\emptyset$)
4: return $K^c$
```

Algorithm 2 Conversation Checking Depth-First Traversal: *CDF*(s_x, s_m, S_v)

```
1: for $s \in S^f$ where $(s_x, l, s) \in T^f$ do
2:     if $l$ is satisfied then
3:         if $s = s_{fin}$ then
4:             Add $(s_m, s_{fin})$ to $T^{kc}$
5:         end if
6:         if $s \in S_m$ then
7:             if $s \in S^{kc}$ then
8:                 Add $(s_m, s)$ to $T^{kc}$
9:             else
10:                Add $s$ to $S^{kc}$
11:                Add $(s_m, s)$ to $T^{kc}$
12:                CDF($s$, $s$, $\emptyset$)
13:            end if
14:        else
15:            if $s \notin S^v$ then
16:                Add $s$ to $S_v$
17:                CDF($s$, $s_m$, $S_v$)
18:            end if
19:        end if
20:    end if
21: end for
22: return
```

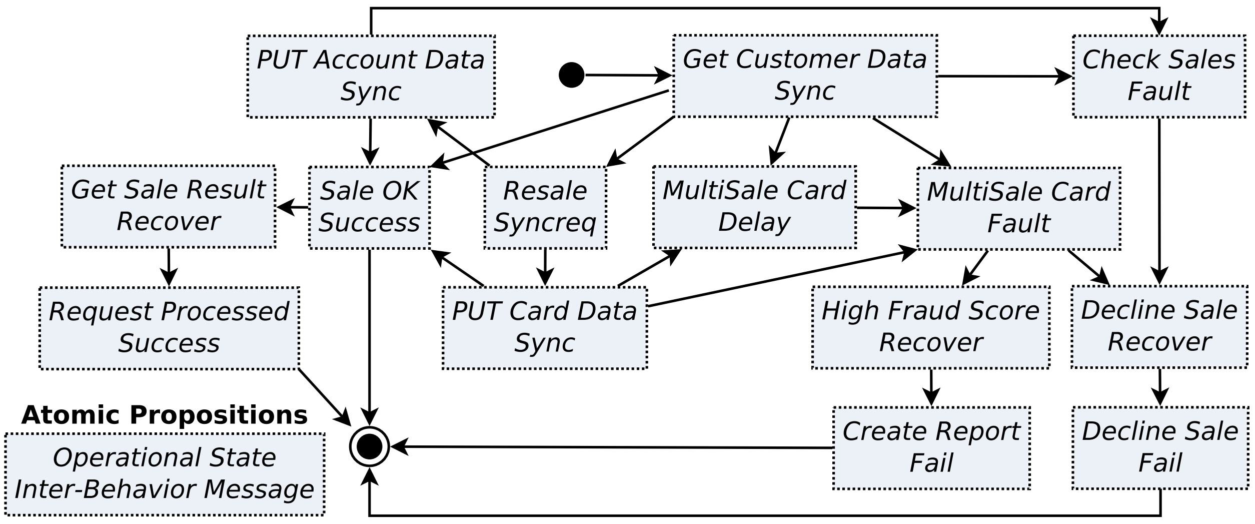


Figure 3.7: Reduced Kripke structure for verification against conversation rules

Figure 3.7 shows the Kripke structure generated by *CKSR* with the input of the flattened behavior model shown in Figure 3.6. This reduces the original models from two statecharts with total 28 states and 14 inter-behaviour messages, to a structure containing 16 states.

The temporal logic representations of conversation rules in Table 3.10 assume, through the use of X operators, that a new inter-behavior message is exchanged with every state. As shown in Figure 3.7, the reduced Kripke structures produced by *CKSR* meet this criteria. However, in order to use the temporal logic properties in for model checking a specific Kripke structure, the properties must be implemented such that they apply to the states it contains. For example, in place of $Sync$, the model checking input contains every state that contains $Sync$ as an atomic proposition, separated with $\vee$ operators. This pattern is used for every inter-behavior message in the conversation rules. Table 3.12 contains the temporal logic properties for verifying the Kripke structure of Figure 3.7. `MS5` is omitted as the model does not contain any *Ping* messages, making in unnecessary to include. We implement this in our prototype tool, and show how model checking is applied to these temporal logic properties and Kripke structures in Chapter 6.

Table 3.12: Temporal logic conversation rules for verifying a Kripke structure

Rule	**Temporal Logic Property**
CSR1	$initial \cup (Sync\_GetCustomerData \vee Sync\_PUTCardData \vee Sync\_PUTAccountData)$
CSR2	$AG(\exists F(final))$
	$G((Sync\_GetCustomerData \vee Sync\_PUTCardData \vee Sync\_PUTAccountData \vee Recover\_GETSaleResult \vee Recover\_HighFraudScore \vee Recover\_DeclineSale \vee Fault\_CheckSales \vee Fault\_MultiSaleCard) \rightarrow X(\neg final))$
MS1	$G((Sync\_GetCustomerData \vee Sync\_PUTCardData \vee Sync\_PUTAccountData \vee Recover\_GETSaleResult \vee Recover\_HighFraudScore \vee Recover\_DeclineSale) \rightarrow X(Success\_SaleOK \vee Success\_RequestProcessed \vee Fail\_CreateReport \vee Fail\_DeclineSale \vee Fault\_CheckSales \vee Fault\_MultiSaleCard \vee Syncreq\_Resale \vee Delay\_MultiSaleCard))$
MS2	$G(Delay\_MultiSaleCard \rightarrow X(Success\_SaleOK \vee Success\_RequestProcessed \vee Fail\_CreateReport \vee Fail\_DeclineSale \vee Syncreq\_Resale \vee Fault\_CheckSales \vee Fault\_MultiSaleCard))$
MS3	$G(Syncreq\_Resale \rightarrow X(Sync\_GetCustomerData \vee Sync\_PUTCardData \vee Sync\_PUTAccountData \vee final))$
MS4	$G((Fault\_CheckSales \vee Fault\_MultiSaleCard) \rightarrow X(Recover\_GETSaleResult \vee Recover\_HighFraudScore \vee Recover\_DeclineSale))$
MS5	N/A
MS6	$G((Success\_SaleOK \vee Success\_RequestProcessed) \rightarrow X(final \vee Recover\_GETSaleResult \vee Recover\_HighFraudScore \vee Recover\_DeclineSale))$
MS7	$G((Fail\_CreateReport \vee Fail\_DeclineSale) \rightarrow Xfinal)$

3.5 Summary

In this chapter, we present our approach for modeling transactional service-oriented processes at design-time, and a verification process for ensuring well-formed transactional behavior. This enables reliability issues in the transactional behavior of the process to be identified and resolved at design-time, to avoid costly redevelopment or runtime consequences.

Our modeling approach separates process behavior into two models, control and operational behavior, which provide transactional and functional views of the process respectively. These models exchange a set of inter-behavior messages to pass instructions and remain aware of each other's status. The transactional behavior of the process can be checked for reliability issues by verifying the conversations between behavior models. We define a set of conversation rules, that ensure inter-behavior conversations initiate and terminate in a valid way, while avoiding deadlock.

Model checking can be used to verify transactional service-oriented processes for reliability issues with our conversation rule set. The conversation rules are formalized in LTL and CTL, before model checking is applied to formally and exhaustively verify the process models against them. Furthermore, we reduce the state space of the model prior to verification via a reduced Kripke structure algorithm, to improve verification performance for large and complex models.

CHAPTER 4

Temporal Logic Templates for Application-Dependent Transactional Requirements

In the previous chapter, we outlined an approach to verify service-oriented processes at design-time against rules for well-formed transactional behavior. While this verification method can identify violations of application-independent correctness criteria, the issue of ensuring transactional requirements drawn from business logic remains. Such requirements include acceptable states for process termination, specific fault-handling measures for certain component tasks, and components considered critical for successful execution, among other possibilities. To address this, we apply the transactional service-oriented process modeling approach introduced in the previous chapter towards ensuring application-*dependent* requirements at design-time.

We first overview work in service-oriented processes and temporal logic patterns related to our approach, and identify the open issues we aim to fill with our contribution. We apply the same comparison criteria as the previous chapter, but also compare approaches according to the specification methods they used for formalizing transactional requirements. Our survey found that most approaches automatically conduct transactional behavior in a way that preserves requirements specified by the user, rather than verifying manually specified transactional behavior. However, our analysis of the requirement specification methods used identified that many are prone to scalability issues with large or revised models, or are unable to formalize complex requirements.

In order to provide a specification method with strong scalability and expressibility, aur approach uses *temporal logic templates* to elicit and formalize transactional requirements from developers. These templates enable developers to manually specify and verify transactional service-oriented processes according to their own business logic. Our temporal logic template set is divided into sets, for specifying both component-level and process-level transactional requirements. The developer can formalize transactional requirements in temporal logic by assigning variables from a template that suits the requirement. Our approach enables complex and varied transactional requirements to be formalized without requiring expertise in formal methods.

Model checking is used to ensure these requirements are satisfied by the design. We show how we address state-space explosion prior to model checking, in order to improve verification performance with large and complex models and requirement sets. An algorithm is applied that generates a reduced Kripke structure from the service-oriented process, modeled as control and operational behaviors.

4.1 Related Work

Our work follows recent efforts in ensuring that application-dependent transactional requirements are satisfied in service-oriented processes. Furthermore, our approach uses temporal logic templates, which builds on research in temporal logic patterns. We provide an overview of both areas and clarify the contribution of our work below.

4.1.1 Ensuring Transactional Requirements in Service-Oriented Processes

For providing an overview of the research in this area, we first define the criteria used to compare each approach. Next, we apply our criteria to our surveyed approaches in order to identify the issues to address with our work.

4.1.1.1 Comparison Criteria

We apply the same set of comparison criteria defined in the previous chapter (Table 3.1) for approaches to ensure application-independent requirements. However, as this section addresses transactional requirements drawn from business logic, each approach requires a method to elicit and formalize such requirements from the developer. Therefore, we include an additional criterion in the *Method* category called *Specification Method*. We identify four properties to consider during our comparison of specification methods:

- *Expressiveness* refers to the complexity of requirements that the method is able to specify. For example, a highly expressive method may be able to specify requirements over the transactional behavior of both the components and the process, and may consider numerous fault types, or a wide range of fault-handling behavior. A less expressive method would be restricted to much simpler requirements.
- *Flexibility* indicates how well the specification method can support cases of simple *and* complex requirement sets. A flexible specification method can support transactional requirement sets varying in complexity, making it applicable to a wider range of business scenarios.
- *Scalability* considers how laborious the process of requirements specification becomes, or how feasible the necessary verification or assurance method remains, as the size and complexity of the process model increases.
- *Usability* refers to the ease of requirements specification for the developer. For example, a method with poor usability may require expert knowledge in a formal language, while a method with high usability may only require a set of variables to be assigned.

4.1.1.2 Survey

We first give an overview of design-time verification approaches, followed by service provisioning strategies, development and execution frameworks, as well as others that do not fit into these categories.

Kokash and Arbab [85] developed a tool that uses model checking to verify the design of long-running transactions against temporal logic properties. The design is specified using the coordination language *Reo*, which comprises of components and services connected with different types of communication channels. The model also makes use of constraint automata to specify data constraints over node input and output. Temporal logic properties are specified manually by the developer using LTL and *Alternating-time Stream Logic* (ASL) [84], which is a variant

Table 4.1: *Overview* criteria of application-dependent requirements approaches.

Approach	Lifecycle Phase	Implementation
Kokash and Arbab [85]	Design	Verification tool
WebTransact [122, 123]	Design, development	Composition framework
Vidyasankar and Vossen [149]	Design, development	Web Service composition model
Bhiri et al. [18]	Design, development	Static service provisioning strategy
Montagut et al. [111]	Design, development	Static service provisioning strategy
El Haddad et al. [53]	Design, development	Static service provisioning strategy
Johny [81]	Design, development	Static service provisioning strategy, WS-BPEL generator
FACTS [100]	Design, development	Composition framework
Zheng and Lyu [161]	Design, development	Fault tolerance strategy selection
Kovács et al. [87]	Development	WS-BPEL process formal modeling for model checking
eFlow [34]	All	Composition and execution framework
FENECIA [12]	All	Composition and execution framework
Montagut et al. [112]	All	Distributed workflow composition and execution framework
MASC [57]	Runtime	Service management middleware
TRAP/BPEL [58]	Runtime	Execution framework
Baresi and Guinea [10]	Runtime	Execution framework
Simmonds et al. [135]	Runtime	Execution framework

of *Computational Tree Logic* (CTL) that is able to express conditions on data flow with regular expressions.

We identified several approaches that apply a service selection and provisioning strategy focused on ensuring transactional requirements. Bhiri et al. [18] use a workflow skeleton of tasks for representing the process with no provisioned Web services. The developer is required to to implement an *Accepted Termination States* (ATS) model [83] to exhaustively define the valid termination states. To satisfy the ATS model, Web services are selected to fit the workflow skeleton from candidate sets of functionally-equivalent services according to their transactional properties and a set of transactional composition rules. As in the authors' related work [17], transactional dependencies between component services, inferred from the transactional properties of services and the workflow patterns used, determine the fault-handling behavior of the composition. This work has been followed by other service-provisioning strategies. In the approach by Montagut et al. [111], the ATS model is specified using a top-down method, by reducing a larger set of valid termination states, determined from execution dependencies in the workflow. El Haddad et al. [53] use an algorithm to select Web services according to QoS properties as well as transactional

Table 4.2: *Support* criteria of application-independent requirements approaches.

Approach	Process Model					
	AND	XOR	IOR	Itr	TP	Notes
Kokash and Arbab [85]	√	√				Abstract data domain
WebTransact [122, 123]		√			√	Data dependency links between components
Vidyasankar and Vossen [149]	√	√			√	
Bhiri et al. [18]	√	√			√	
Montagut et al. [111]	√	√			√	
El Haddad et al. [53]	√	√			√	
Johny [81]	√				√	
FACTS [100]	√	√			√	
Zheng and Lyu [161]	√	√		√		
Kovács et al. [87]	√	√	√	√		WS-BPEL equivalent, process variables with transactional states
eFlow [34]	√	√				
FENECIA [12]	√	√	√	√		Also supports selection and exclusive merge, and rendezvous patterns
Montagut et al. [112]	√	√			√	
MASC [57]	√	√	√	√		Microsoft Windows Workflow Foundation [5] equivalent
TRAP/BPEL [58]	√	√	√	√		WS-BPEL equivalent
Baresi and Guinea [10]	√	√	√	√		WS-BPEL equivalent
Simmonds et al. [135]	√	√	√	√		WS-BPEL equivalent

properties. Instead of an ATS model for transactional requirements, the developer specifies a binary variable signifying whether the resulting composition should be compensatable or not. This general approach is adapted in a broker-based framework proposed by Johny [81].

Several development and execution frameworks have been proposed for Web service compositions. A prominent example is the *Failure Endurable Nested-Transaction Based Execution of Composite Web Services* (FENECIA) framework [12]. FENECIA is a combination of three components: the *WS-SAGAS* transactional model for Web service composition specification, the *THROWS* execution architecture, and a QoS estimation and analysis model. The WS-SAGAS transaction model supports a large set of workflow patterns [147]. Dynamic provisoning, reprovisioning, and rollback of component services is enabled according to their transactional properties. FENECIA uses mandatory tasks as a means for developers to specify transactional requirements. During execution, if a fault occurs at a mandatory task, and the service cannot be replaced, then the composition must be aborted.

The *eFlow* framework [34] enables composition, deployment, and execution of Web services

Table 4.3: *Method* criteria of application-dependent requirements approaches.

Approach	**Transactional Requirements**	**Specification Method**	**Verification or Assurance**
Kokash and Arbab [85]	Temporal logic over rollback-enabled data-aware processes	LTL and ASL properties	Model Checking
WebTransact [122, 123]	Atomic termination, mandatory and desirable services for commit	A set of mandatory tasks for commit	Service provisioning
Vidyasankar and Vossen [149]	Atomic termination, preference ordering of alternatives	Set of acceptable pivot operations	Composition rules
Bhiri et al. [18]	Relaxed atomicity conditions	ATS model	Service provisioning
Montagut et al. [111]	Relaxed atomicity conditions	ATS model	Service provisioning
El Haddad et al. [53]	Composition compensatability	Binary parameter	Service provisioning
Johny [81]	Composition compensatability	Binary parameter	Service provisioning
FACTS [100]	Fault-tolerance, mandatory tasks for commit	Mandatory tasks for commit	Verification algorithm
Zheng and Lyu [161]	Fault tolerance	Optimization problem constraints	Optimization problem with heuristic algorithm
Kovács et al. [87]	LTL properties over process behavior and variable states	Manually specified LTL properties	Model checking
eFlow [34]	Configurable atomicity and isolation levels in process regions	Atomic regions and required rollback tasks	ACID transactions and rollback
FENECIA [12]	Fault-tolerance, mandatory tasks for commit	Mandatory tasks for commit	Runtime reprovisioning and rollback
Montagut et al. [112]	Relaxed atomicity conditions	ATS model of critical regions	Dynamic partner allocation
MASC [57]	Component-service level fault-tolerance requirements	XML-based fault-handling policies	Policies attached to component services
TRAP/BPEL [58]	Component-service level fault-tolerance requirements	XML-based policy file	Dynamic adaptation via generic proxy
Baresi and Guinea [10]	Pre and post-conditions for integrity, reliability, and availability	WSCoL assertion language	Monitoring interactions and invoking user-defined recovery strategies
Simmonds et al. [135]	Application-dependent safety and liveness requirements	Colored labeled transition systems	Dynamic generation and ranking of recovery strategies

Table 4.4: *Transactional behavior* criteria of application-dependent requirements approaches.

Approach	**Faults**	**Fault-handling**	**Specification**
Kokash and Arbab [85]	Behavioral, timeout, communication	Rollback, compensation, cancellation	Manual
WebTransact [122, 123]	Behavioral	Retrial, rollback, 2PC	Automated
Vidyasankar and Vossen [149]	Behavioral	Retrial, rollback, compensation, alternative	Automated
Bhiri et al. [18]	Behavioral	Retrial, rollback, cancellation	Automated
Montagut et al. [111]	Behavioral	Retrial, rollback, cancellation	Automated
El Haddad et al. [53]	Behavioral	Retrial, rollback	Automated
Johny [81]	Behavioral	Retrial, rollback	Automated
FACTS [100]	Behavioral, contract violations, availability, logical	Retrial, rollback, waiting, cancellation, alternative, redundancy, ignore, log, skip	Semi-automated
Zheng and Lyu [161]	Behavioral, availability, communication, logical	Retrial, alternative, redundancy, n-version strategy	Automated
Kovács et al. [87]	Behavioral, communication	Rollback, compensation, unspecified forward recovery	Manual
eFlow [34]	Behavioral	Rollback, cancellation	Manual
FENECIA [12]	Behavioral, communication	Rollback, cancellation, reprovisioning	Automated
Montagut et al. [112]	Behavioral	Retrial, rollback, cancellation, reprovisioning	Automated
MASC [57]	Behavioral, contract violations, communication	Retrial, reprovisioning, redundancy, restructuring	Manual
TRAP/BPEL [58]	Behavioral, timeout	Retrial, rollback, reprovisioning	Manual
Baresi and Guinea [10]	Behavioral, communication, availability	Retrial, rollback, reprovisioning, log	Semi-automated
Simmonds et al. [135]	Behavioral, requirement violations	Retrial, rollback, user intervention, reprovisioning, restructuring	Semi-automated

using ACID transactions for component service invocation and applies rollback in response to faults. Another publication by Montagut et al. [112] presents a system for transactional support of long-running distributed workflow execution, which uses an ATS model defined for *critical zones* of the workflow, where transactional requirements must be satisfied. The *Framework for Fault-Tolerant Composition of Web Services* (FACTS) [100] enables specification and Web service compositions with transactional behavior, and verification that faults occurring at services specified as mandatory are validly handled. *WebTransact* [122, 123] is a Web service composition framework that aggregates functionally equivalent candidate services for each operation. These aggregations form *mediator services*, and can be composed by the developer according to *interaction patterns*.

The remaining approaches to discuss in this section are outside the above categories. Vidyasankar and Vossen [149] present a Web service composition model that constructs a tree structure from the pivot operations in the process, such that the leaves of each node are the alternative operations ordered by developer preference. Users of the process can specify their own execution requirements in the form of acceptable pivot operations. This prunes invalid alternatives from the pivot the tree as necessary. Kovács et al. [87] propose a method to formally model WS-BPEL processes, with fault-handling and process variables, to enable verification through model checking. *Manageable and Adaptable Service Composition* (MASC) [57] is a service management middleware to enable the specification and enforcement of extensive fault recovery policies of component services. Lastly, Simmonds et al. [135] present an execution framework for user-guided recovery for violation of safety and liveness behavioral properties in WS-BPEL processes. This approach computes recovery plans in response to violations, which are a sequence of backward and forward recovery actions to meet the goal specified by the property.

4.1.2 Temporal Logic Patterns

The approach we employ to elicit transactional requirements from the developer adapts existing work in *temporal logic patterns*. These patterns are empty structures of temporal logic taken from common or useful properties [52]. They were developed to aide the use of temporal logic for formal verification, by reducing human error and effort for specifying common properties. We provide a brief overview of the state-of-the-art in temporal logic patterns before clarifying our research direction.

Dwyer et al. [52] defined a set of configurable patterns based on common LTL and CTL property structures found in surveyed specifications. This enabled formal verification tools such as model checkers to be used more broadly by reducing the need for expertise in temporal logic. These patterns are abstract temporal logic structures for expressing behavior such as *absence*, *existence*, and *precedence* of states or events. Furthermore, *scope* values can be applied to limit states of the system where the patterns must hold.

The temporal logic patterns proposed by Dwyer et al. have been adapted and extended in subsequent research. Cardinality between variables and other configuration options were introduced into patterns by Smith et al. [136]. Other approaches include Elgammal et al. [54], who propose a framework to compose atomic patterns to specify more complex properties, and Yu et al. [158], who apply temporal logic patterns to business rule compliance in WS-BPEL schemas. Our work draws on contributions from these approaches, such as allowing properties to be adjusted according to scope and cardinality values. However, our focus is on transactional requirements for service-oriented processes, so our proposed template set is specialized and more applicable to this domain.

The most comparable approach to our work is the Web service composition verification framework proposed by Fantechi et al. [59]. Their approach uses model checking to verify service-oriented systems against properties implemented from temporal logic templates. The templates are constructed in *Service-oriented computing Language* (SocL), a branching-time logic. Unlike our work, the focus of these templates is to formalize interaction requirements of component services, such as specifying that a service may only accept one request between responses. Our approach focuses on the transactional requirements that apply to such components, but also extend the focus on requirements applicable to the scope of the process.

4.1.3 Research Direction

As with the approaches limited to application-independent transactional requirements, these application-*dependent* approaches mostly utilize automated transactional behavior (52.94% automated and a further 17.65% semi-automated). Furthermore, the *Implementation* field of Table 4.1 and the *Verification or Assurance* field of Table 4.3 indicate that verification is not commonly used in this area (in 11.76% of approaches). As our aim is to ensure prior to development that the transactional requirements of the process are satisfied, our approach uses manually defined transactional behavior. Verifying manually specified transactional behavior allows developers to maintain complete control of how their process behaves in response to faults, and identify major issues early in the process lifecycle. For example, determining if a set of transactional requirements are unrealizable in their current form, or identifying conflicts between their own transactional requirements and pre-established requirements for interacting with a business partner involved in the process.

Table 4.3 indicates that the transactional requirement specification methods used in the state-of-the-art are often either highly expressive or easy to use. For example, ATS models allow an exhaustive definition of valid process termination states, but it is laborious to define for large models, and requires non-trivial revisions if the structure of the process is modified. Similarly, manually specified temporal logic properties are highly expressive, but require language expertise and can be error-prone to define. In contrast, binary variables or mandatory tasks for commit are simple methods that are flexible to changes or scaling in the process model, but are unable to express more complex transactional requirements. We aim to address this with a requirement specification method that is usable, expressive, and scalable to large process models.

The body of work in temporal logic patterns is, to the best of our knowledge, is yet to be adapted towards the specification of transactional requirements in service-oriented processes. These would provide a benefit over many of the specification methods proposed in related work, as they allow complex temporal logic properties to be formalized without expertise. The initial set of patterns provided by Dwyer et al. [52] support high-level properties that are broadly applicable to many systems. In contrast, we aim to enable verification of a specific model. Therefore, we plan to adapt the concept of temporal logic patterns (as temporal logic *templates*), while proposing our own set of properties that are specialized towards transactional requirements for service-oriented processes. The transsactional requirements that our template set will be able to specify will be gathered from common and useful requirements in existing work.

Our approach to address these issues is to verify service-oriented processes at design-time against transactional requirements elicited from the developer using a specification method that is both expressive and easy to use. We plan to build on the contribution presented Chapter 3, by applying the same modeling approach of control and operational behaviors with manually defined transactional behavior. This is appropriate as it provides a view of the functional and transactional states of the process, and allows developers to specify detailed transactional behavior through messages.

4.2 Formalizing Transactional Requirements

A developer must have confidence that a potentially long-running process of distributed and heterogeneous services will conform to a set of transactional requirements for containing and handling faults. These requirements could apply to failures of individual components, e.g., required recovery operations to undo the component's effect, or to the scope of the process, such as components critical for success, or relaxed atomicity conditions for failure. Violating such requirements could compromise internal business policies, leading to issues with consistency of distributed data, consumer relations, or legal obligations. Therefore, it is crucial to identify and resolve any compliance issues prior to development, which requires a formal specification method.

To make formalizing complex and diverse transactional requirements feasible for non-experts, we propose the use of *temporal logic templates*. This approach adapts previous work in temporal logic patterns [52], which simplify property specification by identifying common temporal logic property structures and presenting them for use in an implementable form. Similarly, our work identifies common transactional requirements for service-oriented processes and provides temporal logic properties in a template form, which can be implemented by developers by assigning variables. This leverages the expressive power of temporal logic without requiring users to have expert knowledge in the language. It also reduces the effort and the capacity for human error during requirement formalization.

As transactional requirements can vary in scope, we propose two sets of templates. *Component-level* templates can be used to formalise requirements specific to certain component services in the process, such as defining acceptable alternatives or recovery operations following failure. In contrast, *process-level* templates implement transactional requirements that apply to the scope of the whole process, e.g., to specify pre-conditions for valid committing or cancellation of the process. Both template sets can be used to formalize transactional requirements for a process modeled as interacting control and operational behavior models, as introduced in Section 3.2. Both LTL and CTL is used, in order to increase the range of transactional requirements that our templates are able to specify.

We overview each set of templates below, and provide a full specification of one from each set. Full specifications of all templates are omitted from this chapter for readability, but can be found in Appendix A. We do not claim that our template set is complete, but rather provide a foundational set that is simple to extend. Further templates could be added by defining further specifications for requirements not already supported. Alternatively, the templates could be added to expand the set into other types of requirements, such as more general business rules [54].

4.2.1 Component-Level Templates

Component-level templates specify requirements for handling the failures of individual component services in the process. For example, considering the online payment process in Figure 3.4, after the failure of the *Resale* operation, either *PUT Card Data* or *PUT Account Data* can be attempted as alternative operations to continue the sale. From analyzing the common transactional properties applied to component services in related work (Table 4.2), we identify six component-level templates. We list and briefly describe each below.

- **CompensateFailure** specifies that the failure of a certain component must be recovered with further operations. The developer specifies a component and a condition, such that the failure of the component requires the condition to be satisfied in the future.

- **CompensateSuccess** is used when the effect of a component must be undone after it has previously completed. This is required if a process, following the successful completion of that component, must be aborted or compensated. A component and condition is specified by the developer, such that the condition is required to be satisfied prior to abortion or during compensation, if the component has completed earlier.

- **Alternative** specifies that following the failure of a component, one or several alternative operations, are considered acceptable replacements. Potential alternatives are expressed as a condition variable.

- **NonRetriable** defines a component that should not be retried following failure in the same process instance.

- **RetriablePivot** specifies a component that may be retried, but its effect once completed cannot be undone. Following successful completion of the component, the process must commit.

- **NonRetriablePivot** specifies that a component may not be retried after failure or undone once completed. The process must therefore commit or abort depending on the success of the execution of the component.

Table 4.5 shows the complete specification of the `CompensateFailure` template. A full specification of a template contains several fields to enable developers to fully understand and utilize the template. The variables field lists the variables that must be assigned by the developer in order to implement the template. These can be single components, as in *Component*, a basic propositional logic condition, as in *Recovery*, or a selection from a discrete set of options, such as *Card* and *Scope*. The *Card* value (short for *cardinality*) is used to specify the relationship between two variables, while *Scope* restricts when the requirement is applicable. These two variables increase the flexibility of the template and remove ambiguity from the formalization.

The *Description* field provides information on what requirements the template can be used to implement. *Prerequisite* identifies any basic features that the model must include before the transactional requirement can be verified. For example, in order to specify a `CompensateFailure`

Table 4.5: Template specification for `CompensateFailure`

Name	`CompensateFailure <Component,Recovery,Card,Scope>`		
Type	Component-level		
Variables	*Component*	An operational behavior state that requires recovery upon failure.	
	Recovery	A condition that undoes the effect of the failure. This can be a single component or a set of components structured with $\vee$ operators.	
	Card	One of the cardinality options below.	
	Scope	One of the scope options below.	
Description	The failure of *Component* leaves an impact an effect, which must be compensated by *Recovery* becoming true in the future.		
Prerequisites	A `Fault` message originating from *Component* in the operational behavior is necessary for this requirement to be verified.		
Cardinality	1:1	*Recovery* undoes one failure of *Component*.	
	Many:1	*Recovery* can undo many failures of *Component*.	
Scope	G	The template applies in all executions.	
	P	Applies during the satisfaction of a condition P.	
	$\neg P$	Applies during the negation of a condition P.	
	Before P	*Recover* must precede the satisfaction of P.	
LTL	1:1	G	$G(Component.FAULT \rightarrow ((\neg(Activated \wedge Component) \cup Recovery) \wedge F(Recovery))$
		P	$F(P) \rightarrow G(Component.FAULT \rightarrow ((\neg(Activated \wedge Component) \cup Recovery) \wedge F(Recovery))$
		$\neg P$	$G(\neg P) \rightarrow G(Component.FAULT \rightarrow ((\neg(Activated \wedge Component) \cup Recovery) \wedge F(Recovery))$
		Before P	$G(Component.FAULT \rightarrow (((\neg(Activated \wedge Component) \wedge \neg P) \cup Recovery) \wedge F(Recovery))$
	Many:1	G	$G(Component.FAULT \rightarrow F(Recovery))$
		P	$F(P) \rightarrow G(Component.FAULT \rightarrow F(Recovery))$
		$\neg P$	$G(\neg P) \rightarrow G(Component.FAULT \rightarrow F(Recovery))$
		Before P	$G(Component.FAULT \rightarrow ((\neg P \cup Recovery) \wedge F(Recovery)))$

requirement for a component in the model, a *Fault* message originating from that component must be included, to indicate the presence of potential failures that require recovery operations.

Finally, the specification provides the temporal logic forms that the requirement can take according to the selections of *Card* and *Scope* variables. *Card* is used to state the precise relationship between other variables. In Table 4.5, it specifies whether the *Recovery* condition is needed for every failure of *Component*, or if one instance can recover from all failures preceding it. *Scope* is used to limit when the requirement is applicable, such as only during execution that satisfy or violate a certain property.

The properties for this template are formulated in LTL. Depending on how the *Card* and *Scope* variables are set, there are 8 basic structures that the property may take. The properties of this template, and several others in Appendix A, can become non-trivial to manually and interpret. Our template set avoids this capacity for human error and effort.

Specifying a `CompensateFailure` requirement using our template instead of manual temporal logic removes a lot of burden from the developer, while maintaining sufficient options and expressibility. For example, this template can be used to formalize the requirement that if `Check Sales` has a fault, then `Decline Sale` must be used to recover the process, by assigning those components to *Component* and *Recovery* respectively, setting *Card* to 1:1, and *Scope* to Global. The LTL property expressing this requirement takes the form:

$$G(CheckSales.FAULT \rightarrow ((\neg(Activated \wedge CheckSales) \cup DeclineSale) \wedge F(DeclineSale))$$

This property formulates a condition that must be satisfied after every instance of a *Fault* message originating from the `Check Sales` activity. Firstly, that `Check Sales` must not be executed again until the *Recovery* condition (as *Card* is set to 1:1), and finally that *Recovery* must be satisfied at some point in the future.

4.2.2 Process-Level Templates

Process-level templates are able to specify requirements over the entire process, such as preconditions, triggers, or reachability conditions for entering control behavior states. An example process-level requirement taken from the online payment composition is that in cases where a high fraud score is detected, *Create Report* must be executed before the *Aborted* state is reached. Our proposed set of process-level templates are listed below.

- **ControlStateCritical** specifies a condition that is *critical* to satisfy before entering a certain control behavior state. This template requires a condition, control behavior state, and scope set by the developer.
- **ControlStateTrigger** specifies a condition that once satisfied, must trigger the activation of a control behavior state at some point in the future.

Table 4.6: Template specification for `ControlStateCritical`

Name	`ControlStateCritical<ControlState,Condition,Scope>`	
Type	Process-level	
Variables	*ControlState*	The control behavior state this critical condition applies to.
	Condition	The precondition for entering this control behavior state. This can be a single component or a set structured with $\wedge$ and $\vee$ operators.
	Scope	One of the scope options below.
Description	*Condition* denotes the precondition for entering *ControlState*. When *ControlState* is entered, *Condition* must have been met previously on the execution path.	
Prerequisites	N/A	
Cardinality	N/A	
Scope	G	The template applies in all executions.
	P	Applies during the satisfaction of a condition P.
	$\neg P$	Applies during the negation of a condition P.
	Before P	*ControlState* is entered before P is met.
LTL	G	$G(\mathit{ControlState} \rightarrow O(\mathit{Condition}))$
	P	$F(P) \rightarrow G(\mathit{ControlState} \rightarrow O(\mathit{Condition}))$
	$\neg P$	$G(\neg P) \rightarrow G(\mathit{ControlState} \rightarrow O(\mathit{Condition}))$
	Before P	$G(\mathit{ControlState} \rightarrow (O(\mathit{Condition}) \wedge H(\neg P)))$

- **ControlStateReachable** specifies that a control behavior state must be reachable when a condition provided by the developer is satisfied.
- **ControlStateUnreachable** specifies a condition that once satisfied, indicates that a control state should not be reachable in the future.
- **Compensation** specifies a condition that must be met during compensation. Compensation is a service or sub-process to undo the effect of a committed process instance.
- **ConditionalCompensation** specifies a condition for compensation and a condition for execution, such that the compensation condition is only necessary when the execution condition has been satisfied earlier in the execution.

The full specification of the `ControlStateCritical` template is shown in Table 4.6. The variables include a control behavior state that the requirement applies to, and a vital pre-condition for entering that state. Like the `CompensateFailure` template in Table 4.5, the applicability of the requirement can be restricted using a scope variable. For example, the template could formalize the requirement that before entering the `Aborted` state, if a `High Fraud Score` was detected (assigned as P in *Scope*), then `Create Report` must have been previously executed. The LTL property to formalize this requirements takes the form:

$$F(\mathit{HighFraudScore}) \rightarrow G(\mathit{Aborted} \rightarrow O(\mathit{CreateReport}))$$

This property uses the F operator to only apply to executions of the model contains the `HighFraudScore` activity at any point. In these execution, every instance of the `Aborted` control behavior state must be preceded at some point by the `Create Report` activity.

4.3 Enabling Formal Verification Through Model Checking

The process for enabling verification against transactional requirements specified using our template set is similar to the one defined in Chapter 3.4 for conversation rules. We apply model checking to ensure that a service-oriented process design satisfies the temporal logic forms of implemented transactional requirement templates. In order to improve verification performance and feasibility for large and complex processes, we also use an algorithm to reduce the state space of the model prior to model checking.

4.3.1 Transactional Requirement Formalization

As shown in Tables 4.5 and 4.6, each template comes with temporal logic property structures that formalize the intended requirement. As an example, we define 3 transactional requirements applying to the online payment composition that can be implemented using our templates, as shown in Table 4.7. The client can set the template variables as shown, and they will be mapped to concrete temporal logic properties.

Table 4.8 shows the temporal logic forms of these 3 requirements after mapping the values set to the LTL structures in their specifications. Generating properties this complex automatically removes a lot of potential for human error in requirement specification.

Table 4.7: Transactional requirements for the online payment process

ID	Template	Variables	Values
CLR1	Alternative	Component	$Resale$
		Recovery	$PUTAccountData \vee PUTCardData$
		Cardinality	1:1
		Scope	G
CLR2	RetriablePivot	Component	$CheckSales$
		Scope	G
PLR1	ControlState Critical	Control State	$Aborted$
		Condition	$CreateReport$
		Scope	$HighFraudScore$

Table 4.8: Online payment process transactional requirements implemented using our template set

ID	Temporal Logic
CLR1	$G((Resale.SYNCREQ) \rightarrow ((\neg(Activated \wedge Resale) \cup (Activated \wedge PUTAccountData) \vee (Activated \wedge PUTCardData))) \wedge F((Activated \wedge PUTAccountData) \vee (Activated \wedge PUTCardData))))$
CLR2	$G((CheckSales \wedge X(\neg DeclineSale.FAIL)) \rightarrow F(Done))$
PLR1	$F(HighFraudScore) \rightarrow G((Aborted) \rightarrow O(CreateReport))$

4.3.2 State Space Reduction

For reducing the model state space prior to verification against transactional requirements, we propose a second algorithm called *Transactional Requirement Kripke Structure Reduction* (*TSKR*), shown in Algorithm 3, to produce a Kripke structure K^t. Since the model is verified against a set of application-dependent transactional requirements, the Kripke structure must capture the states related to those requirements. Therefore, a different algorithm than Algorithm 1 for conversation rule checking is required.

The temporal logic forms of specified requirements can include operational behavior states, control behavior states, and inter-behavior messages. Therefore, the atomic propositions at each state of the Kripke structure K^t will be the active value of each of these three features. The input to this algorithm is the flattened behavior model B^f, the control behavior model B^c, and a set of all operational and control behavior states defined as template variables TV. *TKSR* uses a recursive procedure *Transactional Requirements Depth-First Traversal* (*TDF*), shown in Algorithm 4, to traverse B^f in a depth-first order.

First, *TKSR* creates the K^t with initial and final states, before invoking the *TDF* procedure

Algorithm 3 Transactional Requirements Kripke Structure Reduction: *TKSR*

```
1: $K^t \leftarrow \langle \mathcal{S}^{kt}, \mathcal{T}^{kt}, \mathcal{L}^t \rangle$
2: Add $s_0$ and $s_{fin}$ to $S^{kt}$
3: *TDF*($NotActivated$, $s_0$, $s_0$, $\emptyset$)
4: **return** $K^t$
```

Algorithm 4 Transactional Requirements Depth-First Traversal: *TDF*(s_c, s_x, s_k, AP_v)

```
1: **if** $s_x = s_{fin}$ **then**
2:     Add ($s_k$, $s_{fin}$) to $T^{kt}$
3:     **return**
4: **end if**
5: **if** $s_x \neq s_0$ **then**
6:     **if** $s_x \in S_m$ **then**
7:         Update $s_c$
8:     **end if**
9:     **if** ($s_c$,$s_x$)$\in AP_v$ **then**
10:        **return**
11:    **end if**
12:    **if** $s_c \in TV \vee s_x \in TV$ **then**
13:        **if** $s \in S^{kt}$ with atomic propositions ($s_c$,$s_x$) **then**
14:            Add ($s_k$, $s$) to $T^{kt}$
15:            **return**
16:        **else**
17:            Add $s$ with atomic propositions ($s_c$,$s_x$) to $S^{kt}$
18:            Add ($s_k$, $s$) to $T^{kt}$
19:            $AP_v \leftarrow \emptyset$
20:            $s_k \leftarrow s$
21:        **end if**
22:    **end if**
23:    Add ($s_c$,$s_x$) to $AP_v$
24: **end if**
25: **for** $s \in S^f$ where ($s_x$, $l$, $s$) $\in T^f$ **do**
26:    **if** $l$ is satisfied **then**
27:        *TDF*($s_c$, $s$, $s_k$, $AP_v$)
28:    **end if**
29: **end for**
30: **return**
```

to traverse that states of B^f. *TDF* requires as input the current control behavior states s_c, active flattened behavior model states s_x, and the most recent Kripke state added at this point in the traversal s_k. The fourth input parameter is the set of visited control and flattened behavior model states since s_k was added, represented as a set of atomic propositions AP_v, such that $AP_v \subseteq S^c \times S^f$.

Firstly, if the traversal has reached the final state, *TDF* creates a Kripke transition to the final state and returns (lines `1-4`). Next, *TDF* determines whether states or transitions need to be added to K^t at this point in the traversal (lines `5-24`). The steps in this process include transitioning the control behavior state if a message state is encountered (lines `6-8`), and backtracking if a cycle has occurred in the traversal (lines `9-11`). If either of the current atomic propositions (s_c, s_x) are in the template variable set TV (line `12`), a Kripke transition is created to either an existing state (lines `13-15`) or new state (lines `16-21`) in S^{kt}. If an existing state is linked to the procedure can return, as the outgoing transitions in B^f from that state have already been explored (line `15`). Following any updates to the Kripke structure, the visited atomic propositions set AP_v is also updated with (s_c, s_x) (line `23`). Finally, *TDF* recursively calls itself on every state targeted by satisfied transitions outgoing from s_x (lines `25-29`). The procedure returns once the initial invocation with s_0 terminates.

The control and operational behavior states specified by the user as template variables are input for the template Kripke reduction algorithm, as specified in Algorithm 3. We implement two requirements with component-level templates (`CLR1` and `CLR2`), and one requirement with a process-level template (`PLR1`), as shown in Table 4.7. Given the input from these requirements, the control and flattened behavior model is reduced to the Kripke structure of Figure 4.1. This Kripke structure contains only the states required for verifying the transactional requirement set, and their temporal dependencies. The algorithm has reduced the initial behavior models totalling 28 states with 14 inter-behavior messages to a Kripke structure of 12 states.

This Kripke structure and the temporal logic forms of implemented templates can be used as input for a model checking tool. However, the temporal logic properties have to apply to the states of the Kripke structure they are verifying. In place of control states, operational states, or inter-behavior messages in the temporal logic properties, all Kripke states that contain that atomic

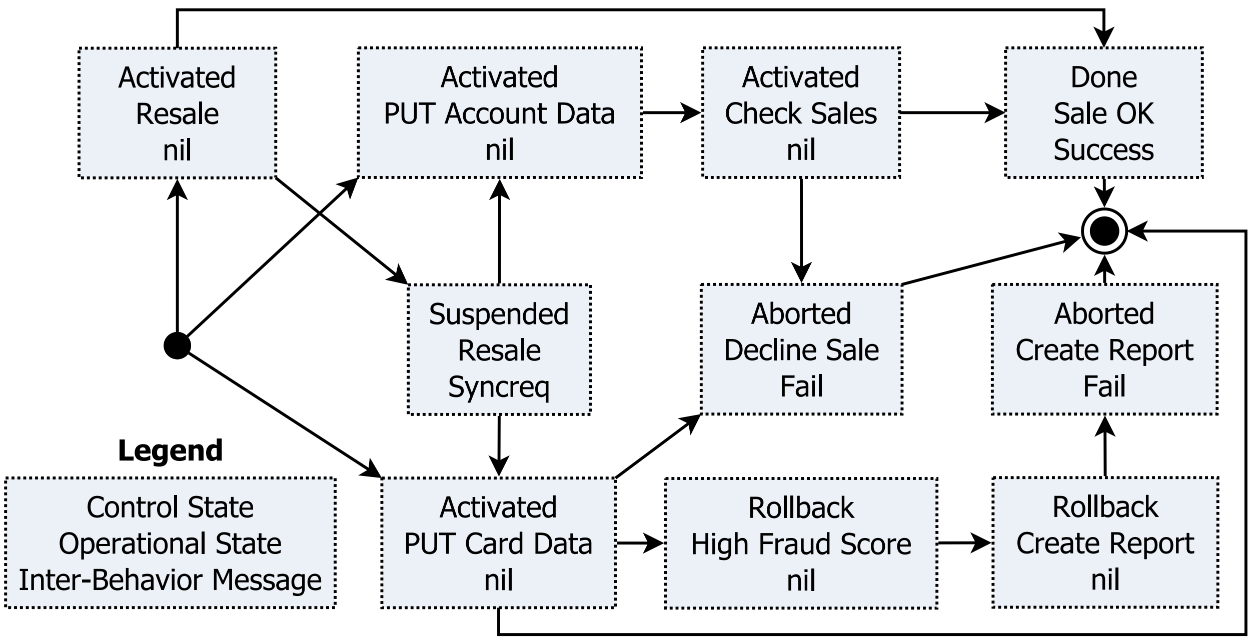


Figure 4.1: Reduced Kripke structure for transactional requirement verification

Table 4.9: Temporal logic formalizations of the online payment process transactional requirements

ID	Temporal Logic
`CLR1`	$G((Suspended\_Resale\_Syncreq) \rightarrow ((\neg(Activated\_Resale) \cup (Activated\_PUTAccountData \vee Activated\_PUTCardData)) \wedge F(Activated\_PUTAccountData \vee Activated\_PUTCardData)))$
`CLR2`	$G((Activated\_CheckSales \wedge X(\neg Aborted\_DeclineSale\_Fail)) \rightarrow F(Done\_SaleOK))$
`PLR1`	$F(Rollback\_HighFraudScore) \rightarrow G((Aborted\_DeclineSale \vee Aborted\_CreateReport) \rightarrow O(Rollback\_CreateReport))$

proposition must be included. If more than one such state exists in the Kripke structure, they are separated with $\vee$ operators.

Table 4.9 contains the temporal logic formalizations of the requirements specified in Table 4.7, for verifying the Kripke structure in Figure 4.1. Chapter 6 provides more details on how our prototype tool uses model checking to verify the Kripke structure and against these properties.

4.4 Summary

Formalizing transactional requirements for service-oriented processes at design-time allows developers to verify that their model behaves in an acceptable manner in the presence of faults. It provides assurance to the developers that the transactional behavior of the process avoids undesirable consequences, such as business rule violations or mismanagement of funds and resources.

From conducting an analytical survey of research in ensuring application-dependent transactional requirements in service-oriented processes, we identify several issues to address. Firstly, the majority of approaches conduct automatically defined transactional behavior, and do not attempt to ensure requirements provided by the developer until later stages in the process lifecycle. Furthermore, the specification methods for transactional requirements used in these approaches often unable to express diverse and complex requirements, require expertise in formal methods, or become laborious to use for large and complex processes.

In our approach, we adopt temporal logic patterns towards this issue to develop a design-time verification method for service-oriented processes against application-dependent transactional requirements. We provide a set of templates for specifying component-level and process-level transactional requirements for service-oriented processes. These templates can be implemented by developers by assigning variables, which are then automatically mapped into LTL or CTL properties. This allows developers to formalize transactional requirements drawn from business logic in a way that does not require expertise in formal methods, while reducing human error and effort.

Model checking is used to verify service-oriented processes modeled as control and operational behaviors against a set of requirements implemented using our template set. We use an algorithm to reduce the impact of state-space explosion during model checking, by producing a minimal Kripke structure from the control and operational behaviors. By enabling this verification at design-time, we allow developers to ensure their requirements early in the process lifecycle and avoid costly redevelopment or requirement violations at later stages.

CHAPTER 5

Transactional Behavior Verification in Business Process as a Service Configuration

In this chapter, we extend our approach for verifying application-dependent transactional requirements towards Business Process as a Service (BPaaS) configuration. BPaaS is an emerging type of cloud service that offers configurable and executable business processes to clients over the Internet. Our design-time verification approach detailed in Chapters 3 and 4 can be adapted towards the *configuration* of BPaaS, to ensure that provisioned service satisfies the transactional requirements of clients.

As BPaaS is still in early years of research, many open issues remain, as shown by our comparison of related work. Managing the configuration of BPaaS builds on such areas as software product lines and configurable business processes. The problem has concerns to consider from several perspectives, such as the different types of variable features, constraints between configuration options, and satisfying the requirements provided by the client. In our approach, we use our temporal logic templates set to elicit transactional requirements from clients that the configured service must adhere to. As a configurable BPaaS model, we extend the separated behaviors presented in previous chapters by defining operational behavior with BPMN. This allows for control flow connectors that not supported by statecharts, and enables configurable resources and data objects to be mapped to activities. For formalizing constraints over configuration, feature models are used. These allow for many types of relations between configuration choices to be expressed, in a way that allows relations to cross perspectives. For example, the selection of an activity may require a certain resource.

To manage all these concerns during BPaaS configuration, we develop a structured process that applies several formal methods towards identifying a configuration solution. This process directs the client through specifying transactional requirements and selecting configurable features. Firstly, our temporal logic templates capture the transactional requirements the client wishes for the configured BPaaS to conform to. Then, *Binary Decision Diagram* (BDD) analysis is used to verify that the selected configurable features do not violate any constraints. Finally, model checking is applied to verify the configured service against the transactional requirement set. We expand on the algorithm proposed in the previous section and further reduce the model state space by dividing the model checking problem into two phases.

5.1 Related Work

Our work in verifying and managing configurable BPaaS follows increasing research interest in related areas such as configurable cloud service applications [110, 129], and configurable or adaptive business process models [71, 94, 146]. Our survey compares 23 approaches from these areas according to criteria including their support for business process characteristics, configuration constraints, correctness criteria, and client requirements they ensure. From this analysis, we determine that configuration of expressive BPaaS models, including resource and data object configurability, that ensures configuration constraints and complex client transactional requirements is an open issue for our work to address.

5.1.1 Comparison Criteria

We compare works according to two sets of criteria, namely, *Support* and *Correctness criteria*. Table 5.1 shows our criteria set with example values. The support criteria is related to the business process expressiveness enabled by the approach. The *Process Formalism* criterion identifies how the approach expresses business processes. If business process structures are not explicitly incorporated in the approach, the value is left blank. The next criterions specify whether it is capable of modeling resource and data and configurability. Finally, the means of expressing domain constraints are identified for comparison.

Table 5.1: Comparison criteria overview

Category	Criteria	Values
Support	Process Formalism	Petri-nets, BPMN, statecharts
	Resources	√ or -
	Data	√ or -
	Domain Constraints	Feature model, OVM, hierarchy model
Correctness Criteria	Process Model	Syntax, Soundness
	Configurability	Circular dependency-free, contradiction-free
	Client Requirements	Feature selections, business rules, QoS parameters

Correctness criteria identifies the requirements that are ensured during the configuration or adaptation approach. Process model criteria refers to structural or behavioral correctness of the process model, such as soundness [148] or syntactical correctness [109]. Some approaches may also analyze the configurability of the model to identify issues such as contradictions, dead features, or circular dependencies [71, 95]. The final criterion identifies the client requirements that are input into the process, such as selections of features [92, 102], or more complex behavioral requirements [91].

5.1.2 Survey

Tables 5.2 and 5.3 show the results of applying our comparison criteria to the work related to BPaaS configuration. Below, we provide a brief summary of these approaches, before analyzing the comparison results for identifying a research direction.

5.1.2.1 Configurable Cloud Services

Configurability has been identified as a key property to increase the potential market of cloud services, by allowing clients to adjust the service behavior or QoS guarantee to meet their requirements [9, 49, 129]. Configuring SaaS shares many challenges with BPaaS, such as enforcing domain constraints, and has received increasing research attention in recent years [92, 102, 110].

Nitu [114] analyzes the different aspects of SaaS that may be configurable, such as user interface, program structure, data, and access control, and proposes an enabling architecture. One module of the architecture contains a set of *template data*, relating to the various features that can

Table 5.2: *Support* criteria for comparing work related to BPaaS configuration

Approach	**Process Formalism**	**Resources**	**Data**	**Domain Constraints**
Nitu [114]	-	√	√	Configuration templates
Mietzner et al. [110]	-	√	√	OVM
Lizhen et al. [102]	-	√	√	Metagraph relationships
Schoreter et al. [128]	-	√	√	QCL Contracts
Banerjee [9]	-	√	√	Variation pre and post-conditions
Kumara et al. [92]	-	√	√	Feature model
Schroeter et al. [129]	-	√	√	Extended feature model
Mendling et al. [109]	C-EPC	-	-	XPath Statements
van der Aalst et al. [144]	Workflow nets	-	-	-
van der Aalst et al. [145]	Workflow nets, C-EPC	-	-	-
Kumar and Yao [91]	WS-BPEL	√	√	If-then statements
La Rosa et al. [93, 94]	C-iEPC	√	√	Hierarchy model
La Rosa et al. [95]	-	-	-	Configuration Model
Wang et al. [152]	WS-BPEL	√	√	-
van der Aalst [142]	C-nets	-	-	C-net bindings
Tsai and Sun [141]	Customizable workflow	√	√	OVM
Liu et al. [101]	Petri-nets	-	-	-
van Dongen et al. [148]	EPC, Petri-nets	-	-	-
Gröner et al. [71]	BPMN	√	√	Feature Model
van der Aalst et al. [146]	Petri-nets	-	-	-
Jiang et al. [79]	Dependency structures	-	-	-
Hallerbach et al. [73]	Workflow model	-	-	Basic logic
Gottschalk et al. [68]	LTS, C-EPC	-	-	-

be integrated into the service, such as UI or data elements. Configuration is handled by developers and pre-defined for each type of client using these templates and a module holding *configuration data*. While this framework allows service providers to maintain control over the configuration of their product, other work in this area has aims to enable the clients themselves to drive configuration.

Mietzner et al. [110] apply techniques from *Software Product Lines* (SPL) [124] to support configurable SaaS. The authors adapt the *Orthogonal Variability Model* (OVM) [19] for modeling SaaS configurability. An OVM captures mandatory and optional variation points, as well as the concrete choices for each point, with dependencies between. OVMs can capture any type of SaaS feature as abstract variation points, such as workflows, GUI elements, or security allowances. The variability information can be used to generate a *customization flow*, which is a workflow procedure to guide clients through configuration choices.

Lizhen et al. [102] adapt metagraphs [11] for modeling and support of configurable SaaS

Table 5.3: *Correctness Criteria* for comparing work related to BPaaS configuration

Approach	**Process Model**	**Configurability**	**Client Requirements**
Nitu [114]	-	-	-
Mietzner et al. [110]	-	-	Feature selections
Lizhen et al. [102]	-	-	Feature selections
Schoreter et al. [128]	-	-	Feature selections
Banerjee [9]	-	-	Required services
Kumara et al. [92]	-	-	Feature selections
Schroeter et al. [129]	-	-	Feature selections
Mendling et al. [109]	Syntax	-	Function and connector variants
van der Aalst et al. [144]	Syntax and Soundness	-	Blocked and hidden activities
van der Aalst et al. [145]	Syntax and Soundness	-	Blocked and hidden activities
Kumar and Yao [91]	Syntax	-	Business rules
La Rosa et al. [93, 94]	Syntax	-	Blocked and optional functions, resources, data objects, connectors
La Rosa et al. [95]	-	No circular dependencies or contradictions	Boolean questionnaire answers
Wang et al. [152]	-	-	Set of business policy types
van der Aalst [142]	-	-	Blocked activities
Tsai and Sun [141]	-	-	Structure, GUI, resources, data, QoS
Liu et al. [101]	Fairness, deadlock-free, reachability	-	Adding, deleting, modifying elements
van Dongen et al. [148]	Relaxed soundness	-	Termination states
Gröner et al. [71]	-	Control flow and domain constraint conflicts	Feature selections
van der Aalst et al. [146]	Weak-termination	-	Blocked and hidden activities
Jiang et al. [79]	Deadlock-free	-	Blocked activities
Hallerbach et al. [73]	Soundness	-	Security, maintenance, and workload context variables
Gottschalk et al. [68]	-	-	Blocked, hidden, and optional activities

applications. Metagraphs model variants of GUI, data schemas and business process elements as abstract *customization points*. At each configuration decision, an algorithm is applied to check consistency with dependencies modeled in the metagraph.

Schoreter et al. [128] propose a runtime architecture for configurable SaaS that negotiates the configuration concerns of providers, clients, and users. The authors propose to enable this by extending *MQuAT* [69], an architecture for dynamically adaptive systems in order to meet tenant-based requirements and shared resource support. This framework would be able to configure cloud services dynamically according to all stakeholder concerns.

Several approaches have addressed the problem of *evolving* multi-tenant SaaS at runtime to handle either new clients or changed requirements, while still supporting the existing clients. Banerjee [9] proposes a formal SaaS framework to manage configuration and customization of services to handle new clients. The SaaS contains a set of *base* and *auxiliary* features, which are applicable to *all* and *some* clients respectively. Additionally, a set of invariants must be preserved over all configurations of the SaaS. Min-max multi-objective optimization can be applied to determine how to support new clients with minimal changes and cost to the provider. The SPL-based framework proposed by Kumara et al. [92] categorizes the different types of changes that can be necessary to a SaaS, and the general impacts each potentially create. The changes and their propagated impacts are realized by code updates by developers. Schroeter et al. [129] propose a dynamic SaaS configuration method that responds to stakeholders being added or removed from the service. SPL techniques are applied to model service configurability, and the roles and access control of stakeholders. A formal configuration process is proposed that utilizes a CSP solver to ensure consistency with configuration constraints.

5.1.2.2 Configurable Business Processes

While configurable BPaaS is still a new area of research, work in related domains such as configurable business processes [91, 94, 146] and reference models [71, 145] has been ongoing for several years. Below, we provide an overview of how common configuration and verification problems in these domains have been addressed. Most approaches are enabled during the configuration of business processes, while others verify configurable business processes for properties, or obtain a set of all valid configurations for potential clients to provision from.

Mendling et al. [109] propose an algorithm for generating semantically correct business process configurations from *Configurable Event-driven Process Chain* (C-EPC) models. Clients are able to remove configurable functions and connectors, or set them to optional to defer the decision to runtime, as long as they are in line with attached XPath constraints. The configuration algorithm manages all client decisions at once rather than incrementally. Configurable connectors and functions are managed in individual steps, with reduction rules to exclude unnecessary paths.

An alternate configuration method proposed by Van der Aalst et al. [144, 145] preserves soundness in business processes. Domain constraints are captured in propositional logic and reduced by a *Binary Decision Diagram* (BDD) solver to efficiently handle large and complex systems. As the client makes configuration choices, the framework incrementally recalculates and discards incorrect options.

Kumar and Yao [91] manage configuration and customization of business process variants given a rule set provided by the client and an algorithm for applying mutations. Client rules can be defined using an English-like syntax, and the algorithm ensures mutations to the business process activity structure, resource allocation, or related data objects, are applied in a valid sequence. Business process variants are generated as WS-BPEL schemas.

La Rosa et al. [93, 94] extend C-EPCs to include resource and data object configurability, as *C-iEPC*. The relationships between options for resources and data objects are represented as hierarchy models, which show the alternate specializations of resources to fill a general role. The authors propose an algorithm to generate syntactically correct configured models by focusing on different types of business process elements individually.

For managing adaptive distributed business processes at runtime, Wang et al. [152] propose an aspect-oriented programming approach. Three kinds of policies can be specified for runtime governance, such as safety constraints and how the process should react if they are violated. Runtime governance is enabled by dynamically weaving aspects into WS-BPEL execution according to these policies.

Van der Aalst [142] develops a novel formalism called *Casual Nets* (C-nets) to support and analyze configurable business processes. Each activity in a C-net has a series of potential input and output bindings to act as domain constraints. Configurations are considered valid when all predecessor and successor activities agree on bindings, and no pending obligations remain.

Tsai and Sun [141] propose an approach to manage the configuration, customization, and restructuring of workflow-structured SaaS in a semi-automated way. Client requirements in this approach can apply to features such as workflow structure, GUI, services used, data, and QoS. An OVM is used to express variation points and constraints in the workflow. During configuration, tenant mining is used on the choices made by existing tenants to guide clients towards a valid configuration.

Liu et al. [101] address the evolution of cloud services that have a business process structure. The service is modeled as a reflective Petri-net [31], enabling simulation and verification of changes such as adding, modifying, or deleting functions, business rules, or the process structure. Guard checking is applied to new evolutions of the service before reification to check reachability of activities, fairness, and deadlock-freedom.

Several approaches verify configurable business process models to ensure certain properties are met. Van Dongen et al. [148] verify reference models according to relaxed soundness and client-specified acceptable termination states. Process models adapted from reference models are expressed as an *Event-driven Process Chain* (EPC), to which reduction measures can be applied to the state space for analysis and verification. By translating the reduced EPC into a Petri-net, and receiving client input on the possible initial events, the various termination states of the process can be deduced. A coloring algorithm verifies the process model with respect to valid termination states specified by the client. Gröner et al. [71] use *Business Process Model and Notation* (BPMN) to model the control flow constraints of reference models, and *feature models* [82] to specify the configuration constraints. Mappings between these models connect configurable features to their related business process activities. Validation is applied to ensure that adapted business processes respect the constraints of the original reference model, by identifying strong and potential inconsistencies between control flow and configuration constraints.

Another general approach is to obtain all valid configurations from a configurable business process, through exhaustive verification or other means. Van der Aalst et al. [146] apply *partner synthesis* [157] to obtain all correct configurations of a process at design-time. This can then be used as a configuration guideline to check future configurations for correctness. Instead of soundness, the relaxed *weak-termination* criteria is used for configurations, meaning that processes may contain dead parts if the client wishes. Similarly, Jiang et al. [79] incorporates configuration with verification at design-time to obtain all deadlock-free process models. Using *dependencies structures* (formerly *protocol structures* [80]) as business process models, all valid configurations can be obtained by first separating the structure into a set of deadlock-free atomic models, and applying an algorithm to determine all deadlock-free merges. Hallerbach et al. [73] take the approach of verifying all possible configurations for correctness given basic context information provided by the client. Context is specified according to discrete variables for security level, workload, and maintenance, and is used in conjunction with domain constraints to limit the configuration space. This reduced set of configurations is then verified to ensure process soundness.

Other approaches use generic models to accelerate or ease business process integration. For example, Gottschalk et al. [68] propose an approach whereby generic business process solutions are kept in a repository to be configured and integrated in client systems as needed. Analysis and configuration of these reference models is discussed using both *Labelled Transition Systems* (LTS) and C-EPC. In the cloud service domain, Petcu and Stankovski [121] envision a BPaaS marketplace where services are deployed and enabled using business process patterns. Services are published with patterns that provide clients with information on the behavior of the service, and rules for integrating the service with larger systems.

5.1.3 Research Direction

Table 5.2 shows that among the 23 approaches we identified, only 65% apply themselves to business process structures. Furthermore, only one third of those approaches support resource and data configuration. As indicated by the configurable cloud service approaches in our survey, resource and data configurability is an important feature for allowing clients to tailor services towards their requirements. Allowing clients to configure resources can have an impact on the running costs of the service, in addition to increasing the potential market with greater configurability. Data object configuration allows clients to make the service more similar to existing or expected business practises. Therefore, we consider it important for research into BPaaS configuration to consider configuration from these perspectives.

Also, as the *Client Requirements* field in Table 5.3 indicates, most approaches only support simple requirements such as selecting and removing features. Feature selections are all the client provides in 26% percent of our surveyed approaches, while other basic binary selections, such as blocking or hiding activities, make up a further 39%. Other approaches provide some business rule support [91, 152], such as basic if-then conditions. To the best of our knowledge, transactional requirements important to clients, such as those supported by our template set, are not yet supported by any business process configuration method.

Another issue identified by our survey is that very little research so far has addressed BPaaS explicitly. Approaches such as Liu et al. [101] target their configuration method towards SaaS that has a business process structure of activities, while some configurable business process methods consider variability of resources and data associated with activities. However, BPaaS has their own unique concerns such as configurable use of third-party services, and the transactional concerns inherent in such service-oriented processes.

In our approach, we aim to manage BPaaS configuration in a way that addresses the issues identified above. Firstly, our BPaaS model will enable configuration from numerous perspectives important to BPaaS clients, namely, activities, resources, and data objects. Our configuration method will aim to elicit and ensure complex transactional requirements from clients, by adapting the temporal logic template set from our existing work.

5.2 Transactional Business Process as a Service Modeling

In this section, we present our methods for expressing BPaaS configurability, domain constraints, and client transactional requirements. The formalisms proposed in this section aim to enable clients to configure a BPaaS, while also providing a transactional perspective that allows for verification against requirements. The goal of these formalisms is to enable BPaaS configuration, and verification against domain constraints and client transactional requirements.

Our BPaaS model aims to capture configurability of the various business process perspectives managed by the provider. These include activities and control flow, resources, and data objects. We define a configurable business process as containing constant and optional activities, resources, and data objects. To reduce verbosity, we model constant and optional activities, but only model optional resources and data objects. We justify our modeling formalism below with a motivating scenario.

5.2.1 Motivating Example

We consider the scenario of a configurable Web store checkout BPaaS. The clients of this process will be small and medium sized Web stores, while the users will be customers. This BPaaS targets businesses selling physical or digital goods, as standard orders or pre-orders. The process places shipping orders for physical goods and retrieves download links for digital goods. Specific tasks included in the process include validating customers, obtaining payment details, updating inventory and accounting systems, and processing customer payment amongst others.

Clients can configure the structure of this process to suit their business requirements. For example, stores only selling digital goods can remove all tasks related to product shipping, while stores who do not store customer details can restrict the process to handling unregistered guest customers.

This process provisions both constant and configurable external resources. Examples of configurable resources include the optional payment services used by the BPaaS, such as Paypal[1], or Epoch[2] and eWay[3] credit card transactional managers. Similarly, the data objects used by the process can be configured as appropriate, such as including or excluding fields in customer and product details.

[1]https://developer.paypal.com/
[2]https://www.epoch.com/en/index.html
[3]http://www.eway.com.au/developers/api/overview

5.2.2 BPaaS Model

We use BPMN for modeling activities and control flow, as it formalizes the BPaaS structure in a format that is easily readable for clients. Furthermore, BPMN is a widely supported and used notation for formalizing and executing business processes, which increases the potential client and provider base of our approach. Configuring BPMN is also a less complex activity than alternatives such as C-EPC, as the correctness criteria of alternating events and functions do not need to be maintained [94].

Figure 5.1 shows the BPMN model of the configurable Web store checkout BPaaS. Configurable activities are shown in a lighter shade. For readability, tasks related to preparing orders, delivering products, and validating customers have been organized into sub-processes.

Numerous activities in the checkout BPaaS utilize configurable resources and data objects, which are shown in Tables 5.4 and 5.5 respectively. For example, the `Process Payment` activity can be implemented using Paypal, Epoch, or eWay transactional managers, according to the client's requirements. Figure 5.2 shows the four configurable resources and the data object mapped to the `Place Shipping Order` activity. These show the difference postage options for implementing shipping orders, as well as the necessary data object for Shipping Address, which is not applicable to all configurations. Other optional resources include Microguru[4] for inventory management, SaaSu[5] for accounting, and FTP or FTPS for digital product file transfer. Cloud storage is offered by the provider for a customer repository and digital product hosting. Data object configurability includes physical and digital product details, enabling product quantities, and payment and shipping details.

While BPMN provides lifecycle statecharts that represent the transactional state of individual activities, a view of the transactional state of the entire process is necessary for verification against process-level transactional requirements. Such requirements include activities *critical* for successful execution, *necessary* activities to execute prior to aborting, or *requirements* for valid process compensation. To provide this view, we adapt the separated behavior model detailed in Chapters 3 and 4, and use the control behavior model to represent the global transactional state of the process.

[4]http://www.microguru.com/

[5]http://www.saasu.com/

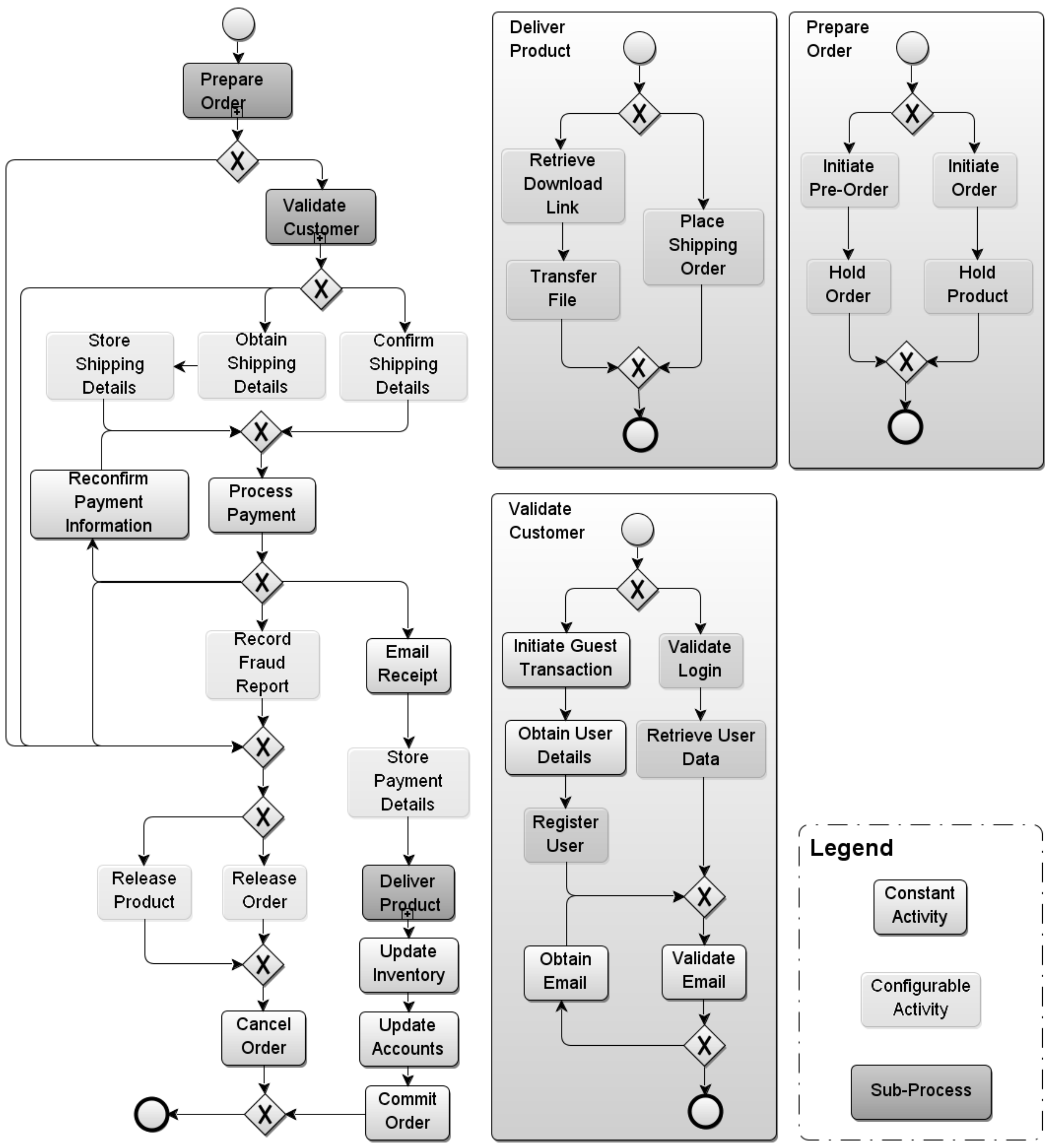


Figure 5.1: BPMN model of the configurable checkout BPaaS

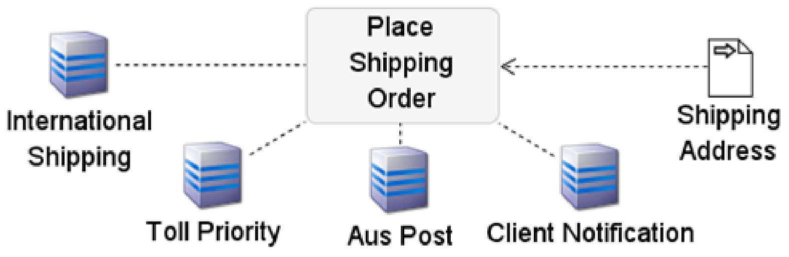


Figure 5.2: Configurable resources and data objects mapped to a BPMN activity

Table 5.4: Configurable resources for the checkout BPaaS

Resources	**Associated Activities**
Private Inventory System, Microguru	Update Inventory, Hold Order, Hold Product, Release Order, Release Product
International Shipping, Toll Priority, AusPost, Client Notification	Place Shipping Order
Provider Storage, Private Customer Repository	Store Payment Details, Validate Login, Confirm Shipping Details, Register User, Store Shipping Details
Private Accounting System, SaaSu	Update Accounts
External Cloud Storage, Provider Storage, External Server	Retrieve Download Link
FTP, FTPS	Transfer File

Figure 5.3 shows an example of how the control behavior model can direct and communicate with the checkout BPMN using the inter-behavior messages defined in Section 3.2.3. In this conversation session, the process becomes suspended after processing payment fails, but the process is able to commit successfully after the user is asked to reconfirm their payment information.

Table 5.5: Configurable data objects for the checkout BPaaS

Data Objects	**Associated Activities**
Physical Product Details, Digital Product Details, Quantity	Initiate Order, Initiate Pre-Order
Shipping Address	Store Shipping Details, Place Shipping Order, Obtain User Details, Retrieve User Data, Obtain Shipping Details
Payment Details	Process Payment, Retrieve User Data, Obtain User Details, Reconfirm Payment Information

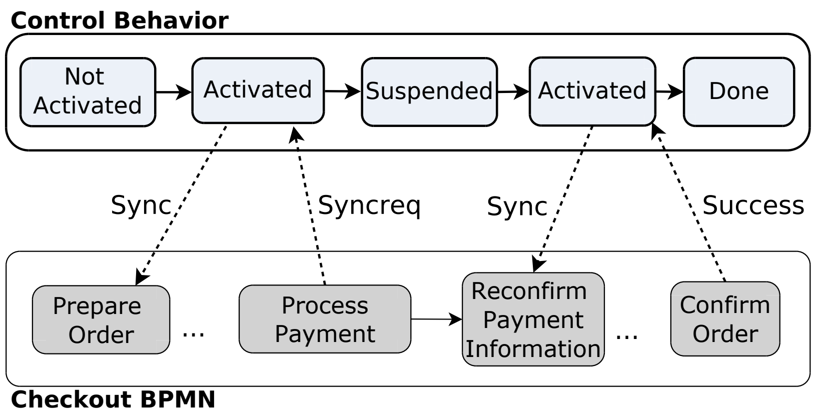


Figure 5.3: Inter-behavior messages used to enable communication between the checkout BPMN and the control behavior model

5.3 Configuration Domain Constraints

Domain constraints allow providers to restrict BPaaS configuration to valid choices, such as ensuring at least one payment resource is selected. We adapt feature models from software product line engineering [82] to express domain constraints formally and visually. Feature models are typically used to express variability in a configurable system, by modeling the constraints between optional *features*. In our approach, they define constraints between the selection of configurable activities, resources, and data objects. By using one feature model, we are able to define constraints that cross these configuration perspectives.

We apply six feature model relationship structures, shown in Figure 5.4, to model domain constraints. The first four relationships applies to one or more leaf features if the head feature is selected. For example, the *Mandatory* and *Optional* structures define that if feature `A` is selected, then feature `B` is essential or optional respectively. *Implication* and *Exclusion* can be defined between any two features in the model, regardless of their level in the tree structure.

A feature model capturing the domain constraints for the configuration of the checkout BPaaS is shown in Figure 5.5. The root `Checkout Service` feature contains all other features as children, and allows constraints to cross between activity, resource, and data object perspectives. For example, `Validate Login` is an optional feature, but it requires `Register User`, `Retrieve User Details`, either `Private Customer Repository` or `Provider Storage`, and enables `Store Payment Details` to be selected. A selection of features

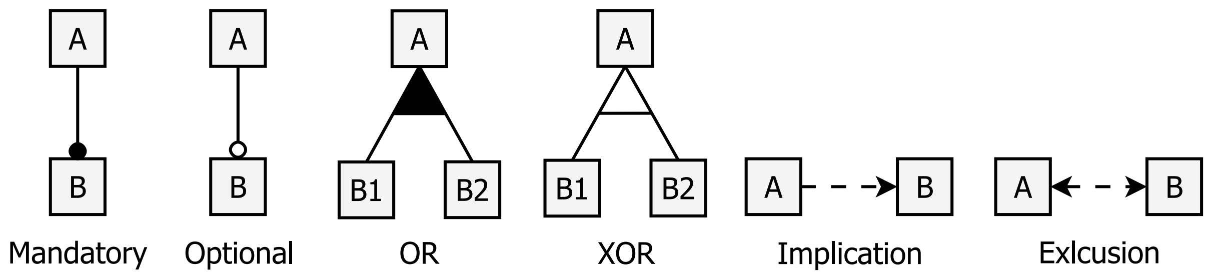


Figure 5.4: Feature model constraints used in our approach

that satisfy all constraints in this model, therefore conforming to all configuration requirements of the provider, is a *valid configuration* of the BPaaS.

Feature models have been used extensively in SPL research and other similar fields since the early 1990s [13]. By adapting them for formalizing domain constraints, we also enable our approach to be extended with several types of feature model reasoning, such as detecting satisfiability, unreachable features, contradictions, false optionals, and other properties.

5.4 Business Process as a Service Configuration and Verification

In this section, we propose a BPaaS configuration process that applies formal methods to ensure that i) the configuration is valid with respect to provider domain constraints, and ii) the process satisfies transactional requirements drawn from the business rules of the client. First, we provide an overview of the process which guides clients through BPaaS configuration, then we provide details on how BDD analysis and model checking is used at certain steps.

5.4.1 BPaaS Configuration Process

The aim of our BPaaS configuration process is to produce a *configuration solution* that satisfies transactional requirements provided by the client while respecting domain constraints. The provider inputs of this process include the BPMN model and the feature model. The client inputs are their transactional requirements formalized using templates, and additional features that they require. Our configuration process applies formal methods and simple client interaction to identify a configuration that i) is valid with respect to domain constraints, and ii) conforms to the client

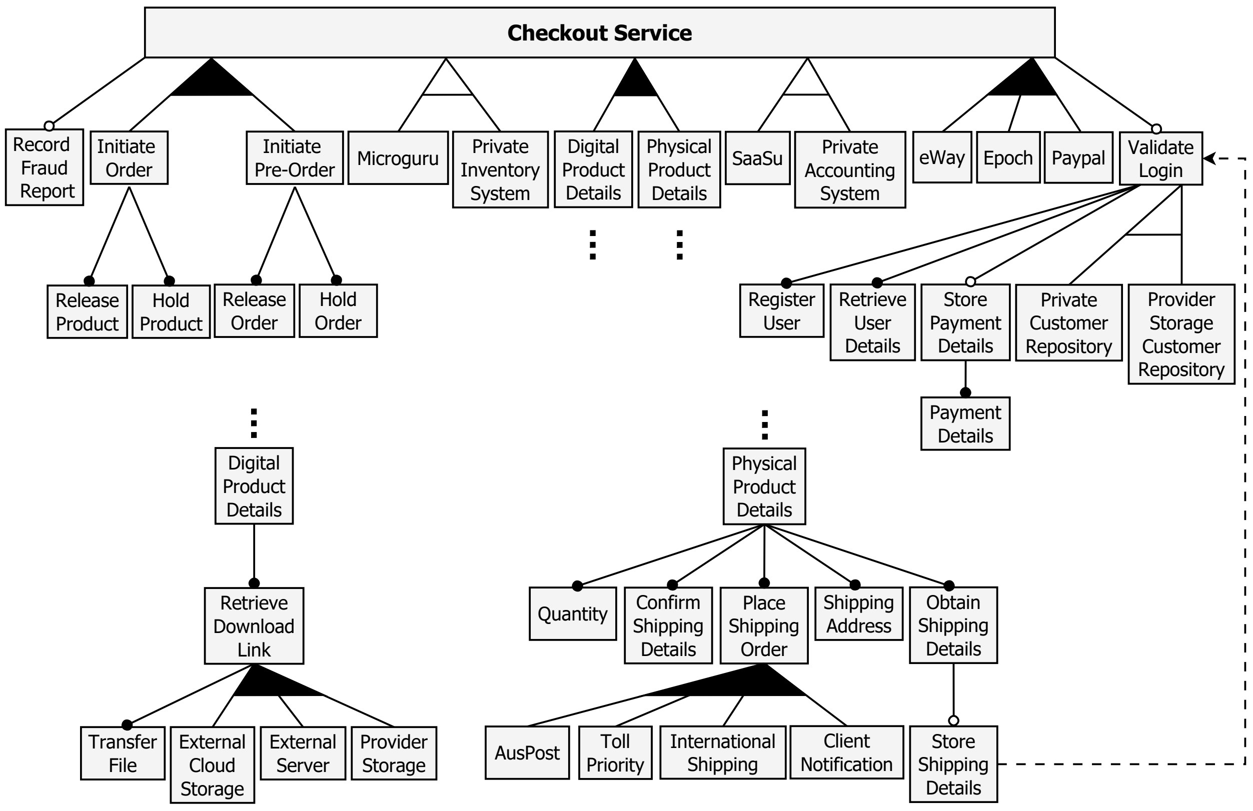


Figure 5.5: A feature model representing configuration constraints of the checkout BPaaS

transactional requirements set. This increases client trust that the service is going to behave in a manner consistent with internal business policies and requirements, without having to perform their own analysis of the service behavior.

The structure of our configuration and verification process is shown in Figure 5.6. The first step is for the client to formalize transactional requirements into temporal logic using our template set. Then, *Binary Decision Diagram* (BDD) analysis is used to ensure that all configurable features included in the client transactional requirement set can be selected in a valid configuration. If this is successful, the client may select other configurable features that they require for the service, which can be verified using BDD analysis again. If the domain constraints are found to be violated at any stage, the client must revised the selected features.

The model checking phases are used in order to ensure the transactional requirements of the client are satisfied. We divide the task into two phases in order to reduce the state space of the model and complexity of the temporal logic properties, in order to make our approach more time

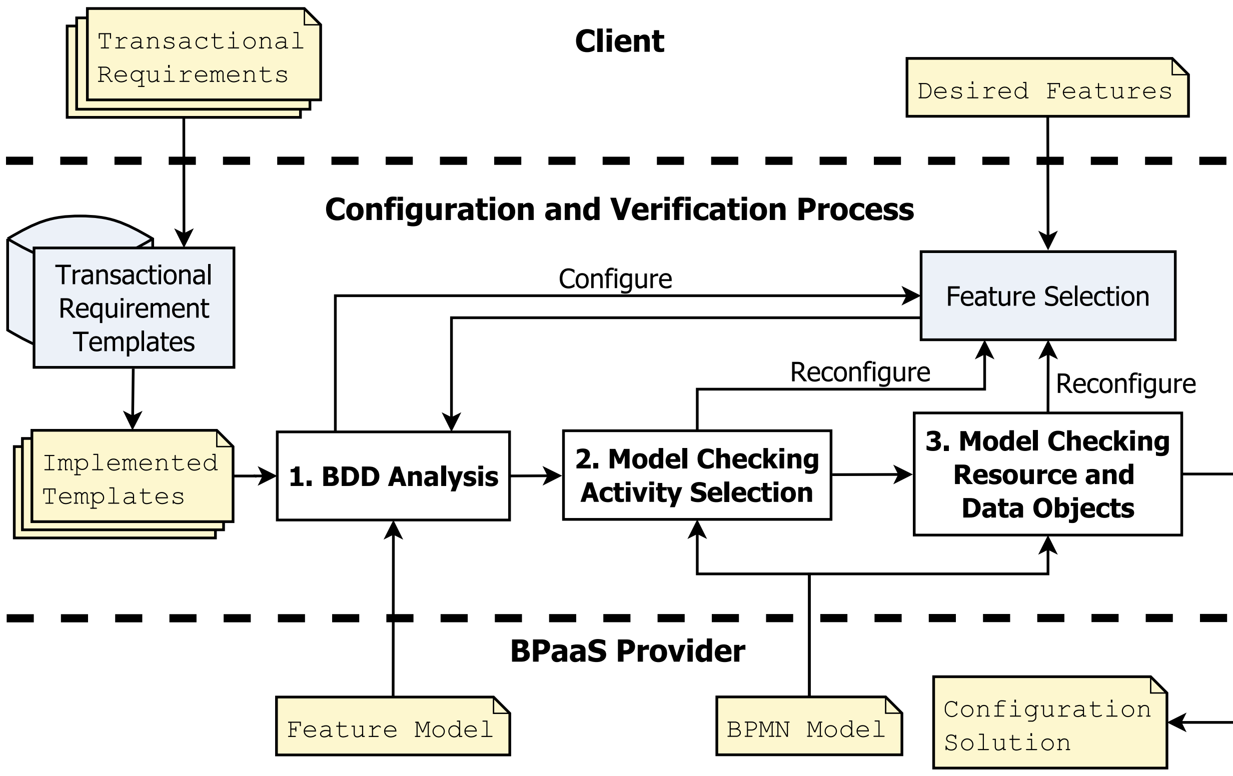

Transactional Requirements
Client
Desired Features
Configuration and Verification Process
Transactional Requirement Templates
Configure
Feature Selection
Reconfigure
Reconfigure
Implemented Templates
1. BDD Analysis
2. Model Checking Activity Selection
3. Model Checking Resource and Data Objects
BPaaS Provider
Feature Model
BPMN Model
Configuration Solution

Figure 5.6: Overview of the BPaaS configuration and verification process

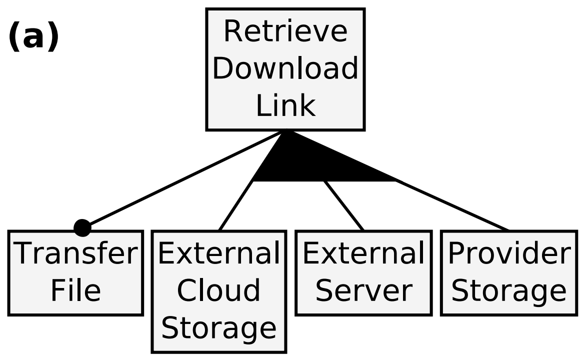
efficient for large and complex services. If model checking is successful, then a configuration solution has been identified. Feature selection is used by the client in order to add desired functionality or reconfigure the service in the event that model checking determines the current configuration to be invalid. Below, we describe in detail how BDD analysis and model checking is implemented in our configuration process.

5.4.2 BDD Analysis for Ensuring Domain Constraints

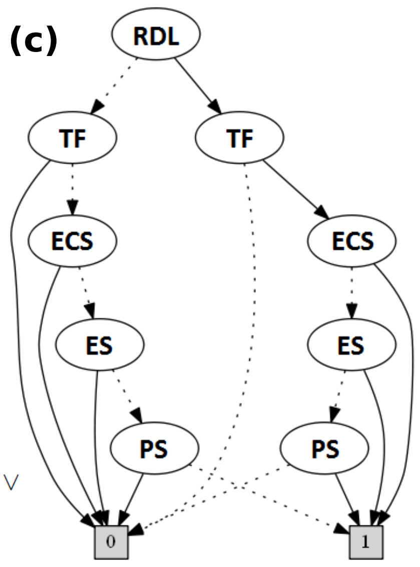
The first verification step in our approach is to identify all features required for the transactional requirements specified by the client, and determine whether they can all be selected while satisfying the feature model constraints. Therefore, we must determine that at least one valid configuration exists using the activities, resources, and data objects specified in the requirements set, or extra features required by the client.

To this end, we use BDD based analysis, which has been proven as an effective method for de-

(a)



(c)



(b)

$$(TransferFile \leftrightarrow RetrieveDownloadLink) \wedge$$
$$(RetrieveDownloadLink \leftrightarrow (ExternalCloudStorage \vee ExternalServer \vee ProviderStorage))$$

Figure 5.7: A selection of the feature model (a) transformed into propositional logic (b) and a BDD (c)

termining feature model satisfiability [13]. A BDD is an acyclic graph visualization of a propositional logic formula. Variables of the formula are represented as nodes with two outgoing branches, indicating their true or false assignment. The graph is constructed in such a way that each complete path from head node to terminal node represents the assignment of boolean variables. All paths terminate at a final true or false node, which determine whether the variable assignments of that path satisfy the propositional logic formula. Example BDDs are shown in Figure 5.7(c), and Appendix C.

Our analysis transforms the feature model with selections into a propositional logic formula, then into a BDD which can be checked for satisfiability. Figure 5.7 shows an example of these two transformation steps using a part of the checkout feature model. The BDD in Figure 5.7(c) uses acronyms of feature names for space considerations, and *0* and `1` nodes for the *false* and *true* respectively. A solid line from a node indicates true assignment, while a dashed line indicates false. By traversing solid lines from selected features, and dashed lines from unselected features, the `1` and *0* nodes indicate whether the preceding path was a valid or invalid selection.

A feature model can be transformed into a propositional logic formula according to the constraint conversions in Table 5.6. Figure 5.7(b) shows a propositional logic formula expressing both the *mandatory* and *XOR* relations between the features in Figure 5.7(a). To determine the satisfia-

Table 5.6: Propositional logic representations of the feature model constraints of Figure 5.4

Constraint	**Propositional Logic**
Mandatory	$A \leftrightarrow B$
Optional	$B \rightarrow A$
OR	$A \leftrightarrow (B1 \vee B2)$
XOR	$(B1 \leftrightarrow (\neg B2 \wedge A)) \wedge (B2 \leftrightarrow (\neg B1 \wedge A))$
Implication	$A \rightarrow B$
Exclusion	$\neg(A \wedge B)$

Table 5.7: A selection of features from the checkout BPaaS

Feature Type	**Selections**
Activities	Obtain Shipping Details, Register User, Place Shipping Order, Initiate Order, Release Product, Hold Product, Validate Login, Retrieve User Details, Confirm Shipping Details
Resources	AusPost, International Shipping, eWay, Paypal, Private Customer Repository, Microguru, SaaSu
Data Objects	Physical Product Details, Quantity, Shipping Address

bility of all features specified in the client's transactional requirements, they are added to the end of the formula separated by $\wedge$ operators. We use the JDD library[6] to automatically construct BDDs from these propositional logic formulas to check their satisfiability. The feature selection satisfies the formula if the BDD contains at least one transition to the final true node. If so, at least one valid configuration using the selected features exists. Otherwise, the client is faced with the choice of revising their transactional requirement set, or accepting that this BPaaS is unable to provide the necessary transactional integrity necessary for their operations.

BDD analysis is able to efficiently solve satisfiability problems, but the size of the BDD dependent on variable ordering in the underlying propositional logic property. While finding the most efficient ordering is an NP-complete problem [13], ordering variables from a depth-first traversal of the feature model has been shown as an effective strategy [160]. Therefore, we adopt this approach in our work when transforming the feature model to propositional logic.

For example, to check the validity of the feature selections shown in Table 5.7, the propositional logic formula in Figure 5.9 would be generated. Lines 1-20 of the property are a propositional logic representation of the checkout process feature model, constructed by a depth-first traversal of the constraints as they are ordered in the feature model. The selection of features from Table 5.7 are

[6]http://javaddlib.sourceforge.net/jdd/index.html

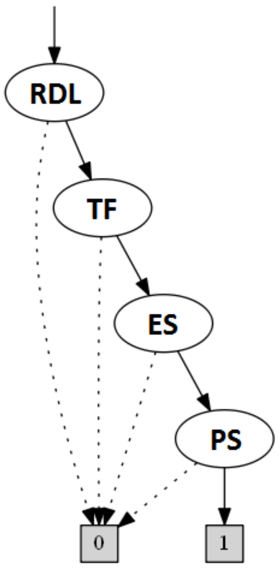


Figure 5.8: Binary Decision Diagram form of the feature model in Figure 5.7 with four feature selections

added to the end of the property using $\wedge$ operators, as shown in Line 21.

Due to its size and complexity, the BDD constructed from this property is displayed in Appendix C. However, as this BDD contained at least one path to the final true node, this selection of features is shown to be valid. For a smaller BDD example, we consider the section of the feature model shown in Figure 5.7(a), and select the features *Retrieve Download Link*, *Transfer File*, *External Server*, and *Provider Storage*. The BDD generate to confirm the validity of these selections is shown in Figure 5.8. The selection of `External Cloud Storage` is no longer relevant to the satisfaction of the constraints, as the OR relation containing it has already been satisfied. Therefore, JDD omits it from the updated BDD. What this BDD shows is that at least one solution is possible with these four features selected. The Java code for using JDD to construct the BDDs in Figure 5.7(c) and Figure 5.8 can be found in Appendix B.

5.4.3 Model Checking Against Transactional Requirements

In order to verify the process against a set of transactional requirements, we adapt the verification of our temporal logic templates from Chapter 4. Clients can use our temporal logic template set to formalize their transactional requirements for the configured process to conform to. Model

1. $(RecordFraudReport \rightarrow CheckoutService)\wedge$
2. $((InventorySaaS \leftrightarrow (\neg PrivateInventorySystem \wedge CheckoutService))\wedge$
   $(PrivateInventorySystem \leftrightarrow (\neg InventorySaaS \wedge CheckoutService)))\wedge$
3. $(CheckoutService \leftrightarrow (Epoch \vee Eway \vee Paypal))\wedge$
4. $(InitiateOrder \leftrightarrow (ReleaseProduct \wedge HoldProduct))\wedge$
5. $(InitiatePreOrder \leftrightarrow (ReleaseOrder \wedge HoldOrder))\wedge$
6. $((AccountingSaaS \leftrightarrow (\neg PrivateAccountingSystem \wedge CheckoutService))\wedge$
   $(PrivateAccountingSystem \leftrightarrow (\neg AccountingSaaS \wedge CheckoutService)))\wedge$
7. $(TransferFile \leftrightarrow (FTP \vee FTPS))\wedge$
8. $(TransferFile \leftrightarrow RetrieveDownloadLink)\wedge$
9. $(RetrieveDownloadLink \leftrightarrow (ExternalCloudStorage \vee ExternalServer\vee$
   $ProviderStorage))\wedge$
10. $(DigitalProductDetails \leftrightarrow RetrieveDownloadLink)\wedge$
11. $(RetrieveShippingDetails \rightarrow ObtainShippingDetails)\wedge$
12. $(PhyscialProductDetails \leftrightarrow (Quantity \wedge ObtainShippingDetails \wedge PlaceShippingOrder\wedge$
    $ShippingAddress))\wedge$
13. $(PhyscialProductDetails \leftrightarrow (AusPost \vee TollPriority \vee InternationalShipping\vee$
    $ClientShippingNotification))\wedge$
14. $(CheckoutService \leftrightarrow (DigitalProductDetails \vee PhyscialProductDetails))\wedge$
15. $(StorePaymentDetails \rightarrow ValidateLogin)\wedge$
16. $((PrivateCustomerRepository \leftrightarrow (\neg ProviderStorageCustomerRepository\wedge$
    $ValidateLogin)) \wedge (ProviderStorageCustomerRepository \leftrightarrow$
    $(\neg PrivateCustomerRepository \wedge ValidateLogin)))\wedge$
17. $(ValidateLogin \leftrightarrow (RetrieveUserData \wedge PaymentDetails \wedge RegisterUser))\wedge$
18. $(ValidateLogin \rightarrow CheckoutService)\wedge$
19. $(RetrieveShippingDetails \rightarrow ValidateLogin)\wedge$
20. $CheckoutService\wedge$
21. $ObtainShippingDetails \wedge RegisterUser \wedge PlaceShippingOrder \wedge InitiateOrder \wedge ReleaseProduct \wedge HoldProduct \wedge ValidateLogin \wedge RetrieveUserDetails \wedge ConfirmShippingDetails \wedge AusPost \wedge InternationalShipping \wedge eWay \wedge Paypal \wedge PrivateCustomerRepository \wedge Microguru \wedge SaaSu \wedge PhysicalProductDetails \wedge Quantity \wedge ShippingAddress$

Figure 5.9: Propositional logic form of the checkout process feature model with the feature selections of Table 5.7

checking can then be applied to formally verify that the process meets the expected behavior.

The verification approach for our template set applies a state space reduction algorithm, shown in Algorithm 3, to reduce the impact of the state space explosion problem inherent in model checking. This algorithm identified the properties (control or operational behavior states and inter-behavior messages) that were required by the requirements, and reduced the model to a minimal Kripke structure containing the temporal dependencies between those properties. As the inclusion of resource and data objects in BPaaS increases the model state space even further, we also address state space explosion by dividing the model checking problem between two phases: *activity selection* and *resource and data object selection*. Each phase aims to verify a different configuration perspective, and allows the state space of the model to be reduced further. Furthermore, some temporal logic properties may not be necessary to include in both phases, thereby simplifying the model checking problem. For example, if a requirement has no specification of resources and data objects, it will be verified during the activity selection phase and does not need to be included in the next.

In each model checking phase, a configuration is generated from the current feature selections, and verified against the transactional requirement set. Applying model checking in phases allows the client to focus on configuration perspectives separately. This simplifies the configuration and verification process for the client, in particular with services containing many configurable features. The two model checking phases are described below.

5.4.3.1 Activity Selection

In this phase, if a requirement specifies the inclusion of resources or data objects for an activity, only the presence of the activity in the configured process is verified. This reduces the state space of the process model verified by the model checker, simplifies the properties, and allows the client to focus primarily on activity configuration. Further state space reduction is enabled by using our *TKSR* algorithm (Algorithm 3) for generating reduced Kripke structures.

To verify activity selection against transactional requirements, the atomic propositions of interest include configurable activities selected from the feature model, and other activities or control

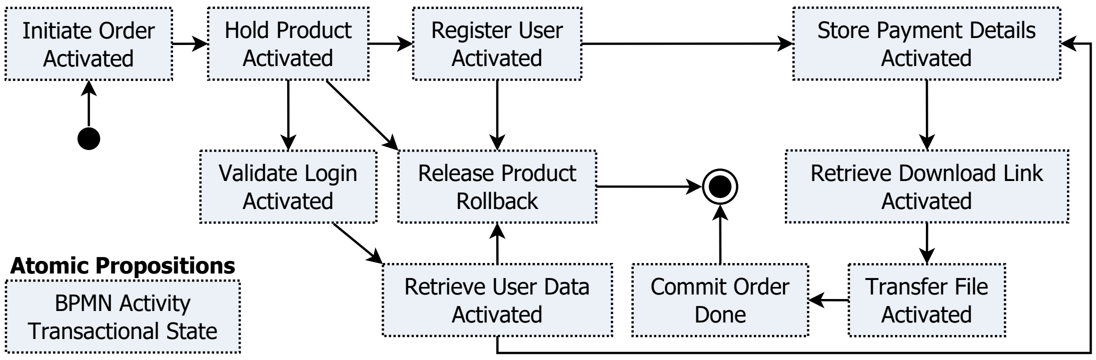


Figure 5.10: Kripke structure example for verifying the checkout activity selection

behavior states used by the temporal logic templates. For example, considering the checkout service, Figure 5.10 shows the Kripke structure generated to verify this requirement:

`Retrieve User Data` or `Register User` is necessary before entering the `Done` transactional state.

The template `ControlStateCritical` in Table 4.6 can be used to formalize this requirement. The Kripke structure contains the activities and transactional state necessary for verifying the requirement. In addition, the structure captures the control flow relation between other configurable activities selected by the client. These are included so that violating stack traces produced by the NuSMV model checker can indicate the configurable activities that led to the violation.

Model checking verifies the Kripke structure against the temporal logic forms of the client's transactional requirements. If a violation is found, the client must reconfigure the service by interpreting the model checking output. If the verification of the activity selection is successful, the second model checking phase can be undertaken.

5.4.3.2 Resource and Data Object Selection

In this model checking phase, only transactional requirements with resources or data objects in their specification need to be verified. All other requirements have been ensured in the activity selection phase. This phase is necessary because an activity can have more than one resource provisioned, in order to defer the selection to the user at runtime. For example, a configuration

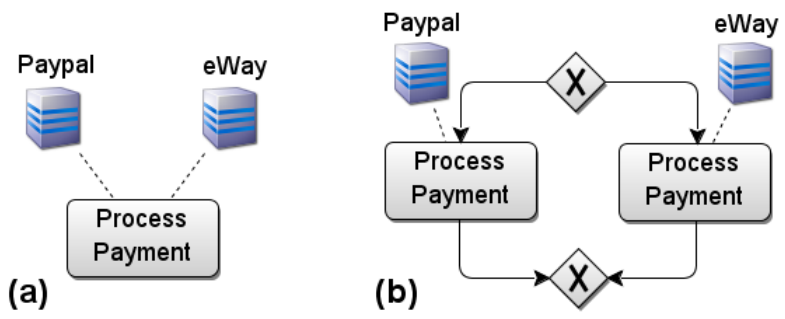


Figure 5.11: An example of how activities with multiple resources are traversed for the second model checking phase

may include several resources for the `Process Payment` activity. When verifying the activity selection, `Process Payment` will be treated as a single state. However, when considering resources, it becomes several states in an XOR structure. Figure 5.11 shows an example of this with *Process Payment* with both *PayPal* and *eWay* provisioned as resources. During this verification phase, the activity (Figure 5.11(a)) is replaced with the XOR structure in Figure 5.11(b), as both resources are not used simultaneously. Unlike resources, selected data objects are assumed to apply to every runtime instance and do not require process restructuring during verification.

A second Kripke structure must be generated for this phase. We again employ the *TKSR* algorithm for state space reduction of the BPaaS model. This time, the atomic propositions of this Kripke structure must be expanded to include resources and data objects used by the activities, in order to verify their selection against the requirements. For example, given the following requirement:

`Process Payment` with `Paypal` or `eWay` may be retried if failed, but the process must commit if it is successful.

The `RetriablePivot` template (Table A.6 of Appendix A) can be used to formalize this as two requirements: one with `Process Payment.Paypal` as the *Component* variable and the other with `Process Payment.eWay`. Figure 5.12 shows a Kripke structure generated by *TKSR* to verify these requirements.

If both model checking phases are successful, then a configuration solution has been found. For example, Figure 5.13 shows the BPMN of the configuration solution identified following ver-

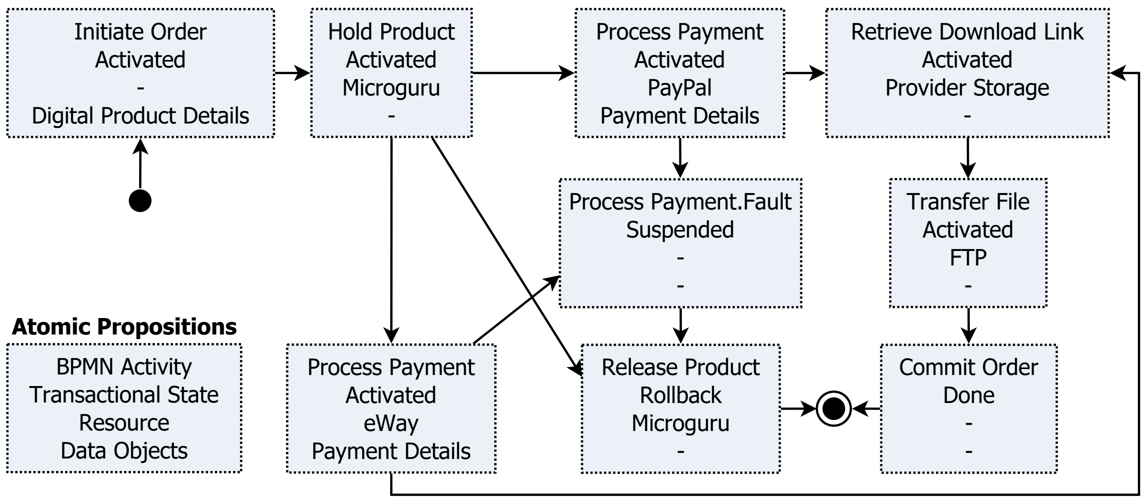


Figure 5.12: Kripke structure example for verifying resource and data object selection

ification of the Kripke structures in Figures 5.10 and 5.12, and the feature selections in Table 5.7. We aim to implement these model checking phases, as well as BDD analysis for checking configuration validity, into a structured configuration process for BPaaS clients.

5.4.3.3 Feature Selection

Feature selection is a task that is used potentially numerous times during configuration. Firstly, after conducting BDD analysis on all features in the transactional requirements set, but prior to model checking, further features must be selected by the client until the configuration is valid and complete with respect to the feature model. For example, all OR groups in the feature model must have at least one feature selected. Invalid configurations violate the provider's domain constraints, and can create incorrect process structures, such as omitting mandatory tasks. Therefore, our configuration process does not undertake the model checking phases until the client has selected enough desired features to satisfy all constraints.

To reduce the client burden for selecting from a potentially large number of features, once a feature is selected, all mandatory and implied child features are also selected automatically. Therefore, before a configuration is complete, the client must simply select at least one feature from every OR constraint, and one from every XOR constraint, which has a selected parent. Once the client has finished selecting desired features, BDD analysis must be performed once more to

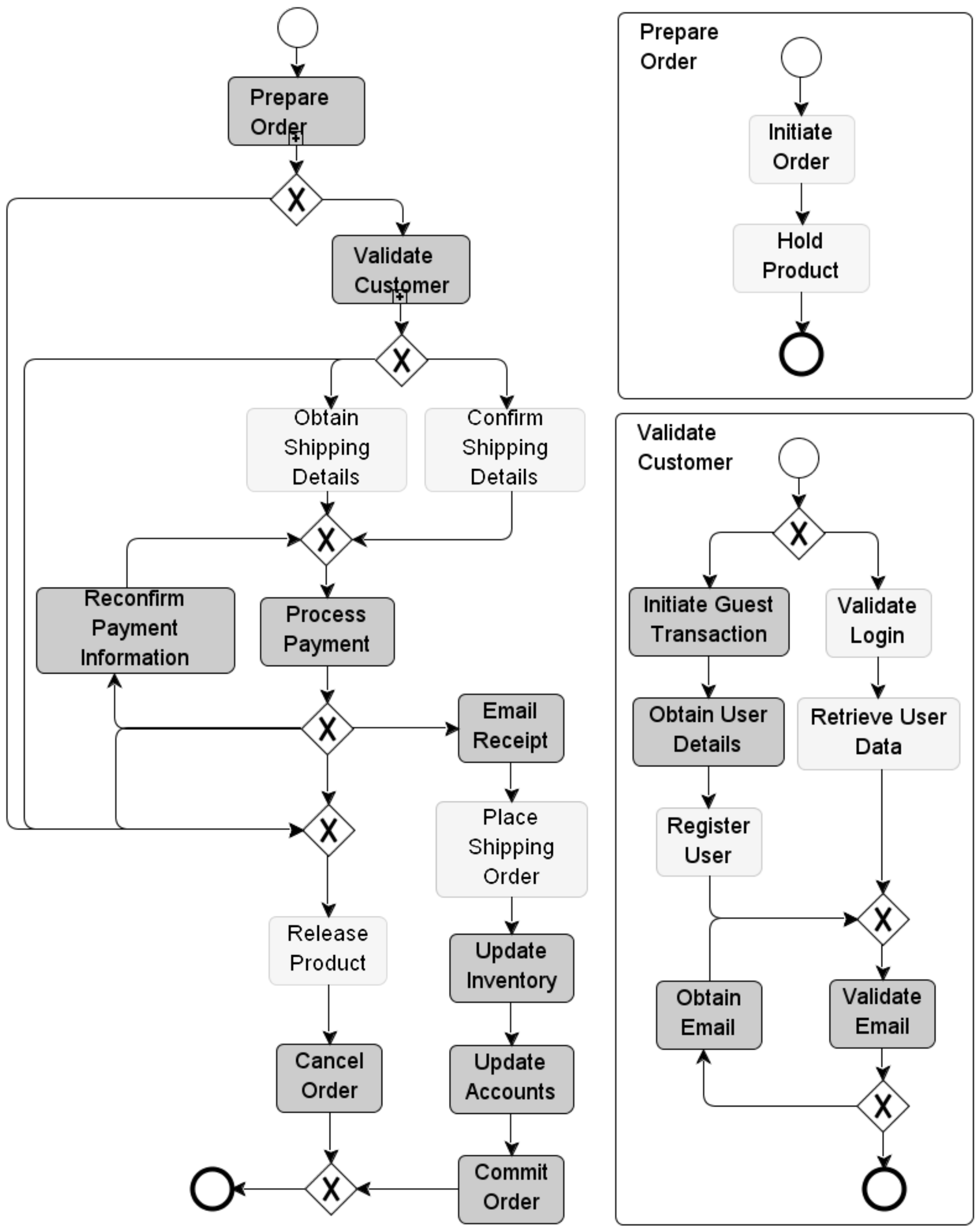


Figure 5.13: BPMN of a configuration solution of the Checkout BPaaS

ensure the configuration is valid (e.g. checking that exclusionary features have not been selected).

Feature selection is also used in order to revise the configuration if either model checking phase fails. This enables the client to change the behavior of the process, in order to satisfy violated transactional requirements. Following reconfiguration, BDD analysis is applied once more to ensure that the new feature selection does not violate the domain constraints, before model checking is re-attempted.

5.5 Summary

The increase in cloud computing adaptations in recent years has produced the concept of *Business Process as a Service* (BPaaS), whereby service providers are able to offer common or proven business processes to clients looking to automate and outsource parts of their operations. In order to increase the potential client base and exploit economies of scale, services may be configurable to individual client's requirements, such as including or excluding specific activities, resources, and data objects used in the business process.

In this chapter, we address the problem of managing BPaaS configuration in a way to ensure that the resulting service i) is valid with respect to configuration constraints of the provider, and ii) satisfies transactional requirements drawn from the business rules of the client. Our approach utilizes several modelling techniques, including BPMN for business process structure, statecharts for transactional state, feature models for configuration constraints. Using these models, we develop a process that applies formal methods to configure BPaaS.

Firstly, our temporal logic templates for transactional requirements can be adapted to help clients specify required application-dependent transactional behavior. Further configurable features (activities, resources, and data objects) can also be selected as configuration requirements by the client. To ensure that the features selected do not violate the domain constraints of the service provider, we employ *Binary Decision Diagram* (BDD) analysis. Following this, model checking is applied to verify the configured BPaaS against the transactional requirements provided by the client. To reduce the impact of state-space explosion, we employ the state-space reduction algorithm defined in Algorithm 3, and split the model checking into two phases. These phases verify different configuration perspectives separately, and allow for the state space and temporal logic properties to be reduced further.

CHAPTER 6

Prototype Implementation and Experimental Analysis

In order to evaluate the contributions we present in Chapters 3-5, we implement them in a prototype tool. This tool, named TL-VIEWS, enables developers to model and verify service-oriented processes at design time according to conversation rules and transactional requirements they provide. Furthermore, it also provides an interface to manage BPaaS configuration from the client perspective.

First, we give an overview of the implementation of the architecture of TL-VIEWS, which contains five primary modules. Two interfaces provide the means for either modeling and veriftying service-oriented processes at design-time, or configuring BPaaS in accordance with the process described in Chapter 5. A verification module handles model checking by using the NuSMV tool, while also managing temporal logic transformations and state space reduction. BPaaS configuration is enabled through a controller module, while a BDD analysis module utilizes the JDD library for verifying feature selections against domain constraints.

We also provide a comprehensive experimental analysis of the verification and configuration approaches implemented in TL-VIEWS. Firstly, we perform two service-oriented process validation scenarios in order to demonstrate the effectiveness of our design-time conversation rule checking and temporal logic template verification approaches. Then, we use two BPaaS configuration scenarios to show how our configuration process can detect and domain constraint and transactional requirement violations. Finally, we perform a series of performance tests to verify the effectiveness of our state space reduction measures when applying model checking to large and complex models.

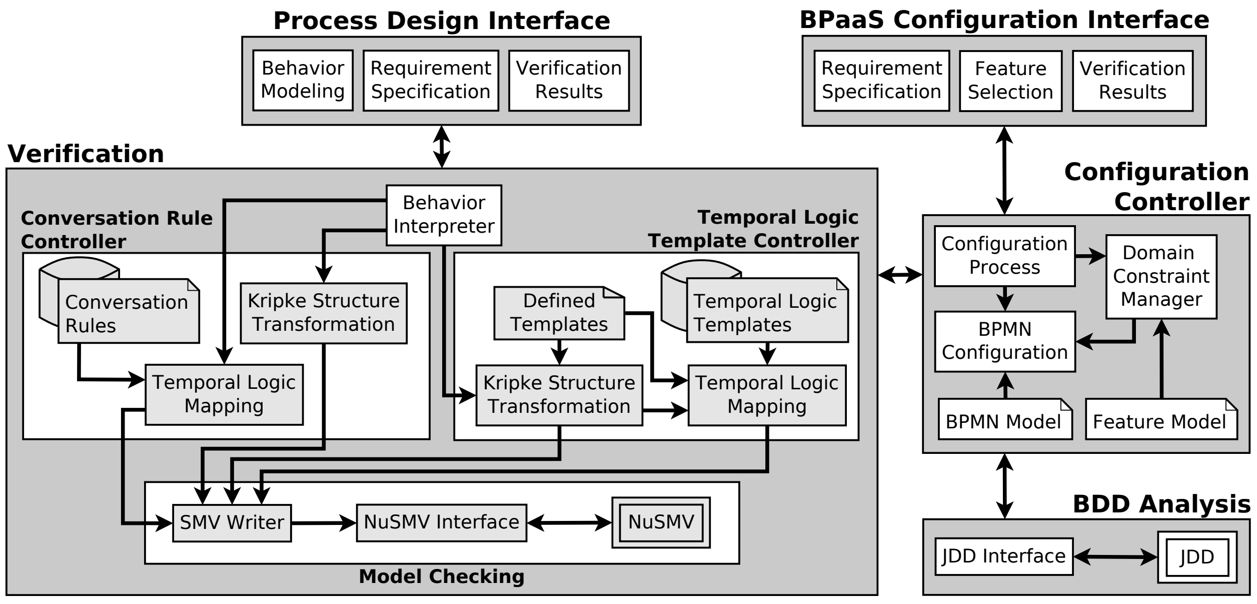


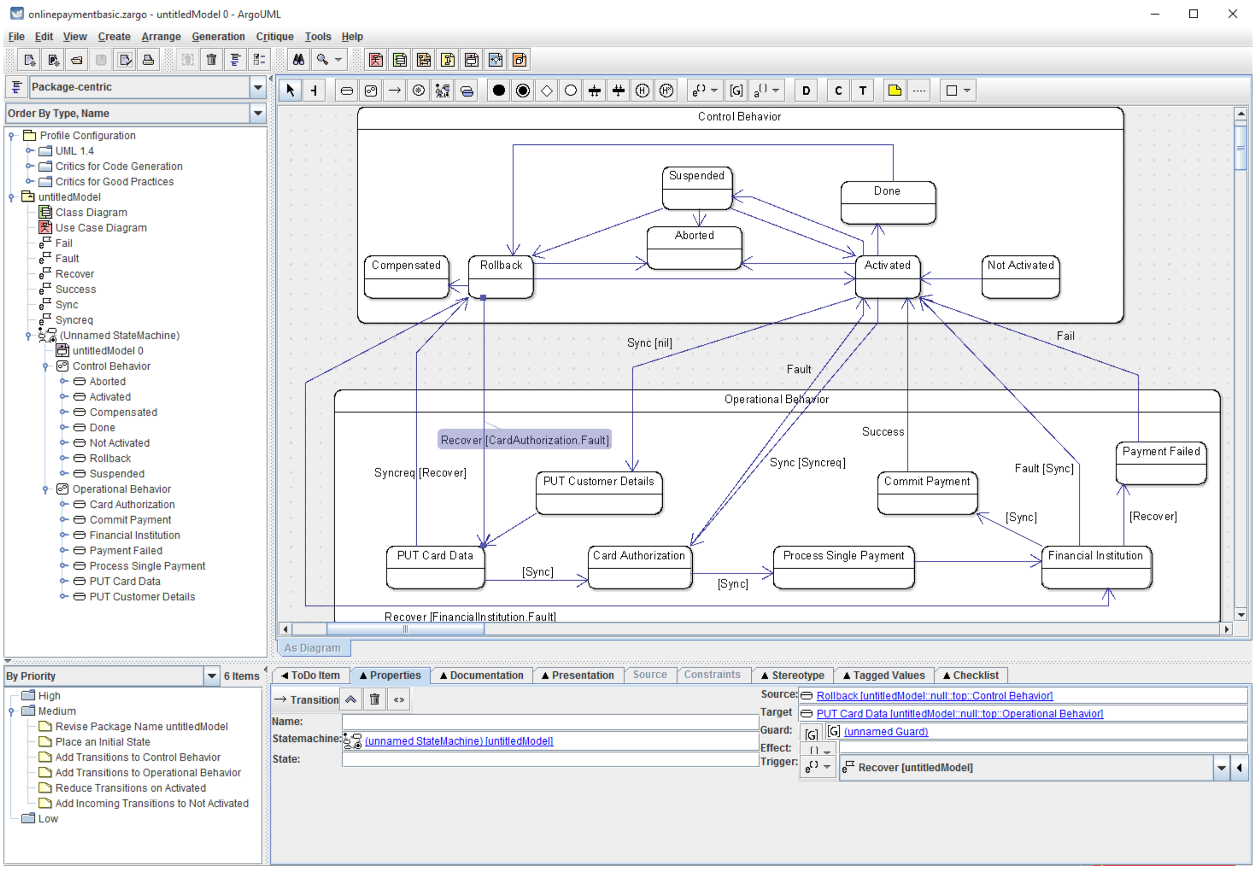
Figure 6.1: TL-VIEWS architecture

6.1 Implementation Architecture

The name of our prototype is TL-VIEWS (*Temporal Logic VerIfication of transactional bEhavior of Web Services*) [24]. TL-VIEWS is developed in Java and utilizes existing modeling and verification tools. Figure 6.1 shows the TL-VIEWS architecture, containing five primary modules: *Process Design Interface*, *Verification*, *BPaaS Configuration Interface*, *Configuration Controller*, and *BDD Analysis*. Below, we outline the role of each module in implementing model verification against conversation rules, transactional requirements specified using templates, and configuring BPaaS.

6.1.1 Verification Against Conversation Rules and Templates

To implement our conversation rule and transactional requirement template verification processes, TL-VIEWS contains modules for modeling control and operational behaviors, managing the verification process, and handling model checking.



Name:
Statemachine: (unnamed StateMachine) [untitledModel]
State:
Source: Rollback [untitledModel::null::top::Control Behavior]
Target: PUT Card Data [untitledModel::null::top::Operational Behavior]
Guard: (unnamed Guard)
Effect:
Trigger: Recover [untitledModel]

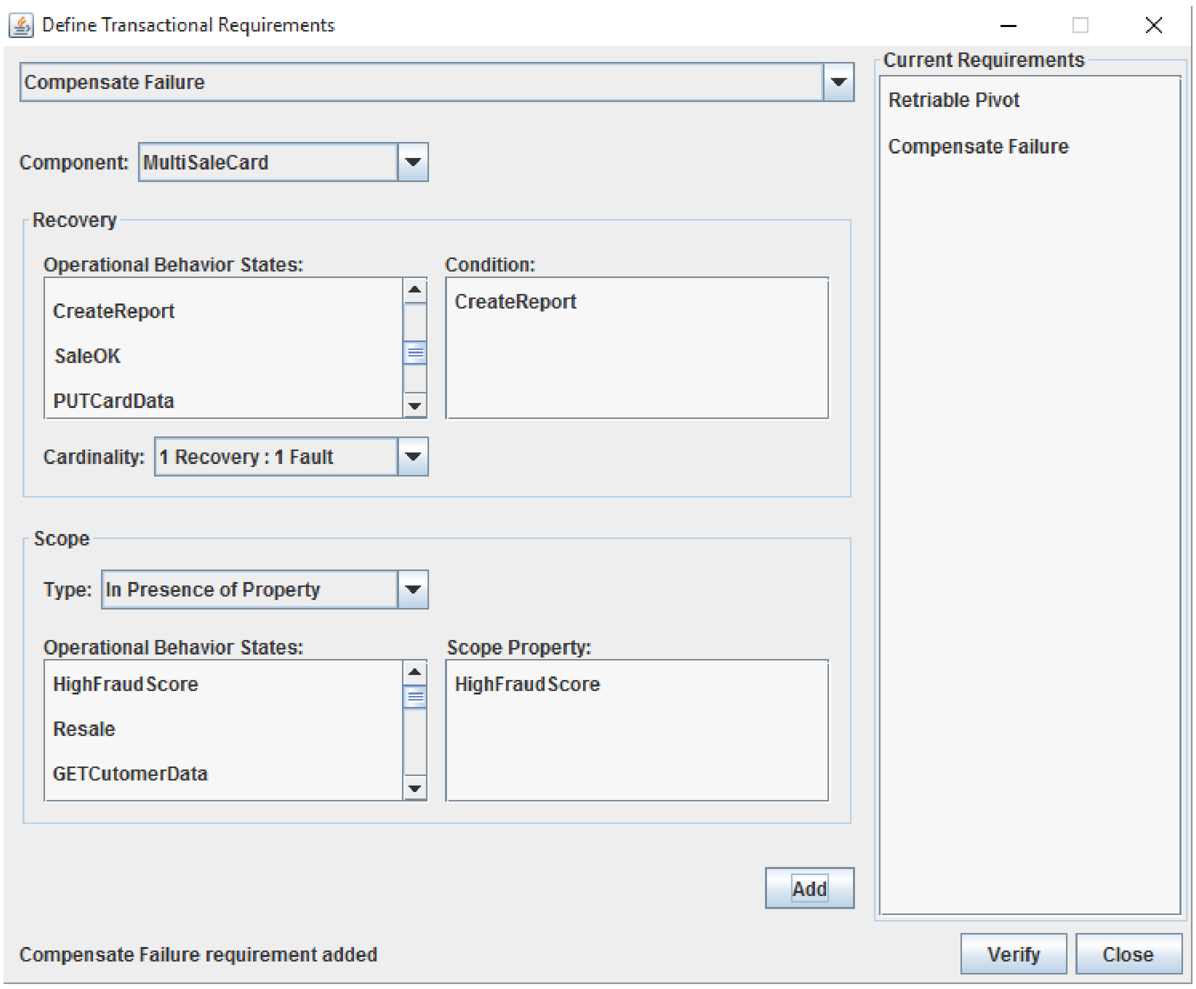
Figure 6.2: TL-VIEWS process design interface

6.1.1.1 Process Design Interface

The purpose of this interface is to enable developers to design service oriented processes as interacting control and operational behavior models. The interface also allows verification of both inter-behavior conversations and specified transactional requirements, as detailed in Chapters 3 and 4.

We leverage the open source UML diagramming application ArgoUML[1] for this interface. Figure 6.2 shows the interface in use for modeling control and operational behaviors, with inter-behavior messages. An extension to the ArgoUML GUI allows developers to formalise transactional requirements using the template set (Figure 6.3). Verification results are presented as shown in Figure 6.4.

[1]http://argouml.tigris.org/


Define Transactional Requirements

Compensate Failure

Component: MultiSaleCard

Recovery

Operational Behavior States: CreateReport, SaleOK, PUTCardData

Condition: CreateReport

Cardinality: 1 Recovery : 1 Fault

Scope

Type: In Presence of Property

Operational Behavior States: HighFraudScore, Resale, GETCutomerData

Scope Property: HighFraudScore

Current Requirements: Retriable Pivot, Compensate Failure

Compensate Failure requirement added

Figure 6.3: TL-VIEWS requirement specification window

6.1.1.2 Verification

This module handles control and operational behavior verification against the conversation rule set and transactional requirements specified by the process developer or BPaaS client. Due to its complexity, we split this into four sub-modules.

Behavior Interpreter

The *Behavior Interpreter* serves as an interface between the control and operational behavior models and the various verification modules that analyze them. This module generates and maintains the flattened behavior model. The flattened behavior model is stored internally and invisible to the user, it only exists as a transformation step in the verification process. Transforming the control and operational behavior models into a flattened behavior model is a trivial process (described in Chapter 3), but it simplifies the generation of Kripke structures for model checking.

Conversation Rule Controller

This controller serves two main purposes. Firstly, it applies *CKSR* (Algorithm 1) to produce a Kripke structure from the flattened behavior model of the Behavior Interpreter. Secondly, it formalizes the conversation rules according to the temporal logic transformations defined in Chapter 3.4.1. Only the rules necessary for the model are formalized. For example, if the model contains no *Ping* messages, then conversation rule `MS5` is unnecessary and omitted.

Temporal Logic Template Controller

Like the Conversation Rule Controller, the Temporal Logic Template Controller manages Kripke transformation and temporal logic mapping. *TKSR* (Algorithm 3) transforms the flattened behavior model into a reduced Kripke structure as described in Chapter 4.The templates the developer uses to formalize transactional requirements contain a temporal logic field, which is used to map the requirement to concrete LTL or CTL properties.

Model Checking

Model checking [43] is a formal method that exhaustively verifies that the behavior of a given model conforms to a set of properties. We use the NuSMV[2] model checker to verify Kripke structures against the temporal logic forms of conversation rules and transactional requirements. Our verification uses the symbolic BDD-based model checking feature of NuSMV. This constructs BDDs from the input model to apply several verification techniques, including fair CTL model checking and LTL model checking (by an algorithm that reduces the problem to CTL model checking [44]). We employ NuSMV instead of alternatives such as SPIN [76] and UPPAAL [14], as it provides support for properties formulated in both LTL and CTL. Further technical details on NuSMV can be found in publications [41, 42] by members of the development team.

[2]http://nusmv.fbk.eu/

The model checking module contains the interface for invoking NuSMV and interpreting output. The Kripke structures and concrete temporal logic properties are written in to SMV files, the input language for NuSMV. Appendixes D and E contain example SMV files for both conversation rule checking and verification against temporal logic templates.

The results from the verification process are then presented to the developer in a results window. Interpreting conversation rule or transactional requirement violating stacktraces identified by NuSMV can the help developer refine an erroneous model by highlighting the invalid behavior.

6.1.2 BPaaS Configuration

Managing BPaaS configuration requires three additional modules to those defined above. These handle tasks such as enabling client interaction, managing BDD analysis, and modifying configurable BPMN models.

6.1.2.1 BPaaS Configuration Interface

This interface allows clients to configure a BPaaS using the process outlined in Chapter 5. Clients may use this interface to specify requirements in the same manner as in the process design interface (Figure 6.3). The interface also allows clients to select additional desired features and view the verification results from BDD analysis and model checking phases.

6.1.2.2 Configuration Controller

The configuration controller manages the steps in the configuration process proposed in Chapter 5.4.1. A BPMN file of the complete BPaaS is used, with a feature model to indicate configurable activities, resources, and data objects. The feature selections and domain constraints are interpreted and managed by a *Domain Constraint Manager*, while *BPMN Configuration* manages the construction of configured BPMN models. By default, all configurable activities are included in the BPMN model, and it is configured by removing unselected activities and redirecting their incoming branch to the target of their outcoming branch. If the removal of an activity leaves a branch joining two connectors (such as an XOR split and join), then the branch is also removed.

The *Configuration Process* manages the use of these tools, and invokes the verification and BDD analysis modules as necessary. Figure 5.6 shows how the formal methods are used for configuration..

6.1.2.3 BDD Analysis

This module uses the JDD library[3] for BDD creation and analysis. JDD is a Java library that enables construction and manipulation of BDDs and *Zero-suppressed Decision Diagrams* (Z-BDDs). In TL-VIEWS, it is used to encode propositional logic properties into a BDD, which can be checked for satisfiability. This is done using the BDD class provided by JDD, and adding properties according to a depth-first traversal of the feature model. After the feature model is represented as a BDD, the feature selections are appended. An example of BDD generated by JDD is shown in Figure 5.9 in the previous chapter.

The configuration process uses this functionality to verify that a selection of features satisfies the domain constraints of the provider, as described in Chapter 5.4.2. This is done to ensure the satisfiability of i) all feature specified in the transactional requirement set provided by the client, and ii) all subsequent feature selections and revisions during the configuration process. This BDD analysis must be performed any time features are added or removed from the configuration in verify that the domain constraints remain satisfied.

6.2 Experimental Analysis

The experimental analysis of our contributions comprises of i) a series of validation scenarios, and ii) a performance evaluation using test suites of increasingly large and complex models. With these tests we aim to show how the effectiveness of our approaches, as well as their feasibility with large models and requirement sets.

[3]http://javaddlib.sourceforge.net/jdd/index.html

6.2.1 Validation Scenarios

We apply two validation scenarios to each contribution of our work in order to demonstrate their use and effectiveness. Two Web service composition designs are verified according to conversation rules and transactional requirement sets. We also propose two scenarios for configuring a BPaaS according to client transactional requirements.

6.2.1.1 Online Payment Service-Oriented Process

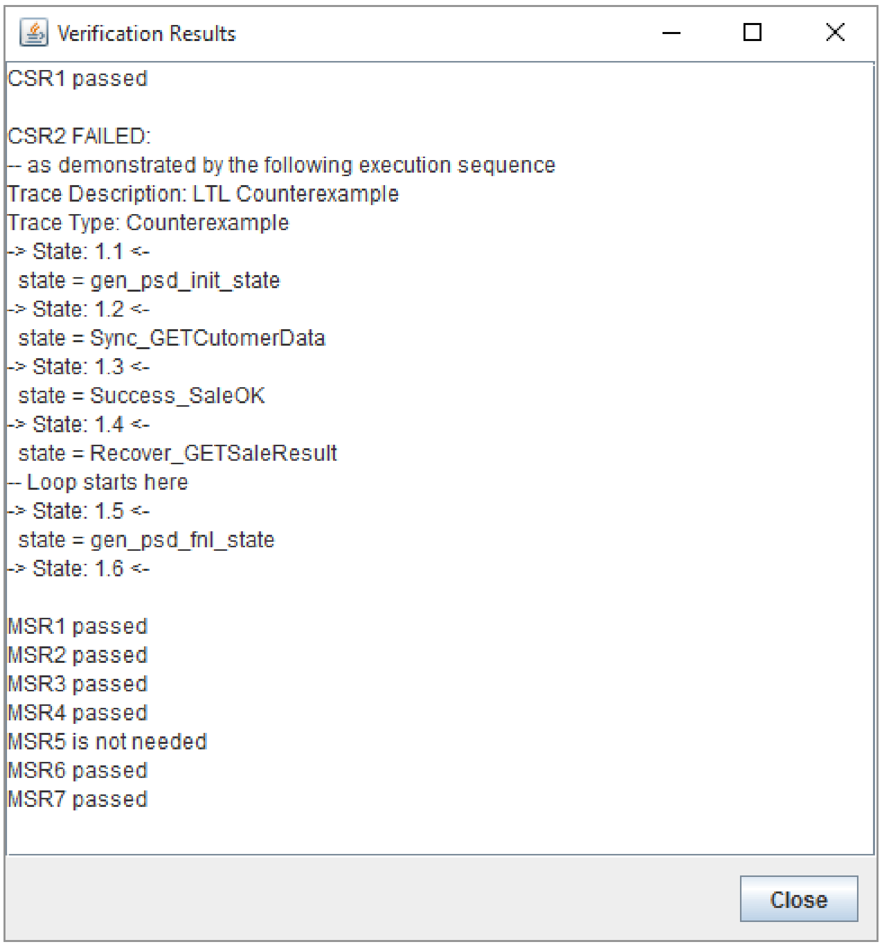
The online payment Web service composition, shown in Figure 3.4, enables payments to be processed using either credit card or direct debit. This example scenario has important requirements for transactional integrity. The process usees real-world services, and carries consequences for faulty transactional behavior such as mismanaged funds, loss of consumer confidence, and interruption of business operations.

Conversation Rule Checking

For conversation rule verification, we use two example scenarios of service-oriented processes which have important requirements for transactional integrity. These processes use real-world services, and carry consequences for faulty transactional behavior such as mismanaged funds, loss of consumer confidence, data inconsistency across sites, and interruption of business operations. We firstly employ the online payment process that appears in Chapters 3 and 4, followed by a process for enrolling in educational courses.

In Chapter 3, we demonstrated our *CKSR* algorithm to reduce this model to a Kripke structure for verification (Figure 3.7). This Kripke structure is formalized in SMV, and the conversation rules are translated to LTL and CTL. These are included in a single input file for the NuSMV model checker. Appendix D contains this input file in full.

TL-VIEWS uses NuSMV to confirm that our current model satisfies the conversation rule set. To demonstrate the ability of TL-VIEWS to detect violations, we insert a manual error. In our erroneous model, the task `Refund` indicates successful compensation with a *Success* message,



```
CSR1 passed

CSR2 FAILED:
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  state = gen_psd_init_state
-> State: 1.2 <-
  state = Sync_GETCutomerData
-> State: 1.3 <-
  state = Success_SaleOK
-> State: 1.4 <-
  state = Recover_GETSaleResult
-- Loop starts here
-> State: 1.5 <-
  state = gen_psd_fnl_state
-> State: 1.6 <-

MSR1 passed
MSR2 passed
MSR3 passed
MSR4 passed
MSR5 is not needed
MSR6 passed
MSR7 passed
```

Figure 6.4: TL-VIEWS results window following unsuccessful conversation rule verification

but `Non-Refundable` does not.

After running conversation rule checking following this change, TL-VIEWS detects a violation shown in Figure 6.4. The stack trace shows a sequence of inter-behavior messages that violate the rule `CSR2`. `Recover` is the final message in this conversation session, but as shown in the `CSR2` definition in Table 3.8, it is not a valid terminating message type. This stack trace can be used to identify the design problem that the compensation activities beginning with `GET Sale Result` do not always complete and respond with a valid message.

Verification Against Transactional Requirements

We identify six critical transactional requirements that apply to the online payment design, which can be implemented using our template set:

OP1: The sale of refundable items require `Refund` to be undone.

OP2: When `Resale` fails, the sale can be retried by the user re-entering card or account data.

OP3: Once a direct deposit sale has been confirmed as successful, the transaction must commit.

OP4: When a high fraud score has been returned, `Create Report` must be executed before the sale aborts.

OP5: Even if `Resale` is not successful, the sale should be able to complete using alternative services.

OP6: Commit should be reachable even if the outcome of `MultiSale Card` is delayed.

Table 6.1 shows how each requirement can be implemented using one of our templates. An input file was generated for NuSMV containing the temporal logic forms of the transactional requirements together with the reduced Kripke structure generated by TKSR. This input file is included in full in Appendix D.

NuSMV verified the reduced Kripke structure against these requirements and determined that the design conforms to all of them except `OP6`. Interpreting the stack trace shows that the `Done` control behavior state is unreachable following a `Delay` message to `MultiSale Card`. All Kripke states containing a `Delay` message to `MultiSale Card` only have transitions leading to `Abort`.

To satisfy `OP6`, the design must be refined to enable `MultiSale Card` to be retried or replaced with another operation following a delay. The following changes were made to the model:

- A *Syncreq* message originating from `MultiSale Card` to `Activated` was added to indicate that retrial is needed.
- The responding *Sync* message is sent back to `MultiSale Card` with a guard condition that prevents it from being used in other scenarios.

The model checking in TL-VIEWS confirmed that these changes satisfied `OP6`. It also determined that the changes did not violate the other requirements or conversation rule set.

Table 6.1: Transactional requirements for the online payment model

ID	Template	Variables	Values
OP1	Compensate Success	*Component*	Sale OK
		Recovery	Refund $\vee$ Non-Refundable
		Card	1:1
		Scope	Global
OP2	Alternative	*Component*	Resale
		Recovery	PUT Account Data $\vee$ PUT Card Data
		Card	1:1
		Scope	Global
OP3	Retriable Pivot	*Activity*	Check Sales
		Scope	Global
OP4	ControlState Critical	*Control State*	Aborted
		Condition	Create Report
		Scope	High Fraud Score
OP5	ControlState Reachable	*Control State*	Done
		Condition	Resale.Syncreq
		Scope	Global
OP6	ControlState Reachable	*Control State*	Done
		Condition	MultiSale Card.Delay
		Scope	Global

6.2.1.2 Course Enrolment Service-Oriented Process

This process is owned by an educational institution that is offering courses for mature-age students on a variety of subjects. Registered students may enrol in these courses through a web form interface that utilizes several third party Web services. The process is invoked once a student selects a specific course they wish to enrol in.

The enrolment process must first retrieve the details of the student, or make a temporary profile to be verified by administrators afterwards. Registration for courses must consider several factors, such as if registration is open, the courses and lessons the student has already enrolled in, the lessons that are part of the course, and the current academic calendar of the student. To enrol in a course, a student must enrol in one or more individual lessons, while ensuring that no unresolvable clashes occur with their existing calendar. Once that is completed, the course registration is confirmed, and the invoice attached to the student's account is updated.

The management of courses and students is done using the *GlobalTeach* Web service API [4].

[4]http://www.programmableweb.com/api/globalteach-lms

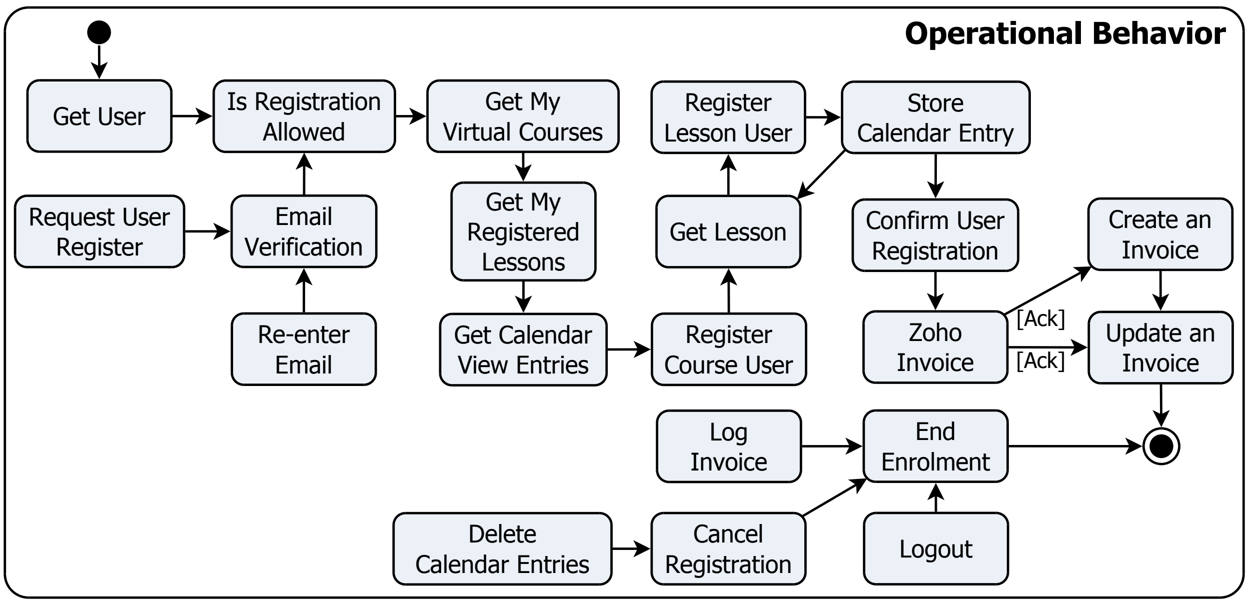


Figure 6.5: Operational behavior model of the lesson enrolment composition

The *Zoho Invoice* API [5] is used to manage billing of new registrations by creating and updating invoices. A *Verify Email* [6] Web service ensures that the email address provided by the user really exists.

Using these services, we model the operational behavior model of this composition as shown in Figure 6.5. Two transitions in the model contain a guard condition for `Ack`, because the invoicing services should not be invoked if the Zoho servers cannot acknowledge their liveness. The inter-behavior messages defined for this model are listed in Table 6.2. These messages enable behavior such as an automatic user logout following a long delay between inputs, and logging invoices for future processing if the Zoho services are experiencing an outage.

Conversation Rule Checking

After modeling this design in the TL-VIEWS interface, conversation rule checking was triggered, producing the output shown in Figure 6.6. The rule `MSR2`, as shown in Table 3.9, specifies that the valid responses to a *Delay* message are *Syncreq*, *Fault*, *Success*, and *Fail*. From the output, we can determine that following the *Delay* message to the *Register Course User* activity, the control

[5]http://www.zoho.com/invoice/api/v3/
[6]https://www.mashape.com/fetch/verify-email

Table 6.2: The inter-behavior messages used in the lesson enrolment composition

Message	Source	Target	Condition
Sync	*Activated*	*Get User*	`nil`
	Activated	*Request User Register*	`Get User.Syncreq`
	Activated	*Re-enter Email*	`Email Verification.Syncreq`
Syncreq	*Get User*	*Activated*	–
	Email Verification	*Activated*	–
Delay	*Activated*	*Register Course User*	–
Fault	*Is Registration Allowed*	*Activated*	–
	Get Lesson	*Activated*	–
	Create an Invoice	*Activated*	–
	Update an Invoice	*Activated*	–
Success	*Update an Invoice*	*Activated*	–
Fail	*End Enrolment*	*Rollback*	–
Recover	*Rollback*	*Delete Calendar Entries*	`Get Lesson.Fault`
	Rollback	*Logout*	`Register Course User.Fault`
	Rollback	*Log Invoice*	`Create an Invoice.Fault` ∨ `Update an Invoice.Fault` ∨ `Zoho Invoice.Ping`
	Rollback	*End Enrolment*	`Is Registration Allowed.Fault`
Ping	*Activated*	*Zoho Invoice*	–
Ack	*Zoho Invoice*	*Activated*	–

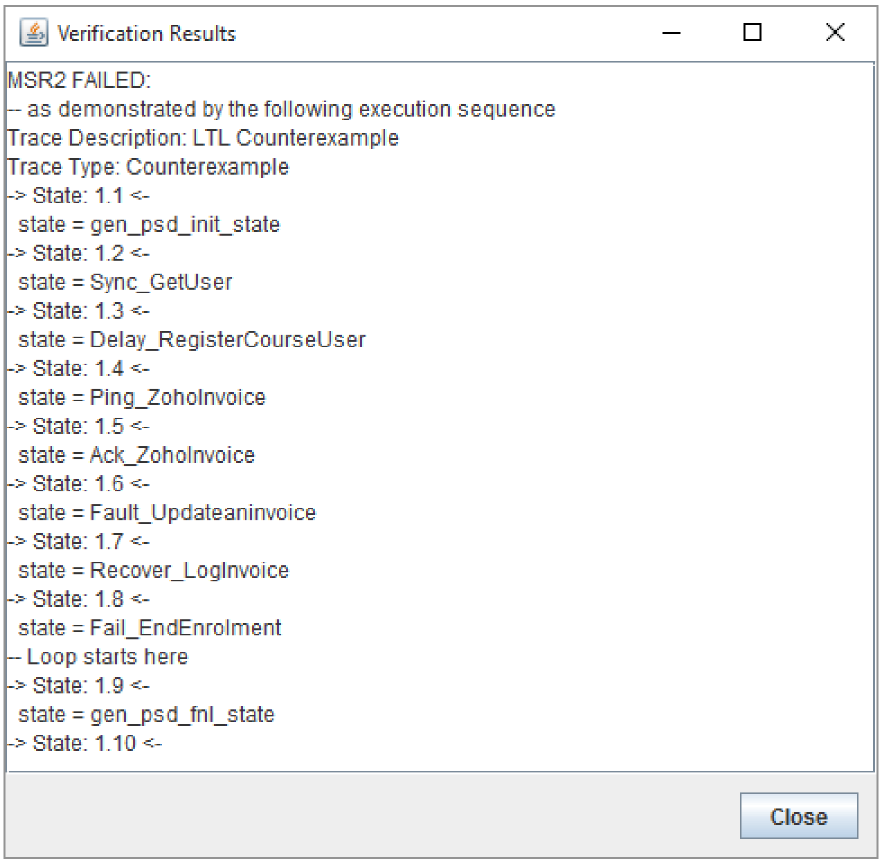
behavior is able to *Ping* Zoho Invoice. This not only violates `MSR2`, but skips several essential activities between *Register Course User* and invoicing. It indicates that the ping sent to the Zoho Invoice servers lacks sufficient specification on when it should be used.

The simplest way to resolve this design issue is to attach a guard condition to the *Ping* message to ensures it is only sent once the invoicing activities are reached. Attaching the condition `[Sync]` to the *Ping* message will prevent it from being sent by the control behavior while waiting for a response to *Delay*. After this update, TL-VIEWS confirms that the model satisfies all conversation rules.

Verification Against Temporal Logic Templates

The course enrolment process has a set of 9 transactional requirements provided by the developer:

`CE1`: If an invoice is unable to *updated* using the Zoho Invoice services, then the changes must be logged locally to be entered later.



```
MSR2 FAILED:
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  state = gen_psd_init_state
-> State: 1.2 <-
  state = Sync_GetUser
-> State: 1.3 <-
  state = Delay_RegisterCourseUser
-> State: 1.4 <-
  state = Ping_ZohoInvoice
-> State: 1.5 <-
  state = Ack_ZohoInvoice
-> State: 1.6 <-
  state = Fault_Updateaninvoice
-> State: 1.7 <-
  state = Recover_LogInvoice
-> State: 1.8 <-
  state = Fail_EndEnrolment
-- Loop starts here
-> State: 1.9 <-
  state = gen_psd_fnl_state
-> State: 1.10 <-
```

Figure 6.6: Conversation rule checking output for the online enrolment model

`CE2:` If an invoice is unable to *created* using the Zoho Invoice services, then the changes must be logged locally to be entered later.

`CE3:` `Cancel Registration` is required to undo the effect of `Register Course User` when the process needs to be aborted.

`CE4:` If user data cannot be found, then a temporary account may be created to be verified later.

`CE5:` If `Email Verification` fails, the user can re-enter a new address to be verified.

`CE6:` If registration for this course in not permitted, the process must terminate.

`CE7:` If a new invoice must be created, then it also needs to be updated before the process commits.

`CE8:` Once invoicing is reached, the process must briefly suspend while the liveness of Zoho Invoice servers is confirmed.

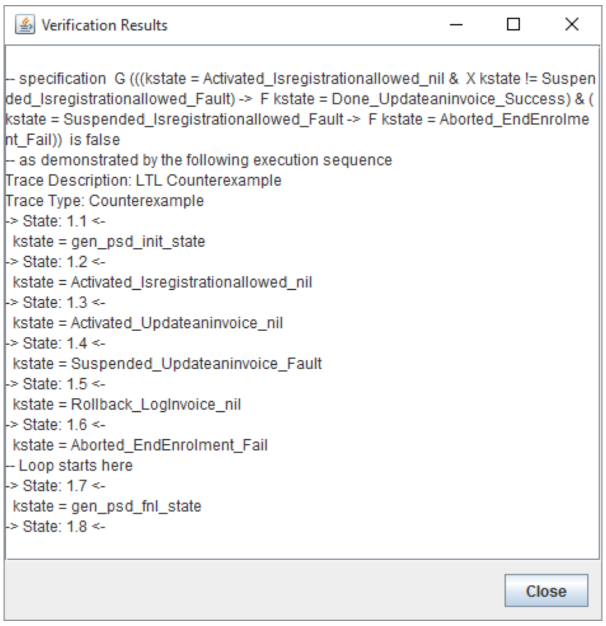
Table 6.3: Transactional requirements for the course enrolment model

ID	Template	Variables	Values
CE1	Compensate Failure	*Component*	Update Invoice
		Recovery	Log Invoice
		Card	1:1
		Scope	Global
CE2	Compensate Failure	*Component*	Create an Invoice
		Recovery	Log Invoice
		Card	1:1
		Scope	Global
CE3	Compensate Success	*Component*	Register Course User
		Recovery	Cancel Registration
		Card	1:1
		Scope	Global
CE4	Alternative	*Component*	Get User
		Recovery	Request User Registration
		Card	1:1
		Scope	Global
CE5	Alternative	*Component*	Email Verification
		Recovery	Re-enter Email
		Card	1:1
		Scope	Global
CE6	NonRetriable Pivot	*Activity*	Is Registration Allowed
		Scope	Global
CE7	ControlState Critical	*Control State*	Done
		Condition	Update Invoice
		Scope	Create an Invoice
CE8	ControlState Trigger	*Control State*	Suspended
		Condition	Zoho Invoice
		Scope	Global
CE9	ControlState Unreachable	*Control State*	Aborted
		Condition	Confirm User Registration
		Scope	Global

CE9: Once registration has been confirmed, the process can no longer be aborted.

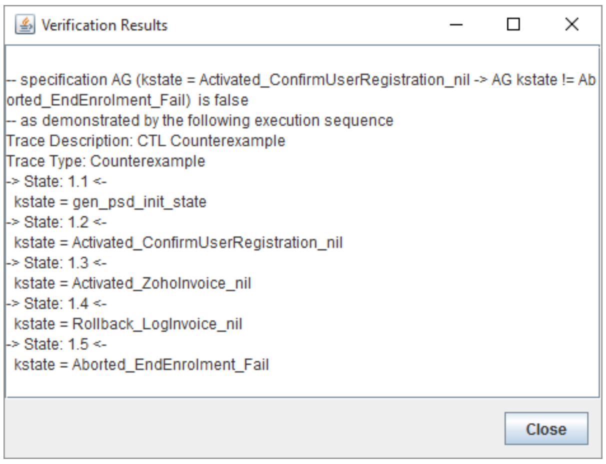
Table 6.3 shows how the requirements can be implemented using our template set. The complete SMV input file with temporal logic properties and reduced Kripke structure can be found in Appendix E. The temporal logic properties and Kripke structure were input to NuSMV as the shown in the file in Figures E.3 and E.3 respectively.

NuSMV was able to confirm that the course enrolment process as specified in Figure 6.5 and Table 6.2 satisfies all requirements except CE4 and CE9. The stack trace violating CE4 (Figure 6.7(a)) showed that faults during invoicing allows the process to abort, therefore violating the requirement that it must commit after Is Registration Allowed successfully completes.



```
-- specification  G (((kstate = Activated_Isregistrationallowed_nil &  X kstate != Suspen
ded_Isregistrationallowed_Fault) ->  F kstate = Done_Updateaninvoice_Success) & (
kstate = Suspended_Isregistrationallowed_Fault ->  F kstate = Aborted_EndEnrolme
nt_Fail))  is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  kstate = gen_psd_init_state
-> State: 1.2 <-
  kstate = Activated_Isregistrationallowed_nil
-> State: 1.3 <-
  kstate = Activated_Updateaninvoice_nil
-> State: 1.4 <-
  kstate = Suspended_Updateaninvoice_Fault
-> State: 1.5 <-
  kstate = Rollback_LogInvoice_nil
-> State: 1.6 <-
  kstate = Aborted_EndEnrolment_Fail
-- Loop starts here
-> State: 1.7 <-
  kstate = gen_psd_fnl_state
-> State: 1.8 <-
```

(a)



```
-- specification AG (kstate = Activated_ConfirmUserRegistration_nil -> AG kstate != Ab
orted_EndEnrolment_Fail)  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  kstate = gen_psd_init_state
-> State: 1.2 <-
  kstate = Activated_ConfirmUserRegistration_nil
-> State: 1.3 <-
  kstate = Activated_ZohoInvoice_nil
-> State: 1.4 <-
  kstate = Rollback_LogInvoice_nil
-> State: 1.5 <-
  kstate = Aborted_EndEnrolment_Fail
```

(b)

Figure 6.7: Transactional requirement verification output for the course enrolment model

A second stack trace that violated `CE9` (Figure 6.7(b)) exhibited similar behavior. `CE9` requires the `Aborted` control behavior state to be unreachable after `Confirm User Registration`, but the stack trace showed that if the Zoho Invoice services fail after `Confirm User Registration` has completed, the process is able to abort.

To satisfy `CE4` and `CE9`, the process must be revised in a way that allows execution to continue after recovering from faults during invoicing. Therefore, the following changes to the process are proposed:

- The outgoing transition from `Log Invoice` to `End Enrolment` is removed, and replaced with a *Syncreq* message to `Rollback`.
- An activity called `Invoicing Complete` is added, with an incoming transition from `Update Invoice`.
- The origin of the *Success* message sent from `Update Invoice` is changed to `Invoicing Complete`.
- A *Sync* message is sent from `Activated` to `Invoicing Complete`, with a guard con-

dition `Log Invoice.Syncreq`, ensuring it can only be triggered after the `Log Invoice` recovery completes.

These changes mean that once an invoice is logged as a recovery action, the process briefly resumes to record that invoicing has been completed before committing successfully. TL-VIEWS confirms that the revised process satisfies `CE4` and `CE9`, and that the conversation rules and remaining requirements were not compromised by the changes.

6.2.1.3 Web Store Checkout BPaaS Configuration

Our validation scenarios for BPaaS configuration aim to show how our configuration process can handle diverse clients of the same service, while correctly enforcing domain constraints and client transactional requirements. To validate our BPaaS configuration approach, we demonstrate two scenarios using the Web store checkout service as it is presented in Section 5.2. These scenarios show two clients with quite different transactional and configuration requirements, to demonstrate how our configuration process is suitable for highly configurable BPaaS.

Web Store Checkout: Scenario A

The first client is a medium-sized business looking to expand into online sales with a Web store. The business offers physical products only, and does not do pre-orders. Inventory and accounting are to be managed with SaaS, but the client has an existing customer repository that will be used. Payment may be made with Paypal and eWay services.

This client has provided 8 transactional requirements that the checkout service must satisfy to provide the necessary trust to outsource their business operations:

SA1: `Release Product` is necessary prior to `Aborted` after `Initiate Order` with `Physical Product Details`.

SA2: `Place Shipping Order` should always lead to `Done`.

SA3: `Update Accounts` using `SaaSu` is necessary prior to `Done`.

Table 6.4: Transactional requirements for BPaaS Scenario A

ID	Template	Variables	Values
SA1	ControlState Critical	*Control State*	Aborted
		Condition	Release Product
		Scope	Initiate Order.Physical Product Details
SA2	ControlState Trigger	*Control State*	Done
		Condition	Place Shipping Order
		Scope	Global
SA3	ControlState Critical	*Control State*	Done
		Condition	Update Accounts.SaaSu
		Scope	Global
SA4	ControlState Unreachable	*Control State*	Aborted
		Condition	Update Accounts.SaaSu $\vee$ Place Shipping Order $\vee$ Update Inventory.Microguru
		Scope	Global
SA5	ControlState Critical	*Control State*	Done
		Condition	Obtain Shipping Details
		Scope	Global
SA6	Compensate Failure	*Activity*	Process Payment
		Recovery	Reconfirm Payment Information $\vee$ Cancel Order
		Scope	Global
SA7	Retriable Pivot	*Activity*	Process Payment. Paypal
		Scope	Global
SA8	Retriable Pivot	*Activity*	Process Payment. eWay
		Scope	Global

SA4: Once Update Accounts, Place Shipping Order, or Update Inventory have completed, the process cannot abort.

SA5: Obtain Shipping Details is necessary prior to Done.

SA6: Reconfirm Payment Information or Cancel Order must be executed following the failure of Process Payment.

SA7: Process Payment using Paypal cannot be undone.

SA8: Process Payment using eWay cannot be undone.

The template implementations of these requirements are in Table 6.4. The BDD generated from the domain constraints of Figure 5.5 and the configurable activities, resources, and data objects in Table 6.4 indicates that these features can be selected in a valid configuration.

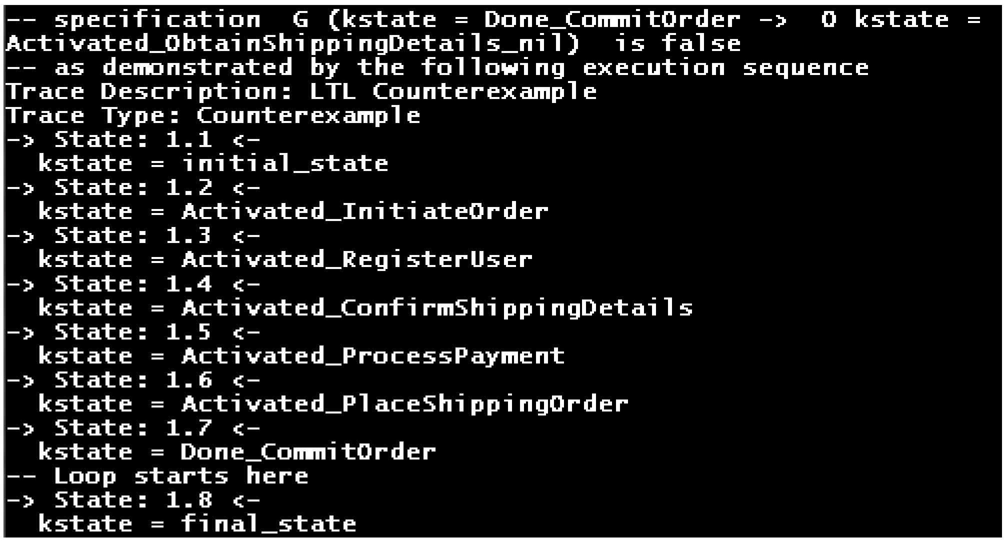

```
-- specification  G (kstate = Done_CommitOrder ->  O kstate =
Activated_ObtainShippingDetails_nil)  is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  kstate = initial_state
-> State: 1.2 <-
  kstate = Activated_InitiateOrder
-> State: 1.3 <-
  kstate = Activated_RegisterUser
-> State: 1.4 <-
  kstate = Activated_ConfirmShippingDetails
-> State: 1.5 <-
  kstate = Activated_ProcessPayment
-> State: 1.6 <-
  kstate = Activated_PlaceShippingOrder
-> State: 1.7 <-
  kstate = Done_CommitOrder
-- Loop starts here
-> State: 1.8 <-
  kstate = final_state
```

Figure 6.8: Violating stack trace produced by NuSMV

In addition to the features specified in these requirements, the client also selects the additional features `AusPost`, `International Shipping` for handling shipping orders, and the `Confirm Shipping Details` activity to streamline the process for recurring customers. BDD analysis confirms that these selections satisfy the domain constraints. The BDD generated at this step is shown in Figure 5.9.

During the activity selection model checking phase, NuSMV identified a Kripke stack trace violating `SA5`, shown in Figure 6.8. From analyzing the stack trace and the BPMN structure in Figure 5.1, it can be determined that the inclusion of `Confirm Shipping Details` is enabling `Obtain Shipping Details` to be bypassed, thereby violating the requirement that it is critical for successful execution. In order to complete this verification step successfully, the client can i) remove `Confirm Shipping Details` from the configuration, or ii) revise the violating requirement to include it. Following either of these revisions, both model checking phases are successful and a configuration solution is identified.

Web Store Checkout: Scenario B

An online music store with a local focus wishes to use the BPaaS to handle the checkout of sales done through their website. The store offers digital downloads, CDs, and merchandise of local artists, with shipping currently restricted to domestic customers. Furthermore, the checkout process must be able to handle pre-order sales of upcoming releases. Payment must be handled through the owner's Paypal account, who also maintains customer details, accounting, and inventory.

This client provides the following transactional requirements for the BPaaS:

SB1: For sales that are not pre-orders, `Transfer File` or `Place Shipping Order` must be successful before the process commits.

SB2: A receipt must be sent to customers following every successful sale.

SB3: Customers must be able to re-enter their `Paypal` details and retry if payment is initially unsuccessful.

SB4: If `Process Payment` is unsuccessful, either it is retried with new details, or the hold on the product or pre-order is released before cancelling the sale.

SB5: `Process Payment` using `Paypal` cannot be undone.

Table 6.5 shows these transactional requirements formalized with templates. BDD analysis conducted of the configurable features included in this requirement set confirm that they are valid with respect to the domain constraints of the service provider. In addition, the client selects the configurable activities, resources, and data objects shown in Table 6.6.

Once these features are added, the constructed BDD is not satisfiable. JDD automatically removes redundant paths from BDDs, so the output produced is an empty BDD only consisting of a transition to the `0` node. Therefore, this specific selection of features violates the domain constraints.

From analyzing the feature model in Figure 5.5, we can see that the client has omitted `Validate Login`, but has selected several features that are directly dependent on it. The client plans to handle all customer details management internally, but the BPaaS requires the security of a login procedure in order to conduct sales. In order to satisfy this constraint, the client removes integration with a `Private Customer Repository` and the `Store Payment Details`

Table 6.5: Transactional requirements for BPaaS Scenario B

ID	Template	Variables	Values
SB1	ControlState Critical	*Control State*	Done
		Condition	Transfer File $\vee$ Place Shipping Order
		Scope	Initiate Order
SB2	ControlState Critical	*Control State*	Done
		Condition	Email Receipt
		Scope	Global
SB3	ControlState Reachable	*Control State*	Done
		Condition	Reconfirm Payment Information
		Scope	Global
SB4	Compensate Failure	*Activity*	Process Payment.Paypal
		Recovery	Release Order $\vee$ Release Product $\vee$ Reconfirm Payment Information
		Scope	Global
SB5	Retriable Pivot	*Activity*	Process Payment.Paypal
		Scope	Global

Table 6.6: Additional feature selections for BPaaS Scenario B

Activities	Initiate Pre-Order, Store Payment Details
Resources	Private Inventory System, Private Accounting System, Private Customer Repository, Provider Storage, AusPost
Data Objects	Digital Product Details, Physical Product Details

activity from the configuration. This activity and resource will be managed completely internally, due to a manageable level of anticipated customers. BPaaS running and provisioning costs may also be reduced if a smaller number of features are included.

After removing those two features from the configuration, BDD analysis passed successfully. Next, the configuration was verified against the transactional requirements formalized in Table 6.5 across our two model checking phases. Both phases were successful, confirming that the configured BPaaS model satisfies the domain constraints and transactional requirement set.

6.2.2 Performance Analysis

For each use of model checking in the TL-VIEWS architecture, a series of experiments were conducted to verify the feasibility of verification with large and complex models. We evaluate the effectiveness of the state space reduction algorithms employed in each verification approach by comparing model checking time with and without their use.

All experiments were run on an Intel Core i7 3.40GHz 4GB RAM system. The execution times are recorded in seconds, and are mean average values taken from 10 executions.

6.2.2.1 Conversation Rule Checking

To evaluate the effectiveness of *CKSR*, we employ a test model suite of six designs of increasing size and complexity, containing 100 to 1,000 operational behavior states $\mathcal{S}^o$. These were created from an initial model of 100 operational behavior states, by adding a further 100 states with identical transitions and inter-behavior messages for each increasingly complex model.

Table 6.7 shows a comparison between the conversation rule verification time of our model suite, with and without Kripke structure reduction. For the unreduced tests, the complete control and operational behavior models were transformed into the input language of NuSMV with no reductions to the state space. Our results show that even while considering the additional overhead of our algorithm, a net verification performance benefit is created in our approach. The performance benefit of our approach increases as the model under testing becomes larger, from a 76.01% reduction in verification time (100 $\mathcal{S}^o$ states), to a 98.79% reduction (1000 $\mathcal{S}^o$ states). These results indicate that our approach enables conversation rule checking at a speed that does not impair or inconvenience the design process, even for very large models. This is especially vital when multiple design revisions and model checking executions are required.

Table 6.7: Verification time (in seconds) of conversation rules with and without Kripke structure reduction

	With Kripke Reduction			
$\mathcal{S}^o$ **States**	**CKSR**	**NuSMV**	**Total**	**Unreduced**
100	0.003	0.062	**0.065**	**0.271**
200	0.004	0.093	**0.097**	**0.816**
300	0.006	0.140	**0.146**	**1.860**
400	0.006	0.192	**0.198**	**3.017**
500	0.007	0.227	**0.234**	**6.124**
1,000	0.015	0.468	**0.483**	**39.865**

Table 6.8: NuSMV execution times for individual templates

Template	Cardinality	NuSMV (ms)
`CompensateFailure`	1:1	7.3
	Many:1	7.0
`CompensateSuccess`	1:1	7.8
	Many:1	7.6
`Alternative`	1:1	7.4
	Many:1	7.4
`NonRetriable`	-	7.1
`RetriablePivot`	-	7.2
`NonRetriablePivot`	-	7.5
`ControlStateCritical`	-	7.1
`ControlStateTrigger`	-	7.1
`ControlStateReachable`	-	6.5
`ControlStateUnreachable`	-	6.4
`Compensation`	-	7.4
`ConditionalCompensation`	-	7.3

6.2.2.2 Verification Against Temporal Logic Templates

To evaluate our approach for verification against transactional requirements formalized using our templates, we first evaluate the performance of each temporal logic template individually. Table 6.8 shows the verification time for minimal Kripke structures, as generated by TKSR, against one requirement specified by each template. The *Scope* variable of every template that requires it was set to *Global*, in each test in order to obtain an even comparison. These results indicate that the templates with the greatest model checking performance demand are `CompensateSuccess`, `Alternative`, `NonRetriablePivot`, and `Compensation`. Therefore, our evaluation will utilize these templates in an even ratio.

We employ the same test suite of service-oriented process models used for conversation rule verification, but use each model in several tests with requirement sets of increasing size. The aim of this is to test the impact of our state space reduction measures when models *and* transactional requirement sets become large and complex.

The results of our test suite evaluation against transactional requirement sets are shown in Table 6.9. Our approach was able to reduce verification time, in a range from 79.90% (100 $\mathcal{S}^o$ states with 100 requirements), to 98.63% (1,000 $\mathcal{S}^o$ states with 500 requirements). These results also show that verification time for complex models against large sets of requirements can reach un-

Table 6.9: Verification time (in seconds) of temporal logic templates with and without Kripke structure reduction

$\mathcal{S}^o$ States	Requirements	With Kripke Reduction			Unreduced
		TKSR	**NuSMV**	**Total**	
100	25	0.005	0.230	**0.235**	**1.334**
	50	0.006	0.542	**0.548**	**2.771**
	100	0.007	1.137	**1.144**	**5.692**
200	100	0.010	1.401	**1.411**	**11.755**
	150	0.012	2.175	**2.187**	**17.802**
	200	0.013	3.036	**3.049**	**23.951**
300	200	0.020	3.598	**3.618**	**42.249**
	250	0.020	4.635	**4.655**	**52.633**
	300	0.021	5.739	**5.760**	**63.071**
400	300	0.026	6.697	**6.723**	**94.326**
	350	0.026	7.982	**8.008**	**110.721**
	400	0.027	9.504	**9.531**	**126.530**
500	400	0.035	10.524	**10.559**	**222.406**
	450	0.036	12.123	**12.159**	**249.973**
	500	0.036	13.755	**13.791**	**278.924**
1,000	500	0.066	24.780	**24.846**	**1,807.452**
	750	0.068	39.848	**39.916**	**2,743.530**
	1,000	0.068	56.893	**56.961**	**3,620.442**

reasonable lengths without our reduction measures. For example, our tests verifying an unreduced model with 1,000 operational behavior states against 500, 750, and 1,000 requirements took over 30 minutes, 45 minutes, and one hour respectively to complete. In contrast, our approach reduced the total verification time to under one minute for all three requirement sets.

6.2.2.3 Configuration Business Process as a Service Verification

We perform two series of performance verification tests for our BPaaS configuration process. The first set of tests aims to validate the performance benefit on our approach using a straightforward model with smaller requirements sets. The configured BPaaS model contains a total of 100 activities (30 configurable), and a combined total of 30 configurable resources and data objects. This model is a BPaaS that has already been configured through feature selection, rather than a configurable BPaaS with 100 total *possible* activities. We developed this model by extending the BPaaS of Figure 5.1 with pseudo-states. Using this model, we compare our multi-step model checking approach against a single step model checking approach that does not apply any state space reduction measures. For the verifying without state space reduction, the NuSMV input was manually

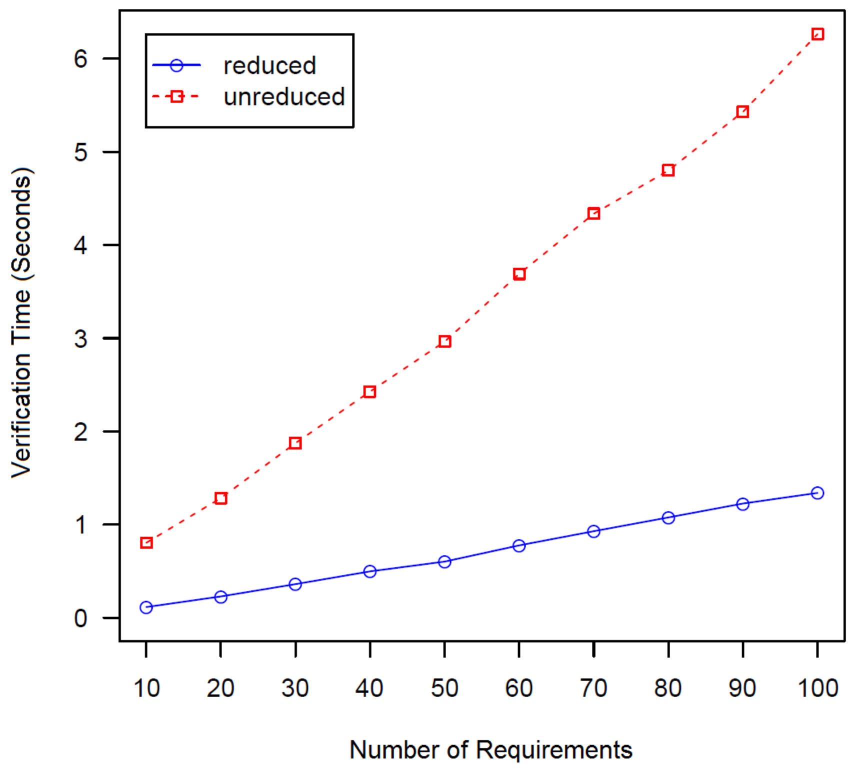


Figure 6.9: Verification times during configuration with and without reduction for 10 to 100 requirements

written based on a complete implementation of the model.

We performed 10 tests, verifying the configured BPaaS model given sets of requirements from sizes 10 to 100. The requirements used the same four templates that were used in the previous section. Figure 6.9 plots the verification times of both the reduced and unreduced models. In the case of unreduced, the verification time is determined from the sum of both model checking phases, and the applications of the TKSR algorithm before each phase. These results indicate that our approach provides a significant performance benefit in verification time, and that the benefit becomes greater as the requirements sets become larger.

The second set of tests verifies the model checking performance of our BPaaS configuration approach with large and complex models. This suite contains the model from the first set of performance test, but extends it to create a further 3 models that are 200, 300, and 500 activities in size. Like the model used in our first test, these descriptions apply to a configured BPaaS. These models are too large to display in a readable manner, but to provide an indication of their size

Table 6.10: Details of the configurable BPaaS test suite for performance analysis

Model ID	Total Activities	Configurable Activities	Resources & Data Objects
`B1`	100	30	30
`B2`	200	60	60
`B3`	300	90	90
`B4`	500	150	150

Table 6.11: Verification time (in seconds) of increasingly complex configuration scenarios

		NuSMV with reduction					NuSMV
Model	**Reqs**	**Act. TKSR**	**Act. NuSMV**	**RDO TKSR**	**RDO NuSMV**	**Total**	**unreduced**
`B1`	25	0.007	0.141	0.006	0.107	**0.261**	**1.520**
	50	0.007	0.308	0.008	0.280	**0.603**	**2.963**
	100	0.008	0.725	0.008	0.599	**1.340**	**6.264**
`B2`	100	0.015	0.835	0.015	0.715	**1.580**	**13.512**
	150	0.022	1.324	0.017	0.963	**2.326**	**20.164**
	200	0.023	1.921	0.021	1.591	**3.556**	**28.365**
`B3`	200	0.027	2.116	0.022	1.782	**3.947**	**46.538**
	250	0.027	2.775	0.022	2.290	**5.114**	**60.972**
	300	0.030	3.468	0.026	2.788	**6.312**	**73.553**
`B4`	300	0.053	4.246	0.042	3.391	**7.732**	**191.103**
	400	0.053	5.861	0.045	4.636	**10.595**	**256.413**
	500	0.057	7.643	0.047	5.963	**13.710**	**323.896**

and complexity, Table 6.10 shows the total number of activities, selected configurable activities, resources, and data objects of each model.

For each configuration, three tests were conducted with requirement sets of increasing size (from 25 to 500), to compare our approach against a model checking implementation with no reduction measures. Unreduced configurations were achieved by manually translating the BPaaS configuration model, complete with BPMN, control behavior model, and inter-behavior messages, into the input language of NuSMV. These tests aim to demonstrate the effectiveness of our approach as both the model complexity and transactional requirements increase.

The performance of our approach is determined from the sum of the Kripke structure reduction algorithm (TKSR) and NuSMV execution time from both the Activity Selection (Act. NuSMV) and Resources and data Objects Selection (RDO NuSMV) phases. The unreduced model checking was performed by verifying all transactional requirements in one phase against a complete representation of the configured BPaaS in the input language of NuSMV. All requirements in our tests are successful.

The results of our experiments shown in Table 6.11 indicate that the model checking time of unreduced representations of the configured BPaaS can reach infeasible lengths for large models. This is a greater concern considering that our configuration approach applies model checking iteratively against reconfigured process models until a solution is reached. Furthermore, when verifying a large BPaaS configuration, such as the last few models in Table 6.11, clients may wish to verify against a large requirement set incrementally. Considering that the verification times for this model reaches several minutes in length for large requirement sets, this creates an explosion in the total time of the configuration process. In contrast, our approach is shown to provide a substantial performance benefit for large BPaaS configurations, and enable manageable model checking time. The reduction in model checking time shown in our experiments ranges from 78.61%, when verifying our 100 activity configuration against 100 requirements, to 95.95%, from our 500 activity configuration against 300 requirements.

6.3 Summary

The contributions proposed in Chapters 3-5 have been implemented in a prototype tool called TL-VIEWS. This tool enables developers to model service-oriented process at design-time, perform verification against conversation rules, and verify against their own transactional requirements using our template set. TL-VIEWS also contains the functionality to configure BPaaS models to satisfy domain constraint and transactional requirements, as described in Chapter 5. We utilize the ArgoUML modelling tool and NuSMV model checker to implement our proposed configuration and verification approaches.

To validate our contributions, we demonstrate four verification scenarios for service-oriented processes at design-time, and two configuration scenarios for BPaaS. These cases use real-world services, and demonstrate the ability of our approach to identify violations of transactional requirements or domain constraints. Our validation scenarios showed how design-time verification against conversation rules and transactional requirements can identify transactional behavior issues that developers need to address. The BPaaS configuration scenarios we employed showed the effectiveness of our configuration process in handling diverse clients of the same configurable service, while also ensuring domain constraints and transactional requirements.

Finally, we perform an extensive performance evaluation for each of these main features of TL-VIEWS, as they use model checking, an exhaustive verification method susceptible to state-space explosion. These tests were performed using two test suites of models up to 1,000 activities in size, with sets of up to 1,000 temporal logic properties. For each use of model checking, our state-space reduction measures were shown to have a substantial impact on verification time, reducing it by 78.61% to 98.63%. Therefore, our state space reduction measures are able to make exhaustive verification and configuration of even highly complex models feasible.

CHAPTER 7

Conclusion

With the increased use of Web services and cloud services over the last decade, service-oriented processes have become an attractive means for fast and low-risk implementation of business operations. We propose three major contributions in this area: two for verification of transactional behavior at design-time, and one for managing configuration of BPaaS. Firstly, we propose a design-time method for verifying service-oriented processes against a set of conversation rules, which can identify application-independent transactional issues such as invalid termination and deadlocking scenarios. Then, we enable verification against complex and varied transactional requirements drawn from business logic, through temporal logic templates. Finally, we develop a BPaaS configuration approach that enables activity, resource, and data object configuration while preserving domain constraints and transactional requirements drawn from the service client.

All three of these approaches employ state space reduction measures to improve the feasibility and performance of verification when considering large and complex models and property sets. Our validation scenarios and performance tests show that are methods are effective in identifying design or requirement conformance issues, and capable of handling very large verification problems in reasonable time.

Papers of each of these contributions - conversation rule checking [22], temporal logic templates [23], and BPaaS configuration [25] - have been published in high quality conference proceedings. Our prototype tool TL-VIEWS was also accepted as a formal demonstration [24] at ICSOC 2014. We also currently have two articles under review for top quality journals. To conclude, we provide a short summary of each of our main contributions, and discuss future directions to take.

7.1 Summary

To summarize, we revisit the three goals stated in Chapter 1.2 and discuss how our work accomplished each.

7.1.1 Conversation Rule Checking for Well-Formed Transactional Behavior

The goal behind this work was to uncover and resolve issues in the transactional behavior of service-oriented processes at design-time. To do this, we developed a verification approach using a statechart-based model that separates process behavior into *control* and *operational* behavior models. The benefit of this model is that it provides separate views of the functional and transactional behavior of the process, allowing each to be designed and refined individually by relevant domain experts.

Expressive transactional behavior can be defined on this model as inter-behavior messages. These allow the behavior models to pass instructions or notifications, and remain aware of each other's status. These inter-behavior conversations can be analyzed for transactional issues in the process design, such as deadlock or inconsistency between the behavior models. We defined a set of *conversation rules* to identify the transactional behavior design issues that lead to these events.

To verify service-oriented processes against our conversation rule set, we employ temporal logic and model checking. However, as model checking is sensitive to state-space explosion, we advance an algorithm called *CKSR* to reduce the model state space before verification. This algorithm produces a minimal Kripke structure for verifying inter-behavior conversations, which can be used as model checking input along with temporal logic representations of our conversation rule set.

Our prototype tool TL-VIEWS enables users to model service-oriented processes as interacting control and operational behaviors, and verify them against the conversation rules using with model checking. TL-VIEWS has a graphical user interface that extends the UML modeling tool ArgoUML, while enabling model checking with the NuSMV model checker. Two validation scenarios were used to demonstrate the effectiveness of our approach in detecting transactional behavior issues. We also performed an extensive performance evaluation of our verification approach with large and complex models. The largest model in our test suit contains 1,000 operational behavior states, and our state-space reduction algorithm was able to reduce verification time by 98.79% when compared to an unreduced model.

7.1.2 Application-Dependent Transactional Requirement Verification

Our second goal was to enable complex application-dependent transactional requirements to be formalized for verification in an expressive but easy-to-use manner. We adapt temporal logic patterns for transactional requirement specification, and propose a set of *temporal logic templates*. These templates are skeleton structures of temporal logic properties expressing various transactional requirements, that can be implemented and adjusted by assigning simple variables. By assigning variables, these templates allow developers to formalize complex and varied transactional requirements in temporal logic, without requiring expertise in the language. Our set of templates were developed to verify service-oriented processes expressed using the control and operational model, which allows for modelling complex transactional behavior.

We identify common or useful transactional requirements from the service-oriented process literature, and use them as the basis of our template set. Two categories of templates are produced: for component-level and process-level transactional requirements. Component-level templates specify requirements for the transactional management of specific components in the process, such as necessary recovery operations, alternatives, and whether retrial is safe upon failure. In contrast, requirements implemented by process-level templates apply to the transactional behavior of the entire process, such as compensating completed process executions, or valid conditions for aborting or committing the process.

Given a set of variables, such as applicable components, scope, or recovery conditions, these templates automatically map the transactional requirement to formal temporal logic properties. The use of templates greatly reduces human error and effort when compared with manual specification, while still allowing complex and diverse requirements to be formalized. Model checking can again be applied to ensure the requirements are satisfied at design-time. We advance a second algorithm for state-space reduction. Unlike the algorithm used during conversation checking, this reduction removes all states unnecessary for verifying an implemented set of requirements, without compromising the behavior of the model under verification.

TL-VIEWS contains an interface for specifying transactional requirements using templates, and automatically employs our state space reduction and verification. Our validation scenarios demonstrate how the templates can be used to formalize complex application-dependent requirements and identify violating behavior in the design. We perform a series of tests to verify the performance of our verification approach with large and complex models and transactional requirement sets. The results show verification time reductions between 79.90% and 98.63%.

7.1.3 Configuration of Transactional Business Process as a Service

Our third goal was to contribute a BPaaS configuration method that enables clients to provide transactional requirements that the service must satisfy, while also considering domain constraints over valid configuration. This configuration approach extends work related to BPaaS configuration by enabling complex requirements from the client to be verified, while also considering BPaaS configuration from multiple perspectives. We aim to enable BPaaS configuration of activities, resources, and data objects to provide greater flexibility for clients. We employ BPMN with as an expressive BPaaS model that includes configurable activities, resources, and data objects. Combined with our separation of behaviors modelling approach, we also include detailed transactional behaviour in the model. Feature models are used to model domain constraints over configurable features in the BPaaS. Our temporal logic templates are employed for formalizing application-dependent transactional requirements.

These models can be combined into a configuration approach that satisfies our goal using formal methods, as they can be used to verify large models against complex properties. BDD analysis is used to ensure that the configurable features included in the transactional requirements and the client's selections do not violate the constraints expressed in the feature model. As in our design-time verification approach, model checking is used to ensure transactional requirements expressed in temporal logic are satisfied. However, due to the impact that resource and data object configuration has on the model state-space, we divide the model checking problem into two phases. The first phase focuses verifies the BPaaS configuration according to selected activities, while the second only verifies requirements that include specific resources and data objects. This two-phase approach allows for both the model and transactional requirement set to be reduced in complexity in each phase.

Several modules in TL-VIEWS are dedicated to implementing this configuration approach. Using a Web store checkout BPaaS, we demonstrate how our configuration process can be used to handle diverse requirements for features and transactional behavior with two validation scenarios. We also perform an series of performance analysis tests to determine the effectiveness of our state-space reduction methods, and the feasibility of our approach with large and complex BPaaS. Our analysis uses four models of increasing size with several large requirement sets for each to be verified against. The results indicate a verification time reduction between 78.61% and 95.95%. With our largest model (500 activities with 300 configurable features) verified against our largest transactional requirements set (500 requirements), each model checking phase was still able to complete in under 10 seconds.

7.2 Future Directions

The contributions behind these three goals have each received publication in high quality conferences in recent years [22, 23, 24, 25], as well as submissions to high quality journals. However, at the current stage of this project, some limitations remain that provide an opportunity for future work. Below, we highlight three avenues for future work and discuss how they could create further contributions to the field.

7.2.1 Diagnosing Conversation Rule and Transactional Requirement Violations

Currently, in all three main functions of TL-VIEWS that use model checking, the output of violations presented to the user i) the property violated, and ii) a violating stack-trace generated by NuSMV. This is helpful to the user as it demonstrates the behavior that needs to be revised.

However, there are limitations to this approach that could be improved:

- The stack trace contains states from the Kripke structure used as model checking input, which the user does not have access to. While the Kripke structure states have states from the model as atomic propositions, this requires extra manual interpretation from the user.
- For some CTL properties, NuSMV is unable to produce stack trace violations. For example, when a requirement using the `Control State Reachable` template is not satisfied, it is because no reachable state satisfies the property. In these cases, since there is not violating stack trace, the user has to determine manually how the property is not satisfied.
- The stack trace violations are still completely up to the user to interpret. Given the complexity of some temporal logic properties in our approach, it can be a non-trivial task to determine exactly what revisions need to be made to the model.

Other approaches using temporal logic templates or patterns use the knowledge of property structures to help diagnose model checking violations. For example, Elgammal et al. [54] construct *Current Reality Trees* (CRTs) [48] to diagnose violations of properties constructed from a set of temporal logic patterns. As the structures of temporal logic properties are pre-defined in our approach, mechanisms for explaining why violations occured could be explored. Futhermore, common factors across all models, such as control behavior states, could be used to automatically diagnose violations in high detail (such as indicating when an inter-behavior message is missing). This information could be represented to the user as suggestions, or even in a visual manner on the model itself.

Pursuing this research direction would greatly increase the usability of our verification and configuration approaches. Users will not need any experience with model checking and formal methods to use our tool as all complexities will be hidden and interpreted automatically. It would also increase the feasibility of our approach for large models, where cause of the violations of complex temporal logic properties can be difficult to manually determine.

7.2.2 Preserving Transactional Requirements During Dynamic Configuration

At present, our BPaaS configuration approach is a static process that is conducted before the client provisions the service for use. As cloud services aim to respond dynamically to changes in workload and environment [7, 16], another goal for BPaaS could be to adapt to these events. Furthermore, existing work in SaaS management has addressed services that *evolve* over time, in order to handle new clients or environmental changes [92, 101, 131]. Changes to third party services utilized by the BPaaS may also need to be addressed dynamically, such as outages, QoS variations, or updates to the service interface or behaviour [3, 118].

As future work, methods could be explored to make dynamic changes to the BPaaS in a way that does not violate requirements. This presents two immediate challenges:

- The method to ensure transactional requirements remain satisfied must be very time efficient in order to be used dynamically at runtime. As we have shown, transactional requirements can be very complex when formalized, but through appropriate reduction measures, can be verified in a reasonable time frame.
- Changes to the BPaaS structure or resources, such as replacing resources that are not responsive, may impact one or many clients [92]. The requirements of all clients, rather than single clients, may need to be addressed.

Addressing all these challenges would change our configuration process to a comprehensive BPaaS management framework capable of responding to changes in the environment and managing client transactional requirements. This added functionality would also reduce the number of unpredicted faults of component services and prevent suspensions or failures of the BPaaS execution. Existing approaches that manage cloud service evolution focus on requirements such as process structure fairness [101], and preserving elasticity [118] and multi-tenancy [110]. To the best of our knowledge, complex application-dependent properties such as transactional requirements are yet to received attention in this area.

7.2.3 Business Process as a Service Configuration Framework

The implementation of our BPaaS configuration process in TL-VIEWS validates that BDD analysis and model checking can be applied as formal methods during configuration. Currently, TL-VIEWS applies these formal methods to configure basic workflow skeletons. Our validation scenarios and performance tests have shown the feasibility of our approach as a configuration method for BPaaS.

The next step is to develop an appropriate interface for clients that hides the details and complexity of the service aside from the transactional requirement specification and feature selection. The modules of the architecture for managing configuration would need to be expanded in order to configure implemented business processes in an executable language, while preserving basic syntactical correctness. After a configuration solution has been found, the framework would need to apply the necessary configurations to an implemented service-oriented process, such as a WS-BPEL implementation.

As BPaaS is still an emerging technology, a framework such as this, especially one that enables clients to ensure complex transactional requirements, would be a significant contribution to the field. The work presented in this thesis, in addition to the future directions discussed in this section, will hopefully provide avenues for addressing a host of the opportunities present in the early stages of BPaaS research.

REFERENCES

[1] R. Accorsi. Business Process as a Service: Chances for Remote Auditing. In *The 35th Annual Computer Software and Applications Conference Workshop*, pages 398–403. IEEE, 2011.

[2] E. Al-Masri and Q. H. Mahmoud. Investigating Web Services on the World Wide Web. In *Proceedings of the 17th International Conference on World Wide Web*, pages 795–804. ACM, 2008.

[3] A. Alhosban, K. Hashmi, Z. Malik, B. Medjahed, and S. Benbernou. Bottom-Up Fault Management in Service-Based Systems. *Transactions on Internet Technology*, 15(2):7, 2015.

[4] G. Alonso, D. Agrawal, M. Kamath, R. Günthör, and C. Mohan. Advanced Transaction Models in Workflow Contexts. In *Proceedings of the 12th International Conference on Data Engineering*, pages 574–581. IEEE, 1996.

[5] P. Andrew, J. Conard, and S. Woodgate. *Presenting Windows Workflow Foundation*. Sams, Indianapolis, IN, USA, 2005.

[6] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, and S. Thatte. *Business Process Execution Language for Web Services*. Packt Publishing, 2003.

[7] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, and I. Stoica. A View of Cloud Computing. *Communications of the ACM*, 53(4):50–58, 2010.

[8] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT press, 2008.

[9] A. Banerjee. A Formal Model for Multi-Tenant Software-as-a-Service in Cloud Computing. In *Proceedings of the 5th ACM COMPUTE Conference: Intelligent & Scalable System Technologies*, page 18. ACM, 2012.

[10] L. Baresi and S. Guinea. Self-Supervising BPEL Processes. *IEEE Transactions on Software Engineering*, 37(2):247–263, 2011.

[11] A. Basu and R. W. Blanning. A Formal Approach to Workflow Analysis. *Information Systems Research*, 11(1):17–36, 2000.

[12] N. Ben Lakhal, T. Kobayashi, and H. Yokota. FENECIA: Failure Endurable Nested-transaction Based Execution of Composite Web Services with Incorporated State Analysis. *The International Journal on Very Large Databases*, 18(1):1–56, 2009.

[13] D. Benavides, S. Segura, and A. Ruiz-Cortés. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Information Systems*, 35(6):615–636, 2010.

[14] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi. *UPPAAL A Tool Suite for Automatic Verification of Real-Time Systems*. Springer, 1996.

[15] P. A. Bernstein, D. W. Shipman, and W. S. Wong. Formal Aspects of Serializability in Database Concurrency Control. *IEEE Transactions on Software Engineering*, (3):203–216, 1979.

[16] C. Bezemer and A. Zaidman. Multi-Tenant SaaS Applications: Maintenance Dream or Nightmare? In *Proceedings of the Joint ERCIM Workshop on Software Evolution and International Workshop on Principles of Software Evolution*, pages 88–92. ACM, 2010.

[17] S. Bhiri, C. Godart, and O. Perrin. Transactional Patterns for Reliable Web Services Compositions. In *Proceedings of the 6th International Conference on Web Engineering*, pages 137–144. ACM, 2006.

[18] S. Bhiri, O. Perrin, and C. Godart. Ensuring Required Failure Atomicity of Composite Web Services. In *Proceedings of the 14th International World Wide Web Conference*, pages 138–147. ACM, 2005.

[19] G. Böckle, F. J. van der Linden, and K. Pohl. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer Science & Business Media, 2005.

[20] M. L. Bote-Lorenzo, Y. A. Dimitriadis, and E. Gómez-Sánchez. Grid Characteristics and Uses: a Grid Definition. In *Grid Computing*, pages 291–298. Springer, 2004.

[21] S. Bouchenak, G. Chockler, H. Chockler, G. Gheorghe, N. Santos, and A. Shraer. Verifying cloud services: present and future. *SIGOPS Operating Systems Review*, 47(2):6–19, 2013.

[22] S. Bourne, C. Szabo, and Q.Z. Sheng. Ensuring Well-Formed Conversations Between Control and Operational Behaviors of Web Services. In *Proceedings of the 10th International Conference on Service-Oriented Computing*, pages 507–515. Springer, 2012.

[23] S. Bourne, C. Szabo, and Q.Z. Sheng. Verifying Transactional Requirements of Web Service Compositions using Temporal Logic Templates. In *Proceedings of the 14th International Conference on Web Information System Engineering*, pages 243–256. Springer, 2013.

[24] S. Bourne, C. Szabo, and Q.Z. Sheng. TL-VIEWS: A Tool for Temporal Logic Verification of Transactional Behavior of Web Service Compositions. In *Proceedings of the 12th International Conference on Service-Oriented Computing*, pages 418–422. Springer, 2014.

[25] S. Bourne, C. Szabo, and Q.Z. Sheng. Managing Configurable Business Process as a Service to Satisfy Client Transactional Requirements. In *Proceedings of the 11th International Conference on Services Computing*, pages 154–161. IEEE, 2015.

[26] M. Bozkurt, M. Harman, and Y. Hassoun. Testing and Verification in Service-Oriented Architecture: a Survey. *Software Testing, Verification and Reliability*, 23(4):261–313, 2013.

[27] F. Cabrera, G. Copeland, M. Feingold, R. W. Freund, T. Freund, S. Joyce, J. Klein, D. Langworthy, M. Little, and F. Leymann. Web Services Business Activity Framework (WS-BusinessActivity). *IBM Web Service Transactions Specifications*, 2005.

[28] F. Cabrera, G. Copeland, M. Feingold, T. Freund, J. Johnson, S. Joyce, C. Kaler, J. Klein, and D. Langworthy. Web Services Atomic Transaction (WS-AtomicTransaction). *MSDN Library*, 2005.

[29] J. Cao, J. Luo, S. Zhang, X. Zheng, B. Liu, G. Zhu, and B. Zhang. A Context-Aware Recovery Mechanism for Web Services Business Transaction. In *Proceedings of the 9th International Conference on Services Computing*, pages 352–359, 2012.

[30] Z. Cao, X. Zhang, W. Zhang, X. Xie, J. Shi, and H. Xu. A Context-Aware Adaptive Web Service Composition Framework. In *Proceedings of the International Conference on Computational Intelligence & Communication Technology*, pages 62–66. IEEE, 2015.

[31] L. Capra and W. Cazzola. Self-Evolving Petri Nets. *Journal of Universal Computer Science*, 13(13):2002–2034, 2007.

[32] Y. Cardinale, J. El Haddad, M. Manouvrier, and M. Rukoz. Transactional-aware Web Service Composition: A Survey. *IGI Global-Advances in Knowledge Management Book Series*, 2011.

[33] Y. Cardinale and M. Rukoz. A Framework for Reliable Execution of Transactional Composite Web Services. In *Proceedings of the International Conference on Management of Emergent Digital Ecosystems*, pages 129–136. ACM, 2011.

[34] F. Casati, S. Ilnicki, L. Jin, V. Krishnamoorthy, and M. Shan. Adaptive and Dynamic Service Composition in eFlow. In *Proceedings of the 12th International Conference on Advanced Information Systems Engineering*, pages 13–31. Springer, 2000.

[35] A. Ceponkus, P. Furniss, A. Green, S. Dalal, and M. Little. Business Transaction Protocol. *Change*, 2002.

[36] L. Chen and A. Avizienis. N-version Programming: A Fault-tolerance Approach to Reliability of Software Operation. In *Proceedings of the 8th IEEE International Symposium on Fault-Tolerant Computing*, pages 3–9, 1978.

[37] F. Chong and G. Carraro. Architecture Atrategies for Catching the Long Tail. *MSDN Library*, pages 9–10, 2006.

[38] H. Chong, J. S. Wong, and X. Wang. An Explanatory Case Study on Cloud Computing Applications in the Built Environment. *Automation in Construction*, 44:152–162, 2014.

[39] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL) 1.1, 2001.

[40] P. K Chrysanthis and K. Ramamritham. ACTA: A Framework for Specifying and Reasoning about Transaction Structure and Behavior. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, volume 19, pages 194–203. ACM, 1990.

[41] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An Opensource Tool for Symbolic Model Checking. In *Computer Aided Verification*, pages 241–268. Springer, 2002.

[42] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a New Symbolic Model Checker. *International Journal on Software Tools for Technology Transfer*, 2(4):410–425, 2000.

[43] E. Clarke. Model Checking. In *Foundations of Software Technology and Theoretical Computer Science*, pages 54–56. Springer, 1997.

[44] E. Clarke, O. Grumberg, and K. Hamaguchi. Another Look at LTL Model Checking. In *Computer Aided Verification*, pages 415–427. Springer, 1994.

[45] E. Clarke, D. Kroening, and F. Lerda. A Tool for Checking ANSI-C Programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176. Springer, 2004.

[46] S. Cranen, J. F. Groote, J. J. A. Keiren, F. P. M. Stappers, E. P. de Vink, W. Wesselink, and T. A. C. Willemse. An Overview of the mCRL2 Toolset and its Recent Advances. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 199–213. Springer, 2013.

[47] U. Dayal, M. Hsu, and R. Ladin. Business Process Coordination: State of the Art, Trends, and Open Issues. In *Proceedings of the 27th Very Large Databases Conference*, volume 1, pages 3–13, 2001.

[48] H. W. Dettmer. *Goldratt's Theory of Constraints: a Systems Approach to Continuous Improvement*. ASQ Quality Press, 1997.

[49] T. Dillon, C. Wu, and E. Chang. Cloud Computing: Issues and Challenges. In *The 24th IEEE International Conference on Advanced Information Networking and Applications*, pages 27–33, 2010.

[50] D. Dranidis, E. Ramollari, and D. Kourtesis. Run-Time Verification of Behavioural Conformance for Conversational Web Services. In *Proceedings of the 7th European Conference on Web Services*, pages 139–147. IEEE, 2009.

[51] E. Duipmans and L. F. Pires. Business Process Management in the Cloud: Business Process as a Service (BPaaS). Technical report, University of Twente, 2012.

[52] Matthew B Dwyer, George S Avrunin, and James C Corbett. Patterns in Property Specifications for Finite-State Verification. In *Software Engineering, 1999. Proceedings of the 1999 International Conference on*, pages 411–420. IEEE, 1999.

[53] J. El Hadad, M. Manouvrier, and M. Rukoz. TQoS: Transactional and QoS-Aware Selection Algorithm for Automatic Web Service Composition. *IEEE Transactions on Services Computing*, 3(1):73–85, 2010.

[54] Amal Elgammal, Oktay Turetken, Willem-Jan van den Heuvel, and Mike Papazoglou. Root-Cause Analysis of Design-Time Compliance Violations on the Basis of Property Patterns. In *Proceedings of the 8th International Conference on Service-Oriented Computing*, pages 17–31. Springer, 2010.

[55] A. Elmagarmid. *Transaction Models for Advanced Database Applications*. Morgan Kaufmann Publishers Inc., 1991.

[56] E. A. Emerson. Temporal and Modal Logic. *Handbook of Theoretical Computer Science*, 2:995–1072, 1990.

[57] A. Erradi, P. Maheshwari, and V. Tosic. Recovery Policies for Enhancing Web Services Reliability. In *Proceedings of the International Conference on Web Services*, pages 189–196. IEEE, 2006.

[58] O. Ezenwoye and S. M. Sadjadi. TRAP/BPEL-A Framework for Dynamic Adaptation of Composite Services. In *WEBIST*, volume 1, pages 216–221, 2007.

[59] A. Fantechi, S. Gnesi, A. Lapadula, F. Mazzanti, R. Pugliese, and F. Tiezzi. A Logical Verification Methodology for Service-Oriented Computing. *ACM Transactions on Software Engineering and Methodology*, 21(3):16, 2012.

[60] R. T. Fielding. *Architectural Styles and the Design of Network-Based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[61] J. P. Friese, T.and Müller and B. Freisleben. Self-healing Execution of Business Processes Based on a Peer-to-Peer Service Architecture. In *Systems Aspects in Organic and Pervasive Computing*, pages 108–123. Springer, 2005.

[62] W. Gaaloul, S. Bhiri, and M. Rouached. Event-based Design and Runtime Verification of Composite Service Transactional Behavior. *IEEE Transactions on Services Computing*, 3(1):32–45, 2010.

[63] V. Gabrel, M. Manouvrier, and C. Murat. Optimal and Automatic Transactional Web Service Composition with Dependency Graph and 0-1 Linear Programming. In *Proceedings of the 12th International Conference on Service-Oriented Computing*, pages 108–122. Springer, 2014.

[64] M. Gagnaire, F. Diaz, C. Coti, C. Cerin, K. Shiozaki, Y. Xu, P. Delort, J. Smets, J. Le Lous, and S. Lubiarz. Downtime Statistics of Current Cloud Solutions. Technical report, 2012.

[65] M. Gajewski, M. Momotko, H. Meyer, H. Schuschel, and M. Weske. Dynamic Failure Recovery of Generated Workflows. In *Proceedings of the 16th International Workshop on Database and Expert Systems Applications*, pages 982–986. IEEE, 2005.

[66] L. Gao, S. D. Urban, and J. Ramachandran. A survey of Transactional Issues for Web Service Composition and Recovery. *International Journal of Web and Grid Services*, 7(4):331–356, 2011.

[67] H. Garcia-Molina and K. Salem. SAGAS. In *Proceedings of the SIGMOD International Conference on Management of Data*, volume 16. ACM, 1987.

[68] F. Gottschalk, W. M. P. van der Aalst, and M. H. Jansen-Vullers. Configurable Process Models A Foundational Approach. In *Reference Modeling*, pages 59–77. Springer, 2007.

[69] S. Götz, C. Wilke, S. Richly, and U. Assmann. Approximating Quality Contracts for Energy Auto-Tuning Software. In *Proceedings of the First International Workshop on Green and Sustainable Software*, pages 8–14. IEEE Press, 2012.

[70] J. Gray. The Transaction Concept: Virtues and Limitations. In *Proceedings of the Very Large Database Conference*, pages 144–154, 1981.

[71] G. Gröner, M. Bošković, F. S. Parreiras, and D. Gašević. Modeling and Validation of Business Process Families. *Information Systems*, 38(5):709–726, 2013.

[72] C. Hagen and G. Alonso. Exception Handling in Workflow Management Systems. *IEEE Transactions on Software Engineering*, 26(10):943–958, 2000.

[73] A. Hallerbach, T. Bauer, and M. Reichert. Guaranteeing Soundness of Configurable Process Variants in Provop. In *IEEE Conference on Commerce and Enterprise Computing*, pages 98–105. IEEE, 2009.

[74] D. Harel and A. Naamad. The STATEMATE Semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, 1996.

[75] A. Höing, G. Scherp, S. Gudenkauf, D. Meister, and A. Brinkmann. An Orchestration as a Service Infrastructure Using Grid Technologies and WS-BPEL. In *Service-Oriented Computing*, pages 301–315. Springer, 2009.

[76] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*, volume 1003. Addison-Wesley Reading, 2004.

[77] S. Hwang, E. Lim, C. Lee, and C. Chen. Dynamic Web Service Selection for Reliable Web Service Composition. *IEEE Transactions on Services Computing*, 1(2):104–116, 2008.

[78] K. Jensen. Coloured Petri Nets. In *Petri Nets: Central Models and Their Properties*, pages 248–299. Springer, 1987.

[79] J. M. Jiang, S. Zhang, P. Gong, and Z. Hong. Configuring Business Process Models. *SIGSOFT Software Engineering Notes*, 38(4):1–10, 2013.

[80] J. M. Jiang, S. Zhang, P. Gong, Z. Hong, and H. Yue. Modeling and Analyzing Mixed Communications in Service-Oriented Trustworthy Software. *Science China Information Sciences*, 55(12):2738–2756, 2012.

[81] K. Johny. A New Broker-Based Architecture for TQoS Driven Web Services Composition. In *Proceedings of the 2nd International Conference on Advances in Computer Engineering*, pages 159–161, 2011.

[82] K. C. Kang, J. Lee, and P. Donohoe. Feature-Oriented Product Line Engineering. *IEEE Software*, 19(4):58–65, 2002.

[83] W. Kim. *Modern Database Systems: The Object Model, Interoperability, and Beyond.* ACM Press/Addison-Wesley Publishing Co., 1995.

[84] S. Klüppelholz and C. Baier. Alternating-Time Stream Logic for Multi-Agent Systems. *Science of Computer Programming*, 75(6):398–425, June 2010.

[85] N. Kokash and F. Arbab. Formal Design and Verification of Long-Running Transactionsl with Extensible Coordination Tools. *IEEE Transactions on Services Computing*, 6:186–200, 2013.

[86] J. Korhonen, L. Pajunen, and J. Puustjarvi. Automatic Composition of Web Service Workflows Using a Semantic Agent. In *Proceedings of the International Conference on Web Intelligence*, pages 566–569. IEEE, 2003.

[87] M. Kovács, D. Varró, and L. Gönczy. Formal Modeling of BPEL Workflows Including Fault and Compensation Handling. In *Proceedings of the Workshop on Engineering Fault Tolerant Systems*. ACM, 2007.

[88] R. Kowalski and M. Sergot. A Logic-based Calculus of Events. In *Foundations of Knowledge Base Management*, pages 23–55. Springer, 1989.

[89] Saul Kripke. Semantical Considerations on Modal Logic. *Acta Philosophica Fennica*, 16(1963):83–94, 1963.

[90] A. Kumar, A. K. Sen, M. H. Sundari, and A. Bagchi. Semantic Notions of Weakly Correct AND/XOR Business Workflows Based on Partial Synchronization. In *Proceedings of the International Conference on Services Computing*, pages 128–135. IEEE, 2011.

[91] A. Kumar and W.Z Yao. Design and Management of Flexible Process Variants using Templates and Rules. *Computers in Industry*, 63(2):112–130, 2012.

[92] I. Kumara, J. Han, A. Colman, and M. Kapuruge. Runtime Evolution of Service-Based Multi-Tenant SaaS Applications. In *Service-Oriented Computing*, pages 192–206. Springer, 2013.

[93] M. La Rosa, M. Dumas, A. H. M. ter Hofstede, and J. Mendling. Configurable Multi-Perspective Business Process Models. *Information Systems*, 36(2):313–340, 2011.

[94] M. La Rosa, M. Dumas, A. H. M. ter Hofstede, J. Mendling, and F. Gottschalk. Beyond Control-Flow: Extending Business Process Configuration to Roles and Objects. In *Proceedings of the 27th International Conference on Conceptual Modeling (ER 2008)*, pages 199–215.

[95] M. La Rosa, W. M. P. van der Aalst, M. Dumas, and A. H. M. ter Hofstede. Questionnaire-Based Variability Modeling for System Configuration. *Software & Systems Modeling*, 8(2):251–274, 2009.

[96] N. Laranjeiro and M. Vieira. Towards Fault Tolerance in Web Services Compositions. In *Proceedings of the Workshop on Engineering Fault Tolerant Systems*. ACM, 2007.

[97] G. Li, L. Liao, D. Song, and Z. Zheng. A Fault-Tolerant Framework for QoS-Aware Web Service Composition via Case-Based Reasoning. *International Journal of Web and Grid Services*, 10(1):80–99, 2014.

[98] Y. Li, Y. Liu, L. Zhang, G. Li, B. Xie, and J. Sun. An Exploratory Study of Web Services on the Internet. In *Proceedings of the International Conference on Web Services*, pages 380–387. IEEE, 2007.

[99] M. Little. Transactions and Web Services. *Communications of the ACM*, 46(10):49–54, 2003.

[100] A. Liu, Q. Li, L. Huang, and M. Xiao. FACTS: A Framework for Fault-tolerant Composition of Transactional Web Services. *IEEE Transactions on Services Computing*, 3(1):46–59, 2010.

[101] Y. Liu, B. Zhang, G. Liu, M. Zhang, and J. Na. Evolving SaaS Based on Reflective Petri Nets. In *Proceedings of the 7th Workshop on Reflection, AOP and Meta-Data for Software Evolution*, pages 7:1–7:4. ACM, 2010.

[102] C. Lizhen, W. Haiyang, J. Lin, and H. Pu. Customization Modeling Based on Metagraph for Multi-Tenant Applications. In *The 5th International Conference on Pervasive Computing and Applications*, pages 255–260. IEEE, 2010.

[103] N. Looker, M. Munro, and J. Xu. Increasing Web Service Dependability Through Consensus Voting. In *Proceedings of the 29th Annual International of Computer Software and Applications Conference*, volume 2, pages 66–69. IEEE, 2005.

[104] T. Lynn, J. Mooney, M. Helfert, D. Corcoran, G. Hunt, L. Van Der Werff, J. Morrison, and P. Healy. Towards a Framework for Defining and Categorising Business Process-As-A-Service (BPaaS). In *21st International Product Development Management Conference*, 2014.

[105] H. E. Mansour and T. Dillon. Dependability and Rollback Recovery for Composite Web Services. *IEEE Transactions on Services Computing*, 4(4):328–339, 2011.

[106] S. Marston, Z. Li, S. Bandyopadhyay, J. Zhang, and A. Ghalsasi. Cloud Computing The Business Perspective. *Decision Support Systems*, 51(1):176–189, 2011.

[107] S. Mehrotra, R. Rastogi, A. Silberschatz, and H. F. Korth. A Transaction Model for Multidatabase Systems. In *Proceedings of the 12th International Conference on Distributed Computing Systems*, pages 56–63. IEEE, 1992.

[108] X. Mei, A. Jiang, F. Zheng, and S. Li. Execution Semantics Analysis Based Composition Compensation Mechanism in Web Services Composition. In *Proceedings of the WRI World Congress on Computer Science and Information Engineering*, volume 7, pages 820–824. IEEE, 2009.

[109] J. Mendling, J. Recker, M. Rosemann, and W. M. P. van der Aalst. Generating Correct EPCs from Configured C-EPCs. In *Proceedings of the Symposium on Applied Computing*, pages 1505–1510. ACM, 2006.

[110] R. Mietzner, A. Metzger, F. Leymann, and K. Pohl. Variability Modeling to Support Customization and Deployment of Multi-Tenant-Aware Software as a Service Applications. In *Proceedings of the ICSE Workshop on Principles of Engineering Service Oriented Systems*, pages 18–25. IEEE, 2009.

[111] F. Montagut, R. Molva, and S.T. Golega. Automating the Composition of Transactional Web Services. *International Journal of Web Services Research*, 5(1):24–41, 2008.

[112] F. Montagut, R. Molva, and S. Tecumseh Golega. The Pervasive Workflow: A Decentralized Workflow System Supporting Long-Running Transactions. *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, pages 319–333, 2008.

[113] J. E. B. Moss. Nested Transactions: An Approach to Reliable Distributed Computing. Technical report, 1985.

[114] M. Nitu. Configurability in saas (software as a service) applications. In *Proceedings of the 2nd Annual Conference on India Software Engineering*, pages 19–26, 2009.

[115] J. Ouyang, A. Sahai, and V. Machiraju. An Approach to Optimistic Commit and Transparent Compensation for E-Service Transactions. In *Proceedings of the 14th International Conference on Parallel and Distributed Computing Systems*, pages 142–149, 2001.

[116] M. P. Papazoglou. Service-Oriented Computing: Concepts, Characteristics and Directions. In *Proceedings of the 4th International Conference on Web Information Systems Engineering*, pages 3–12. IEEE, 2003.

[117] M. P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann. Service-Oriented Computing: State of the Art and Research Challenges. *IEEE Computer*, 40(11):38–45, 2007.

[118] M. P. Papazoglou and W. van den Heuvel. Blueprinting the Cloud. *IEEE Internet Computing*, 15(6):74–79, 2011.

[119] M. Pathirage, S. Perera, I. Kumara, and S. Weerawarana. A Multi-Tenant Architecture for Business Process Executions. In *Proceedings of the 9th International Conference on Web Services*, pages 121–128. IEEE, 2011.

[120] C. Peltz. Web Services Orchestration and Choreography. *IEEE Computer*, 36(10):46–52, 2003.

[121] D. Petcu and V. Stankovski. Towards Cloud-Enabled Business Process Management based on Patterns, Rules and Multiple Models. In *Proceedings of the 10th International Symposium on Parallel and Distributed Processing with Applications*, pages 454–459. IEEE, 2012.

[122] P. F. Pires, M. Benevides, and M. Mattoso. WebTransact: A Framework for Specifying and Coordinating Reliable Web Services Compositions. Technical report, Federal University of Rio De Janeiro, 2002.

[123] P. F Pires, M. R. F. Benevides, and M. Mattoso. Building Reliable Web Services Compositions. In *Web, Web-Services, and Database Systems*, pages 59–72. Springer, 2003.

[124] K. Pohl, G. Böckle, and F. J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer Science & Business Media, 2005.

[125] B. Randell and J. Xu. The Evolution of the Recovery Block Concept. *Software Fault Tolerance*, 3:1–22, 1995.

[126] J. Rao and X. Su. A Survey of Automated Web Service Composition Methods. In *Proceedings of the 1st International Conference on Semantic Web Services and Web Process Composition*, pages 43–54. Springer-Verlag, 2005.

[127] F. Rosenberg, P. Leitner, A. Michlmayr, P. Celikovic, and S. Dustdar. Towards Composition as a Service - a Quality of Service Driven Approach. In *Proceedings of the 25th International Conference on Data Engineering*, pages 1733–1740. IEEE, 2009.

[128] J. Schroeter, S. Cech, S. Götz, C. Wilke, and U. Aßmann. Towards Modeling a Variable Architecture for Multi-Tenant SaaS-Applications. In *Proceedings of the 6th International Workshop on Variability Modeling of Software-Intensive Systems*, pages 111–120. ACM, 2012.

[129] J. Schroeter, P. Mucha, M. Muth, K. Jugel, and M. Lochau. Dynamic Configuration Management of Cloud-based Applications. In *Proceedings of the 16th International Software Product Line Conference (SPLC 2012)*, pages 171–178.

[130] H. Schuldt, G. Alonso, C. Beeri, and H. Schek. Atomicity and Isolation for Transactional Processes. *ACM Transactions on Database Systems*, 27(1):63–116, 2002.

[131] B. Sengupta and A. Roychoudhury. Engineering Multi-Tenant Software-as-a-Service Systems. In *Proceedings of the 3rd International Workshop on Principles of Engineering Service-Oriented Systems*, pages 15–21. ACM, 2011.

[132] Q. Z. Sheng, Z. Maamar, L. Yao, C. Szabo, and S. Bourne. Behavior Modeling and Automated Verification of Web Services. *Information Sciences*, 258:416–433, 2014.

[133] Q. Z. Sheng, X. Qiao, A. V. Vasilakos, C. Szabo, S. Bourne, and X. Xu. Web Services Composition: A Decades Overview. *Information Sciences*, 280:218–238, 2014.

[134] Q.Z. Sheng, Z. Maamar, H. Yahyaoui, J. Bentahar, and K. Boukadi. Separating Operational and Control Behaviors: A New Approach to Web Services Modeling. *IEEE Internet Computing*, (3):68–76, 2010.

[135] J. Simmonds, S. Ben-David, and M. Chechik. Guided Recovery for Web Service Applications. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 247–256, 2010.

[136] Rachel L Smith, George S Avrunin, Lori A Clarke, and Leon J Osterweil. PROPEL: an Approach Supporting Property Elucidation. In *Proceedings of the 24th International Conference on Software Engineering*, pages 11–21. ACM, 2002.

[137] S. Stein, T. R. Payne, and N. R. Jennings. Flexible Provisioning of Web Service Workflows. *ACM Transactions on Internet Technology*, 9(1):2, 2009.

[138] F. Tartanoglu, V. Issarny, A. Romanovsky, and N. Levy. Dependability in the Web Services Architecture. In *Architecting Dependable Systems*, pages 90–109. Springer, 2003.

[139] A. H. M. Ter Hofstede, W. M. P. van der Aalst, M. Adams, and N. Russell. *Modern Business Process Automation: YAWL and its Support Environment*. Springer Science & Business Media, 2009.

[140] P. Townend, P. Groth, N. Looker, and J. Xu. FT-Grid: A Fault-tolerance System for e-Science. *Concurrency and Computation:Practice and Experience*, 20(3):297–309, 2005.

[141] W. Tsai and X. Sun. SaaS Multi-Tenant Application Customization. In *The 7th International Symposium on Service Oriented System Engineering*, pages 1–12. IEEE, 2013.

[142] W. M. P. van der Aalst. Business Process Configuration in the Cloud: How to Support and Analyze Multi-tenant Processes? In *Proceedings of the 9th European Conference on Web Services (ECOWS 2011)*, pages 3–10.

[143] W. M. P. Van Der Aalst. Configurable Services in the Cloud: Supporting Variability While Enabling Cross-Organizational Process Mining. In *On the Move to Meaningful Internet Systems*, pages 8–25. Springer, 2010.

[144] W. M. P. Van Der Aalst, M. Dumas, F. Gottschalk, A. H. M. ter Hofstede, M. La Rosa, and J. Mendling. Correctness-Preserving Configuration of Business Process Models. In *Fundamental Approaches to Software Engineering*, pages 46–61. Springer, 2008.

[145] W. M. P. van der Aalst, M. Dumas, F. Gottschalk, A. H. M. ter Hofstede, M. La Rosa, and J. Mendling. Preserving Correctness During Business Process Model Configuration. *Formal Aspects of Computing*, 22(3-4):459–482, 2010.

[146] W. M. P. van der Aalst, N. Lohmann, M. Rosa, and J. Xu. Correctness Ensuring Process Configuration: An Approach Based on Partner Synthesis. In *Proceedings of the 8th International Conference on Business Process Management (BPM 2010)*, volume 6336 of *Lecture Notes in Computer Science*, pages 95–111.

[147] W. M. P. van Der Aalst, A. H. M. Ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow Patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.

[148] B. F. van Dongen, M. H. Jansen-Vullers, H. M. W. Verbeek, and W. M. P. van der Aalst. Verification of the SAP Reference Models using EPC Reduction, State-Space Analysis, and Invariants. *Computers in Industry*, 58(6):578–601, 2007.

[149] K. Vidyasankar and G. Vossen. A Multi-Level Model for Web Service Composition. In *Proceedings of the IEEE International Conference on Web Services*, pages 462–469. IEEE, 2004.

[150] J. Vonk, W. Derks, P. Grefen, and M. Koetsier. Cross-organizational Transaction Support for Virtual Enterprises. In *Proceedings of the 7th International Conference on Cooperative Information Systems*, pages 323–334. Springer, 2000.

[151] F. Wagner, F. Ishikawa, and S. Honiden. Robust Service Compositions with Functional and Location Diversity. *IEEE Transactions on Service Computing*, PP(99), 2013.

[152] M. X. Wang, K. Y. Bandara, and C. Pahl. Process as a Service Distributed Multi-Tenant Policy-Based Process Runtime Governance. In *The International Conference on Services Computing*, pages 578–585. IEEE, 2010.

[153] Y. Wang, Y. Fan, and A. Jiang. A Paired-net Based Compensation Mechanism for Verifying Web Composition Transactions. In *4th International Conference on New Trends in Information Science and Service Science*, pages 1–6. IEEE, 2010.

[154] Y. Wei and M B. Blake. Service-Oriented Computing and Cloud Computing: Challenges and Opportunities. *Internet Computing*, (6):72–75, 2010.

[155] G. Weikum and H. J. Schek. Database Transaction Models for Advanced Applications. chapter Concepts and Applications of Multilevel Transactions and Open Nested Transactions, pages 515–553. Morgan Kaufmann Publishers Inc., 1992.

[156] S. A. White. Introduction to BPMN. *IBM Cooperation*, 2, 2004.

[157] K. Wolf. Does My Service Have Partners? In *Transactions on Petri Nets and Other Models of Concurrency II*, pages 152–171. Springer, 2009.

[158] J. Yu, T. Manh, J. Han, Y. Jin, Y. Han, and J. Wang. Pattern Based Property Specification and Verification for Service Composition. In *Proceedings of the International Conference on Web Information Systems Engineering*, pages 156–168, 2006.

[159] Q. Zhang, L. Cheng, and R. Boutaba. Cloud Computing: State-of-the-Art and Research Challenges. *Journal of Internet Services and Applications*, 1(1):7–18, 2010.

[160] W. Zhang, H. Yan, H. Zhao, and Z. Jin. A BDD-Based Approach to Verifying Clone-Enabled Feature Models Constraints and Customization. In *High Confidence Software Reuse in Large Systems*, pages 186–199. Springer, 2008.

[161] Z. Zheng and M. R. Lyu. Selecting an Optimal Fault Tolerance Strategy for Reliable Service-Oriented Systems with Local and Global Constraints. *IEEE Transactions on Computers*, 64(1):219–232, 2015.

APPENDIX A

Temporal Logic Template Specifications

This appendix contains the complete specifications of our temporal logic template set for specifying transactional requirements for service-oriented processes. Each specification provides developers or clients with enough information to use the template to formalize complex and diverse requirements in temporal logic without any knowledge of the language required.

The component-level template specifications are shown in Tables A.1-A.7. Due to space considerations, the temporal logic field of the `CompensateSuccess` template are split into their own table: Table A.3. The process-level template specifications comprise of Tables A.8-A.13. An overview of the purpose of each field in these specifications can be found in Chapter 4.2.

The component-level templates are for specifying transactional requirements related to the fault-handling of individual components. `CompensateFailure` is used to specify necessary recovery actions following the failure of a component, while `CompensateSuccess` specify the actions to undo the effect of a completed component. For `CompensateFailure`, a *Fault* message originating from the component in order to indicate its failure, while `CompensateSuccess` requires a *Recovery* message to trigger backwards recovery operations. `Alternative` specifies valid alternative actions upon the failure of a component, and requires either *Fault* or *Syncreq* message from the component to indicate the failure. `NonRetriable` specifies components that cannot or should not be retired after failing, while `RetriablePivot` can be used for components that *can* be retried but not undone, and `NonRetriablePivot` components can be neither retried or undone.

The first four process-level templates specify transactional requirements that apply conditions related to the activity of a certain control behavior states. `ControlStateCritical` specifies a condition that must be satisfied before entering the state, `ControlStateTrigger` specifies a condition that must lead to the state once it has been satisfied, `ControlStateReachable` indicates that the state is still possible to reach when the condition is satisfied, and `ControlState Unreachable` states that the state is no longer reachable once the condition is satisfied. As `ControlState Reachable` and `ControlStateUnreachable` use CTL for temporal logic, the *Scope* variables used in other templates cannot be used due to language restrictions.

Finally, `Compensation` is used to specify a condition for the valid compensation of the process after it has been committed. `ConditionalCompensation` also specifies valid compensation, but only for compensating successful executions that have satisfied a certain condition during their execution.

Table A.1: Template specification for `CompensateFailure`

Name	`CompensateFailure <Component,Recovery,Card,Scope>`		
Type	Component-level		
Variables	*Component*	An operational behavior state that requires recovery upon failure.	
	Recovery	A condition that undoes the effect of the failure. This can be a single component or a set of components structured with $\vee$ operators.	
	Card	One of the cardinality options below.	
	Scope	One of the scope options below.	
Description	The failure of *Component* leaves an impact an effect, which must be compensated by *Recovery* becoming true in the future.		
Prerequisites	A `Fault` message originating from *Component* in the operational behavior is necessary for this requirement to be verified.		
Cardinality	1:1	*Recovery* undoes one failure of *Component*.	
	Many:1	*Recovery* can undo many failures of *Component*.	
Scope	G	The template applies in all executions.	
	P	Applies during the satisfaction of a condition P.	
	$\neg P$	Applies during the negation of a condition P.	
	Before P	*Recover* must precede the satisfaction of P.	
LTL	1:1	G	$G(\mathit{Component}.FAULT \rightarrow ((\neg(\mathit{Activated} \wedge \mathit{Component}) \cup \mathit{Recovery}) \wedge F(\mathit{Recovery}))$
		P	$F(P) \rightarrow G(\mathit{Component}.FAULT \rightarrow ((\neg(\mathit{Activated} \wedge \mathit{Component}) \cup \mathit{Recovery}) \wedge F(\mathit{Recovery}))$
		$\neg P$	$G(\neg P) \rightarrow G(\mathit{Component}.FAULT \rightarrow ((\neg(\mathit{Activated} \wedge \mathit{Component}) \cup \mathit{Recovery}) \wedge F(\mathit{Recovery}))$
		Before P	$G(\mathit{Component}.FAULT \rightarrow (((\neg(\mathit{Activated} \wedge \mathit{Component}) \wedge \neg P) \cup \mathit{Recovery}) \wedge F(\mathit{Recovery}))$
	Many:1	G	$G(\mathit{Component}.FAULT \rightarrow F(\mathit{Recovery}))$
		P	$F(P) \rightarrow G(\mathit{Component}.FAULT \rightarrow F(\mathit{Recovery}))$
		$\neg P$	$G(\neg P) \rightarrow G(\mathit{Component}.FAULT \rightarrow F(\mathit{Recovery}))$
		Before P	$G(\mathit{Component}.FAULT \rightarrow ((\neg P \cup \mathit{Recovery}) \wedge F(\mathit{Recovery})))$

Table A.2: Template specification for `CompensateSuccess`, minus temporal logic

Name	`CompensateSuccess <Component,Recovery,Card,Scope>`	
Type	Component-level	
Variables	*Component*	An operational behavior state that requires recovery upon failure.
	Recovery	A condition that undoes the effect of the *Component*. This can be a single component or a set of components structured with $\vee$ operators.
	Card	One of the cardinality options below.
	Scope	One of the scope options below.
Description	Specifies *Component* and *Recovery*, such that when the composition must be undone, *Recovery* must be satisfied to undo the effect of *Component*.	
Prerequisites	A `Recover` message able to trigger a compensation process must be present between the behavior models.	
Cardinality	1:1	*Recovery* undoes one failure of *Component*.
	Many:1	*Recovery* can undo many failures of *Component*.
Scope	G	The template applies in all executions.
	P	Applies during the satisfaction of a condition P.
	$\neg P$	Applies during the negation of a condition P.
	Before P	*Recover* must precede the satisfaction of P.

Table A.3: LTL for the implementations of `CompSuccess`

1:1	G	$G((\neg(Activated \wedge Component) \wedge O((Activated \wedge Component) \wedge X(!Component.SYNCREQ \wedge \neg Component.FAULT))) \rightarrow X(\neg(Activated \wedge Component))\wedge$ $G(((Done \wedge O((Activated \wedge Component) \wedge X(\neg Component.SYNCREQ \wedge \neg Component.FAULT)) \wedge F(RECOVER)) \rightarrow F(Recovery))$
	P	$F(P) \rightarrow G((\neg(Activated \wedge Component) \wedge O((Activated \wedge Component) \wedge X(!Component.SYNCREQ \wedge \neg Component.FAULT))) \rightarrow X(\neg(Activated \wedge Component))\wedge$ $G(((Done \wedge O((Activated \wedge Component) \wedge X(\neg Component.SYNCREQ \wedge \neg Component.FAULT)) \wedge F(RECOVER)) \rightarrow F(Recovery))$
	$\neg P$	$G(\neg P) \rightarrow G((\neg(Activated \wedge Component) \wedge O((Activated \wedge Component) \wedge X(!Component.SYNCREQ \wedge \neg Component.FAULT))) \rightarrow X(\neg(Activated \wedge Component))\wedge$ $G(((Done \wedge O((Activated \wedge Component) \wedge X(\neg Component.SYNCREQ \wedge \neg Component.FAULT)) \wedge F(RECOVER)) \rightarrow F(Recovery))$
	Before P	$G((\neg(Activated \wedge Component) \wedge O((Activated \wedge Component) \wedge X(!Component.SYNCREQ \wedge \neg Component.FAULT))) \rightarrow X(\neg(Activated \wedge Component))\wedge$ $G(((Done \wedge O((Activated \wedge Component) \wedge X(\neg Component.SYNCREQ \wedge \neg Component.FAULT)) \wedge F(RECOVER)) \rightarrow ((\neg P \cup Recovery) \wedge F(Recovery))))$
Many:1	G	$G(((Done \wedge O((Activated \wedge Component) \wedge X(\neg Component.SYNCREQ \wedge \neg Component.FAULT)) \wedge F(RECOVER)) \rightarrow F(Recovery))$
	P	$F(P) \rightarrow G(((Done \wedge O((Activated \wedge Component) \wedge X(\neg Component.SYNCREQ \wedge \neg Component.FAULT)) \wedge F(RECOVER)) \rightarrow F(Recovery))$
	$\neg P$	$G(\neg P) \rightarrow G(((Done \wedge O((Activated \wedge Component) \wedge X(\neg Component.SYNCREQ \wedge \neg Component.FAULT)) \wedge F(RECOVER)) \rightarrow F(Recovery))$
	Before P	$G(((Done \wedge O((Activated \wedge Component) \wedge X(\neg Component.SYNCREQ \wedge \neg Component.FAULT)) \wedge F(RECOVER)) \rightarrow ((\neg P \cup Recovery) \wedge F(Recovery)))$

Table A.4: Template specification for `Alternative`

Name	`Alternative <Component,Alt,Card,Scope>`		
Type	Component-level		
Variables	*Component*	The component that can be substituted with one or several others upon failure.	
	Alt	The component or components that can be used to replace the Component variable, expressed as a Boolean condition.	
	Card	One of the cardinality options below.	
	Scope	One of the scope options below.	
Description	Following the failure of a component, one or several alternative operations, expressed as a condition, are considered acceptable replacements.		
Prerequisites	A `Syncreq` or `Fault` message originating from *Component* in the operational behavior is necessary for this requirement to be verified.		
Cardinality	1:1	*Recovery* undoes one failure of *Component*.	
	Many:1	*Recovery* can undo many failures of *Component*.	
Scope	G	The template applies in all executions.	
	P	Applies during the satisfaction of a condition P.	
	$\neg P$	Applies during the negation of a condition P.	
	Before P	*Recover* must precede the satisfaction of P.	
LTL	1:1	G	$G((Component.FAULT \vee Component.SYNCREQ) \rightarrow ((\neg(Activated \wedge Component) \cup (Activated \wedge Alt)) \wedge F(Activated \wedge Alt)))$
		P	$F(P) \rightarrow G((Component.FAULT \vee Component.SYNCREQ) \rightarrow ((\neg(Activated \wedge Component) \cup (Activated \wedge Alt)) \wedge F(Activated \wedge Alt)))$
		$\neg P$	$G(\neg P) \rightarrow G((Component.FAULT \vee Component.SYNCREQ) \rightarrow ((\neg(Activated \wedge Component) \cup (Activated \wedge Alt)) \wedge F(Activated \wedge Alt)))$
		Before P	$G((Component.FAULT \vee Component.SYNCREQ) \rightarrow (((\neg P \wedge \neg(Activated \wedge Component)) \cup (Activated \wedge Alt)) \wedge F(Activated \wedge Alt)))$
	Many:1	G	$G((Component.FAULT \vee Component.SYNCREQ) \rightarrow F((Activated \wedge Alt) \vee (Activated \wedge Component)))$
		P	$F(P) \rightarrow G((Component.FAULT \vee Component.SYNCREQ) \rightarrow F((Activated \wedge Alt) \vee (Activated \wedge Component)))$
		$\neg P$	$G\neg P) \rightarrow (G((Component.FAULT \vee Component.SYNCREQ) \rightarrow F((Activated \wedge Alt) \vee (Activated \wedge Component)))$
		Before P	$G((Component.FAULT \vee Component.SYNCREQ) \rightarrow ((\neg P \cup ((Activated \wedge Alt) \vee (Activated \wedge Component))) \wedge F((Activated \wedge Alt) \vee (Activated \wedge Component))))$

Table A.5: Template specification for `NonRetriable`

Name	`NonRetriable<ControlState,Scope>`	
Type	Component-level	
Variables	*Component*	A component that cannot be retried following failure.
	Scope	One of the scope options below.
Description	Following failure of *Component*, retrial is either not possible, or the user is not interested in it.	
Prerequisites	N/A	
Cardinality	N/A	
Scope	G	The template applies in all executions.
	P	Applies during the satisfaction of a condition P.
	$\neg P$	Applies during the negation of a condition P.
LTL	G	$G((\neg(Activated \wedge Component) \wedge O(Component.SYNCREQ \vee Component.FAULT)) \rightarrow X(\neg(Activated \wedge Component)))$
	P	$F(P) \rightarrow G((\neg(Activated \wedge Component) \wedge O(Component.SYNCREQ \vee Component.FAULT)) \rightarrow X(\neg(Activated \wedge Component)))$
	$\neg P$	$G(\neg P) \rightarrow G((\neg(Activated \wedge Component) \wedge O(Component.SYNCREQ \vee Component.FAULT)) \rightarrow X(\neg(Activated \wedge Component)))$

Table A.6: Template specification for `RetriablePivot`

Name	`RetriablePivot<ControlState,Scope>`	
Type	Component-level	
Variables	*Component*	The component that can be retried but not undone.
	Scope	One of the scope options below.
Description	A component that may be retried, but not undone. Following its execution, the service must commit.	
Prerequisites	N/A	
Cardinality	N/A	
Scope	G	The template applies in all executions.
	P	Applies during the satisfaction of a condition P.
	$\neg P$	Applies during the negation of a condition P.
LTL	G	$G(((Activated \wedge Component) \wedge X(\neg Component.SYNCREQ \wedge \neg Component.FAULT)) \rightarrow F(Done))$
	P	$F(P) \rightarrow G(((Activated \wedge Component) \wedge X(\neg Component.SYNCREQ \wedge \neg Component.FAULT)) \rightarrow F(Done))$
	$\neg P$	$G(\neg P) \rightarrow G(((Activated \wedge Component) \wedge X(\neg Component.SYNCREQ \wedge \neg Component.FAULT)) \rightarrow F(Done))$

Table A.7: Template specification for `NonRetriablePivot`

Name	`NonRetriablePivot<ControlState,Scope>`	
Type	Component-level	
Variables	*Component*	The component that cannot be retried or undone.
	Scope	One of the scope options below.
Description	A component that may not be retried upon failure, and cannot be undone following success. The service must commit if this component executes successfully.	
Prerequisites	N/A	
Cardinality	N/A	
Scope	G	The template applies in all executions.
	P	Applies during the satisfaction of a condition P.
	$\neg P$	Applies during the negation of a condition P.
LTL	G	$G((((Activated \wedge Component) \wedge X(\neg Component.SYNCREQ \wedge \neg Component.FAULT)) \rightarrow F(Done)) \wedge ((Component.SYNCREQ \vee Component.Fault) -> F(Aborted)))$
	P	$F(P) \rightarrow G((((Activated \wedge Component) \wedge X(\neg Component.SYNCREQ \wedge \neg Component.FAULT)) \rightarrow F(Done)) \wedge ((Component.SYNCREQ \vee Component.Fault) -> F(Aborted)))$
	$\neg P$	$G(\neg P) \rightarrow G((((Activated \wedge Component) \wedge X(\neg Component.SYNCREQ \wedge \neg Component.FAULT)) \rightarrow F(Done)) \wedge ((Component.SYNCREQ \vee Component.Fault) -> F(Aborted)))$

Table A.8: Template specification for `ControlStateCritical`

Name	`ControlStateCritical<ControlState,Condition,Scope>`	
Type	Process-level	
Variables	*ControlState*	The control behavior state this critical condition applies to.
	Condition	The precondition for entering this control behavior state. This can be a single component or a set structured with $\wedge$ and $\vee$ operators.
	Scope	One of the scope options below.
Description	*Condition* denotes the precondition for entering *ControlState*. When *ControlState* is entered, *Condition* must have been met previously on the execution path.	
Prerequisites	N/A	
Cardinality	N/A	
Scope	G	The template applies in all executions.
	P	Applies during the satisfaction of a condition P.
	$\neg P$	Applies during the negation of a condition P.
	Before P	*ControlState* is entered before P is met.
LTL	G	$G(ControlState \rightarrow O(Condition))$
	P	$F(P) \rightarrow G(ControlState \rightarrow O(Condition))$
	$\neg P$	$G(\neg P) \rightarrow G(ControlState \rightarrow O(Condition))$
	Before P	$G(ControlState \rightarrow (O(Condition) \wedge H(\neg P)))$

Table A.9: Template specification for `ControlStateTrigger`

Name	`ControlStateTrigger<ControlState,Condition,Scope>`	
Type	Composition-level	
Variables	*ControlState*	The control behavior state that is triggered by this requirement condition applies to.
	Condition	The trigger for entering this control behavior state. This can be a single component or a set structured with $\wedge$ and $\vee$ operators.
	Scope	One of the scope options below.
Description	*Condition* specifies a condition that must trigger the entering of *ControlState* at some point in the future.	
Prerequisites	N/A	
Cardinality	N/A	
Scope	G	The template applies in all executions.
	P	Applies during the satisfaction of a condition P.
	$\neg P$	Applies during the negation of a condition P.
	Before P	*ControlState* is entered before P is met.
LTL	G	$G((Condition \wedge X(\neg Condition.FAULT \wedge \neg Condition.SYNCREQ)) \rightarrow F(ControlState))$
	P	$F(P) \rightarrow G((Condition \wedge X(\neg Condition.FAULT \wedge \neg Condition.SYNCREQ)) \rightarrow F(ControlState) \rightarrow F(ControlState))$
	$\neg P$	$G(\neg P) \rightarrow G((Condition \wedge X(\neg Condition.FAULT \wedge \neg Condition.SYNCREQ)) \rightarrow F(ControlState) \rightarrow F(ControlState))$
	Before P	$G((Condition \wedge X(\neg Condition.FAULT \wedge \neg Condition.SYNCREQ)) \rightarrow F(ControlState) \rightarrow (F(ControlState) \wedge (\neg P \cup ControlState))$

Table A.10: Template specification for `ControlStateReachable`

Name	`ControlStateReachable<ControlState,Condition>`	
Type	Composition-level	
Variables	*ControlState*	The control behavior state that is reachable, given the satisfaction of *Condition*.
	Condition	The condition that, while true, indicates that *ControlState* is reachable.
Description	Whenever *Condition* is met, *ControlState* should be reachable. However, this does not imply that *ControlState* is unreachable when *Condition* is false.	
Prerequisites	N/A	
Cardinality	N/A	
Scope	N/A	
CTL	$AG(Condition \rightarrow EF(ControlState))$	

Table A.11: Template specification for `ControlStateUnreachable`

Name	`ControlStateUnreachable<ControlState,Condition>`	
Type	Composition-level	
Variables	*ControlState*	The control behavior state that is unreachable, given the satisfaction of *Condition*.
	Condition	The condition that implies that *ControlState* should no longer be reachable.
Description	The requirement specifies that when *Condition* is met, it is no longer possible to reach *ControlState*.	
Prerequisites	N/A	
Cardinality	N/A	
Scope	N/A	
CTL	$AG(Condition \rightarrow AG(\neg ControlState))$	

Table A.12: Template specification for `Compensation`

Name	`Compensation<CompCondition>`	
Type	Composition-level	
Variables	*CompCondition*	A condition that must be met during the compensation of the service.
Description	A requirement that must be met during any compensation of the service.	
Prerequisites	N/A	
Cardinality	N/A	
Scope	N/A	
LTL	$G((Done \wedge F(RECOVER)) \rightarrow F(CompCondition))$	

Table A.13: Template specification for `ConditionalCompensation`

Name	`ConditionalCompensation<ExecCondition,CompCondition>`	
Type	Composition-level	
Variables	*ExecCondition*	A condition that can be satisfied during successful execution.
	CompCondition	The compensatory process required following *ExecCondition*.
Description	A requirement for satisfactory compensation of the service in cases where a condition is met during execution.	
Prerequisites	N/A	
Cardinality	N/A	
Scope	N/A	
CTL	$G((Done \wedge F(RECOVER) \wedge O(ExecCondition)) \rightarrow F(CompCondition))$	

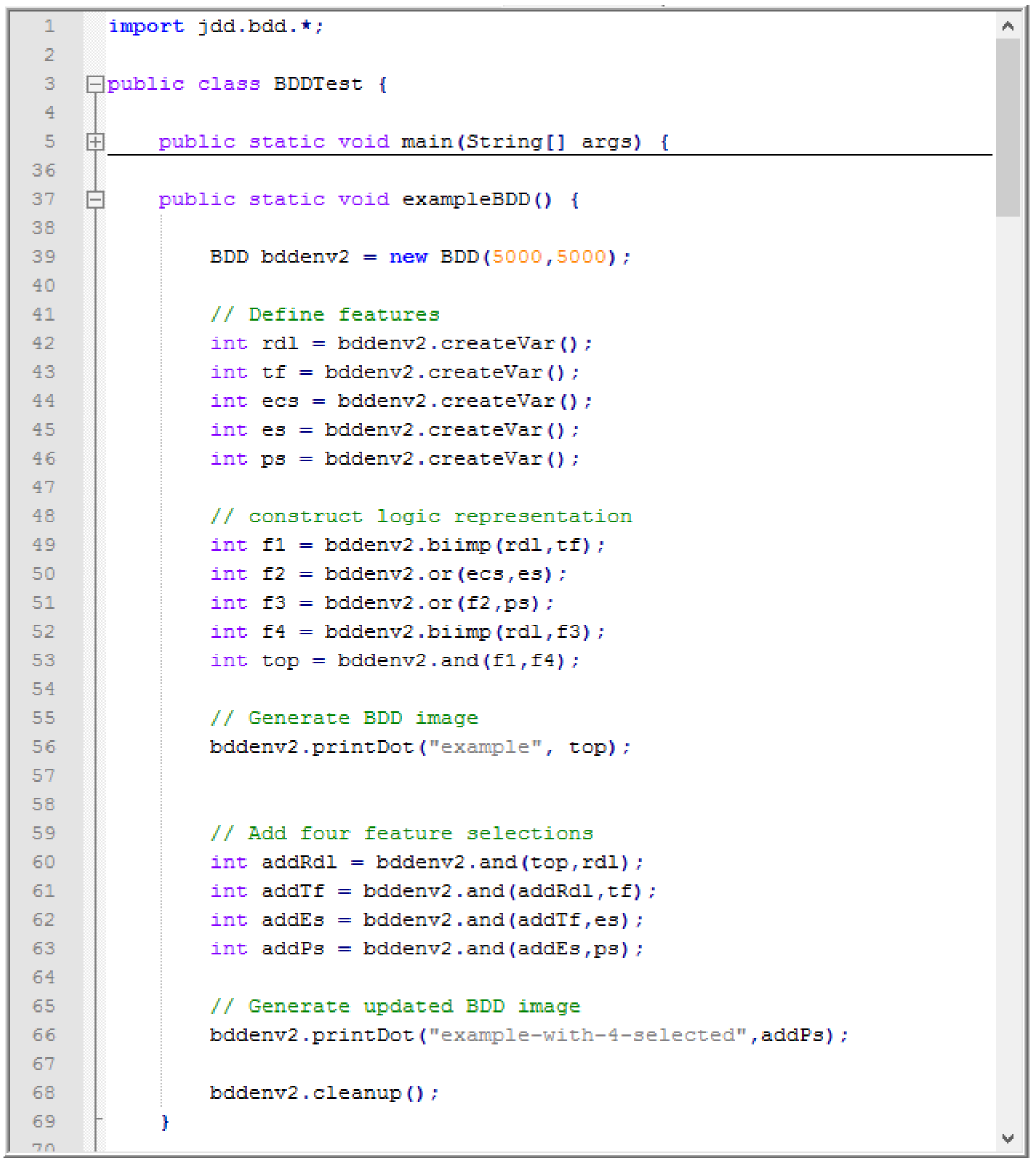
APPENDIX B

Using JDD for BDD Construction

Figure B.1 shows an example of a Java class using the JDD library to construct a BDD. In the `exampleBDD()` method, a BDD is constructed from the small feature model and propositional logic property in Figure 5.7 (lines 39-53). Next, four feature selections are made using the AND relation (lines 60-63), before the updated BDD is produced. The updated BDD is shown in Figure 5.8.

A memory cache for BDD construction and manipulation is provisioned and released with BDD constructor and `cleanup()` methods respectively. The `printDot(string,int)` methods are used to generate BDD images using *Graphviz*[1], a software tool for visualizing graph structures.

[1]http://www.graphviz.org/



```
import jdd.bdd.*;

public class BDDTest {

    public static void main(String[] args) {

    public static void exampleBDD() {

        BDD bddenv2 = new BDD(5000,5000);

        // Define features
        int rdl = bddenv2.createVar();
        int tf = bddenv2.createVar();
        int ecs = bddenv2.createVar();
        int es = bddenv2.createVar();
        int ps = bddenv2.createVar();

        // construct logic representation
        int f1 = bddenv2.biimp(rdl,tf);
        int f2 = bddenv2.or(ecs,es);
        int f3 = bddenv2.or(f2,ps);
        int f4 = bddenv2.biimp(rdl,f3);
        int top = bddenv2.and(f1,f4);

        // Generate BDD image
        bddenv2.printDot("example", top);


        // Add four feature selections
        int addRdl = bddenv2.and(top,rdl);
        int addTf = bddenv2.and(addRdl,tf);
        int addEs = bddenv2.and(addTf,es);
        int addPs = bddenv2.and(addEs,ps);

        // Generate updated BDD image
        bddenv2.printDot("example-with-4-selected",addPs);

        bddenv2.cleanup();
    }
```

Figure B.1: A Java class for implementing a BDD using the JDD library

APPENDIX C

Checkout Configuration BDD

Figure C.1 shows a BDD generated generated by the JDD library during the BPaaS configuration process. The final true and false nodes are labelled `1` and `0` respectively. Instead of labeling each BDD node with the name of the boolean property in the property, JDD numbers them according to their variable ordering. For example, the root node `v1` corresponds to the root `Checkout Service` in the feature model of Figure 5.5, as it is the first feature in the depth-first traversal variable ordering. A node may appear many times, but always on the same level of the tree. Furthermore, unnecessary nodes are removed automatically. For example, `v2` (`Microguru`) is not included as it is impossible to select along with the features already chosen.

This tree represents the propositional logic property shown in Figure 5.9, which corresponds to the domain constraints of the checkout service feature model, and a set of feature selections. As this BDD contains at least one branch to the final `1` node, this selection of features is satisfiable.

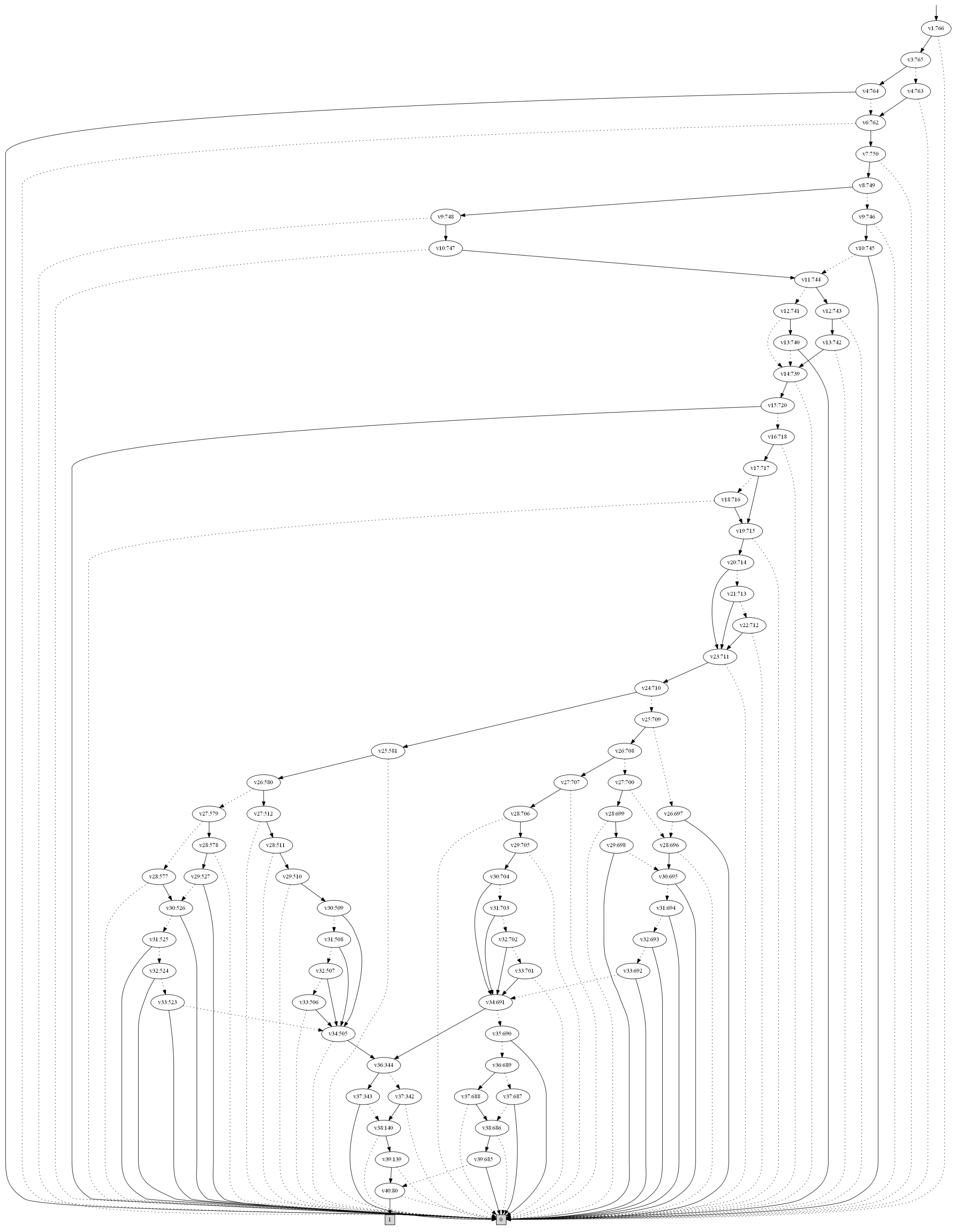


Figure C.1: Binary Decision Diagram form of the propositional logic property of Figure 5.9

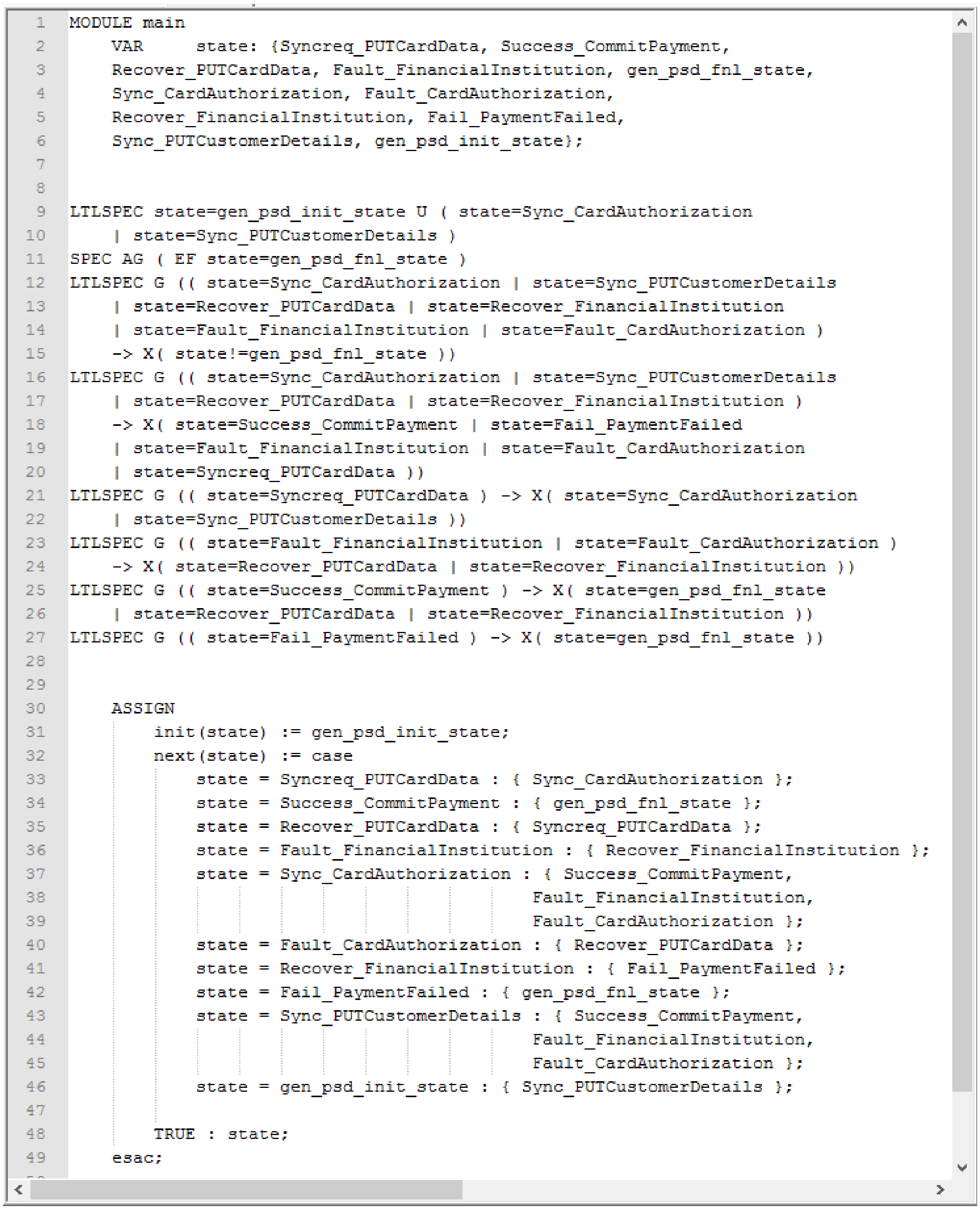
APPENDIX D

Implementation of Online Payment Scenario

This service-oriented process implementing an online payment scenario was defined to demonstrate our approaches for modeling and verification. The operational behavior model of the process is shown in Figure 3.4, and the inter-behavior messages are defined in Table 3.11. The validation scenario for conversation rule checking and transactional requirement verification is presented in Chapter 6.2.1.1.

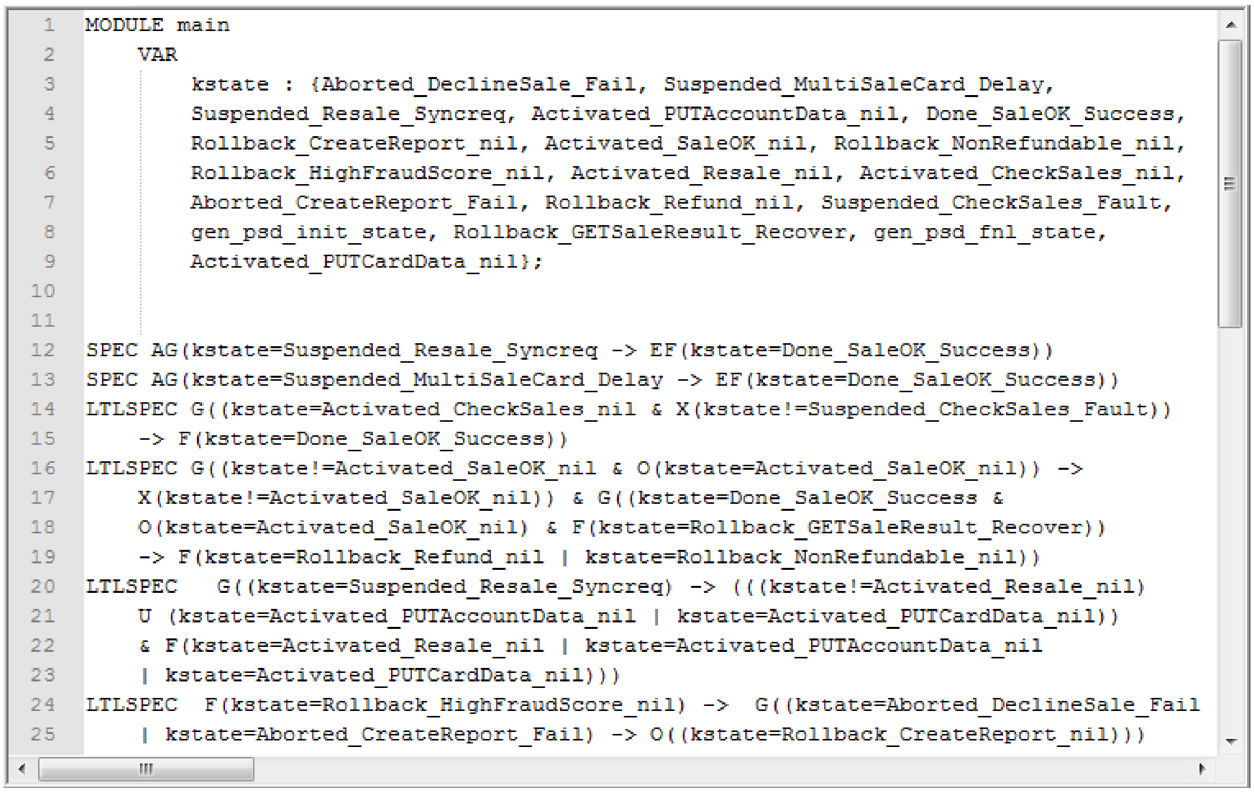
For conversation rule checking, the SMV input file shown in Figure D.1 is used. This input file contains the Kripke states definition (lines 2-6), the temporal logic forms of the conversation rules (lines 9-27), and the Kripke relation, which is defined using a case structure (lines 31-48).

The SMV file for verification against the transactional requirements of Table 6.1 is split between Figures D.2 and D.3. Figure D.2 shows the definition of Kripke structure states and temporal logic implementations of the transactional requirements, whereas Figure D.3 shows the Kripke relation.



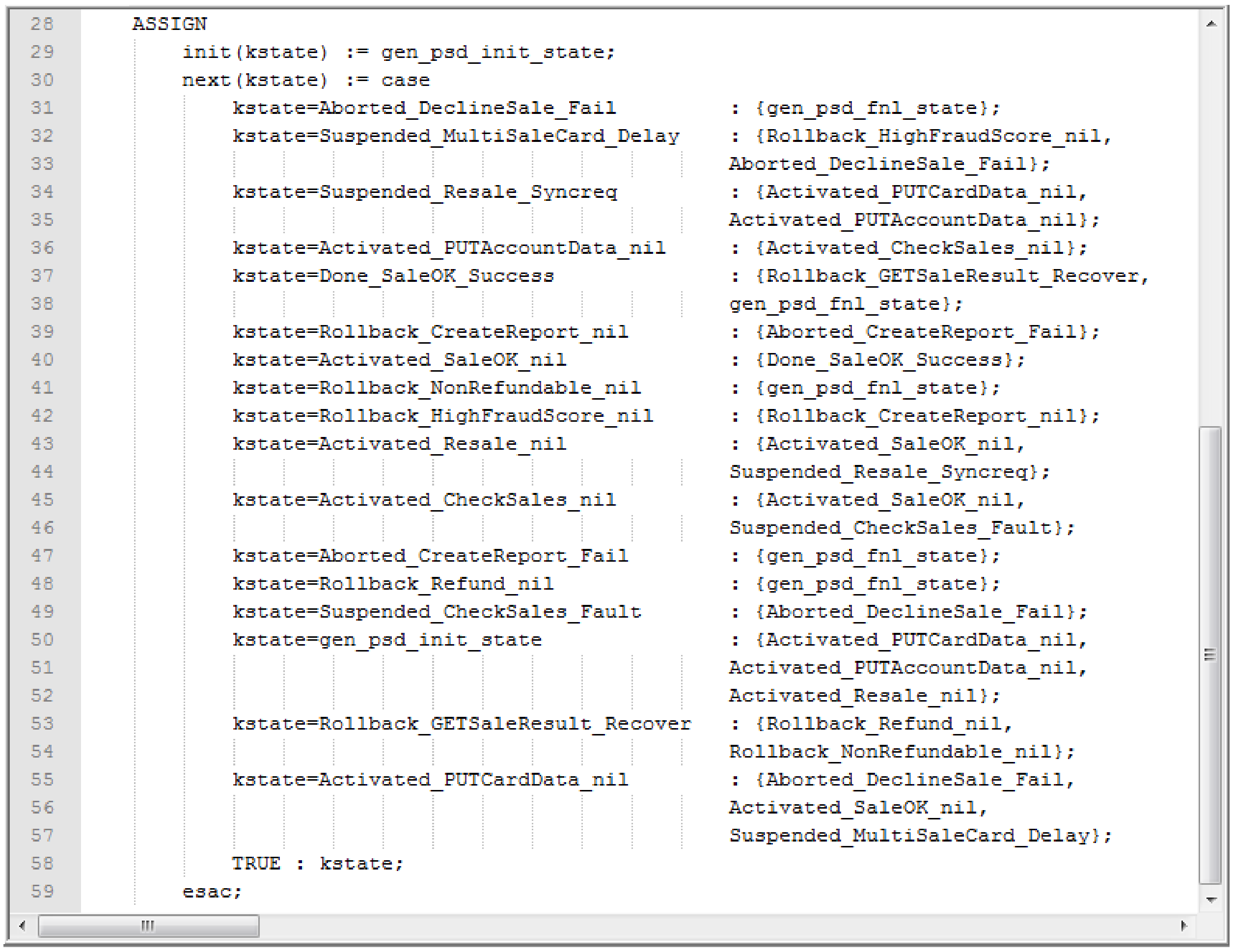
```
MODULE main
    VAR     state: {Syncreq_PUTCardData, Success_CommitPayment,
    Recover_PUTCardData, Fault_FinancialInstitution, gen_psd_fnl_state,
    Sync_CardAuthorization, Fault_CardAuthorization,
    Recover_FinancialInstitution, Fail_PaymentFailed,
    Sync_PUTCustomerDetails, gen_psd_init_state};


LTLSPEC state=gen_psd_init_state U ( state=Sync_CardAuthorization
    | state=Sync_PUTCustomerDetails )
SPEC AG ( EF state=gen_psd_fnl_state )
LTLSPEC G (( state=Sync_CardAuthorization | state=Sync_PUTCustomerDetails
    | state=Recover_PUTCardData | state=Recover_FinancialInstitution
    | state=Fault_FinancialInstitution | state=Fault_CardAuthorization )
    -> X( state!=gen_psd_fnl_state ))
LTLSPEC G (( state=Sync_CardAuthorization | state=Sync_PUTCustomerDetails
    | state=Recover_PUTCardData | state=Recover_FinancialInstitution )
    -> X( state=Success_CommitPayment | state=Fail_PaymentFailed
    | state=Fault_FinancialInstitution | state=Fault_CardAuthorization
    | state=Syncreq_PUTCardData ))
LTLSPEC G (( state=Syncreq_PUTCardData ) -> X( state=Sync_CardAuthorization
    | state=Sync_PUTCustomerDetails ))
LTLSPEC G (( state=Fault_FinancialInstitution | state=Fault_CardAuthorization )
    -> X( state=Recover_PUTCardData | state=Recover_FinancialInstitution ))
LTLSPEC G (( state=Success_CommitPayment ) -> X( state=gen_psd_fnl_state
    | state=Recover_PUTCardData | state=Recover_FinancialInstitution ))
LTLSPEC G (( state=Fail_PaymentFailed ) -> X( state=gen_psd_fnl_state ))


    ASSIGN
        init(state) := gen_psd_init_state;
        next(state) := case
            state = Syncreq_PUTCardData : { Sync_CardAuthorization };
            state = Success_CommitPayment : { gen_psd_fnl_state };
            state = Recover_PUTCardData : { Syncreq_PUTCardData };
            state = Fault_FinancialInstitution : { Recover_FinancialInstitution };
            state = Sync_CardAuthorization : { Success_CommitPayment,
                                            Fault_FinancialInstitution,
                                            Fault_CardAuthorization };
            state = Fault_CardAuthorization : { Recover_PUTCardData };
            state = Recover_FinancialInstitution : { Fail_PaymentFailed };
            state = Fail_PaymentFailed : { gen_psd_fnl_state };
            state = Sync_PUTCustomerDetails : { Success_CommitPayment,
                                            Fault_FinancialInstitution,
                                            Fault_CardAuthorization };
            state = gen_psd_init_state : { Sync_PUTCustomerDetails };

        TRUE : state;
    esac;
```

Figure D.1: Conversation rules and Kripke structure formulated in an SMV input file for verifying the online payment process



```
MODULE main
    VAR
        kstate : {Aborted_DeclineSale_Fail, Suspended_MultiSaleCard_Delay,
        Suspended_Resale_Syncreq, Activated_PUTAccountData_nil, Done_SaleOK_Success,
        Rollback_CreateReport_nil, Activated_SaleOK_nil, Rollback_NonRefundable_nil,
        Rollback_HighFraudScore_nil, Activated_Resale_nil, Activated_CheckSales_nil,
        Aborted_CreateReport_Fail, Rollback_Refund_nil, Suspended_CheckSales_Fault,
        gen_psd_init_state, Rollback_GETSaleResult_Recover, gen_psd_fnl_state,
        Activated_PUTCardData_nil};


SPEC AG(kstate=Suspended_Resale_Syncreq -> EF(kstate=Done_SaleOK_Success))
SPEC AG(kstate=Suspended_MultiSaleCard_Delay -> EF(kstate=Done_SaleOK_Success))
LTLSPEC G((kstate=Activated_CheckSales_nil & X(kstate!=Suspended_CheckSales_Fault))
    -> F(kstate=Done_SaleOK_Success))
LTLSPEC G((kstate!=Activated_SaleOK_nil & O(kstate=Activated_SaleOK_nil)) ->
    X(kstate!=Activated_SaleOK_nil)) & G((kstate=Done_SaleOK_Success &
    O(kstate=Activated_SaleOK_nil) & F(kstate=Rollback_GETSaleResult_Recover))
    -> F(kstate=Rollback_Refund_nil | kstate=Rollback_NonRefundable_nil))
LTLSPEC   G((kstate=Suspended_Resale_Syncreq) -> (((kstate!=Activated_Resale_nil)
    U (kstate=Activated_PUTAccountData_nil | kstate=Activated_PUTCardData_nil))
    & F(kstate=Activated_Resale_nil | kstate=Activated_PUTAccountData_nil
    | kstate=Activated_PUTCardData_nil)))
LTLSPEC  F(kstate=Rollback_HighFraudScore_nil) ->  G((kstate=Aborted_DeclineSale_Fail
    | kstate=Aborted_CreateReport_Fail) -> O((kstate=Rollback_CreateReport_nil)))
```

Figure D.2: Kripke structure definition and transactional requirements of the online payment process



```
ASSIGN
    init(kstate) := gen_psd_init_state;
    next(kstate) := case
        kstate=Aborted_DeclineSale_Fail          : {gen_psd_fnl_state};
        kstate=Suspended_MultiSaleCard_Delay     : {Rollback_HighFraudScore_nil,
                                                   Aborted_DeclineSale_Fail};
        kstate=Suspended_Resale_Syncreq          : {Activated_PUTCardData_nil,
                                                   Activated_PUTAccountData_nil};
        kstate=Activated_PUTAccountData_nil      : {Activated_CheckSales_nil};
        kstate=Done_SaleOK_Success               : {Rollback_GETSaleResult_Recover,
                                                   gen_psd_fnl_state};
        kstate=Rollback_CreateReport_nil         : {Aborted_CreateReport_Fail};
        kstate=Activated_SaleOK_nil              : {Done_SaleOK_Success};
        kstate=Rollback_NonRefundable_nil        : {gen_psd_fnl_state};
        kstate=Rollback_HighFraudScore_nil       : {Rollback_CreateReport_nil};
        kstate=Activated_Resale_nil              : {Activated_SaleOK_nil,
                                                   Suspended_Resale_Syncreq};
        kstate=Activated_CheckSales_nil          : {Activated_SaleOK_nil,
                                                   Suspended_CheckSales_Fault};
        kstate=Aborted_CreateReport_Fail         : {gen_psd_fnl_state};
        kstate=Rollback_Refund_nil               : {gen_psd_fnl_state};
        kstate=Suspended_CheckSales_Fault        : {Aborted_DeclineSale_Fail};
        kstate=gen_psd_init_state                : {Activated_PUTCardData_nil,
                                                   Activated_PUTAccountData_nil,
                                                   Activated_Resale_nil};
        kstate=Rollback_GETSaleResult_Recover    : {Rollback_Refund_nil,
                                                   Rollback_NonRefundable_nil};
        kstate=Activated_PUTCardData_nil         : {Aborted_DeclineSale_Fail,
                                                   Activated_SaleOK_nil,
                                                   Suspended_MultiSaleCard_Delay};
        TRUE : kstate;
    esac;
```

Figure D.3: Kripke structure relation of the online payment process

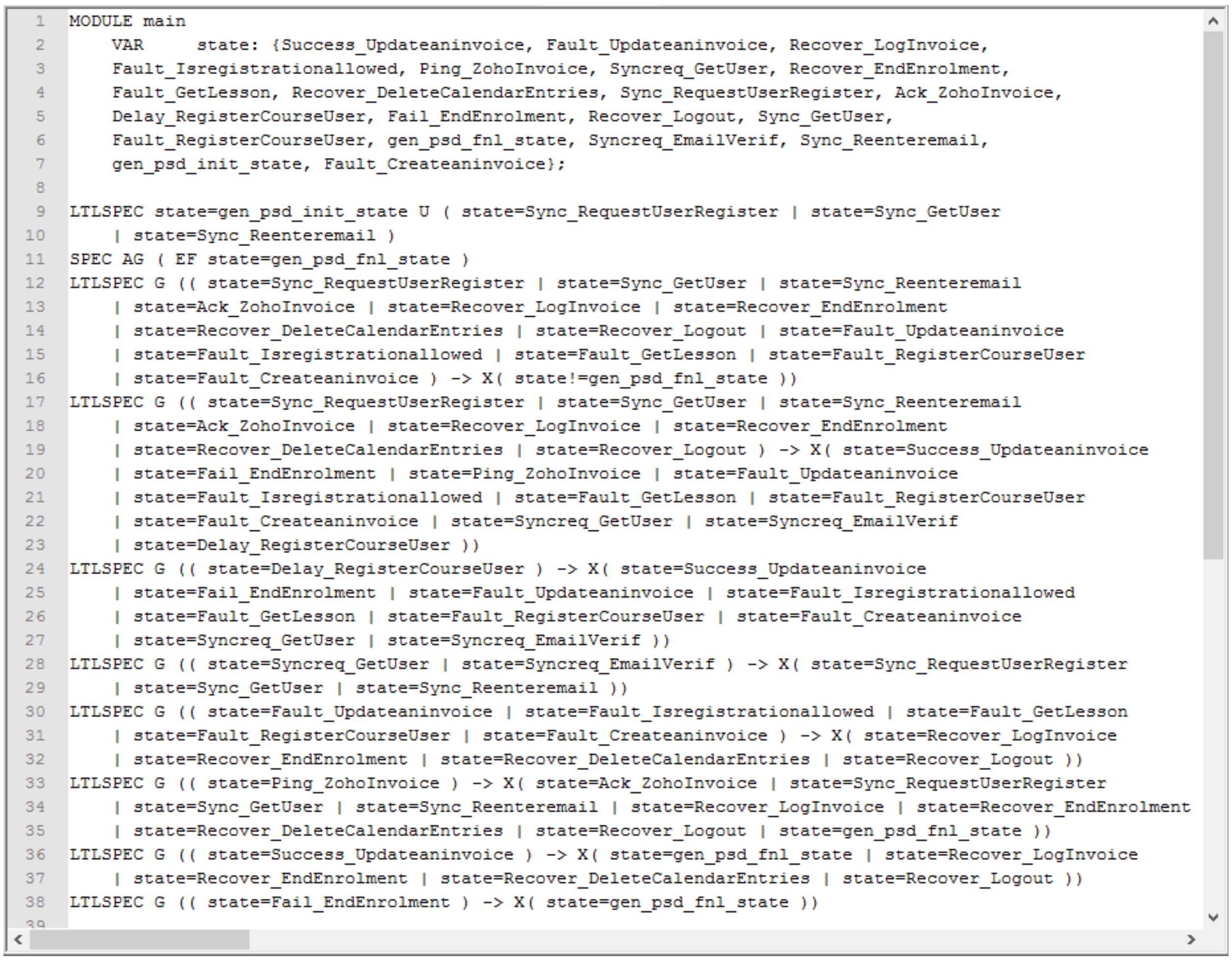
APPENDIX E

Implementation of Course Enrolment Scenario

This course enrolment service-oriented process was defined in Chapter 6 as a validation scenario for conversation rule checking and transactional requirement verification. The operational behavior model and inter-behavior messages of this process can be found in Figure 6.5 and Table 6.2 respectively. Details of the validation scenario for this process are in Chapter 6.2.1.2.

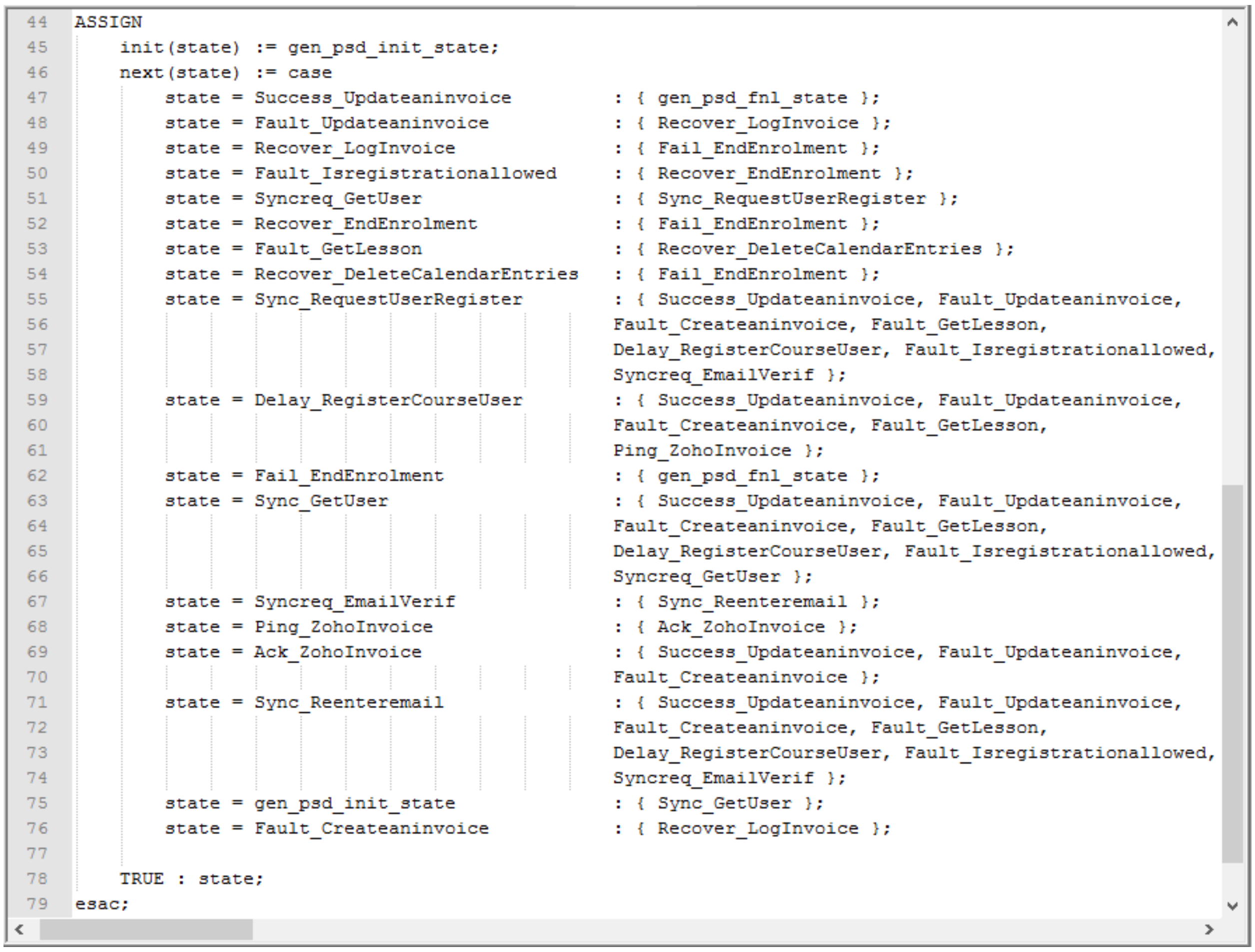
The SMV input file for conversation checking is split between Figure E.1 and E.2. Figure E.1 contains the definition of Kripke states and temporal logic properties, while the Kripke relation is shown in Figure E.2.

Figures E.3 and E.4 contain the SMV input file for transactional requirement verification. The Kripke states definition and temporal logic properties of the transactional requirements are in Figures E.3. Figure E.4 contains the Kripke transition relation, which is defined using a *case* structure.



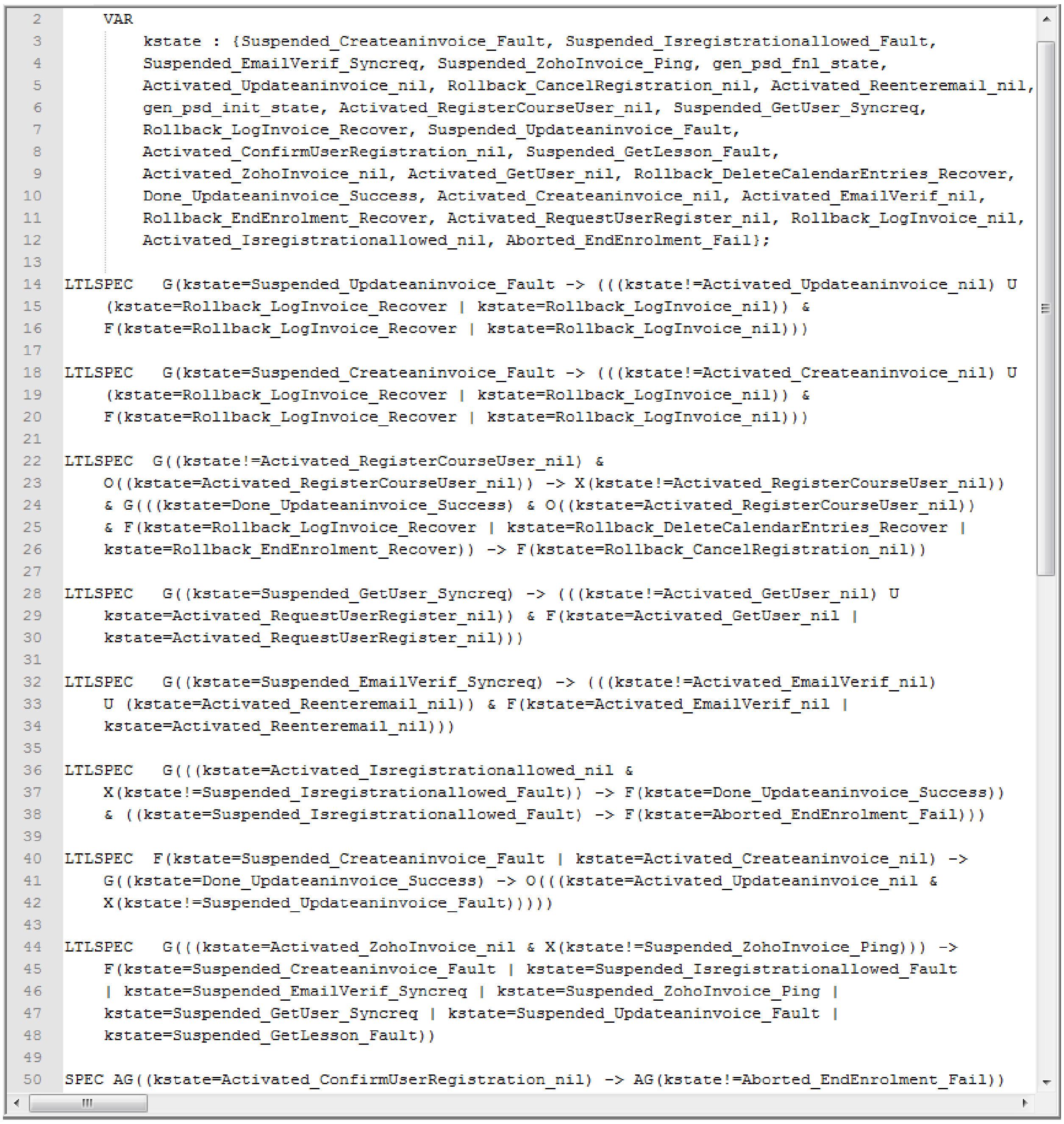
```
MODULE main
    VAR     state: {Success_Updateaninvoice, Fault_Updateaninvoice, Recover_LogInvoice,
    Fault_Isregistrationallowed, Ping_ZohoInvoice, Syncreq_GetUser, Recover_EndEnrolment,
    Fault_GetLesson, Recover_DeleteCalendarEntries, Sync_RequestUserRegister, Ack_ZohoInvoice,
    Delay_RegisterCourseUser, Fail_EndEnrolment, Recover_Logout, Sync_GetUser,
    Fault_RegisterCourseUser, gen_psd_fnl_state, Syncreq_EmailVerif, Sync_Reenteremail,
    gen_psd_init_state, Fault_Createaninvoice};

LTLSPEC state=gen_psd_init_state U ( state=Sync_RequestUserRegister | state=Sync_GetUser
    | state=Sync_Reenteremail )
SPEC AG ( EF state=gen_psd_fnl_state )
LTLSPEC G (( state=Sync_RequestUserRegister | state=Sync_GetUser | state=Sync_Reenteremail
    | state=Ack_ZohoInvoice | state=Recover_LogInvoice | state=Recover_EndEnrolment
    | state=Recover_DeleteCalendarEntries | state=Recover_Logout | state=Fault_Updateaninvoice
    | state=Fault_Isregistrationallowed | state=Fault_GetLesson | state=Fault_RegisterCourseUser
    | state=Fault_Createaninvoice ) -> X( state!=gen_psd_fnl_state ))
LTLSPEC G (( state=Sync_RequestUserRegister | state=Sync_GetUser | state=Sync_Reenteremail
    | state=Ack_ZohoInvoice | state=Recover_LogInvoice | state=Recover_EndEnrolment
    | state=Recover_DeleteCalendarEntries | state=Recover_Logout ) -> X( state=Success_Updateaninvoice
    | state=Fail_EndEnrolment | state=Ping_ZohoInvoice | state=Fault_Updateaninvoice
    | state=Fault_Isregistrationallowed | state=Fault_GetLesson | state=Fault_RegisterCourseUser
    | state=Fault_Createaninvoice | state=Syncreq_GetUser | state=Syncreq_EmailVerif
    | state=Delay_RegisterCourseUser ))
LTLSPEC G (( state=Delay_RegisterCourseUser ) -> X( state=Success_Updateaninvoice
    | state=Fail_EndEnrolment | state=Fault_Updateaninvoice | state=Fault_Isregistrationallowed
    | state=Fault_GetLesson | state=Fault_RegisterCourseUser | state=Fault_Createaninvoice
    | state=Syncreq_GetUser | state=Syncreq_EmailVerif ))
LTLSPEC G (( state=Syncreq_GetUser | state=Syncreq_EmailVerif ) -> X( state=Sync_RequestUserRegister
    | state=Sync_GetUser | state=Sync_Reenteremail ))
LTLSPEC G (( state=Fault_Updateaninvoice | state=Fault_Isregistrationallowed | state=Fault_GetLesson
    | state=Fault_RegisterCourseUser | state=Fault_Createaninvoice ) -> X( state=Recover_LogInvoice
    | state=Recover_EndEnrolment | state=Recover_DeleteCalendarEntries | state=Recover_Logout ))
LTLSPEC G (( state=Ping_ZohoInvoice ) -> X( state=Ack_ZohoInvoice | state=Sync_RequestUserRegister
    | state=Sync_GetUser | state=Sync_Reenteremail | state=Recover_LogInvoice | state=Recover_EndEnrolment
    | state=Recover_DeleteCalendarEntries | state=Recover_Logout | state=gen_psd_fnl_state ))
LTLSPEC G (( state=Success_Updateaninvoice ) -> X( state=gen_psd_fnl_state | state=Recover_LogInvoice
    | state=Recover_EndEnrolment | state=Recover_DeleteCalendarEntries | state=Recover_Logout ))
LTLSPEC G (( state=Fail_EndEnrolment ) -> X( state=gen_psd_fnl_state ))
```

Figure E.1: NuSMV input file containing the temporal logic properties for verifying the course enrolment process against conversation rules



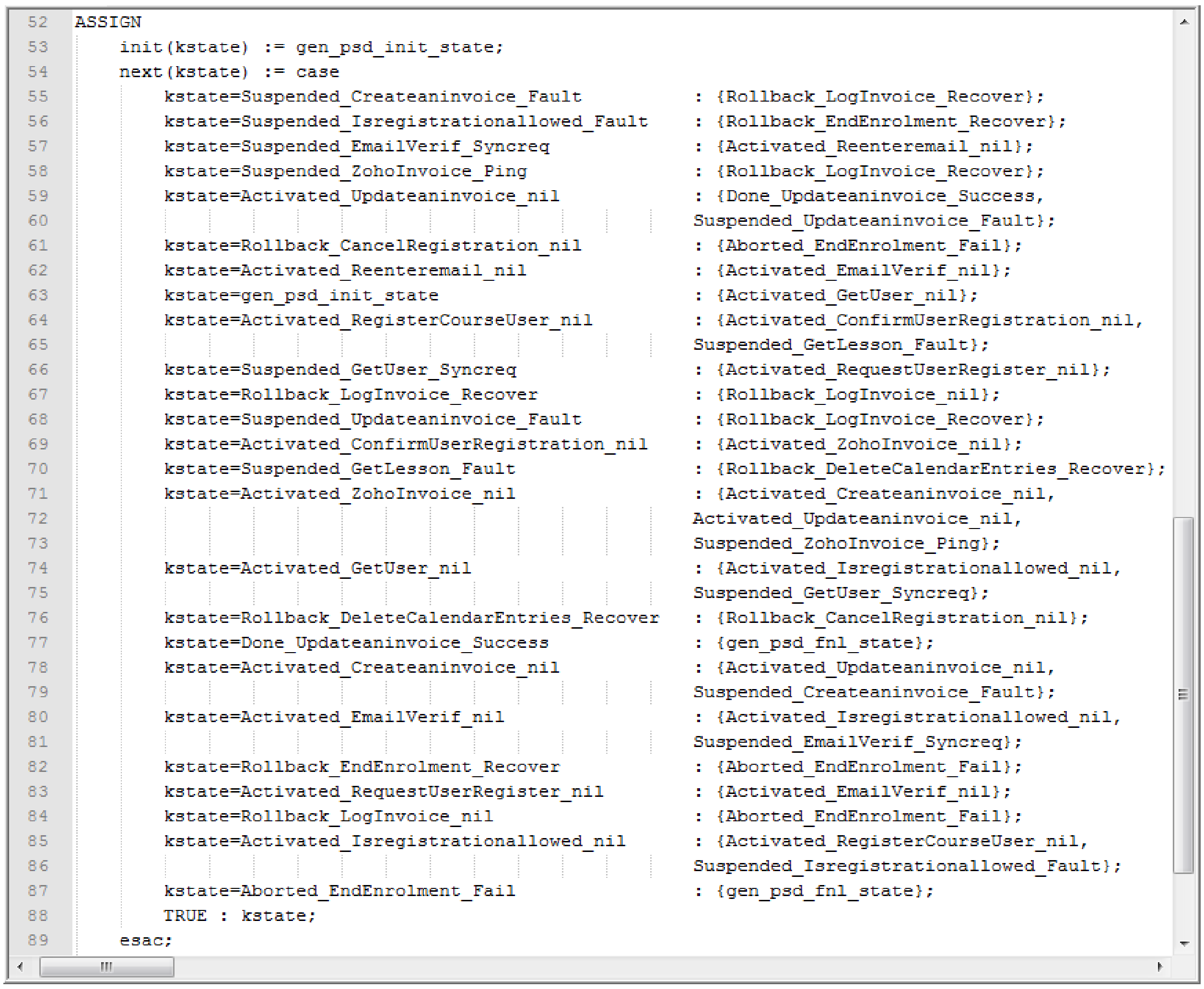
```
ASSIGN
    init(state) := gen_psd_init_state;
    next(state) := case
        state = Success_Updateaninvoice         : { gen_psd_fnl_state };
        state = Fault_Updateaninvoice           : { Recover_LogInvoice };
        state = Recover_LogInvoice              : { Fail_EndEnrolment };
        state = Fault_Isregistrationallowed     : { Recover_EndEnrolment };
        state = Syncreq_GetUser                 : { Sync_RequestUserRegister };
        state = Recover_EndEnrolment            : { Fail_EndEnrolment };
        state = Fault_GetLesson                 : { Recover_DeleteCalendarEntries };
        state = Recover_DeleteCalendarEntries   : { Fail_EndEnrolment };
        state = Sync_RequestUserRegister        : { Success_Updateaninvoice, Fault_Updateaninvoice,
                                                Fault_Createaninvoice, Fault_GetLesson,
                                                Delay_RegisterCourseUser, Fault_Isregistrationallowed,
                                                Syncreq_EmailVerif };
        state = Delay_RegisterCourseUser        : { Success_Updateaninvoice, Fault_Updateaninvoice,
                                                Fault_Createaninvoice, Fault_GetLesson,
                                                Ping_ZohoInvoice };
        state = Fail_EndEnrolment               : { gen_psd_fnl_state };
        state = Sync_GetUser                    : { Success_Updateaninvoice, Fault_Updateaninvoice,
                                                Fault_Createaninvoice, Fault_GetLesson,
                                                Delay_RegisterCourseUser, Fault_Isregistrationallowed,
                                                Syncreq_GetUser };
        state = Syncreq_EmailVerif              : { Sync_Reenteremail };
        state = Ping_ZohoInvoice                : { Ack_ZohoInvoice };
        state = Ack_ZohoInvoice                 : { Success_Updateaninvoice, Fault_Updateaninvoice,
                                                Fault_Createaninvoice };
        state = Sync_Reenteremail               : { Success_Updateaninvoice, Fault_Updateaninvoice,
                                                Fault_Createaninvoice, Fault_GetLesson,
                                                Delay_RegisterCourseUser, Fault_Isregistrationallowed,
                                                Syncreq_EmailVerif };
        state = gen_psd_init_state              : { Sync_GetUser };
        state = Fault_Createaninvoice           : { Recover_LogInvoice };

    TRUE : state;
esac;
```

Figure E.2: NuSMV input file containing the Kripke structure for verifying the course enrolment process against conversation rules



```
    VAR
        kstate : {Suspended_Createaninvoice_Fault, Suspended_Isregistrationallowed_Fault,
        Suspended_EmailVerif_Syncreq, Suspended_ZohoInvoice_Ping, gen_psd_fnl_state,
        Activated_Updateaninvoice_nil, Rollback_CancelRegistration_nil, Activated_Reenteremail_nil,
        gen_psd_init_state, Activated_RegisterCourseUser_nil, Suspended_GetUser_Syncreq,
        Rollback_LogInvoice_Recover, Suspended_Updateaninvoice_Fault,
        Activated_ConfirmUserRegistration_nil, Suspended_GetLesson_Fault,
        Activated_ZohoInvoice_nil, Activated_GetUser_nil, Rollback_DeleteCalendarEntries_Recover,
        Done_Updateaninvoice_Success, Activated_Createaninvoice_nil, Activated_EmailVerif_nil,
        Rollback_EndEnrolment_Recover, Activated_RequestUserRegister_nil, Rollback_LogInvoice_nil,
        Activated_Isregistrationallowed_nil, Aborted_EndEnrolment_Fail};

LTLSPEC   G(kstate=Suspended_Updateaninvoice_Fault -> (((kstate!=Activated_Updateaninvoice_nil) U
    (kstate=Rollback_LogInvoice_Recover | kstate=Rollback_LogInvoice_nil)) &
    F(kstate=Rollback_LogInvoice_Recover | kstate=Rollback_LogInvoice_nil)))

LTLSPEC   G(kstate=Suspended_Createaninvoice_Fault -> (((kstate!=Activated_Createaninvoice_nil) U
    (kstate=Rollback_LogInvoice_Recover | kstate=Rollback_LogInvoice_nil)) &
    F(kstate=Rollback_LogInvoice_Recover | kstate=Rollback_LogInvoice_nil)))

LTLSPEC  G((kstate!=Activated_RegisterCourseUser_nil) &
    O((kstate=Activated_RegisterCourseUser_nil)) -> X(kstate!=Activated_RegisterCourseUser_nil))
    & G(((kstate=Done_Updateaninvoice_Success) & O((kstate=Activated_RegisterCourseUser_nil))
    & F(kstate=Rollback_LogInvoice_Recover | kstate=Rollback_DeleteCalendarEntries_Recover |
    kstate=Rollback_EndEnrolment_Recover)) -> F(kstate=Rollback_CancelRegistration_nil))

LTLSPEC   G((kstate=Suspended_GetUser_Syncreq) -> (((kstate!=Activated_GetUser_nil) U
    kstate=Activated_RequestUserRegister_nil)) & F(kstate=Activated_GetUser_nil |
    kstate=Activated_RequestUserRegister_nil)))

LTLSPEC   G((kstate=Suspended_EmailVerif_Syncreq) -> (((kstate!=Activated_EmailVerif_nil)
    U (kstate=Activated_Reenteremail_nil)) & F(kstate=Activated_EmailVerif_nil |
    kstate=Activated_Reenteremail_nil)))

LTLSPEC   G(((kstate=Activated_Isregistrationallowed_nil &
    X(kstate!=Suspended_Isregistrationallowed_Fault)) -> F(kstate=Done_Updateaninvoice_Success))
    & ((kstate=Suspended_Isregistrationallowed_Fault) -> F(kstate=Aborted_EndEnrolment_Fail)))

LTLSPEC  F(kstate=Suspended_Createaninvoice_Fault | kstate=Activated_Createaninvoice_nil) ->
    G((kstate=Done_Updateaninvoice_Success) -> O(((kstate=Activated_Updateaninvoice_nil &
    X(kstate!=Suspended_Updateaninvoice_Fault)))))

LTLSPEC   G(((kstate=Activated_ZohoInvoice_nil & X(kstate!=Suspended_ZohoInvoice_Ping))) ->
    F(kstate=Suspended_Createaninvoice_Fault | kstate=Suspended_Isregistrationallowed_Fault
    | kstate=Suspended_EmailVerif_Syncreq | kstate=Suspended_ZohoInvoice_Ping |
    kstate=Suspended_GetUser_Syncreq | kstate=Suspended_Updateaninvoice_Fault |
    kstate=Suspended_GetLesson_Fault))

SPEC AG((kstate=Activated_ConfirmUserRegistration_nil) -> AG(kstate!=Aborted_EndEnrolment_Fail))
```

Figure E.3: NuSMV input file containing the course enrolment transactional requirements



```
ASSIGN
    init(kstate) := gen_psd_init_state;
    next(kstate) := case
        kstate=Suspended_Createaninvoice_Fault          : {Rollback_LogInvoice_Recover};
        kstate=Suspended_Isregistrationallowed_Fault    : {Rollback_EndEnrolment_Recover};
        kstate=Suspended_EmailVerif_Syncreq             : {Activated_Reenteremail_nil};
        kstate=Suspended_ZohoInvoice_Ping               : {Rollback_LogInvoice_Recover};
        kstate=Activated_Updateaninvoice_nil            : {Done_Updateaninvoice_Success,
                                                        Suspended_Updateaninvoice_Fault};
        kstate=Rollback_CancelRegistration_nil          : {Aborted_EndEnrolment_Fail};
        kstate=Activated_Reenteremail_nil               : {Activated_EmailVerif_nil};
        kstate=gen_psd_init_state                       : {Activated_GetUser_nil};
        kstate=Activated_RegisterCourseUser_nil         : {Activated_ConfirmUserRegistration_nil,
                                                        Suspended_GetLesson_Fault};
        kstate=Suspended_GetUser_Syncreq                : {Activated_RequestUserRegister_nil};
        kstate=Rollback_LogInvoice_Recover              : {Rollback_LogInvoice_nil};
        kstate=Suspended_Updateaninvoice_Fault          : {Rollback_LogInvoice_Recover};
        kstate=Activated_ConfirmUserRegistration_nil    : {Activated_ZohoInvoice_nil};
        kstate=Suspended_GetLesson_Fault                : {Rollback_DeleteCalendarEntries_Recover};
        kstate=Activated_ZohoInvoice_nil                : {Activated_Createaninvoice_nil,
                                                        Activated_Updateaninvoice_nil,
                                                        Suspended_ZohoInvoice_Ping};
        kstate=Activated_GetUser_nil                    : {Activated_Isregistrationallowed_nil,
                                                        Suspended_GetUser_Syncreq};
        kstate=Rollback_DeleteCalendarEntries_Recover   : {Rollback_CancelRegistration_nil};
        kstate=Done_Updateaninvoice_Success             : {gen_psd_fnl_state};
        kstate=Activated_Createaninvoice_nil            : {Activated_Updateaninvoice_nil,
                                                        Suspended_Createaninvoice_Fault};
        kstate=Activated_EmailVerif_nil                 : {Activated_Isregistrationallowed_nil,
                                                        Suspended_EmailVerif_Syncreq};
        kstate=Rollback_EndEnrolment_Recover            : {Aborted_EndEnrolment_Fail};
        kstate=Activated_RequestUserRegister_nil        : {Activated_EmailVerif_nil};
        kstate=Rollback_LogInvoice_nil                  : {Aborted_EndEnrolment_Fail};
        kstate=Activated_Isregistrationallowed_nil      : {Activated_RegisterCourseUser_nil,
                                                        Suspended_Isregistrationallowed_Fault};
        kstate=Aborted_EndEnrolment_Fail                : {gen_psd_fnl_state};
        TRUE : kstate;
    esac;
```

Figure E.4: NuSMV input file containing the course enrolment Kripke structure relation