



universität
wien

DISSERTATION / DOCTORAL THESIS

Titel der Dissertation / Title of the Doctoral Thesis

Supporting Automated Containment Checking of Software
Behavioural Models

verfasst von / submitted by

Faiz UL Muram, MSc

angestrebter akademischer Grad / in partial fulfilment of the requirements for the degree of

Doktorin der technischen Wissenschaften (Dr. techn.)

Wien / Vienna, 2017

Studienkennzahl lt. Studienblatt / degree programme

code as it appears on the student record sheet:

A 786 880

Dissertationsgebiet lt. Studienblatt / field of study as

it appears on the student record sheet:

Informatik

Betreuer / Supervisor:

Univ.-Prof. Dr. Uwe Zdun

Abstract

A major challenge in software development processes is to detect and fix the deviations of system's intended behaviours at different abstraction levels in early phases. Inconsistencies that are detected at later phases, when the system is already implemented or tested, require huge amounts of time and effort for correction, revision, and verification. Therefore, it is crucial to detect and fix the inconsistencies at early phases of software development, and especially as soon as refined models deviate from their abstract counterparts. This dissertation focuses on a special type of vertical consistency, in particular containment checking that verifies whether the behaviour described by the low-level model encompasses those specified in the high-level counterpart. Previous research has not investigated the containment relationship for behavioural models.

We have performed a systematic review of software behavioural model consistency checking research, and identified a number of gaps and open problems that serve as a foundation for this dissertation. The major contributions of this dissertation are novel concepts and techniques for automatic containment checking of software behaviour models. Containment checking of software behaviour models, in particular activity models, sequence models and service choreographies, is supported using model transformations and model checking. Specifically, the automated transformation of behaviour models into formal specifications and consistency constraints is performed; they are required by model checkers for detecting any discrepancies between the input models and yielding corresponding counterexamples. However, the feedback of model checkers is rather not helpful for users with limited background on the underlying formal methods to analyse and understand the causes of consistency violations. A counterexample analysis approach is therefore proposed for locating the cause(s) of containment violations and presenting appropriate suggestions to stakeholders for their resolution. Dealing with unconditional loops and parallel execution branches are challenging issues in model checking based techniques. Therefore, we introduced, in addition to the model checking based techniques, a lightweight graph-based approach that verifies missing nodes, missing transitive links, and missing cycles.

We have investigated containment checking first generically for the named design models and studied the application in realistic use case scenarios, taken mainly from enterprise information systems. As a second application domain, we have studied applying containment checking in the context of architectural patterns: Architectural patterns impose various kinds of design constraints on the detailed designs and implementations that should not be violated. We studied the application of containment checking for checking those design constraints for three popular architectural patterns: Model-View-Controller, Layers, and Pipe and Filter, as well as their variants. The quantitative evaluation of the realistic use case scenarios shows that the proposed approaches perform reasonably well in typical working environments of software developers and scale well enough for typical sizes of software design models.

Zusammenfassung

Während des Softwareentwicklungsprozesses ist es eine große Herausforderung, bereits in den frühen Phasen auf verschiedenen Abstraktionsebenen festzustellen, ob ein System von beabsichtigtem Verhalten abweicht und etwaige Abweichungen zu korrigieren. Werden Inkonsistenzen erst in späteren Phasen entdeckt, nachdem das System bereits implementiert oder getestet worden ist, führt dies zu enormem Aufwand für Korrektur, Überarbeitung und Verifikation. Es ist daher von großer Bedeutung, diese Inkonsistenzen bereits in einer frühen Phase des Softwareentwicklungsprozesses zu erkennen und zu beheben. Dies gilt vor allem, sobald verfeinerte Modelle von ihren abstrakten Gegenständen abweichen. Diese Dissertation konzentriert sich auf eine spezielle Art von vertikaler Konsistenz, insbesondere Containment Checking. Dabei wird überprüft, ob das vom Low-Level-Modell beschriebene Verhalten auch das im High-Level Gegenstück beschriebene erfasst. Die Containment-Beziehung von Verhaltensmodellen wurde in bereits existierenden Forschungsarbeiten bisher nicht untersucht.

Nach einer systematischen Überprüfung der vorhandenen Forschung zur Konsistenzprüfung von Software-Verhaltensmodellen konnten wir eine Reihe an Lücken und offenen Problemen ermitteln, die als Basis für diese Dissertation dienen. Der hauptsächliche Beitrag dieser Dissertation besteht aus neuen Konzepten und Techniken zum automatischen Containment Checking von Software-Verhaltensmodellen. Mittels Modelltransformationen und Modellprüfung (Model Checking) wird Containment Checking von Software-Verhaltensmodellen, insbesondere Activity Models, Sequence Models und Service Choreography unterstützt. Genauer gesagt wird eine automatische Transformation von Verhaltensmodellen in formale Spezifikationen und Konsistenzbeschränkungen durchgeführt. Diese werden beim Model Checking für das Erkennen von Abweichungen zwischen den Eingabemodellen und daraus folgende Gegenbeispiele benötigt. Das Feedback von Model Checkern ist jedoch für Benutzer mit eingeschränktem Wissen über die zugrundeliegenden formalen Methoden zur Analyse von beziehungsweise dem Verständnis für Konsistenzverstöße meist wenig hilfreich. Wir schlagen daher einen Ansatz vor, der die Analyse von Gegenbeispielen verwendet, um den Grund von Konsistenzverstößen zu finden und den jeweiligen Akteuren passende Lösungsvorschläge zu präsentieren. Endlosschleifen und parallel ausgeführte Programmteile stellen Techniken, die auf Model Checking basieren, vor große Herausforderungen. Wir haben daher zusätzlich einen Ansatz entwickelt, der basierend auf Graphen fehlende Knoten, fehlende transitive Verbindungen und fehlende Abläufe überprüft.

Zunächst wurde Containment Checking allgemein nach benannten Designmodellen untersucht und die Anwendung in realistischen Use-Case-Szenarien geprüft, wobei letztere überwiegend aus Enterprise Informationssystemen entnommen wurden. Als einen zweiten Anwendungsbereich untersuchten wir die Anwendung von Containment Checking im Kontext von Architekturmustern.

Diese legen verschiedene Arten von Designbedingungen für den Detailentwurf und für die Implementierung fest, die nicht verletzt werden dürfen. Wir untersuchten außerdem die Anwendung von Containment Checking für das Prüfen von Designbedingungen bei drei beliebten Architekturmustern: Model-View-Controller, Layers und Pipe and Filter, sowie deren Varianten. Die quantitative Evaluierung der realistischen Anwendungsszenarien zeigt, dass die vorgeschlagenen Ansätze in typischen Arbeitsumgebungen für Softwareentwickler ziemlich gut funktionieren und gut genug für übliche Größen von Softwaredesign-Modellen skalieren.

Acknowledgements

First, I would like to thank Prof. Dr. Uwe Zdun for his trust, support, guidance and encouragement. I am very grateful that I have had the opportunity to conduct research under his supervision. He has strongly influenced my way of thinking and working. Next, I owe my gratitude to Dr. Huy Tran for his support, collaboration and encouragement throughout this dissertation, also for sharing his experience and giving me good advice. I also appreciate the fruitful discussions and supports from my colleagues at the Software Architecture Group. I would also like to thank the Ph.D. committee members for taking the time to read this dissertation. Many thanks go to the University of Vienna and Wiener Wissenschafts-, Forschungs- und Technologiefonds (WWTF) for supporting my research. Last but not least, I want to thank my family for their endless love, support and encouragement.

Contents

List of Figures	ix
List of Tables	xi
1 Introduction	1
1.1 Background	3
1.1.1 Types of Consistency Checking of Software Models	3
1.1.2 Temporal Logics	5
1.1.3 Model Checking	6
1.1.4 Graph Based Search	7
1.2 Publications	7
1.3 Research Methods	8
1.3.1 Design Science Research	9
1.3.2 Systematic Literature Review	9
1.3.3 Case Study	10
1.3.4 Formal Modelling	10
1.3.5 Experimentation	10
1.4 Problem Statements and Research Questions	11
1.5 Major Contributions and Overview of the Dissertation	13
2 Systematic Review of Software Behavioural Model Consistency Checking	19
2.1 Introduction	19
2.2 Systematic Literature Review Process	20
2.2.1 Planning the Review	21
2.2.2 Conducting the Review	24
2.3 Results	28
2.3.1 Historical Development (Q1)	28
2.3.2 Targeted Software Models (Q2.1)	29
2.3.3 Types of Consistency Checking (Q2.2)	30
2.3.4 Consistency Checking Techniques (Q2.3)	33
2.3.5 Inconsistency Handling (Q2.4)	35
2.3.6 Automation Support (Q2.5)	36
2.3.7 Types of Study and Evaluation (Q2.6)	38
2.3.8 Practical Impact (Q3)	42
2.4 Discussions	45
2.4.1 Limitations of the Existing Methods	45
2.4.2 Study Validity	47
2.5 Conclusions	49

3	Model Checking Based Containment Checking of UML Activity Diagrams	51
3.1	Introduction	51
3.2	Containment Checking Approach	54
3.2.1	Step 1: Generating Consistency Constraints from the High-Level Model	55
3.2.2	Step 2: Mapping a UML Activity Diagram into SMV Descriptions	61
3.2.3	Step 3: Containment Checking Using NuSMV Model Checker	72
3.3	Scenario from Industrial Case Study	72
3.3.1	Generating LTL Formulas from the High-Level Model	73
3.3.2	Generating SMV Descriptions from the Low-Level Model	75
3.3.3	Containment Checking Results	76
3.4	Performance Evaluation	78
3.5	Discussion	80
3.6	Related Work	83
3.6.1	Existing Approaches for Consistency Checking	83
3.6.2	Containment Checking	85
3.6.3	Mapping of Models into Formal Descriptions and Specification of Formal Constraints	86
3.6.4	Formalisations of Activity Diagram	87
3.7	Summary	90
4	Model Checking Based Containment Checking of UML Sequence Diagrams	91
4.1	Introduction	91
4.2	Containment Checking Approach for UML Sequence Diagrams	92
4.2.1	Automated Transformation of Sequence Diagrams into LTL and SMV Descriptions	92
4.2.2	Checking Containment between Sequence Diagrams via NuSMV	102
4.3	Evaluation	103
4.3.1	ATM System Scenario	103
4.3.2	Performance Evaluation	104
4.4	Related Work	106
4.5	Summary	106
5	Model Checking Based Containment Checking of Service Choreographies	109
5.1	Introduction	109
5.2	Motivation and Running Example	110
5.3	Approach	112
5.3.1	Generating LTL Constraints from Global Choreography Model	113
5.3.2	Generating SMV Descriptions from Local Choreography Models	116
5.3.3	Containment Checking Using NuSMV Model Checker	121
5.4	Evaluation	122
5.5	Related Work	123
5.6	Summary	124
6	Counterexample Analysis for Supporting Containment Checking	127
6.1	Introduction	127
6.2	Interpretation of Containment Inconsistencies	129
6.2.1	Counterexample Analyser for Locating Causes of Containment Inconsistencies	130
6.2.2	Visual Support for Understanding and Resolving Inconsistencies	133

6.2.3	Use Case from Industrial Case Study	133
6.3	Dealing with Containment Violations for Service Choreographies	136
6.4	Dealing with Containment Violations for Activity Diagrams	138
6.4.1	Locating Causes of Containment Inconsistencies Due to Misplacement of Elements	139
6.4.2	Counterexample Analysis Results for Loan Approval System	141
6.5	Dealing with Containment Violations for Sequence Diagrams	142
6.6	Related Work	142
6.6.1	Behaviour Model Consistency Checking	144
6.6.2	Generating and Analysing Counterexamples	145
6.7	Summary	146
7	Graph-Based Containment Checking of UML Activity Diagrams	147
7.1	Introduction	147
7.2	Approach	149
7.2.1	Activity Models and Check Models	150
	Proof Sketch:	152
7.2.2	Graph-Based Containment Checking	153
7.2.3	Theoretical Complexity Analysis	157
7.3	Quantitative Evaluation and Discussion	158
7.3.1	Quantitative Evaluation	158
7.3.2	Discussions	160
7.4	Related Work	161
7.5	Summary	163
8	Containment Checking of Behaviour in Architectural Patterns	165
8.1	Introduction	165
8.2	Approach Overview	167
8.2.1	Approach Details	168
8.3	Application of Our Approach to Architectural Patterns	172
8.3.1	Containment Checking in MODEL-VIEW-CONTROLLER	172
8.3.2	Containment Checking in LAYERS	177
8.3.3	Containment Checking in PIPE AND FILTER	179
8.4	Related Work	179
8.4.1	Modelling and Formalisation of Architectural Patterns	180
8.4.2	Behavioural Consistency Checking	181
8.5	Summary	182
9	Conclusions and Future Work	183
9.1	Research Questions and Answers	183
9.2	Major Contributions	186
9.3	Future Research Directions	187
	Appendices	191
A	Overview of Selected Primary Studies	191

B Selected Primary Studies	195
C Data Extraction: Data Items and Encoding	203
Bibliography	207

List of Figures

2.1	Overview of the Stages and Results of Our Search Process	26
2.2	Primary Studies per Year Grouped by Publishing Venues	29
2.3	Types of Studied Software Behavioural Models	30
2.4	Primary Studies per Year Grouped by Types of Consistency Checking	32
2.5	Levels of Tool Support per Year Distribution	39
2.6	Sum of Rigour Scores per Dimension	40
2.7	Distribution of the Collective Rigour Metrics R	41
2.8	Trend of Rigour Dimension per Year	41
2.9	Studies Tried in Industrial Settings	45
3.1	Overview of the Containment Checking Approach	54
3.2	Translation of Initial Nodes into SMV Descriptions	63
3.3	Generic Rules for Mapping UML Constructs to SMV Descriptions	63
3.4	Translation of Merge Nodes into SMV Descriptions	65
3.5	Translation of Decision Nodes into SMV Descriptions	66
3.6	Translation of Exception Handlers into SMV Descriptions	67
3.7	Translation of Interruptible Activity Region into SMV Descriptions	69
3.8	Translation of Activity Parameter Nodes into SMV Descriptions	70
3.9	Translation of Loops into SMV Descriptions	71
3.10	High-Level Activity Diagram of Loan Approval System	73
3.11	Low-Level Activity Diagram of Loan Approval System	75
4.1	Translation of Basic Sequence Diagram	95
4.2	Translation of Weak Sequencing Combined Fragment	96
4.3	Translation of Strict Sequencing Combined Fragment	97
4.4	Translation of Alternative Combined Fragment	98
4.5	Translation of Parallel Combined Fragment	99
4.6	Translation of Loop Combined Fragment	100
4.7	Translation of Option Combined Fragment	101
4.8	Translation of Break Combined Fragment	102
4.9	High-Level Sequence Diagram of ATM System	103
4.10	Low-Level Sequence Diagram of ATM System	104
5.1	Travel Booking System: Global Choreography Model	111
5.2	Travel Booking System: Local Choreography Models	112
5.3	Overview of the Containment Checking Approach	113
5.4	Generic Rules for Mapping BPMN Collaboration Constructs to SMV Descriptions	118
5.5	Translation of Exclusive Decision into SMV Descriptions	119

5.6	Translation of Exclusive Merge into SMV Descriptions	120
6.1	Overview of the Counterexample Analysis Approach	129
6.2	High-Level BPMN Model of the Billing Renewal Process	134
6.3	Visual Support for Understanding and Resolving Containment Violations in BPMN	136
6.4	Local Choreography Models After Running Counterexample Analysis	137
6.5	Send Signal Action Preceded by Different Action	139
6.6	Low-Level Activity Diagram of Loan Approval System After Running Counterex- ample Analysis	143
6.7	Low-Level ATM System After Running Counterexample Analysis	144
7.1	Overview of the Graph-Based Containment Checking Approach	149
7.2	An Illustrative Simplified Activity Model of a Travel Agency	151
7.3	The Corresponding Check Model of the Travel Agency Activity Model	153
7.4	Comparison of the Scalability of Two Approaches	160
8.1	Approach Overview	168
8.2	High-Level Model of Itinerary Management System	174
8.3	Feedback of Containment Results in the Low-Level Model	176
8.4	Modelling Layers Architecture Pattern Using Stereotypes	178
8.5	Modelling Pipe And Filter Pattern Using Stereotypes	180

List of Tables

1.1	Overview of Papers, Chapters and Corresponding Research Questions	15
2.1	Literature Search Dimensions and Keywords	23
2.2	Inclusion and Exclusion Criteria	25
2.3	Data Collection	27
2.4	Active Research Groups and Numbers of Studies	29
2.5	Types of Consistency Checking	31
2.6	Support for the Notion of Time in Consistency Checking	33
2.7	Semantic Domains	34
2.8	Consistency Checking Techniques	35
2.9	Degree of Inconsistency Handling	36
2.10	Automation Support for Specification Phase	37
2.11	Evidence of Tool Support for Consistency Checking	38
2.12	Types of Study and Evaluation	39
2.13	Top 10 Most Cited Studies	42
2.14	CASE/IDE Support and/or Integration	43
2.15	Levels of Empirical Evidence	44
3.1	Transformation Rules for Generating LTL Formulas for UML Activity Diagrams .	60
3.2	LTL Formulas Generated from the High-Level Loan Approval Activity Diagram .	74
3.3	Model Size and Translation Time of UML Activity Diagrams	79
3.4	Performance Evaluation Results for UML Activity Diagrams	80
3.5	Overview and Comparison of Related Work	88
4.1	Model Size and Translation Time of UML Sequence Diagrams	105
4.2	Performance Evaluation Results for UML Sequence Diagrams	106
5.1	LTL-Based Transformation Rules for BPMN Global Choreography Model	115
5.2	Model Size and Translation Time of Service Choreographies	122
5.3	Performance Evaluation Results for Service Choreographies	123
6.1	Tracking Back the Causes of Containment Violations and Relevant Countermeasures for BPMN Process Diagrams	132
6.2	Tracking Back the Causes of Violation Due to Misplacement of Elements and Pos- sible Countermeasures	141
7.1	Estimation of Theoretical Complexity of Graph-based Containment Checking . . .	157
7.2	Performance Evaluation and Comparison	159

A.1	Overview of 96 Selected Primary Studies	194
B.1	List of Selected Primary Studies	201
C.1	Evidence of Timing Support	203
C.2	Evidence of Inconsistency Handling (adapted from [SZ01])	203
C.3	Evidence of Automation Support	203
C.4	Evidence of Development Tool Support and Integration	203
C.5	Evidence of Tool Support for Consistency Checking	203
C.6	Level of Empirical Evidence (adapted from [Kit04; Alv+10])	204
C.7	Type of Study and Evaluation (adapted from [CA11])	204
C.8	Rigour (adapted from [IG11])	205

The development of a software system often goes through a number of different stages and iteration cycles, and each of them can introduce new elements or more detailed specifications of the system [Rup10]. In addition, software systems are constantly evolving. In many areas of software engineering, behavioural models are used to represent the behavioural aspects of a software system. Examples of behaviour models are UML (Unified Modelling Language) activity models, state machines, and sequence models [Gro11b], Simulink[®] Stateflow[®] [Mat15], the Business Process Model and Notation (BPMN) [Gro11a], the Business Process Execution Language (BPEL) [OAS07], and Event-driven Process Chains (EPC) [SN00], to name but a few. In the past decades, a substantial number of software engineering research works have been devoted to consistency checking between different models or multiple views of the models that are used in software development [MTZ17a].

An example is ensuring consistency over multiple abstraction levels: Many models are created as “high-level” models [SV06; Str05; Huz+05]. That is, they are mainly used to convey the core concepts or principles of the reality they represent in an abstract and/or concise way (e.g., requirements models or design models). In addition, technical or “low-level” models are often created as refinements of the high-level models with purposes such as providing a precise specification of the source code, executing the model (e.g., in a process engine, interpreter, or virtual machine), or generating executable code directly from the model, e.g., in model-driven software development (MDSD) [Fra02; SV06]. It is crucial that the overlapping parts of the high-level and low-level models are in sync with each other [Str05; SZ01; Huz+05].

Unfortunately, multiple models of the same system (or reality) are often drifting apart over time, and inconsistencies arise among them, when they are created by different stakeholders and evolved independently [Str05; SZ01; Huz+05]. For instance, high-level models might be changed according to new requirements, and low-level models are changed as the implementation is modified. If not each change is systematically propagated to all other models of the same system (or reality), the evolved models may include inconsistencies. We have performed a systematic review of software behavioural model consistency checking research, and identified that a predominance of primary studies concentrate on consistency checking. However, previous research has not investigated the

containment relationship at different levels of abstraction, which is categorized as vertical consistency. An unsatisfied containment relationship implies the deviation of the low-level descriptions from the corresponding high-level specifications and properties.

The major contributions of this dissertation are novel concepts and techniques for automatic containment checking of software behaviour models. Specifically, we have investigated the containment checking problem for activity models, sequence models and service choreographies at different levels of abstraction. In the context of model checking based containment checking, the automated transformation of high-level and low-level models into consistency constraints and formal descriptions are provided. The proposed translation strategies not only bridge the gap between manual specification of formal properties and consistency constraints, but also increase the usability of formal languages in practice.

The outcomes of particular transformations are fed to a model checker for detecting any discrepancies between the input models and yielding corresponding counterexamples. Because the produced results are rather cryptic and verbose, the counterexample analysis approach is proposed for locating the root causes of containment inconsistencies and producing appropriate guidelines as countermeasures. The technique supports users who have limited knowledge of the underlying formalisms, and therefore, are not proficient in analysing the cryptic and verbose counterexamples. By locating actual cause(s) of the inconsistency and providing the relevant countermeasures to alleviate the inconsistencies to the user, the approach significantly reduces the time of manually locating the causes of an inconsistency.

Although the containment checking can be realized based on model checking, but not always the model checking techniques are necessary for addressing the containment checking problem. Specifically, the unconditional loops and parallel execution branches are challenging issues that lead to the state explosion problem. A graph-based containment checking approach is therefore proposed that verifies missing nodes, missing transitive links, and missing cycles.

We have evaluated the containment checking of activity models, sequence models and service choreographies in realistic use case scenarios, taken mainly from enterprise information systems. The application of containment checking is also studied for MODEL-VIEW-CONTROLLER, LAYERS and PIPE AND FILTER patterns as well as their variants. The quantitative evaluation of the realistic use case scenarios demonstrates that the proposed approaches performs reasonably well on typical working environments and scales.

The rest of this chapter is organized as follows: Section 1.1 describes background information. Section 1.2 lists the scientific journal, conference, and workshop publications used in this dissertation. Section 1.3 presents the research methods used in this dissertation. Section 1.4 discusses the problem statements and research questions. Section 1.5 gives a brief introduction to each chapter in the dissertation.

1.1 Background

This section discusses the consistency checking types, temporal logics, model checking and graph-based search that are key concepts in this dissertation.

1.1.1 Types of Consistency Checking of Software Models

According to Spanoudakis and Zisman [SZ01], an inconsistency is described as “*a state in which two or more overlapping elements of different software models make assertions about aspects of the system they describe which are not jointly satisfiable*”. As Spanoudakis and Zisman [SZ01] stated, the problem of inconsistencies in software models have been a big concern of the software engineering community for a long time. As a system is often modelled from different viewpoints by different stakeholders, in different levels of abstraction and granularity [Fin+93; Boi+00], there may exist contradicting information [Eng+01; SZ01]. There are several definitions of consistency and its classification emerging in the literature [Eng+01; Huz+05; Str05; Usm+08; LMT09].

A typical consistency problem happens among different types of representations (e.g., models) of the same aspect of a software system. For instance, for describing the interactions among various objects, a UML sequence diagram and/or a communication diagram can be used with respect to a temporal or structural viewpoint, respectively [Eng+01]. Engels et al. [Eng+01] consider this type of consistency “*horizontal consistency*” whilst Huzar et al. [Huz+05] and Usman et al. [Usm+08] call it “*intra-model consistency*”.

Looking into another dimension, a software model can be refined or transformed into a richer form with more details. This scenario happens quite often in model-driven software development in which model transformations are extensively used to map a high-level, abstract model down to a lower level of abstraction [Fra02; SV06]. This type of consistency is named “*vertical consistency*” [Eng+01], “*inter-model consistency*” [Huz+05], or “*refinement*” [LMT09]. We note that the term “model” is interpreted rather differently by Huzar et al. [Huz+05] (as a set of diagrams) and Engels et al. [Eng+01] (as an umbrella term that embraces the meaning of a diagram) and, therefore, their definition of consistency types are not totally overlapping. Van Der Straeten [Str05] used the term “*evolution consistency*” to indicate the consistency between different versions of the same model, which fits to the category “vertical consistency” in [Eng+01].

In the context of this dissertation, we adopt the umbrella term “*model*” as proposed by Spanoudakis and Zisman [SZ01] and Engels et al. [Eng+01] and use the terms “model” and “diagram” interchangeably with the same meaning unless stated otherwise in case it is necessary to distinguish between these terms. As a result, the interpretation of *horizontal consistency* and *vertical consistency* based on this point of view is used in the dissertation.

In addition, we revealed that several approaches investigate the consistency of a single model itself for correctness, determinism, and so forth. These consistencies do not fit nicely into the two categories mentioned above. Inspired by the philosophical stance presented by Man and Van Gorp [MVG06] in categorizing model transformations, we consider the following definitions of consistency types for software models in this dissertation.

- *Endogenous consistency* indicates consistencies within the same model regarding the properties of itself.
- *Exogenous consistency* denotes consistencies between different models. It can be refined into two sub-categories: horizontal and vertical consistencies.
 - *Horizontal consistency* denotes consistencies among different kinds of models or a model against rules and constraints that are manually specified or derived from other artefacts.
 - *Vertical consistency* shows the consistency among models of the same type at different levels of abstraction. This also includes the consistency between a design model and a corresponding implementation.

Existing studies distinguish between syntactic and semantic consistencies [Eng+01]. *Syntactical consistency* aims to ensure that a model conforms to a predefined syntax specified by a certain meta-model or grammar [Eng+01]. This ensures the well-formedness of the model [Eng+01]. Syntactic (or structural) consistency checking has been extensively discussed in [Huz+05; Usm+08; LMT09].

As the main focus of our dissertation are behavioural models, we will investigate further into semantic consistency. *Semantic consistency* addresses the semantic compatibility of models, for instance, the same identifier that occurs in different models should refer to the same entity (i.e., have same meaning). Semantic consistency is stricter and often requires syntactical consistency beforehand. In the context of horizontal consistency, semantic requires models of different viewpoints to be semantically compatible with regards to the aspects of the system which are specified in both sub-models. For vertical consistency problems, semantic consistency requires that a refined model is semantically consistent with the model it refines. Semantic consistency depends very much on the underlying semantics of the models being used and of the development process. Eshuis and Grefen [EG07] also observed that several model structures are described by different syntaxes but represent the same behaviour. Therefore, the identification of a suitable *semantic domain* is important for consistency checking of software models in general and behaviour models in particular [Eng+01].

1.1.2 Temporal Logics

Temporal logics are modal logics focused towards the description of the temporal ordering of events and states. They provide a convenient way to formalise and verify properties of software systems. There are two types of temporal logics regarding the nature of time [Lam80]: the branching temporal logic (i.e., Computational Tree Logic (CTL) [CE81]) and Linear Temporal Logic (LTL) [Pnu77]. CTL reasons over many possible traces through time. In CTL a time instance has a finite, non-zero number of immediate successors and is EXPTIME-complete. The restricted syntax of CTL limits its expressive power. LTL is a better choice for specification of behavioural properties of the model since such properties are easily expressed in LTL, which may not be expressible in CTL [Roz11].

LTL [Pnu77] is selected in this dissertation for specifying the temporal relationships between the involved elements of the high-level behaviour models, which is PSPACE-complete. In LTL, for each state there is a single successor state, and thus, a unique possible future. This can be represented using linear traces (state sequences), which corresponds to describing the behaviour of a single execution of the system. These features of LTL are useful in the context of behaviour models because they enable explicit reasoning about states and transition executions of the input models for containment relationships. In this dissertation, we adopt a syntactical definition of a well-formed LTL formula φ in terms of the following BNF grammar (note that p is a primitive proposition).

$$\varphi ::= \top \mid \perp \mid p \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \rightarrow \varphi_2 \mid \mathbf{X}\varphi \mid \mathbf{F}\varphi \mid \mathbf{G}\varphi \mid \varphi_1 \mathbf{U} \varphi_2 \mid \varphi_1 \mathbf{R} \varphi_2 \mid \mathbf{Y}\varphi \mid \mathbf{O}\varphi \mid \mathbf{H}\varphi$$

The syntax and standard semantics for LTL operators are as follows: $\mathbf{X}\varphi$ (“neXt”) states that φ will hold in the next state. $\mathbf{F}\varphi$ (“Future/Finally”) means that formula φ will hold in some future state. $\mathbf{G}\varphi$ (“Globally/Always”) indicates that formula φ will continuously hold in all future states. The formula $\varphi_1 \mathbf{U} \varphi_2$ (“Until”) holds if φ_2 holds now or at some state in the future; φ_1 also holds at every point in time until φ_2 holds. $\varphi_1 \mathbf{R} \varphi_2$ (“Release”) states that φ_2 is true until φ_1 becomes true, or φ_2 is true forever. The operator $\mathbf{Y}\varphi$ (“Yesterday/Previous”) means that formula φ held in the previous state. $\mathbf{O}\varphi$ (“Once”) states that formula φ has happened at sometime in the past. $\mathbf{H}\varphi$ (“Historically”) indicates that formula φ always held in the past.

We adopt the conventional semantics of LTL as defined in the field of model checking [CGP99]. For the sake of readability, we have also used the logical exclusive OR operator (hereafter *xor*) which is not part of the traditional LTL definition but often supported by several model checking tools as a useful abbreviation, for instance, $a \text{ xor } b \equiv (a \wedge \neg b) \vee (\neg a \wedge b)$. To make the formulation of some formulas significantly more concise and intuitive, we have also used the LTL past operators, to reason about previous states and transitions. The LTL past formulas, although they do not add expressive power, compared to future only LTL, allow to write more succinct formulas [Gab87]; there is an increase in the size of a formula when the past operator is removed. This, however, do

not increase the complexity of model checking and can be easily converted into future time LTL formulas. For instance, “ $G(a \rightarrow Ob)$ ” means that every a is preceded by b . This can be converted into pure future LTL formula by using the **U** operator “ $(\neg b \text{ U } (a \wedge \neg b))$ ”.

1.1.3 Model Checking

Model checking is an automatic property verification approach that systematically and exhaustively explore the states of software systems. It is most frequently used for the formal verification of safety-critical systems, for example, air traffic control systems, medical equipment systems, train signalling systems, and automotive control systems [Roz11]. The advantage of model checking is that it can be performed in early phases of software modelling and development where no executable products are produced yet. Model checking starts with a model described in a description language and temporal properties, and exhaustively explores violations of a property by traversing the complete state space. If they are valid, an answer true is shown. Otherwise, it generates a counterexample that consists of the execution traces of the descriptions.

We use existing model checkers for formally verifying the containment relationship. LTL model checkers can be classified as explicit or symbolic. Explicit model checkers such as SPIN¹ (Simple Promela Interpreter) construct the state-space of the model explicitly and create the automaton. However, constructing and searching the state space explicitly requires a considerable amount of space [Roz11]. Therefore, the number of states in the finite state representation increases exponentially with the number of variables (i.e., the state explosion problem). The NuSMV (New Symbolic Model Verifier) [Cim+99] model checker used in this dissertation is based on the symbolic model checking technique, and therefore, is able to support the verification of large systems up to 10^{20} states [Bur+92]. The language that underpins the formal descriptions is hereafter called SMV (Symbolic Model Verifier) language, which allows the description of Finite State Machines (FSMs). The SMV description consists of one main module, and a set of state variables and predicates on these variables. A module can contain instances of other modules, allowing a structural hierarchy to be built. The variables can be of type boolean, enumerative or integers. SMV also supports array as a data type. The predicates use the logical operators *AND* (“&”), *OR* (“|”) and *NOT* (“!”). The complete syntax and semantics definitions of SMV description can be found in [Cav+05]. One of the biggest advantages of using the NuSMV model checker is that SMV’s finite state based encoding of the input behaviour models is rather straightforward. That is, each model element is represented by a boolean variable in the SMV description and LTL formulas.

¹See <http://spinroot.com>

1.1.4 Graph Based Search

The containment checking problem can be broken down into smaller graph-based tasks (or functions) that has a number of advantages. First, the tasks are independent from each other, and therefore, can be performed in any order. Moreover, the tasks can also be executed in parallel to gain better performance. Finally, each task produces concrete and precise information about the violation of the containment relationship accordingly. More specifically, no missing nodes (i.e., no missing expected functions), no missing transitive links (i.e., no missing execution paths), and no missing cycles (i.e., no missing loop executions) are implemented. This implementation performs within the boundary of $O(n^3)$ where n is the size of the inputs.

The implementation of no missing transitive links is based on Warshall's algorithm [War62]. Nevertheless, Nuutila has presented heuristics for improving Tarjan's algorithm [Tar72] in detecting Strongly Connected Components (SCC) and uses the improved SCC detection techniques (plus a special representation of successor sets) to achieve better TC finding [Nuu95]. Nuutila's techniques with respect to the use of Tarjan's algorithm is used for no missing cycles. However, we opted not to integrate Nuutila's techniques in order to better analyse individual performance. Moreover, tight integration of Nuutila's techniques implies the dependency between no missing transitive links and no missing cycles, and hence, may nullify the potential parallelizability of our implementation.

1.2 Publications

This doctoral dissertation is based on scientific journal, conference, and workshop publications, which are either published, or submitted (currently under review). In particular, content of the following publications has been used in this dissertation:

- 1 Faiz UL Muram, Huy Tran and Uwe Zdun, "Systematic Review of Software Behavioral Model Consistency Checking", *Journal ACM Computing Surveys (CSUR)*, pages 17:1–17:39, Vol. 50, No. 2, Article 17, April 2017.
- 2 Faiz UL Muram, Huy Tran and Uwe Zdun, "Automated Mapping of UML Activity Diagrams to Formal Specifications for Supporting Containment Checking", In *Proceedings of the 11th International Workshop on Formal Engineering approaches to Software Components and Architectures (FESCA '14)*, pages 93–107, Grenoble, France, April 12, 2014.
- 3 Faiz UL Muram, Huy Tran and Uwe Zdun, "Supporting Automated Containment Checking of Software Behavioural Models Using Model Transformations and Model Checking", submitted to: *Science of Computer Programming* [revised in May 2017].

- 4 Faiz UL Muram, Huy Tran and Uwe Zdun, “A Model Checking Based Approach for Containment Checking of UML Sequence Diagrams”, In Proceedings of the 23rd Asia-Pacific Software Engineering Conference (APSEC '16), Hamilton, New Zealand, December 6-9, 2016.
- 5 Faiz UL Muram, Muhammad Atif Javed, Huy Tran and Uwe Zdun, “Towards a Framework for Detecting Containment Violations in Service Choreography”, In Proceedings of the 14th IEEE International Conference on Services Computing (SCC '17), Honolulu, Hawaii, USA, June 25-30, 2017.
- 6 Faiz UL Muram, Huy Tran and Uwe Zdun, “Counterexample Analysis for Supporting Containment Checking of Business Process Models”, In Proceedings of the 13th International Business Process Management Workshops (BPM '15), pages 515-528, Innsbruck, Austria, August 31-September 3, 2015.
- 7 Huy Tran, Faiz UL Muram and Uwe Zdun, “A Graph-Based Approach for Containment Checking of Behavior Models of Software Systems”, In Proceedings of the 19th IEEE International Enterprise Distributed Object Computing Conference (EDOC '15), pages 84-93, Adelaide, Australia, September 21-25, 2015.
- 8 Faiz UL Muram, Huy Tran and Uwe Zdun, “Towards Containment Checking of Behaviour in Architectural Patterns”, In Proceedings of the 22nd European Conference on Pattern Languages of Programs (EuroPLoP '17), Kloster Irsee, Germany, July 12–16, 2017.
- 9 Muhammad Atif Javed, Faiz UL Muram and Uwe Zdun, “On-Demand Automated Traceability Maintenance and Evolution”, submitted to a journal. This publication is not directly used in the dissertation, but covers related research inspired by the work in this dissertation. The results from this publication have influenced some approaches in this dissertation to consider the implications of traceability between different models and models to code.

1.3 Research Methods

The research conducted in this dissertation adopts the design science framework as a research methodology. It includes analysis of a problem, finding a solution for the problem, and evaluation of the solution. Primary and secondary research methods have been applied for the purpose of data collection, research synthesis and evaluation. In the former case, case studies, formal modelling and laboratory experiments are performed. In the latter case, we have conducted a systematic literature review. In the following subsections we briefly introduce the research methods that have been applied in this dissertation.

1.3.1 Design Science Research

The research presented in this dissertation is based on design science research, which strives to obtain rigorous and succinct results for information systems in the absence of a strong theory base [VK07]. Design science provides the potential to investigate new technologies and to advance accepted practice. In design science research, first a research question is posed, and then the develop/evaluate cycle is continuously repeated until a satisfactory solution for the research question has been obtained [Hev+04]. In the course of design science research, the research question can be altered or refined. Peffers et al. [Pef+08] and Vaishnavi and Kuechler [VK07] specify the following phases of a design science research process:

1. *Identification of the problem and motivation:* The goal of this phase is to identify scope of the problem and significance of the solution.
2. *Design and development artefacts:* This phase focuses on design and development of artefacts, which may require a formal proof to show the correctness.
3. *Evaluation:* In this phase, the developed artefact is evaluated according to the criteria made in the first phase (identification of the problem and motivation).
4. *Conclusion and communication:* It concerns the end of a research cycle or termination of a research project. The final results have to be documented and published in the peer reviewed conferences and journals.

1.3.2 Systematic Literature Review

Systematic Literature Reviews (SLRs) are suitable for identifying, investigating, and interpreting all existing research related to a particular research question or topic [KBB15; KC07]. SLRs are a form of secondary study, while individual studies that contribute to the SLR are considered primary studies. SLRs address clearly formulated questions and use systematic and explicit methods to identify primary studies, selecting primary studies relevant to the questions, critically evaluating studies, synthesizing the data reported in the relevant studies, and reporting the combined results. In this dissertation, we leverage the guidelines for performing SLRs in software engineering recommended in [KC07] with adjustments recommended in [KB13; KBB15]. The phases for conducting the review process are not done sequentially at once but rather in an iterative manner with feedback loops. These phases are defined and described below.

1. *Planning the review:* In this phase, the research questions and methods for developing a review protocol are defined.

2. *Conducting the review*: The review protocol defined in the previous phase will be enacted in this phase.
3. *Reporting the review*: In this phase, the results of the review are documented, validated, and reported.

1.3.3 Case Study

A case study is “an empirical inquiry that investigates a contemporary phenomenon within its real-life context; when the boundaries between phenomenon and context are not clearly evident; and in which multiple sources of evidence are used” [Yin08]. To show the specific properties of a certain situation deep, rich data is needed [Min03]. For instance, it is not enough to define the phenomenon in a small number of variables not covering the entire context. The consideration of large number of cases does not provide additional information about the specific case. Single and multiple case studies are described by Yin [Yin08]. Single case studies examine a group, organization, or system in detail; however, it does not involve variable manipulation, experimental design or controls. Multiple case studies are similar to single case studies, but conducted in a small number of organizations or contexts. Kitchenham et al. [KPP95] point out that the single case studies are not suitable for comparison purposes. They introduce a study design in which the results of a subset of the multiple cases serve as a baseline for the comparison; the method or tool under evaluation is only applied to the rest of the cases.

1.3.4 Formal Modelling

Formal modelling is used to specify certain aspects of a system by applying a mathematical formalism. Examples of formalisms are set theory, category theory, graph theory and logic. Formalisms are used to transform a real or fictitious original into a formal system. The use of formal methods in software development increases understanding of software requirements, avoids ambiguities in the specifications, enables rigorous verification of specifications of the software systems [FKV94]. Formal modelling is especially helpful when a domain is well understood and certain characteristics of a system need to be verified [FKV94].

1.3.5 Experimentation

Experimentation strategies include field experiment, laboratory experiment with human subjects or with software subjects/computer, and experimental simulation [JCP91]. Experimental designs are guided by theories and facilitated by systems development. The experiment results might be used to refine theories and improve the developed systems. In this dissertation we have applied laboratory

experiment with software subjects. The laboratory experiment compares the performance of newly proposed system with other existing systems [CA11]. The focus of this research method in the context of this dissertation is to find out about the performance and scalability of the proposed system/approach.

1.4 Problem Statements and Research Questions

Several techniques and approaches have been proposed in the existing literature to support behavioural model consistency checking. To date, however, no comprehensive systematic review in the area of software behaviour models consistency checking has been published. Therefore, there is a need to systematically select and review the published literature with regard to behavioural model consistency and summarise all existing practices and information in a well-defined and unbiased manner including problems, limitation, future trends, and possible opportunities within the context of software behaviour model consistency. This led us to investigate the following research question:

Research Question RQ1

What is the current state-of-the-art of software behaviour model consistency checking and potential gaps for future research?

Model checking based containment checking is not explicitly considered for the UML behaviour models, business process models and service choreographies. It might be noted that the model checking techniques requires both formal consistency constraints and specifications/descriptions of behaviour models. This makes the particular techniques hard to apply in practice because creating formal specifications and consistency constraints requires considerable knowledge of the underlying formalisms and formal techniques. The creation of formal consistency constraints and specifications is currently done manually, and therefore, labour-intensive and error prone. Accordingly, there is a need to define and develop a fully automated transformation of behaviour models into formal specifications and properties. The generated formal specifications and properties would directly be used by existing model checkers for detecting any discrepancy between the input models and yield corresponding counterexamples. This research gap led us to consider the following research question:

Research Question RQ2

How to perform automated transformation of behavioural diagrams into formal specifications and consistency constraints?

In the design and development of service oriented applications, service choreography models describe the interactions between services at different abstraction levels. The undesired containment violations in service choreographies would cause severe problems; for example, improper identification of services and their corresponding service providers, and therefore affect the delivery of services. Thus, there is a need to investigate the following research question:

Research Question RQ3

How to verify that the behaviour (or interactions) described in the local choreography models collectively encompasses those specified in the global model?

Unfortunately, the results produced by existing model checkers (e.g., counterexamples) are rather cryptic and verbose. As a consequence, the developers and non-technical stakeholders who often have limited knowledge of the underlying formal techniques are confronted with cryptic and lengthy information (e.g., states numbers, input variables over tens of cycles and internal transitions, and so on) in the counterexample. Therefore, there is also a need for an automated approach to interpret the counterexamples with respect to containment checking, in particularly for identifying the causes of containment violations and their resolutions. This led us to formulate the following research question:

Research Question RQ4

Is it possible to facilitate the interpretation of counterexamples for identifying the causes of containment violations and their resolutions?

The costly exhaustive searches employed by model checking are not always necessary for addressing the containment checking problem. The most challenging issues in model checking based techniques are to deal with loops and parallel execution branches. For example, the non-determinism of decision nodes and loop nodes make the translation of the input models to formal properties, e.g., in temporal logics, very difficult and inefficient. The parallel structures, for instance, formed due to the combination of “ForkNodes” and “JoinNodes”, often cause the state explosion problem. In order to alleviate the particular challenges, a lightweight graph-based approach might be developed for supporting the developers in checking the containment relationship between the high-level and low-level behaviour models. This problem led us to consider the following research question:

Research Question RQ5

Does graph-based containment checking provide better support for dealing with the non-determinism of decision nodes and loop nodes, as well as the state explosion problems than model checking based techniques?

The behaviour of architectural patterns must be consistent in terms of the artefacts produced in the various activities of the software development process, such as requirements, software architecture, detailed design and implementation. Previous studies have not considered the checking of architectural patterns' behaviour. Therefore, there is a need to verify whether the high-level design constraints described by an architectural pattern are contained in the low-level design and implementation. This research gap led us to formulate the following research question:

Research Question RQ6

How to ensure that the behaviour of an architectural pattern is consistent across the artefacts produced in the various activities of the software development process?

1.5 Major Contributions and Overview of the Dissertation

This dissertation is divided into nine chapters. The first chapter includes the introduction to software behaviour models consistency checking, background information, publications, problem statements and research questions, which were presented earlier in this chapter. The main body of the dissertation comprises eight chapters; they contain systematic literature review, model checking based containment checking of UML behaviour models, business process models and service choreographies, their counterexample analysis, graph-based containment checking algorithm, containment checking of architectural patterns' behaviour, and conclusions. Table 1.1 shows the research questions together with the papers and chapters, in which they are addressed. Chapters 2 to 8 are based on scientific journal, conference, and workshop publications, which are either published, or currently under revision. In the following, each chapter is briefly outlined.

Chapter 2 presents a Systematic Literature Review (SLR) that was carried out to obtain an overview of the various consistency concepts, problems, and solutions proposed regarding behaviour models. It addressed the Research Question RQ1 and is based on Paper 1 [MTZ17a]. In our study, the identification and selection of the primary studies was based on a well-planned search strategy. The search process identified a total of 1770 studies, out of which 96 have been thoroughly analysed according to our predefined SLR protocol. The SLR aims to highlight the state-of-the-art of software behaviour model consistency checking and identify potential gaps for future research. Based on research topics in selected studies, we have identified seven main categories: targeted software models, types of consistency checking, consistency checking techniques, inconsistency handling, type of study and evaluation, automation support, and practical impact. The findings of the systematic review also reveal suggestions for future research, such as improving the quality of study design and conducting evaluations, and application of research outcomes in industrial settings. For this purpose, appropriate strategy for inconsistency handling, better tool support for consistency checking and/or development tool integration should be considered.

Research Questions	Papers	Chapters
RQ1: What is the current state-of-the-art of software behaviour model consistency checking and potential gaps for future research?	Paper 1: Systematic Review of Software Behavioural Model Consistency Checking	Chapter 2: Systematic Review of Software Behavioural Model Consistency Checking
RQ2: How to perform automated transformation of behavioural diagrams into formal specifications and consistency constraints?	Paper 2: Automated Mapping of UML Activity Diagrams to Formal Specifications for Supporting Containment Checking Paper 3: Supporting Automated Containment Checking of Software Behavioural Models Using Model Transformations and Model Checking Paper 4: A Model Checking Based Approach for Containment Checking of UML Sequence Diagrams Paper 5: Towards a Framework for Detecting Containment Violations in Service Choreography	Chapter 3: Model Checking Based Containment Checking of UML Activity Diagrams Chapter 4: Model Checking Based Containment Checking of UML Sequence Diagrams Chapter 5: Model Checking Based Containment Checking of Service Choreographies
RQ3: How to verify that the behaviour (or interactions) described in the local choreography models collectively encompasses those specified in the global model?	Paper 5: Towards a Framework for Detecting Containment Violations in Service Choreography	Chapter 5: Model Checking Based Containment Checking of Service Choreographies
RQ4: Is it possible to facilitate the interpretation of counterexamples for identifying the causes of containment violations and their resolutions?	Paper 3: Supporting Automated Containment Checking of Software Behavioural Models Using Model Transformations and Model Checking Paper 4: A Model Checking Based Approach for Containment Checking of UML Sequence Diagrams Paper 5: Towards a Framework for Detecting Containment Violations in Service Choreography Paper 6: Counterexample Analysis for Supporting Containment Checking of Business Process Models	Chapter 6: Counterexample Analysis for Supporting Containment Checking
RQ5: Does graph-based containment checking provide better support for dealing with the non-determinism of decision nodes and loop nodes, as well as the state explosion problems than model checking based techniques?	Paper 7: A Graph-Based Approach for Containment Checking of Behavior Models of Software Systems	Chapter 7: Graph-Based Containment Checking of UML Activity Diagrams

(continued on next page)

Research Questions	Papers	Chapters
RQ6: How to ensure that the behaviour of an architectural pattern is consistent across the artefacts produced in the various activities of the software development process?	Paper 8: Towards Containment Checking of Behaviour in Architectural Patterns	Chapter 8: Containment Checking of Behaviour in Architectural Patterns

TABLE 1.1: Overview of Papers, Chapters and Corresponding Research Questions

Chapter 3 concerns the Research Question RQ2 and is based on Paper 2 [MTZ14] and Paper 3. The chapter proposes a novel concept and technique for automatic containment checking of UML activity diagrams. Containment checking aims to verify whether a certain low-level activity diagram, typically created by refining and enhancing a high-level activity diagram, still is consistent with the specification provided in its high-level counterpart based on model checking techniques. In this context, the automated transformation of activity diagrams into formal specifications and consistency constraints is performed; they are used by model checkers for detecting any discrepancies between the input models and yielding corresponding counterexamples. The automated translation strategy is useful to bridge the gap between manual specification of formal properties and consistency constraints for containment checking. Formal modelling and comparative case studies have been utilized to investigate the applicability and technical feasibility of the approach in a typical developer's working environment.

Chapter 4 concerns the Research Question RQ2 and is based on Paper 4 [MTZ16]. The chapter presents a model checking based approach to automatically detect containment inconsistencies between UML 2 sequence diagrams. In contrast to UML activity diagrams, the sequence diagrams represent radically different perspectives of a system and have different semantics. Therefore, we introduce the formalisations of sequence diagrams to track the execution state of an interaction without compromising the containment relationship. This way, our automated translation helps to alleviate the burden of manually encoding consistency constraints, and therefore, increase productivity and avoid potential translation errors. Two research methods have been used for evaluation purposes: formal modelling and comparative case studies.

Chapter 5 concerns the Research Questions RQ2 and RQ3 and is based on Paper 5 [Mur+17]. In this chapter, a technique for automatic containment checking in service choreographies is proposed – that verifies whether the message exchange behaviour (or interactions) described in the joint local choreography models encompasses the behaviour specified in the global model. The global model (aka interaction model) is modelled using BPMN 2.0 choreography diagram, while the local choreography model (aka interconnection model) of each partner is modelled using BPMN

2.0 collaboration diagram. The approach performs automated translation of global choreography model into temporal logic based consistency constraints (i.e., LTL) and local choreography models into formal descriptions (i.e., SMV language), whereas the NuSMV model checker is used for verification. The approach has been validated using formal modelling and comparative case studies.

Chapter 6 concerns the Research Question RQ4 and is based on Paper 3, Paper 4 [MTZ16], Paper 5 [Mur+17] and Paper 6 [MTZ15]. The chapter presents a counterexample analysis approach for supporting containment checking of activity diagrams, sequence diagrams, services choreographies, and business process models. The analysis/interpretation of counterexample consists of two steps. First, the actual causes of the unsatisfied containment relationship are located based on the generated counterexamples and appropriate guidelines to resolve the particular violations are produced. Second, the concise descriptions of the isolation's causes and potential countermeasures, produced in the previous step are annotated in the low-level model such that the developers can easily understand and fix the problems. The applicability of the approach is demonstrated through four use case scenarios taken from industrial case studies.

Chapter 7 concerns the Research Question RQ5 and is based on Paper 7 [TUMZ15]. The chapter presents a lightweight graph-based approach for containment checking that starts by mapping the input UML activity diagrams at different levels of abstraction into equivalent intermediate representations, namely, check models. Subsequently, a lightweight graph-based algorithm verifies whether there are any missing nodes (i.e., missing expected functions), missing transitive links (i.e., missing execution paths), and missing cycles (i.e., missing loop executions). The theoretical complexity of our approach is a cubic polynomial of the number of elements of the input behaviour models. The approach generates feedbacks that are relevant and easy-to-understand for the stakeholders. Comparative case studies and laboratory experiments with software subjects have been performed for evaluation purpose.

Chapter 8 concerns the Research Question RQ6 and is based on Paper 8 [MTZ17b]. The chapter presents a solution to the containment checking problem in architectural patterns' behaviour. The goal is to verify whether the high-level design constraints described by an architectural pattern are contained in the low-level design and implementation. The containment checking of architectural patterns' behaviour not only considers missing elements or interactions but also misplacement of elements at different levels of abstraction. The solution provides informative and comprehensive feedback to the stakeholders to identify the violation causes and their resolutions. In addition, we have selected the stereotypes from the existing vocabulary of design elements as well as we have proposed new stereotypes to model a specific architectural pattern and its variants. The applicability of the proposed solution is demonstrated by applying it on three popular architectural patterns, namely MODEL-VIEW-CONTROLLER, LAYERS, and PIPE AND FILTER, as well as their variants. We have investigated one specification extension of UML sequence diagram for modelling the behaviour of each of those patterns. The proposed solutions can also be applied to other types

of behaviour models, such as state machines, activity diagrams and BPMN models, as well as other architectural patterns.

This chapter presents the systematic review of software behavioural model consistency checking research. It has been conducted to establish a foundation for this dissertation, and therefore concerns the Research Question RQ1. The parts of this chapter focused on the consistency checking techniques, types of consistency checking, inconsistency handling and targeted software models are based on a peer-reviewed journal ACM Computing Surveys [MTZ17a].

2.1 Introduction

In software development, models are often used to represent multiple views of the same system. Such models need to be properly related to each other in order to provide a consistent description of the developed system. Models may contain contradictory system specifications, for instance, when they evolve independently. Therefore, it is very crucial to ensure that models conform to each other. Inconsistencies can also occur due to the multi-view nature of many models [Huz+05; FLP99]. A system can be defined by multiple views that specify different aspects of the system. Inconsistencies can occur due to the overlaps between the views [Huz+05]. Consistency is a general goal to be obtained while building the models. In particular, during system (and model) evolution it is crucial to maintain the consistency between different behaviour models. As Spanoudakis and Zisman [SZ01] emphasized, there are severe negative effects of model inconsistencies that may delay and, therefore, increase the cost of the system development process, jeopardize properties related to the quality of the system, and make it more difficult to maintain the system [SZ01]. These negative effects can be multiplied manifoldly, especially in the context of developing modern large scale software systems that are far more complex and might consist of numerous models and interconnected subsystems (e.g., cloud-based systems, Internet of Things systems, networks of sensors based systems, or astrophysical systems) [Got08; AIM10; Som+12].

Several techniques and approaches have been proposed in the existing literature to support behavioural model consistency checking. There are only a few secondary studies in collecting and analysing evidence regarding the research of model consistency checking and management [SZ01;

[Huz+05](#); [Usm+08](#); [LMT09](#)]. The survey of Spanoudakis and Zisman [[SZ01](#)] presents a broad view and discusses open research issues. Among of those secondary studies, only the SLR conducted by Lucas et al. [[LMT09](#)] has systematically investigated and provided insights regarding consistency concepts, proposals, problems, supported models and the maintainability of consistency management approaches, but only for UML models in the timespan from 2001 to 2007.

We note that the aforementioned studies, except the survey of Spanoudakis and Zisman [[SZ01](#)], mainly focus on UML based modelling and development. They aim to cover broadly both structural and behavioural consistency checking of software artefacts. Although UML is widely used in academia and industry, there are still a considerable number of non-UML modelling and development methods, for instance, in the domains of workflow and business process management, embedded and real-time systems, and service-oriented systems, to name but a few. Considering only UML-based methods and techniques can likely lead to bias against non-UML approaches. Thus, we decided to conduct a systematic literature review of consistency checking with a broader scope and, therefore, research objectives. Our study strives for investigating consistency checking beyond the domain of UML-based software development. As checking of structural aspects has been extensively reviewed [[SZ01](#); [Huz+05](#); [Usm+08](#); [LMT09](#)], we opted to pay special attention to the behavioural aspects of software system modelling. We planned to carry out the SLR at a finer granularity, for instance, we studied in-depth the degrees of automation support, inconsistency handling, tool support, evaluation, and evidence of application in industry settings (these aspects are not yet considered or only in limited form in the previous studies). Chen and Ali Babar [[CA11](#)] emphasized that these aspects are important as they indicate the practical impacts of the research outcomes to both science and industrial practices. Nevertheless, we adopted and extended some fundamental concepts and categories of consistency checking and management that have been proposed in these studies.

The chapter is organized as follows: Section [2.2](#) explains the systematic literature review process used in this review. Section [2.3](#) discusses the results of SLR in relation to the addressed research questions. Section [2.4](#) presents the discussion on the results of our SLR. Finally, Section [2.5](#) concludes the chapter with main findings and potential future research directions.

2.2 Systematic Literature Review Process

An SLR provides a key instrument for identifying, evaluating, and interpreting all existing research related to a particular research question, phenomenon of interest or topic area [[KBB15](#); [KC07](#)]. SLRs are a form of secondary study, while individual studies that contribute to the SLR are considered primary studies. The process of conducting an SLR must be explicitly defined and well executed. To facilitate the planning and execution of this SLR, we leverage the guidelines performing SLRs in software engineering recommended in [[KC07](#)] with adjustments recommended

in [KB13; KBB15]. A clear description of the phases for conducting the review process will be presented in this section.

1. *Planning the review*: The goal of this phase is to define the research questions and methods for developing a review protocol (see Section 2.2.1),
2. *Conducting the review*: In this phase, the review protocol defined in the previous phase will be enacted (see Section 2.2.2),
3. *Reporting the review*: In this phase, the results of the review are documented, validated, and reported (see Sections 2.3 and 2.4).

The aforementioned phases are not done sequentially at once but rather in an iterative manner with feedback loops. In particular, many activities are created during the protocol development phase and should be refined when execution of the review takes place. For example, search terms, inclusion criteria, and exclusion criteria can be refined during the course of the review. The protocol provides details of the plan for the review, such as specifying the search process to be followed and the conditions to apply when selecting relevant primary studies.

2.2.1 Planning the Review

The first phase for undertaking an SLR is related to specifying pre-review activities for conducting the SLR. The planning phase includes identification of the reasons for carrying out the systematic review, specifying the research questions, defining the search strategy, and establishing the inclusion and exclusion criteria.

Need for the Literature Review There are several reasons for undertaking this systematic review. The main objective of this SLR is to systematically select and review the published literature with regard to behavioural model consistency and summarise all existing practices and information in a well-defined and unbiased manner including problems, limitation, future trends, and possible opportunities within the context of software behaviour model consistency.

This SLR aims at identifying the current state-of-the-art of consistency checking of software behaviour models not only in the field of UML based modelling but also in various other domains. As mentioned above, the existing reviews merely covered UML models. Hence, no comprehensive systematic review in the area of software behaviour models consistency checking has been previously published. Our study also aims at a finer level of granularity with regard to automation support, inconsistency handling, tool support, evaluation, and evidence of application in industry settings (which have not yet been well considered in the previous studies) as these aspects indicate the

practical impacts of the research outcomes to both science and industrial practices [CA11]. The perspective taken in this study is both of practitioners and researchers working on behavioural model consistency checking to provide them up-to-date state-of-the-art research and therefore guide them to identify relevant studies that suit their own needs. Furthermore, we aim to appraise evidence of research on consistency checking of software behaviour models as well as to identify challenges and open problems that may provide insights for further investigations.

Research Questions Defining the research questions is one of the most important activities of any systematic review because they guide the selection of primary studies and data extraction. For this purpose, we leverage the Goal-Question-Metric (GQM) approach [BW84], which is a systematic method for organizing measurement programs. The GQM model starts with specifying the certain goal (i.e., purpose, object, issue, and viewpoint). Then, the goal is refined into several questions, each question is then refined into metrics [BW84]. By using GQM, a goal for conducting the SLR is defined. The goal is refined into several research questions, and, subsequently, these questions are refined into metrics that provide means to answer these questions. By providing the answers of the questions, the data can be analysed to identify whether the goals are achieved or not. The goal for our SLR is:

Purpose Understand and characterize ...

Issue ... the consistency checking ...

Object ... of behavioural models used in software development ...

Viewpoint ... from a researcher's and engineer's viewpoint.

Based on the aforementioned goal, we derive the following research questions.

- **Q1:** How did research in consistency checking of software behavioural models develop over time?
- **Q2:** What are the methods, languages, or techniques used in each of the primary studies? This research question is refined into following research questions:
 - **Q2.1:** What types of models have been studied?
 - **Q2.2:** What kinds of consistency problems have been addressed?
 - **Q2.3:** What consistency checking techniques have been used?
 - **Q2.4:** What inconsistency handling techniques have been proposed?
 - **Q2.5:** What levels of automation have been supported?
 - **Q2.6:** What types of study and evaluation have been conducted?
- **Q3:** What is the potential practical impact of the primary studies?

– **Q4:** What are the limitations of the existing methods?

Q1 and Q2 aim at describing the state-of-the-art of research on consistency checking of software behavioural models. The expected outcome will be a comprehensive view of behavioural model consistency checking in various dimensions. This allows us to categorize the state-of-the-art in consistency checking with respect to the behavioural models. Q3 is proposed for learning about the practical impact of the existing methods in academia and industry. It aims to provide both researchers and practitioners with evidence about what methods could be used in practice and to which degree. A lack of evidence or poor evidence could highlight the need for more rigorous studies and applications in real or quasi-real settings. Q4 is formulated to identify gaps in current research that could yield insights regarding the issues or open problems in software behavioural models consistency research and provide directions for further studies.

Dimension	Search Keywords
Type of models	behaviour diagram, behavior diagram, behavioural diagram, behavioral diagram, behaviour model, behavior model, behavioural model, behavioral model, activity diagram, sequence diagram, state diagram, state machine, statechart, collaboration diagram, communication diagram, interaction diagram, timing diagram, workflow, business process, process model, BPMN, WSBPEL, EPC, finite state machine, FSM, state-transition, Stateflow
Type of studies	containment/contain/containing, refine/refining/refinement, inconsistencies/inconsistent/inconsistency, consistent/consistency/consistencies

TABLE 2.1: Literature Search Dimensions and Keywords

Search Strategy Our search strategy aims to find a comprehensive and unbiased collection of primary studies from the literature related to the research questions. Therefore, we devised a search strategy that maximizes the possibility to discover every relevant publications in a search result. For the electronic search, we leveraged the major databases that are widely used in computer science (CS) and software engineering (SE) research as reported in [ZABT11], which are the ACM Digital Library¹, the IEEE Explore Digital Library², SpringerLink³, and ScienceDirect⁴. These are rich and comprehensive databases containing bibliographic information of a plethora of publications from all major publishers of the computing literature. We note that ISI Web of Science (WoS)⁵ is also a large database of scientific studies. However, in the field of software engineering, WoS mainly records publications published in premium journals and only a few conferences, whereas conferences (and sometimes workshops and symposiums) are major outlets for publishing in CS and SE. Thus, we mainly use WoS along with Google Scholar⁶ for cross-checking with the search results from the

¹See <http://dl.acm.org>

²See <http://ieeexplore.ieee.org>

³See <http://link.springer.com>

⁴See <http://www.sciencedirect.com>

⁵See <http://webofknowledge.com>

⁶See <http://scholar.google.com>

chosen sources and for performing some meta-analyses. Our search terms stem from the research questions and can be categorized into two major dimensions, as shown in Table 2.1. On the one hand, we aim at exploring the different behaviour models or diagrams considered in the primary studies. On the other hand, we aim to discover relevant types of studies that have been performed for checking different types of inconsistencies. In order to ensure a sufficient scope of searching, we also consider different alternative words (e.g., behavior/behaviour/behavioural, model/diagram), synonyms (e.g., model, diagram, workflow, process), and abbreviations for the search terms. Then we used the boolean operator “OR” to join the alternate words and synonyms and the boolean operator “AND” to form a sufficient search string.

Inclusion and Exclusion Criteria Inclusion and exclusion criteria are used to identify the suitability of primary studies and making decisions for inclusion or exclusion of an article in the SLR based on the addressed research questions. Our inclusion and exclusion criteria are shown in Table 2.2.

2.2.2 Conducting the Review

The second phase of the SLR is to perform the search strategy defined in the planning stage. The steps involved in conducting the review are primary studies selection, quality assessment, and data extraction and synthesis.

Primary Studies Selection We illustrate the search process and the number of primary studies identified at each stage in Figure 2.1. We strive for a comprehensive list of studies reported without any additional constraints. That is, our search strategy dated back to late eighties since that time period is often considered to foster consistency checking of software artefacts [SZ01]. We started with the search in June 2015 and ended the search process at the end of July 2015 using the search string described in Section 2.2.1. The initial search, however, is performed in 2013. After consolidating the results, overall 1770 studies have been identified. In the initial search process, we identified 1698 studies; whereas the snowballing process added 72 more studies.

Two researchers working independently have identified the relevant studies by quickly scanning parts of each publication such as the title, abstract and keywords (and sometimes the conclusion in case it was difficult to extract information from the abstract). The second selection stage is based on the aforementioned inclusion and exclusion criteria. We note that there were a number of duplications due to different reasons. For instance, some authors published their journal articles which were extended versions of previously published workshop and/or conference papers. Therefore, we worked carefully to eliminate all potential duplications and retained only the most complete

Type	Description
Inclusion	<ol style="list-style-type: none"> 1. Study is internal to software domain. We are only interested in consistency checking for software systems. 2. Study is about consistency checking related to software behavioural models/-diagrams. 3. Study comes from an acceptable source such as a peer-reviewed scientific journal, conference, symposium, or workshop. 4. Study reports issues, problems, or any type of experience concerning software behavioural model consistency. 5. Study describes solid evidence on software behavioural model consistency checking, for instance, by using rigorous analysis, experiments, case studies, experience reports, field studies, and simulation.
Exclusion	<ol style="list-style-type: none"> 1. Study is about hardware or other fields not directly related to software. 2. Study is not clearly related to at least one aspect of the specified research questions. 3. Study reports only syntactic or structural consistency checking of models/diagrams. 4. Secondary literature reviews. 5. Study does not present sufficient technical details of consistency checking related to software behavioural models (e.g., they have a different focus (i.e., version control) and have insufficient detail). 6. Study did not undergo a peer-review process, such as non-reviewed journal, magazine, or conference papers, master theses, books and doctoral dissertations (in order to ensure a minimum level of quality). 7. Study is not in English. 8. Study is a shorter version of another study which appeared in a different source (the longer version will be included).

TABLE 2.2: Inclusion and Exclusion Criteria

(or recent) versions of the duplicates. After the inclusion and exclusion stage (including removing duplications), there are 70 primary studies remained out of 1698 identified studies in initial search.

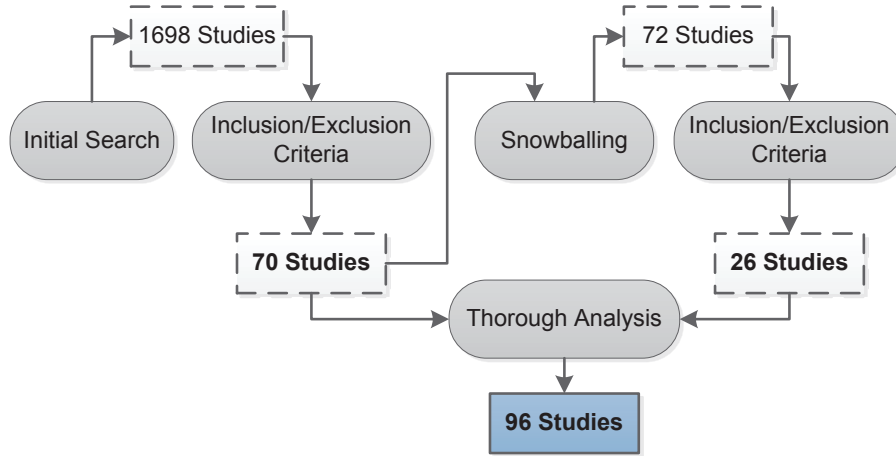


FIGURE 2.1: Overview of the Stages and Results of Our Search Process

We are aware that it is impossible to achieve a total set of publications using the aforementioned automated searches. Therefore, we performed an additional *snowballing* process [Bud+11] (i.e., manually scanning and analysing the references and citations of these primary studies) to ensure that our SLR also covered relevant follow-up works that might exist but have not been included in the search. In particular, we collected references and related works from each of 70 primary studies selected in the automated search phase. The snowballing search continued until no more relevant studies were found. As a result, there are total 72 studies collected from the snowballing process.

Some of these studies did not appear in the results of the aforementioned automated searches. Some others existed but were not found in the first search because either they have too short abstracts or their titles or abstracts do not explicitly include the proposed search terms (but their contents and contribution do). Twenty-six out of the 72 additional studies satisfied the inclusion criteria and exclusion criteria, and therefore, are selected. Finally, we analysed the remaining primary studies thoroughly to ensure that we had obtained the most relevant studies. This in-depth analysis results in total 96 studies within the time period of 1999–2015 that are included for further consideration in our SLR.

Quality Assessment The quality assessment criteria are used to determine the rigorousness and credibility of the used research methods and the relevance of the studies. This assessment is important to limit bias in conducting this SLR, to obtain insight into potential differences, and to support the interpretation of the results. Three main quality assessment criteria have been applied that are based on the assessment criteria introduced in [KC07; Kit+09]. We used a checklist based scoring procedure to evaluate the quality of each selected study and to provide a quantitative comparison between them. The scoring procedure has only three optional answers:

“Yes = 1”, “Partly = 0.5”, or “No = 0”. Therefore, for a given study, its quality score is computed by summing up the scores of the answers to the quality assessment questions.

- *Relevance.* Are all the collected studies relevant to the objective of this literature review? In this SLR, the quality assessment is specifically focused on accumulating only those studies that report adequate information to answer the targeted research questions. The quality assessment has been performed according to our inclusion and exclusion criteria by the first two researchers who have independently reviewed each study. In case of contradicting opinions, the third researcher reviewed and resolved the issues together with the two researchers.
- *Coverage.* Is the literature research cover all relevant studies? In this SLR, the two researchers have independently reviewed each study in a number of iterations to ensure that none of the relevant studies are missed. To achieve this, the search is performed on the entire list of relevant studies following with the screening of the titles, abstracts, keywords, and conclusion. Moreover, a snowballing process is conducted to broaden the scope of selected studies. Finally, in-depth analyses have been performed after accessing the full text.
- *Validation.* Do the collected studies contain adequate data and information? In this SLR, it is analysed whether the primary studies contain the necessary information to answer the targeted research questions. In particular, we devised a number of questions to assess the validation of the relevant studies, such as: Is the technique/tool clearly defined? How rigorously is the technique evaluated? Does the study add value to academia or industry?

Extracted data	Relevant Q
Author(s)	Study overview
Title	Study overview
Year	Study overview, Q1
Studied domains	Study overview
Publication types and venues (c.f. Section 2.3.1)	Study overview, Q1
Active research groups (c.f. Section 2.3.1)	Study overview, Q1
Types of studied behavioural models/diagrams (c.f. Section 2.3.2)	Q2.1
Consistency checking types (c.f. Section 2.3.3)	Q2.2, Q4
Consistency checking techniques (c.f. Section 2.3.4)	Q2.3, Q4
Formalisation methods (c.f. Section 2.3.4)	Q2.3, Q4
Degree of inconsistency handling (c.f. Section 2.3.5)	Q2.4, Q4
Degree of automation support (c.f. Section 2.3.6)	Q2.5, Q4
Tool support for consistency checking (c.f. Section 2.3.6)	Q2.5, Q3, Q4
Types of study and evaluation (c.f. Section 2.3.7)	Q2.6, Q4
Citations (c.f. Section 2.3.8)	Q3
Development tool support and integration (c.f. Section 2.3.8)	Q3, Q4
Levels of application evidence (c.f. Section 2.3.8)	Q3, Q4

TABLE 2.3: Data Collection

Data Extraction and Data Synthesis All remaining 96 primary studies were analysed in-depth and relevant data were extracted from these studies. In the data extraction step, we used spreadsheets to record and correlate the extracted information. In this SLR, we concentrate on extracting the following data items from each study. Table 2.3 shows the corresponding data extracted from the set of selected studies. After the data extraction stage was completed, we synthesized the resulting information such that they are suitable and sufficient for answering the review questions. We will explain the rationale of extracting data and analyse the extracted data in Section 2.3. For further details on the classification and encoding of every aspect related to extract data, please refer to Appendix C.

2.3 Results

In this section, we summarise the main results obtained in the systematic review together with an analysis of the collected data in order to determine the current research trends and identify existing gaps and open problems of the current methods. Table A.1 in Appendix A shows the artefacts studied by each approach, type of semantic domain and consistency checking technique supported by existing literature, and the studied domains. For a list of these selected studies with full details, please refer to Table B.1 in Appendix B.

2.3.1 Historical Development (Q1)

This section presents the result about the research, publication trend (i.e., time and venue) and active research groups in the context of behavioural models consistency checking. Figure 2.2 shows an overview of the distribution of selected studies per year grouped by publication types (e.g., journal, conference, workshop, symposium, etc.). We can see that consistency checking of software behaviour models has drawn considerable attention and became active at the beginning of the 2000s. The number of studies reaches a peak around 2006-2008. Apart from that, the trend of research in behavioural models consistency checking seems stable over time. We did not set a lower boundary for the year of publication in our search process, yet the timeframe of identified studies reflects also the timeframe of activeness and maturation of behavioural models consistency checking field.

With regard to the active research groups within the area of behavioural models consistency checking, we looked at the affiliation details of the selected primary studies. The assignment of contributed studies of each active research group is based on the affiliations that is given in these studies. Table 2.4 presents the active research groups (with at least three publications within behavioural models consistency checking) along with the corresponding number of contributed

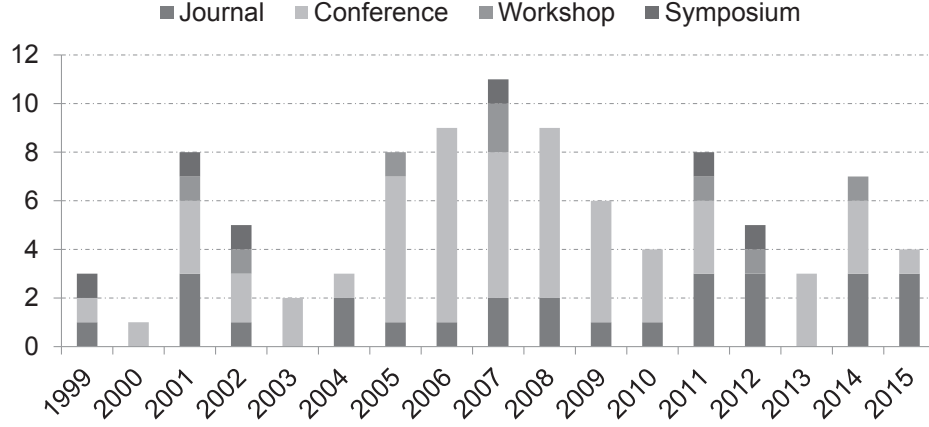


FIGURE 2.2: Primary Studies per Year Grouped by Publishing Venues

studies. The results depict that the University of Paderborn, Germany and Nanjing University, China are the leading ones in terms of the number of publications.

Affiliations	Studies	Total
University of Paderborn, Germany	S16, S17, S26, S29, S30, S39, S55	7
Nanjing University, China	S20, S31, S32, S47, S73, S86	6
University of Potsdam, Germany	S72, S81, S82, S83, S84	5
Universität München, Germany	S5, S37, S64, S92	4
Eindhoven University of Technology, The Netherlands	S19, S77, S78, S82	4
University of Toronto, Canada	S56, S62, S63	3
Lingnan University, Hong Kong	S88, S89, S90	3

TABLE 2.4: Active Research Groups and Numbers of Studies

2.3.2 Targeted Software Models (Q2.1)

This subsection discusses different types of software models tackled in the current literature regarding behaviour model consistency. Table A.1 in the Appendix A shows the results of our SLR regarding targeted software models (i.e., the behavioural models being investigated for consistency checking). We depict in Figure 2.3 the distribution of different behavioural models considered in the selected studies. The UML specification 1.x uses the term “*statecharts*” whilst the UML specification 2.x switches to “*state machines*” [Gro11b, Sec. 15] instead. Please note that, UML statecharts (or UML state machines) are based on or derived from Harel’s statecharts, which extends the classic notion of finite state machines with additional support for hierarchy, concurrency and communication [Har87]. For this reason, we use the term “statecharts” in this chapter to denote the aforementioned variants of Harel’s statecharts and “state machine” to explicitly denote the classic definition of finite state machines. Some other types of behavioural models, such as live sequence charts (LSCs) and Simulink Stateflow are also various extensions of Harel’s statecharts

but target different modelling purposes and application domains. Thus, we opted to separate these types of models as shown in Figure 2.3. We note that “*process model*” is an umbrella term for BPMN, BPEL, and workflow models that are used for describing the behaviour of process-centric information systems (PAIS). The category “Other” indicates a mix of various types of other behavioural models that do not belong to any of the aforementioned categories.

We can see that research on behavioural models consistency checking focuses on four major types of behaviour models, which are statechart (36.59%), sequence diagrams (21.14%), process model (14.63%), and activity diagrams (9.96%). Less attention (i.e., 1.63%) has been paid on collaboration diagrams, LSCs, stateflow and labelled transition systems.

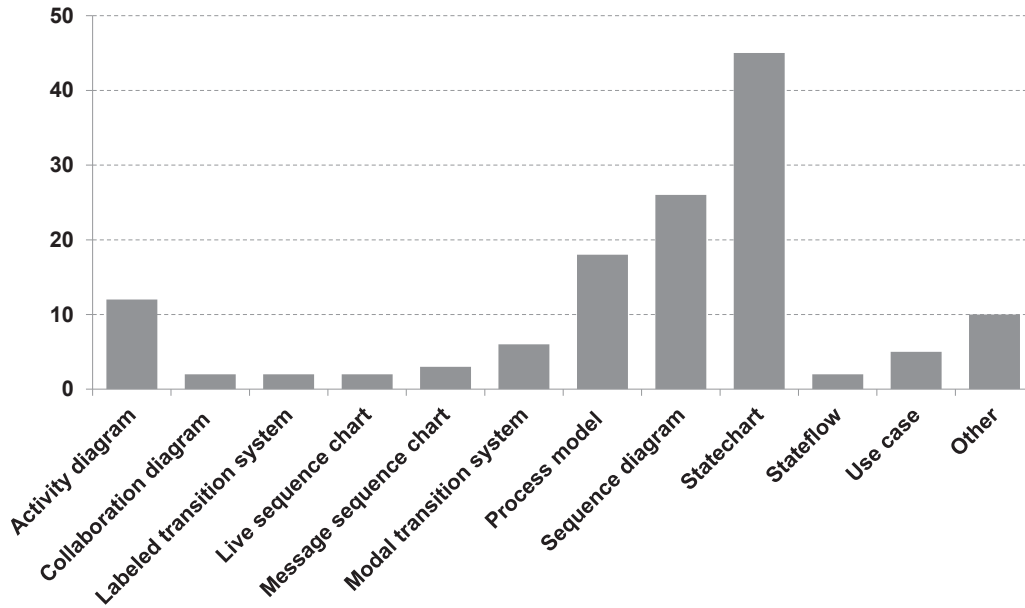


FIGURE 2.3: Types of Studied Software Behavioural Models

2.3.3 Types of Consistency Checking (Q2.2)

Another aspect to be considered in our SLR is the types of consistency checking tackled by the selected publications. Static consistency checking techniques examine and analyse the targeted behavioural models and/or their abstracted versions without running systems. Examples are model checking techniques that systematically and exhaustively explore the states of software systems. The advantage of static checking is that it can be performed in early phases of software modelling and development where no executable products are produced yet. However, static checking can be computationally expensive due to the cost of exhaustive analysis of large and complex models. For instance, model checking often suffers from the problem of state explosion [Cla+11].

In contrast, dynamic consistency checking aims to reveal inconsistencies while the system is running, either by code instrumentation or monitoring. Dynamic checking can achieve better computational performance as it only examines a small subset of system spaces (i.e., the actual execution traces). However, dynamic checking techniques, on the one hand, require a running system. On the other hand, in contrast to static checking, these techniques can not reveal all potential errors.

Apart from static and dynamic consistency checking, there are studies using symbolic execution or simulation approaches to validate the consistency of models. These approaches often consider a subset of input parameters and verify the response and consistency of the systems. As such, these studies do not exactly belong to either of the aforementioned categories. Therefore, we placed them in an additional category, namely, “*symbolic execution/simulation*” (SYM/SIM) to refer to these kinds of studies.

Types	Studies	Total
Static	S1–S51, S53–S69, S71–S96	94
Dynamic	S65	1
SYM/SIM	S48, S50, S52, S70, S17, S96	6
Endogenous	S15, S25, S37, S42, S44, S47, S55, S56, S61, S67–S69, S71, S74, S76, S79	16
Horizontal (Exogenous)	S1–S3, S6–S10, S13–S20, S22–S28, S30, S32–S36, S38, S41, S42, S44–S48, S50, S54, S56–S60, S62, S63, S65–S68, S70, S73, S77, S78, S84–S88, S90–S96	66
Vertical (Exogenous)	S4, S5, S11, S12, S15, S21, S29, S31, S39, S40, S43, S49, S51–S53, S56, S58, S64, S71, S72, S74, S75, S80–S83, S89	27

TABLE 2.5: Types of Consistency Checking

Table 2.5 depicts the distribution of consistency checking types in terms of static, dynamic, and symbolic execution/simulation checking. “Static consistency checking” are addressed prominently in 94 studies (93.07%) whilst “symbolic execution/simulation” is used in six studies (5.94%). Only one study (0.99%) covers dynamic consistency checking. Note that the sum of the numbers of studies on consistency checking types exceeds the total number of studies within a specific category, because the same study could address more than one type of consistency. Four studies (4.17%) out of a total of 96 use a combination of static+simulation consistency checking, whereas only one study (1.04%) uses a combination of static+dynamic consistency checking.

Figure 2.4 presents a bubble-plot distributed over two dimensions regarding: year of publication and consistency checking types. The results show that the majority of research attention has been paid to the static consistency checking throughout the years. Only one study has been found on dynamic consistency checking in the year 2009; however, the SYM/SIM consistency checking has received more attention in the timeframe 2010-2013 but is still at a rather low level.

Based on the aforementioned statistics, we can see that the research on dynamic and simulation consistency checking is much less than the research on static consistency checking. This is perhaps

in accordance with Giannakopoulou and Havelund’s observation that dynamic checking often requires special treatments ranging from extended semantics of temporal logics to monitoring and analysis algorithms [GH01].

We also examined the existing methods from the perspective of distinguishing endogenous and exogenous consistency checking. As mentioned in Chapter 1.1, endogenous consistency concentrates on a single behavioural model whereas exogenous consistency targets various types of models. In addition, exogenous consistency is further classified into vertical and horizontal consistency. Table 2.5 depicts the support of endogenous and exogenous (i.e., vertical and horizontal) consistency types of the selected primary studies. We can see that exogenous consistency (85.32%) is addressed more prominently than endogenous consistency (14.68%). Regarding exogenous consistency, we found that 66 studies (60.55%) focus on horizontal consistencies whilst 27 studies (24.77%) investigate vertical consistencies.

We found that seven of the studies (7.29%) out of 96 tackle the combination endogenous+horizontal consistency and two of the studies (2.08%) endogenous+vertical. There are two studies (2.08%) using the combination horizontal+vertical and one study (1.04%) uses endogenous+horizontal+vertical.

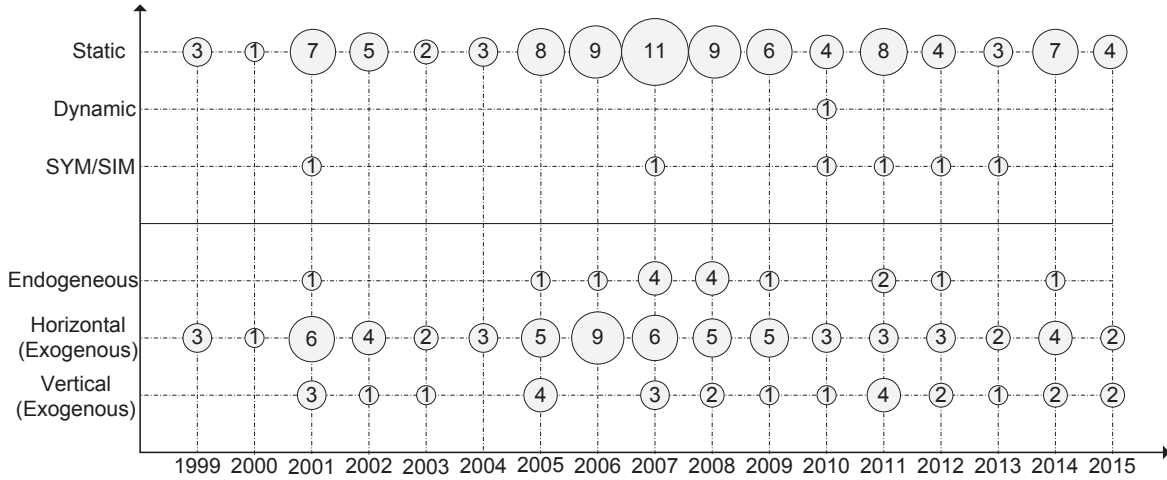


FIGURE 2.4: Primary Studies per Year Grouped by Types of Consistency Checking

The year-wise distribution of endogenous and exogenous consistency checking types is shown in Figure 2.4. We can see that much of research attention has been paid to the endogenous consistency checking from 2007 to 2008. Apart from that, horizontal consistency has received more attention from 2005 to 2009, while comparatively less attention has been given to the vertical consistency checking.

As the complexity of the systems increases, the timing requirements become more and more stringent, especially if the system’s reliability is a key concern. It is therefore necessary to investigate the studies that support for time-related consistency checking. Considering the existing methods

in terms of support for a notion of time during consistency checking, we propose a three-level category of time support in consistency checking, which includes “1=Not considered”, “2=Implicit using of underlying timing model or rules”, and “3=Explicit timing model and analysis”⁷. The first level refers to studies that do not consider time during consistency checking. The second and third level refer to studies that consider implicit usage of an underlying time model and an explicit timing model and analysis (i.e., using explicit timed models or real-time constraints), respectively.

Level	Studies	Total
1	S1–S5, S7, S10–S17, S19, S23–S29, S33, S37, S40, S41, S43, S45, S46, S48, S49, S51–S56, S58, S60–S68, S70–S72, S74–S94, S96	72
2	S6, S18, S22, S30, S32, S34, S39, S47, S50, S57, S73	11
3	S8, S9, S20, S21, S31, S35, S36, S38, S42, S44, S59, S69, S95	13

TABLE 2.6: Support for the Notion of Time in Consistency Checking

Table 2.6 presents the result of our SLR regarding the notion of time in consistency checking. We found that a majority of the existing studies do not consider support for time, i.e., 72 studies (75.00%) out of a total of 96. Only 11 studies (11.46%) support the second level “2=Implicit using of underlying timing model or rules”, whereas, 13 studies (13.54%) explicitly address timed models and real-time constraints. Naturally, these 13 studies stem from the domain of embedded and real-time systems where time critical conditions are often the highest priority.

2.3.4 Consistency Checking Techniques (Q2.3)

This section discusses the techniques used for checking the consistencies of software behavioural models. In particular, we have explored two major aspects of consistency checking techniques including the semantic domains (and correspondingly formal paradigms) employed by existing methods for formalizing the input models and constraints as well as the techniques (e.g., model checking, logical inference, theorem proving, etc.) that are used for performing consistency checking.

The descriptive results of investigating the semantic domain used by each approach are shown in Table A.1 of Appendix A. A semantic domain, when appropriately employed, can help reduce the ambiguity in modelling behaviour of software systems with precise mathematical terms [Eng+01]. In addition, grounding on a solid semantic domain enables the use of several model checkers and theorem provers for consistency checking. We adopt the classification of semantic domains into the following formal paradigms as proposed by Habrias and Frappier [HF06].

- **State transition:** The description defines the transition relation on a set of states.

⁷Please refer to Table C.1 in the Appendix C for further details of timing support scales

- **Algebra:** The description specifies the set of operations and their relations. An event is denoted by a function (also known as an operation). The behaviour of functions is specified by a set of equations (axioms) that describes how these functions are related. A special form of algebra, namely, *process algebra*, in which operations are applied to elementary processes and events, describes how events may occur.
- **Logic:** The behaviour of functions is specified by a set of equations (axioms) that describes how these functions are related [LMT09].
- **Other:** The formal semantic domains that do not exactly belong to either of the aforementioned definitions are included in this particular category.

Semantic Domain	Studies	Total
State Transitions	S2–S6, S8–S14, S16, S18, S20–S25, S27, S28, S30–S34, S36–S39, S43–S48, S50, S51, S53–S57, S60, S63–S66, S68, S69, S71–S74, S76–S79, S82–S84, S86, S87, S90, S91, S93, S94, S95, S97	69
Logic	S4, S11, S13, S18, S19, S21–S23, S26, S32, S34, S39, S42, S44, S45, S47, S50, S54, S57, S61, S67, S79, S84, S92–S94	26
Process Algebra	S1, S8, S15, S29, S35, S40, S41, S52, S58, S59, S62, S80, S85, S87–S89, S96	17
Others	S7, S17, S26, S49, S52, S70, S72, S75, S81	9

TABLE 2.7: Semantic Domains

Table 2.7 presents the studies that are classified according to the aforementioned semantic domains. 57.02% of the studies (i.e., 69 out of 121 studies) are based on state transitions. 17 studies (14.05%) use process algebra as the semantic domain. 26 studies (21.49%) leverage different kinds of logics as their semantic domain. The category “Other” shown in Table 2.7 embraces all studies (9, i.e., 7.44%) that use a semantic domain different from the three domains mentioned above. Please note that a study may employ more than one semantic domain, and therefore, may be based on multiple domains. 20 studies (20.83%) out of a total of 96 use the combination of state transitions+logics as their semantic domains, while two studies (2.08%) use state transitions+process algebra. There are three studies (i.e., 1.04%) using the combinations state transitions+other, logics+other and process algebra+other.

Considering the consistency checking techniques of the existing studies, we adopted and extended the four main categories proposed by Spanoudakis and Zisman [SZ01], which are: model checking (i.e., using or combining with existing model checkers), specialized algorithm (i.e., algorithms designed for analysing models to detect inconsistencies), logical inference (i.e., using formal inference techniques to derive inconsistencies), and theorem proving (i.e., reasoning using theorem provers).

Analysing the selected primary studies we have found that a wide range of studies used specialized algorithm to identify and detect the inconsistencies of software models, i.e., 51 studies (51.00%) out

of 100. The main reason is that these approaches investigate different sets of inconsistency problems on particular domains and, therefore, often propose specific algorithms or heuristics for the particular contexts being studied. About 41 studies (41.00%) use model checking techniques given the existence of several model checkers such as NuSMV⁸, SPIN⁹, FDR¹⁰, UPPAAL¹¹, GROOVE¹², and LSTA¹³, to name but a few.

Technique(s)	Studies	Total
Model Checking	S1, S4, S8, S9, S11, S13–S16, S18, S22, S23, S29, S32, S33, S34, S36, S39, S44–S47, S50, S53, S54, S57–S59, S63, S70, S78, S84, S85, S87, S88, S90–S94, S96	41
Specialized Algorithm	S2, S4–S7, S11, S12, S17, S20, S21, S24–S28, S30, S31, S35, S37, S38, S40, S41, S43, S44, S48, S49, S51, S52, S55, S60, S62, S64, S65, S66, S68, S69, S71–S77, S79, S80–S83, S86, S89, S95	51
Logic Inference	S10, S19, S42, S61, S67	5
Theorem Proving	S3, S56, S67	3

TABLE 2.8: Consistency Checking Techniques

We note that, even though a reasonable number of studies (21.49%) are based on the logic semantic domain as mentioned above, logic inference techniques are used only in five studies for identifying inconsistencies or checking consistency of software models (5.00%). The use of theorem proving techniques is even less frequent, they are used only in three studies (3.00%). These results partially explains the emergence of model checking techniques backed by powerful model checkers [CGP99].

Nonetheless, model checking techniques cannot thoroughly cover every aspect of consistency problems and may suffer from the problem of state explosion [Cla+11]. This could explain why a majority of the existing methods need to develop specialized algorithms for particular purposes. The summary of consistency checking techniques is shown in Table 2.8. Please note that, the sum of the numbers of studies on a technique category exceeds the total number of studies, because some studies used more than one technique. Two studies (2.08%) out of a total of 96 use the combination model checking+specialized algorithm to identify the inconsistencies of software models, while one study (1.04%) uses the combination specialized algorithm+theorem proving, and one study (1.04%) uses the combination logic inference+theorem proving techniques.

2.3.5 Inconsistency Handling (Q2.4)

Detecting behavioural inconsistencies of software systems is important but still far from complete. Handling inconsistencies has been considered a central task in consistency management [LLD98;

⁸See <http://nusmv.fbk.eu>

⁹See <http://spinroot.com>

¹⁰See <https://www.cs.ox.ac.uk/projects/fdr>

¹¹See <http://www.uppaal.org>

¹²See <http://groove.cs.utwente.nl>

¹³See <http://www.doc.ic.ac.uk/ltsa>

[SZ01]. Handling of inconsistencies addresses how to deal with any inconsistencies and analysing the impacts and consequences of particular methods of dealing with inconsistencies [SZ01]. We derive a scale from 1–5 for different degrees of handling with inconsistencies based on the set of activities proposed by Spanoudakis and Zisman [SZ01]:

- 1. Not mentioned / not considered
- 2. Systematic inconsistency diagnosis
- 3. Identifying handling actions
- 4. Evaluating costs and risks
- 5. Automated action selection and execution

Level	Studies	Total
1	S2–S10, S13, S14, S16, S17, S20–S27, S29–S34, S36–S38, S40–S44, S46–S59, S63–S70, S72–S78, S80–S82, S85–S96	78
2	S1, S11, S12, S15, S18, S19, S22, S28, S35, S39, S45, S60–S62, S71, S79, S83, S84	18
3–5	–	0

TABLE 2.9: Degree of Inconsistency Handling

In summary, inconsistencies can be handled by different activities including systematically diagnosing inconsistencies and their causes, identifying necessary actions for resolving inconsistencies, evaluating costs and risks, and automatically selecting and execution the corresponding actions.

Table 2.9 shows our analysis results for inconsistency handling. We have found that most of the existing approaches provide little or even no support for this important aspect. In particular, a large number of studies (78, i.e., 81.25%) do not consider any sort of inconsistency handling. Only 18 studies (18.75%) support certain forms of inconsistency diagnosis, for instance, analysing the counterexamples or error traces back to the origins of the problems. Unfortunately, none of the studies supports any higher levels of inconsistency handling (i.e, from 3–5).

2.3.6 Automation Support (Q2.5)

Some prior secondary studies such as Lucas et al. [LMT09] examined the support for automation in model consistency checking but they propose only two levels “yes” (automated) and “no” (manual). In fact, there exist some techniques where human intervention is partially necessary for specifying models or consistency rules, and the rest can be automatically performed. Moreover, there are different phases in consistency checking including specification, checking, and possibly handling inconsistencies (as mentioned above). Therefore, first of all, we decided to refine the automation

support into a three-level scale in order to cover the aforementioned cases: which are “1=Manual”, “2=Semi-automated” and “3=Fully automated”¹⁴. Moreover, we investigate the automation support in three main phases of consistency checking (i.e., specification, checking and handling inconsistencies) as analysed above and assign the corresponding level for each primary study.

We noticed that the phase consistency checking, due to sound formalisation and well-defined checking algorithms and/or techniques, can be performed automatically (i.e., reaching the level 3 “Fully-automated”). Unfortunately, this is not the case for inconsistency handling. As only a few of the existing approaches consider a limited form of inconsistency handling, which is the diagnosis of the checking results, the level of automation supported by these approaches is 2 (i.e., “Semi-automated”) because human intervention is necessary for investigating the yielded errors. For these reasons, we do not present details for the two phases consistency checking and inconsistency handling, but rather focus only on the specification phase.

Automation Level	Studies	Total
1	S3, S5, S10, S11, S12, S18, S19, S21, S22, S23, S26, S29, S35, S43, S44, S46, S47, S48, S50, S54, S57, S59, S65–S67, S70, S77, S79, S93, S94	30
2	S1, S2, S4, S6, S7, S9, S13–S17, S24, S25, S27, S28, S30–S34, S36, S38, S39–S42, S45, S49, S51–S53, S55, S56, S58, S60–S64, S68, S69, S71–S76, S78, S80–S92, S95, S96	62
3	S8, S20, S37, S74	4

TABLE 2.10: Automation Support for Specification Phase

We present the distribution of the existing approaches in the literature with respect to the aforementioned scale of automation support in the specification phase in Table 2.10. A considerable number of studies (31.25%) falls into Level 1 (i.e., “Manual”) because they either assume the existence of formal logic constraints or require manual efforts in specifying the input consistency constraints (e.g., rules specified in LTL/CTL etc.) or, sometimes, the input models. We note that these manual tasks ask for considerable knowledge of the underlying formalisms and formal techniques, and therefore, are often not easy to use for many software engineers. A larger number of studies (64.58%) assume the existence of the input models but propose automated transformations of these inputs into formal representations. For this reason, we consider them “semi-automated” because strictly speaking the input models assumed by these approaches are often manually created. Nonetheless, these are design models and created during the modelling and development of software systems, anyway, and there is automated translation support. Only four studies (4.17%) can be considered “fully-automated” as they encode consistency rules in the corresponding algorithms or libraries and, therefore, do not require any manual effort in the specification phase. However, the downside could be that the hard-coded consistency rules could lessen the extensibility and generality, and therefore, the scope of application of these approaches.

¹⁴Please refer to Table C.3 in the Appendix C for further details of automation support levels

Tool Support for Consistency Checking (Q2.5, Q3) Another important dimension is to consider tool support provided by the existing approaches for consistency checking. We believe that more or better tool support would showcase a proof of feasibility and attract the attention of practitioners to the corresponding techniques. The lack of any sort of tool support (e.g., research prototypes, demos, etc.) will hinder practitioners to use or at least explore the corresponding proposed methods and techniques, and therefore, hinder the transfer of research results into industrial practice.

We propose three levels of evaluation of tool support as shown in Table C.5: “1=Not mentioned/Not considered”, “2=Only using existing tools/libraries”, and “3=Prototypes (includes using existing tools/libraries)”¹⁵.

Level	Studies	Total
1	S2, S6, S11, S17, S24, S27, S38, S42, S64, S66, S68, S71, S73, S75, S81, S83	16
2	S3, S4, S9, S13, S15, S22, S23, S29, S31, S33, S40, S41, S43, S47, S50, S55–S59, S69, S70, S74, S77, S78, S80, S85, S87–S91, S93, S95	34
3	S1, S5, S7, S8, S10, S12, S14, S16, S18–S21, S25, S26, S28, S30, S32, S34–S37, S39, S44–S46, S48, S49, S51–S54, S60–S63, S65, S67, S72, S76, S79, S82, S84, S86, S92, S94, S96	46

TABLE 2.11: Evidence of Tool Support for Consistency Checking

The distribution of levels of tool support found in the literature on software model consistency checking is shown in Table 2.11. We did not find evidence of tool support for detecting and handling inconsistencies in 16 studies (16.67%). 34 studies (35.42%) used existing tools and libraries to carry out consistency checking, for instance, model checkers such as SPIN, (Nu)SMV, UPPAAL, FDR, GROOVE, LSTA, and Maude or theorem provers such as SPASS and Z/Eves. Out of 96 there are 46 studies (47.92%) that provide specific tool support for checking consistency between software models including the development of prototypes or tool-chains that combine the implementation of various aspects such as model transformations, consistency checking algorithms, and/or existing tools and libraries. Figure 2.5 shows the distribution of primary studies per year grouped by levels of tool support. We can see that the trend of tool support for checking consistency by using existing tools and libraries (i.e., level 2) has received more attention from 2005 to 2008, while comparatively the developing prototypes or tool-chains (i.e., level 3) reached a peak in 2014.

2.3.7 Types of Study and Evaluation (Q2.6)

As a part of this SLR we investigated the nature of evidence presented in the selected primary studies and analysed how an evaluation is performed and reported. The main motivation is that different types of studies provide different strengths of evidence and evaluation. Practitioners

¹⁵Please refer to Table C.5 in the Appendix C for a full description of tool support levels

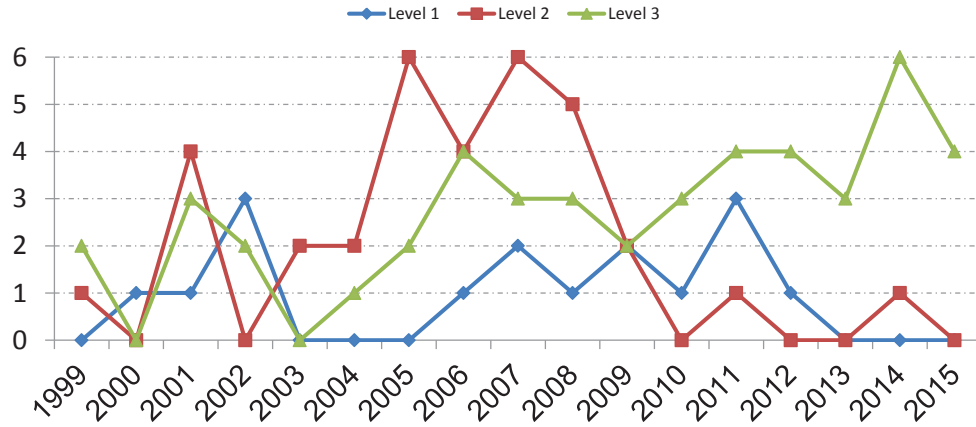


FIGURE 2.5: Levels of Tool Support per Year Distribution

could take the strength of evidence and evaluation into consideration before adopting a specific methodology and tool.

We adopted the classification of types of study and evaluation in six categories as proposed by Chen and Ali Babar [CA11]. The six types of study and evaluation are Rigorous Analysis (RA), Case Study (CS), Discussion (DC), Example (EX), Experience Report (ER), Field Study (FS), Laboratory Experiment with Human Subjects (LH), Laboratory Experiment with Software Subjects (LS), and Simulation (SI)¹⁶.

Types	Studies	Total
DC+EX	S7, S16, S47, S50, S57	5
RA+DC	S2, S13, S30, S31, S52	5
RA+DC+EX	S8, S9, S15, S18–S20, S24, S25, S28, S32, S36–39, S41, S45, S46, S48, S49, S53, S54, S58, S59, S61–S66, S74, S76, S79, S82, S87, S90	35
RA+DC+EX+LS	S12, S22, S26, S35, S55, S72, S81, S83	8
RA+EX	S1, S3, S4, S6, S10, S11, S14, S17, S21, S23, S27, S29, S33, S34, S40, S42–S44, S51, S56, S60, S67, S68, S71, S73, S75, S77, S78, S80, S84–S86, S88, S89, S91, S93–S96	39
RA+EX+ER	S5, S69, S70, S92	4

TABLE 2.12: Types of Study and Evaluation

Analysing the data extracted from the selected studies, we see that a study may be a combination of different types. For instance, S8 presents and discusses formal foundations, consistency problems and gives concrete examples in BPEL. Thus, we clustered the data and divided them into appropriate groups based on the combination employed by each study. Table 2.12 shows the results of data analysis and clustering.

¹⁶Please refer to Table C.7 in the Appendix C for a full description of these categories

We observed that a great amount of the studies (36.46%) use the combinations RA+DC+EX or RA+EX (40.63%). There are eight studies (8.33%) using the combination RA+DC+EX+LS that, in addition to what is done in RA+DC+EX, also perform some sorts of evaluation with software models, for instance, rigorously estimating the scalability, correctness, algorithm precision or comparing with other approaches. Only a few studies (4.17%) report RA+EX+ER, experiences in some academic settings or application scenarios derived from industry practice. Unfortunately, none of the existing studies presents clear evidence using any sort of empirical study or validation.

Along with the aforementioned investigation on types of study and evaluation, we also assessed the level of rigour of the reported evaluations. Rigor is concerned with assessing how an evaluation is performed and reported [Gal+14]. In particular, we examined how well the selected studies report their evaluations with respect to three dimensions: the context of the evaluations (*C*), the design of the studies conducted in the evaluation (*S*), and the validity discussion (*V*). We leverage the score for each rigour dimension as proposed in [IG11] that comprises three levels “1=Weak”, “2=Medium”, and “3=Strong”¹⁷. Figure 2.6 depicts an overview of the results regarding the aforementioned dimensions of the evaluation’s rigour

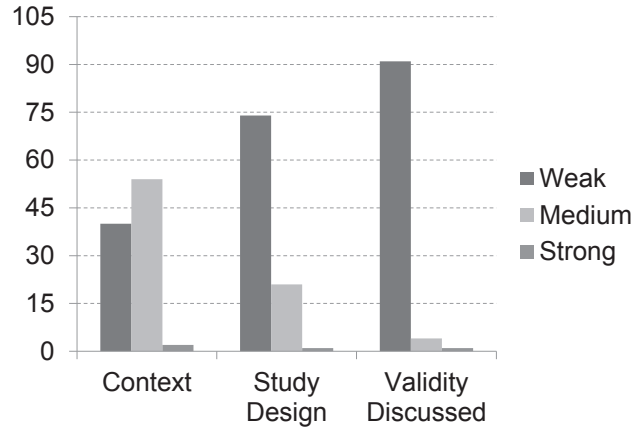


FIGURE 2.6: Sum of Rigour Scores per Dimension

We note that the rigour measure proposed above could be more appropriate for empirical studies. Applying the rigour measure uncovered that many studies mentioned dimensions related to rigour but do not describe these fully. Nonetheless, we consider this measure in this chapter in order to show the lack of rigorous evaluations and provide some insights which researchers can consider when designing and conducting their studies to achieve a reasonable degree of quality.

The results shown in Figure 2.6 indicate that very few studies have been scored the strongest level in each dimension. The evidence regarding describing the study design and the validity discussions is very poor. It also reflects our analysis on the type of study and evaluation where only a small number of studies perform certain rigorous experiments with software models.

¹⁷Please refer to Table C.8 in the Appendix C for a full description of the rigour levels

To illustrate the distribution of rigour measures, we considered a collective rigour metrics for each study, namely, R , which is calculated as $R = C + S + V$, following the suggestions of [IG11] and [Gal+14]. Thus, the value of R for each study is an integer in the range of 3–9. We show the distribution of the collective rigour metrics R in Figure 2.7.

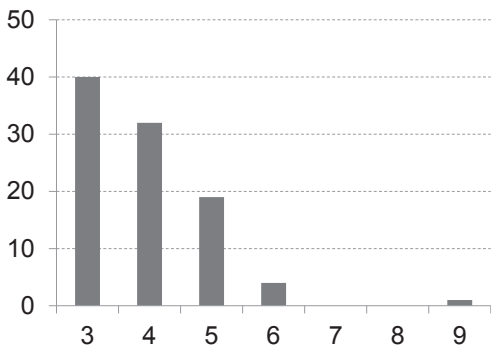


FIGURE 2.7: Distribution of the Collective Rigour Metrics R

In accordance with our prior observation, the collective scores of the rigour metrics of the selected studies are distributed significantly towards the lower side of 3 (40 studies), 4 (32 studies), and 5 (19 studies), respectively. None of these studies could achieve the collective score of 7 or 8. The only study that reaches the best collective score is S26 in which the authors present rigorously and thoroughly all three dimensions of evaluation rigour Figure 2.8 presents the temporal distribution of the rigour metrics for each study. We can see that the trend of rigour evaluations of studies increased in the last ten years, but overall the rigour is still rather low.

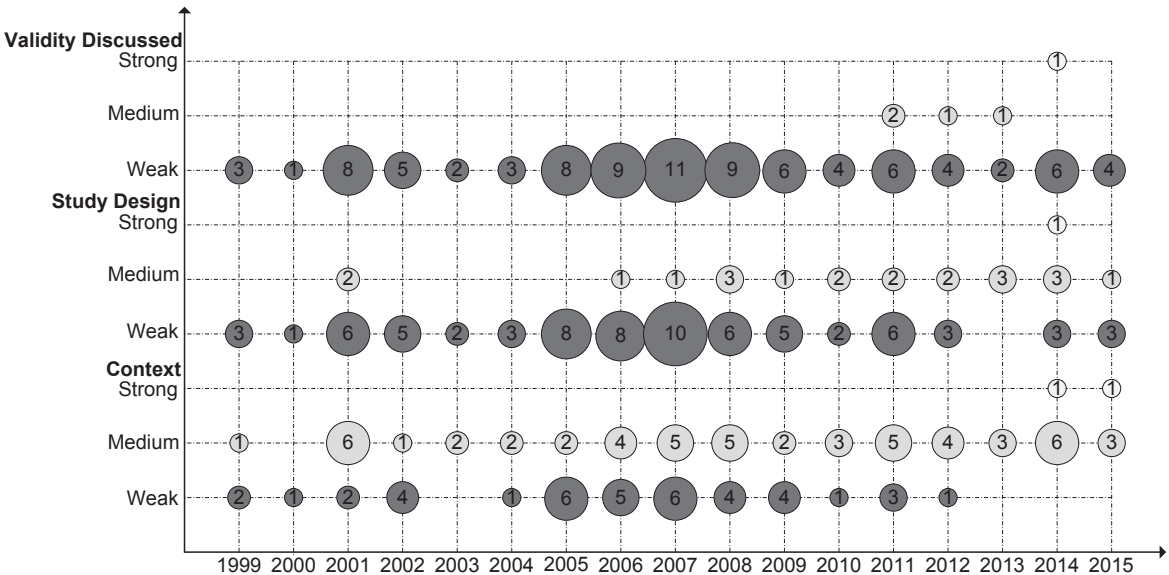


FIGURE 2.8: Trend of Rigour Dimension per Year

2.3.8 Practical Impact (Q3)

Analysing the practical impact, or in other words, the effectiveness, of the existing methods for consistency checking of software behavioural models is a very challenging task because there is no consensus on an ultimate measurement or metrics for this aspect, to the best of our knowledge. After studying other relevant surveys in the field of computer science and software engineering, such as [Alv+10; Gal+14], to name but a few, we propose to examine the practical impact of the existing methods from different dimensions, including citations (how a study influences the others), CASE/IDE tool integration (support in daily working environment of practitioners), tool support (feasibility study: mentioned in Section 2.3.6), degree of evidence, and the applicability of the proposed techniques in industrial settings.

Citations Table 2.13 presents an overview of the top 10 of most highly cited studies along with the year of publishing, the total number of citations, and the average number of citations per year. The numbers of citations are obtained from Google Scholar. We note that these numbers are roughly estimated because Google Scholar may mistakenly count the citations of different authors who have similar or the same names. Also please note that these numbers can change over time. We sampled two times on Aug 9, 2015 and Aug 19, 2015 and noticed slight differences. The presented numbers are obtained as of date Aug 19, 2015. Nonetheless, the average of citations per year is quite stable as the change margins are small.

SID	Author(s)	Year	Total Citations	Avg. Citations per Year
S6	Bernardi et al.	2002	322	25
S63	Schäfer et al.	2001	247	18
S55	Nejati et al.	2007	243	30
S27	Harel et al.	2002	196	15
S36	Knapp et al.	2002	188	14
S15	Engels et al.	2001	169	12
S76	Van der Aalst et al.	2008	152	22
S18	Eshuis & Wieringa	2004	137	12
S28	Hausmann et al.	2002	127	10
S83	Weidlich et al.	2011	124	31

TABLE 2.13: Top 10 Most Cited Studies

Because each of the studies in the top 10 is highly and frequently cited, we can draw the conclusion that at least the studies in the top 10 have substantial influence on other researchers. Unfortunately, we could not find sufficient evidence to conclude on the correlation between the citations and other aspects such as the used semantic domains, methods, techniques, or application domains.

Development Tool Support and Integration The second dimension of the practical impact that should be considered is the integration of the proposed techniques into the working environments of practitioners (such as software analysts, software architects, software developers), i.e., development tool support. Here, specifically, CASE¹⁸ tools and IDEs¹⁹ are relevant. CASE/IDE integration could foster the practical use of the consistency checking techniques. It could help users of the development tools to obtain feedbacks and support them in handling inconsistencies detected in the models. Researchers could benefit by receiving improvement suggestions from the practitioners who are using the corresponding CASE/IDE tools in industrial settings.

We propose a three-level scale to assess the evidence of IDEs/CASE integration that comprises “1=Not mentioned/Not considered”, “2=Proposed/planned integration”, and “3=Fully implemented integration”²⁰. Table 2.14 shows the analysis result of CASE/IDE integration found in the selected studies. The weakest scale is applied for studies that do not consider or mention at all about the integration with any CASE/IDE. There are 80 studies (83.33%) that fall into this level. On the other end of the spectrum, there are only five studies (5.21%) that develop fully working integration with existing CASE/IDE tools, i.e., reaching the strongest level. The rest of the studies (11.46%) (i.e., 11 out of total 96 studies) mainly propose plans for CASE/IDE integration or partially implement the integration.

Level	Studies	Total
1	S2–S4, S6, S7, S9–S15, S17–S23, S26, S27, S29–S33, S35–S43, S45–S47, S49–S52, S55–S59, S61–S68, S71–S78, S80–S96	80
2	S5, S24, S28, S34, S48, S53, S54, S60, S69, S70, S79	11
3	S1, S8, S16, S25, S44	5

TABLE 2.14: CASE/IDE Support and/or Integration

Application in Industrial Settings For many research works in the field of software engineering, it is significant to apply and evaluate newly proposed or improved techniques in a real industrial settings. As Ivarsson and Gorschek [IG11] emphasize, research methods and techniques need to provide tangible evidence of the advantages of using them in order to impact industry. Ivarsson and Gorschek [IG11] also suggest a step-wise validation to enable researchers to test and evaluate in real settings with real users and applications (i.e., *empirical evidence*).

Hence, in our study we examined the evidence of any industrial settings presented in the chosen primary studies. The types of evidence could be critical for researchers to identify new topics for empirical studies, and for practitioners to assess the maturity of a particular method or tool. Some of the approaches use (small-scale) industrial scenarios to illustrate the applicability of their

¹⁸CASE: Computer-Aided/Assisted Software Engineering¹⁹IDE: Integrated Development Environment²⁰Please refer to Table C.4 in the Appendix C for a full description of CASE/IDE support/integration levels

techniques whilst others evaluate their approaches by using small examples or cases (i.e., providing the readers with a rough idea of how to apply and use the proposed approaches). We adopted the 6-level scale of evidence as proposed by Alves et al. [Alv+10]²¹:

- 1. No evidence provided.
- 2. Evidence obtained from demonstration or working out toy examples.
- 3. Evidence obtained from expert opinions or observations.
- 4. Evidence obtained from academic studies.
- 5. Evidence obtained from industrial studies.
- 6. Evidence obtained from industrial practice.

Level	Studies	Total
1	–	0
2	S2–S11, S13–S19, S21, S23–S31, S34, S36, S38, S40–S47, S49, S51, S53, S54, S56–S58, S60, S62–S64, S66–S68, S71, S73–S80, S84–S96	73
3	S1, S12, S22, S32, S33, S39, S44, S48, S50, S59, S61, S72	12
4	–	0
5	S20, S35, S37, S52, S55, S65, S69, S70, S81, S82, S83	11
6	–	0

TABLE 2.15: Levels of Empirical Evidence

The data of our investigation and analysis shows that 73 out of the total 96 studies (76.04%) are tested and evaluated using toy examples. 12 studies (12.50%) use some scenarios obtained from expert opinion or observation. Only 11 studies (11.46%) show a higher level of evidence based on industrial studies. None of the studies support the Levels 4 and 6. Table 2.15 shows the distribution of empirical evidence found in the literature on software model consistency checking. We note that there are no studies of Level 1 either. After closer analysis, we found that this is mainly due to the fact that studies falling into this level have been excluded because they are not presented at an acceptable quality level (c.f. Section 2.2.2).

We map roughly the six corresponding levels of evidence mentioned above into two levels that indicate whether the proposed methods/techniques are tried out in any industrial settings. Levels 4–6 of evidence can be considered being applied or tested in industrial settings whilst Levels 1–3 are not. We summarised the data according to this two-level scale, shown in Figure 2.9. Only 11 studies have validated or evaluated their proposed techniques and tools in any sort of industrial settings. The majority of studies (85, i.e., 88.54%) did not (but rather using other means such as

²¹Please refer to Table C.6 in the Appendix C for a full description of evidence levels

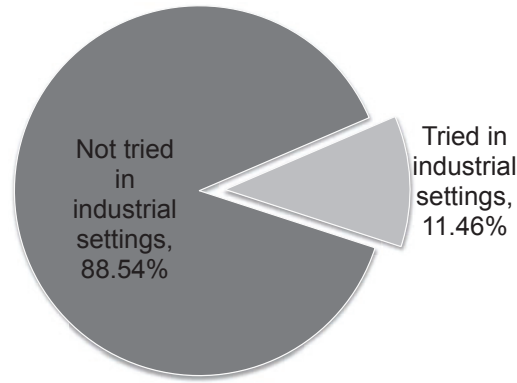


FIGURE 2.9: Studies Tried in Industrial Settings

rigorous analysis, discussion, and toy examples). We also observed that only one study (S70) out of 11 studies (11.46%) was evaluated in industrial settings in the year 2001 while all the others were evaluated between 2007 and 2015. That is, while the level industrial evidence seems rather weak in the total set of studies, it has clearly increased in recent years.

2.4 Discussions

In this section, we will discuss the limitations of existing methods to answer the remaining Research Question Q4 (c.f. Section 2.2.1) and discuss the validity of our study.

2.4.1 Limitations of the Existing Methods

This SLR is an attempt to give a comprehensive view of the state-of-the-art of research in software behavioural model consistency checking. Based on our analysis and interpretation presented in this chapter, we observe a number of limitations and open problems in the existing methods that could potentially point into interesting further research directions.

Unbalanced Focus of Consistency Checking Problems There is a predominance of primary studies concentrating on static consistency checking and little attention has been paid on dynamic checking or consistency checking by means of symbolic execution or simulation. The huge advantage of static checking is to identify many inconsistencies without the need of and/or connection to any running systems. However, static checking also has considerable limitations like suffering from the state explosion problem [Cla+11] or being computationally expensive. A large number of static checking based approaches use model checkers, which only support finite and discrete data types (e.g., boolean, integer, vector) and provide limited or even no support for continuous data types (e.g., floating-point numbers) or strings.

Many distributed software systems are built upon highly flexibly and dynamic architectures like event-driven architectures [MFP06]. Besides, there is also an emerging trend in software development on supporting higher degrees of flexibility [Aal13; GVC15] or on-the-fly adaptations [ME+13] at runtime. Static consistency checking of these types of flexible systems might not be solely enough as several unanticipated changes can happen during system execution. Symbolic execution or simulation can help to partially examine these dynamic changes, but ultimately, dynamic checking, through either system instrumentation or monitoring, should be employed for catching any potential inconsistencies. In the context of exogenous consistency checking the result shows that vertical consistency, has drawn much less research interest than horizontal consistency checking.

Similar observations can be made (in a less severe manner) for the case of support for time-related consistency checking. A rather small portion of the existing methods investigates time-related inconsistencies, especially in the context of real-time constraints. With respect to the niche fragment of the domain of embedded and real-time systems versus the remaining application domains, we do not see this as critical nuisance.

Focus on Specific Model Types Besides consistency checking problems, we identified, research on behavioural models consistency checking focuses on four major types of behaviour models, which are statecharts, sequence diagrams, process models, and activity diagrams. Considerably less attention has been paid on collaboration diagrams, LSCs, stateflow, and labelled transition systems.

Lack of Consideration for Inconsistency Handling Spanoudakis and Zisman [SZ01] emphasized the vital role of inconsistency handling and suggested many aspects to consider for adequately handling consistency issues. Improper or inadequate inconsistency handling could lead to severe negative effects [SZ01] and could be one of the major obstacles to making significant practical impact or to transferring the proposed methods and techniques to industry practice. Unfortunately, we found limited evidence of inconsistency handling on diagnosing inconsistencies. Most of the studies produce consistency checking outcomes in terms of formal representations such as counterexamples [CGP99] or erroneous traces of states. However, this assumption should be reconsidered because most of the practitioners (e.g., software engineers) do not have sufficient knowledge of the underlying formal methods to understand these outcomes. As a result, a considerable amount of time and effort is consumed by the practitioners in order to handle the detected inconsistencies. It would be more pragmatic to present the consistency checking outcomes in appropriate forms of representation (which can be textual or graphical notations or visualization) with suitable abstractions that practitioners can better comprehend and use to resolve the problems detected in the models. Further important activities such as identifying corresponding handling actions, estimating the costs and risks of handling inconsistencies, and eventually choosing and executing handling actions need to be taken into account.

Lack of Rigorous Studies, Evaluation, and Practical Impacts Lack of rigorous studies, evaluation, and practical impacts reflect the quality of studies being conducted in the literature of software behavioural model consistency checking. As we observed in Section 2.3.7, several primary studies tend to focus on introducing new ideas and solutions to consistency checking problems but fail to evaluate their contributions properly and show the validity of their approaches in larger contexts. This issue is mainly due to less consideration of rigorous study design and validity discussion. Moreover, our findings in Section 2.3.7 also show that many of the research results have not been tried out in any sort of industrial settings. As a consequence, these results remain pure academic contributions and exercises. This could be explained as academic results tend to be context-specific, and thus, difficult to generalize to many industrial situations [Alv+10]. While the average rigour of evaluations has increased in the last ten years, it is still at a very low level. The strongest case found in this SLR is a study which has been thoroughly and rigorously designed with test cases based on industry practice. We expect that the categories of types of study and evaluation presented in Table C.7 and discussed in Section 2.3.7 could be considered for design and evaluation of future studies to achieve more scientific and repeatable results that have greater practical impact.

Another aspect that is crucial for gaining more practical impact is to provide adequate tool support for consistency checking and/or integration with existing development tools (such as CASE/IDE tools). It is noted that automation is usually achieved by integrating the techniques with existing software development tools. The benefit of tool support for consistency checking and development tool integration, which has been underlined above, is twofold. It provides intuitive means for fostering the application of the implemented techniques in industrial settings, and it allows for obtaining valuable feedbacks from practitioners who are able to use the tools for their daily tasks. Unfortunately, only a small number of research results from the existing studies have been incorporated in commercially used development tools. The rest is either focused on developing research prototypes or using existing tools and libraries (e.g., model checkers, theorem provers).

2.4.2 Study Validity

The main threats to validity in this systematic review are potential bias in our search strategy, the selection of the studies to be included, and data extraction. We aimed for an exhaustive list of high quality primary studies. Therefore, we followed strictly the guidelines recommended in [KC07] and took into account lessons learned in [Bre+07]. We have developed a clear protocol for searching and choosing primary studies (c.f. Section 2.2) including defining research questions, inclusion and exclusion criteria, and search strategy.

Despite a well-defined systematic procedure, we acknowledge that there may be missing primary studies, for instance, due to the coverage and quality of the search engines and search portals that

we used in our search. It is crucial with respect to the extent where relevant keywords are not standardized or clearly defined [Bre+07]. That is, various studies may use the same terms with different semantics. To deal with this kind of threat, we decided to carry out a “snowballing” process [Bud+11] in which two researchers manually scanned and analysed the references and citations of the primary studies retrieved from the automated search with the search engines/portals. The main goal is to make sure that our SLR also covers follow-up works that might exist but have not been included in the search. As we presented in Section 2.2.2 the snowballing process has caught 72 missing studies out of which 26 studies have been selected.

Bias in study selection may be due to different interpretation and understanding of the researchers involved in the SLR and potential misalignment between our term and category definitions and other definitions. To alleviate these problems, two researchers performed an in-depth analysis and selection of primary studies with respect to the predefined inclusion and exclusion criteria and clearly recorded the reasons for including or excluding. The third researcher reviewed independently and provided corresponding judgments, especially for the cases of discrepancy or “in doubt”. Then all researchers discussed and came to the final conclusion for each study.

During the data extraction phase, we confronted some difficulties, and thus, potential bias, in extracting objective information from the selected primary studies. First of all, this issue could be due to the subjective interpretation and/or poorly described report of relevant terms, methods, techniques, evaluations, and so on. Another difficulty is that different studies could choose different research and evaluation methods as mentioned in Section 2.3.7, and therefore, can compromise the accuracy of data extraction. We tried to minimize this bias by, first of all, defining the extraction strategy and date format with clear encoding descriptions in order to ensure consistent extraction of relevant data. Furthermore, the extraction has been performed independently by two researchers. Any sort of discrepancy from all involved researchers has been adequately recorded. The third researcher then performed the final reviewing and cross-checking. After that, we conducted discussion meetings to resolve any remaining divergence and disagreements. Nonetheless, we acknowledge that there is still a certain possibility of misunderstanding regarding the way we extracted data from the primary studies.

Regarding external validity concerned with generalizing our SLR’s findings, we aimed for a representative set of primary studies with clear research scope and objectives. We set no constraints on the time period such that our SLR can cover from the beginning of research on consistency checking in the eighties to the year 2015. Nonetheless, our findings might not be the same when being generalized to broader scopes or time periods. This could be due to the fact that our current findings are mainly based on qualitative analysis. We could consider rigorous quantitative analysis and inferences to enable the possibility of analytical and statistical generalizations but this is rather beyond the scope of this SLR.

2.5 Conclusions

This chapter addressed the Research Question RQ1 by means of an SLR in the area of software behavioural consistency checking. We aimed to identify the current trend in this field and investigate various dimensions ranging from used methods, languages, techniques to the practical impacts of the identify studies. To achieve this, we have identified a total of 1770 studies by combining automated searches and manual snowballing, out of which 96 have been studied in-depth according to our predefined SLR protocol. Through in-depth analysis and interpretation of the collected data, we obtain many interesting findings along with a number of gaps and open problems that could provide insights for further investigation. In summary, there are promising accomplishments thanks to sound formal foundations that have been employed in most of the existing studies for consistency checking of software behavioural models. The results of our SLR show a real need for improving the quality of study design and conducting evaluations to achieve solid and repeatable scientific results that have greater impact in both academia and industry. The results also indicate the need for inconsistency handling techniques, better tool support for consistency checking and/or development tool integration.

This SLR confirmed that the containment relationship, addressed in this dissertation, has not been studied in the literature so far. In the next chapter, we will introduce our containment checking concepts and tool support, which can be classified in the model developed in this SLR as vertical consistency checking of software behavioural models and inconsistency handling.

3

Model Checking Based Containment Checking of UML Activity Diagrams

The approach presented in this chapter aims to verify whether a certain low-level activity diagram is consistent with the specification provided in its high-level counterpart based on model checking techniques. We interpret the containment checking problem as a model checking problem, which has not received special treatment in the literature so far. This problem addresses Research Question RQ2. Furthermore, the chapter is based on a peer-reviewed workshop paper in the proceedings of 11th International Workshop on Formal Engineering approaches to Software Components and Architectures [MTZ14] and an article submitted to a peer-reviewed journal Science of Computer Programming.

3.1 Introduction

During the modelling phase of a software system, behaviour models such as UML activity diagrams, sequence diagrams, statecharts [Gro11b], Business Process Model and Notation (BPMN) [Gro11a], and Event Driven Process Chain (EPC) [Sch02] are often used to describe how the system behaves. Because of the involvement of different stakeholders in constructing these models and their independent evolution, inconsistencies might occur between the models at different levels of abstraction [SZ01]. This issue becomes extremely relevant for software development processes in which different models are used to derive or generate the system's implementations from its specifications in a stepwise manner. That is, in the long run, the inconsistencies might make the implementations drift apart from the specifications. The inconsistencies that are detected at later stages, when the system is already implemented or tested require huge amounts of time and effort for correction, revision, and verification. Therefore, it is crucial to detect and fix the inconsistencies at early phases of software development, and especially as soon as refined models deviate from their abstract counterparts. This has led to a rich body of work for checking and managing model consistency in the literature [SZ01; LMT09]. Of these existing approaches, only a few deal with consistency checking of behavioural models for software systems [SZ01; LMT09], for instance, checking behavioural models against non-behavioural models [TE00; EW01; Yeu04], or checking

different types of behaviour models [Wan+05; SKM01; KMR02; LP05]. Nonetheless, there are very few studies on checking the deviation of software behavioural models at different abstraction levels.

This work investigates the problem of containment checking for software behaviour models instead of aiming to solve more general consistency checking problems. Containment checking is a special type of consistency checking that verifies whether the behaviour (or functions) described by the low-level model encompasses those specified in the high-level counterpart. It improves the quality and reduces the complexity of big and complex system by determining and resolving the deviations between the low-level behaviour models and its high-level counterpart in the design phase. However, an unsatisfied containment relationship could break the design rather than improve its quality. The containment relationship mainly aims at unidirectional consistency because the low-level behaviour models are often constructed by refining and extending the high-level model. To date, however, none of the published studies has performed a comprehensive exploration of the containment checking problem in software behaviour models.

A general technique employed by most of the existing consistency checking approaches in the literature (c.f. [SZ01]) is to describe the behaviour models under study in form of formal specifications and derive consistency constraints that these specifications must satisfy. These approaches often presume that formal specifications of the systems under consideration and consistency constraints can be easily created. Unfortunately, this makes the approaches hard to apply in practice because creating formal specifications and consistency constraints requires considerable knowledge of the underlying formalisms and formal techniques [SKM01]. Moreover, this task is often accomplished in a laborious and manual manner, which is also error-prone [LP05; SS00]. Besides that, the model checking approach used in our work as a basic technique for realising containment checking is known to be computationally expensive [Bur+92; CW96; CGP99], it is therefore necessary to investigate the applicability and technical feasibility of the approach for realistically sized of models.

The approach presented in this chapter aims to address the aforementioned key shortcomings, which can be summarised by the following fundamental research questions:

- **Q1:** Can we achieve formal specifications and descriptions for containment checking fully automatically?
- **Q2:** Can we design a containment checking approach that offers an acceptable performance for realistically sized input models?

We interpret the containment checking problem of software behaviour models as a model checking problem. That is, the behaviour described in the high-level model can be considered as consistency constraints that the execution of the corresponding low-level model must conform to. This way,

the first research challenge involves two primary tasks: (**T1**) deriving formal specifications/consistency constraints from the behaviour described by the high-level model; and (**T2**) deriving formal descriptions from the corresponding low-level behaviour model. The model checkers will be fed with the outcomes of these tasks to verify their satisfaction. A positive result yielded from the model checker implies that the formal descriptions (resp. the behaviour of the low-level model) satisfy the specification (resp. the behaviour of the high-level model), and vice versa.

Unlike existing model checking based techniques, our approach does not presume the presence of specifications and descriptions. We aim at enabling the automated transformation of the input behaviour models into the corresponding formal specifications and descriptions. This way, our approach can help to alleviate the burden of manually encoding consistency constraints, and therefore, increase productivity and avoid potential translation errors. Whilst Task **T2** might be efficiently achieved by adapting and extending existing approaches on transforming behaviour models to formal descriptions accepted by the model checkers such as those presented in [EW02; Esh06; Lam07; Lam08], no existing techniques can be leveraged for accomplishing Task **T1**. We propose an automated method for transforming a behaviour model into temporal logic based consistency constraints and formal behaviour descriptions, respectively. In particular, Linear temporal logic (LTL) [Pnu77] and the state based SMV language are used for formalising the input models, and the NuSMV model checker [Cim+99] is used for verification. The behaviour models studied in this chapter are UML 2 activity diagrams as they are widely used in both academia and industry for modelling and analysing behaviours of software systems.

In order to address the second research question, we have applied our approach in scenarios from four industrial case studies representing real systems from the banking and e-business domains. Specifically, we investigated the performance of the proposed approach on these scenarios as well as synthetic larger model to assess whether it supports the developers to verify the containment relationship during their development tasks.

The contributions presented in this chapter are as follows. First, we devise an efficient method for transforming the input behaviour models to formal specifications and descriptions. In particular, the formalisation of activity diagrams is based on the explicit representation of the control nodes in the formal models rather than implicit encoding. Second, this research covers a more comprehensive set of modelling constructs used for describing software behaviour. Apart from the fundamental set of modelling constructs, such as exception handlers, interruptible activity regions, parameterized tasks, event actions, and loops, that are widely used for modelling complex circumstances in software systems. Third, we applied our approach in scenarios from four industrial case studies in order to evaluate the applicability and technical feasibility of our approach, and to measure its performance in a typical developer's working environment. Finally, we discuss the generalisation of our approach including the applicability for other kinds of behaviour models, such as sequence

diagrams, statecharts and business process models, as well as the possibility to realise our approach for different formalisations and model checking techniques.

The rest of this chapter is organized as follows: Section 3.2 describes the approach for containment checking based on model checking in detail. The scenario extracted from industrial case study is described in depth to illustrate the proposed approach in Section 3.3. In Section 3.4, we present performance evaluations of the approach using four scenarios from industrial case studies in order to examine the feasibility and applicability of our technique in an industrial context. In Section 3.5, we discuss various aspects and challenges of supporting containment checking. Section 3.6 discusses the related work regarding behavioural consistency checking techniques in general and containment checking in particular, and formal semantics of behavioural models. Finally, Section 3.7 summarises the chapter.

3.2 Containment Checking Approach

Our study aims to address the problem of checking whether the behaviour (or functions) described by the low-level model encompasses those specified in the high-level counterpart, in order to improve the quality of software systems. That is, the “*execution*” of the low-level model must embrace the “*execution*” prescribed in the high-level model. However, a containment violation could break the system’s quality. By assuming that the low-level behaviour model can be represented in terms of a formal description, we could achieve containment checking by verifying that the desired specifications encoded in the corresponding high-level model are satisfied by this formal description.

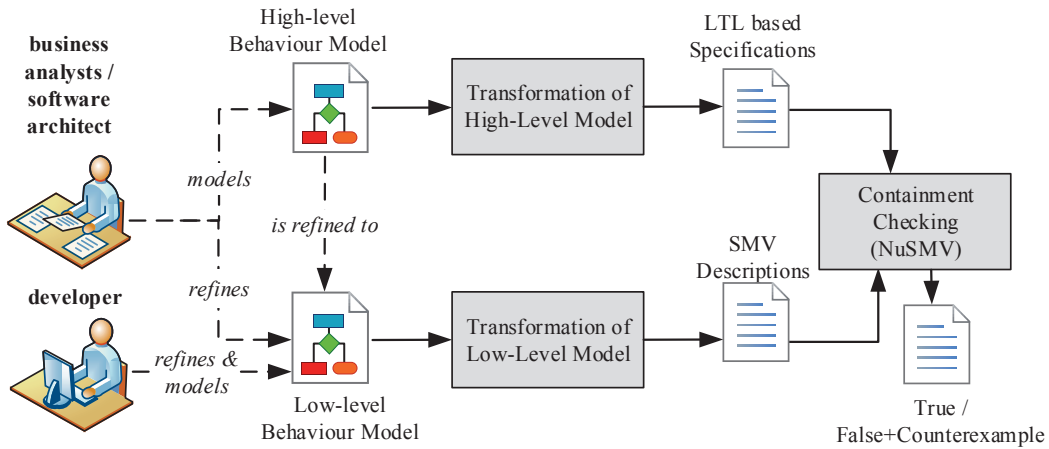


FIGURE 3.1: Overview of the Containment Checking Approach

An overview of our approach is shown in Figure 3.1. The main focus of our approach is represented by the solid lines, while the relevant modelling and developing activities of the involved stakeholders are highlighted by the dashed lines. The proposed containment checking approach consists of three automated steps. First, the high-level behaviour model is transformed into formal consistency

constraints (i.e., LTL formulas in this work). Then, the low-level behaviour model is mapped into formal SMV descriptions. Finally, containment checking is performed on the generated constraints and descriptions using the NuSMV model checker [Cla+96]. If the generated formal description of the low-level model does not satisfy certain constraints generated from the high-level model, then counterexamples are produced by the NuSMV model checker. In the subsequent sections, we explain these steps in detail. The behaviour models considered in this chapter are UML activity diagrams [Gro11b]. Please note that the opposite direction is not essential in containment checking because the low-level behaviour models are often constructed by refining and enriching the high-level model.

3.2.1 Step 1: Generating Consistency Constraints from the High-Level Model

The first step involves an automated transformation of high-level activity diagrams into formal specifications. The main idea is to represent the diagram's elements and their relationships in an appropriate formalism such that the execution order of the activities will become the consistency constraints for the corresponding low-level model in order to incorporate containment of behaviour of activity diagrams. That is, given a certain execution path derived from the input high-level activity diagram, we need to describe the temporal relationship of the involving elements (e.g., actions).

According to OMG UML 2 specification [Gro11b], an activity diagram contains different constructs for expressing the behaviour of software systems. One of the biggest challenges is that the semantics of UML 2 activity diagrams are informal and ambiguous, although it is based on token semantics alike to those of Petri nets [Mur89], where the execution of one node affects the execution of another through directed connections called flows. Thus, our proposed mapping of the constructs of UML 2 activity diagrams to LTL constraints (and formal SMV descriptions presented in the subsequent section) can also be seen as one of few automated approaches in formalisation of UML 2 activity diagrams for supporting model checking. Note that the containment of activity diagrams defines rather loose temporal relationship. The new action(s) in the low-level model, for example, can be inserted between two directly succeeding actions (serial insert), in parallel to one another using fork and join (parallel insert), or with additional condition using decision and merge (conditional insert). The creation of LTL formulas for containment or consistency checking is a very knowledge intensive endeavour. Therefore, we perform the automated creation of the formal constraints by defining LTL-based rules for formally representing constructs of activity diagrams based on containment relationship. Then our approach can take an input UML activity diagram and automatically transform it into corresponding LTL formulas using the LTL-based transformation rules.

The mapping of a UML activity diagram into LTL formulas and SMV descriptions (presented in the subsequent section) are achieved using an extended version of the breadth-first search

Algorithm 1 Mapping UML Activity Diagram A into LTL Formulas

```

1: procedure TRANSLATE( $A$ )
2:    $Q \leftarrow \emptyset$  ▷  $Q$  is the queue of non-visited nodes
3:    $V \leftarrow \emptyset$  ▷  $V$  is the queue of visited nodes
4:    $Q \leftarrow Q \cup \text{get\_initial\_nodes}(A)$  ▷ we start with the initial nodes
5:   for all  $n \in Q$  do
6:      $V \leftarrow V \cup \{n\}$ 
7:      $Q \leftarrow Q \setminus \{n\}$ 
8:     generate_ltl_code( $n$ )
9:     ▷ for mapping of SMV descriptions we use generate_smv_code( $n$ )
10:     $N_{outgoings} \leftarrow \text{get\_outgoing\_nodes}(n)$ 
11:    for all  $m \in N_{outgoings}$  do
12:      if ( $m \notin V$ ) then
13:         $Q \leftarrow Q \cup \{m\}$ 
14:      end if
15:    end for
16:  end for
17: end procedure

```

algorithm as shown in Algorithm 1. To facilitate the representation of an activity diagram in LTL, we define a collection of helper functions to access information of an activity diagram, namely, `get_initial_nodes()`, `generate_ltl_code()`, and `get_outgoing_nodes()`. The function `get_initial_nodes(A)` returns a set of **Initial Nodes** of the input UML activity diagram. An initial node indicates the starting execution point of an activity diagram, and therefore, has no incoming edges. Given a certain node n , its outgoing nodes can be achieved by using the function `get_outgoing_nodes(n)`. A node m is called “outgoing node” of n if there is a control flow going from n to m . Thus, a set of outgoing nodes of n can be achieved by following all of its outgoing edges.

The `generate_ltl_code(n)` function is responsible for generating LTL formulas for each construct of a UML activity diagram. We illustrate the skeleton of the function `generate_ltl_code(n)` in Algorithm 2. The pair of triple apostrophes (‘‘‘) denotes the string templates used for generating code in the our implementation based on Eclipse Xtend framework¹. A pair of guillemots (« and ») is used to denote the parameterized placeholders that will be bound to and substituted with the actual values extracted from the input model elements by the Xtend engine. The `generate_ltl_code(n)` function is not realised as a single function but rather a polymorphism of multiple functions. That is, depending on the type of the input node n , a particular function for generating LTL formulas for that node type will be invoked. The LTL-based transformation rules for initial nodes and sequence of actions, and join nodes are presented in Algorithm 2. In particular, LTL formula for join node requires visited predecessors (incoming flows) that are joined using the logical *AND* operator (“&”) and offered to a join node. The join node cannot execute until

¹See <https://eclipse.org/xtend>

all incoming flows have been received. Table 3.1 summarises the constructs of UML activity diagrams along with their informal descriptions extracted from the UML 2 specification [Gro11b] and LTL-based transformation rules that constitutes the individual function `generate_ltl_code(n)`.

Algorithm 2 Generating LTL Formulas for a Modelling Construct n of a UML Activity Diagram

```

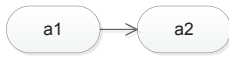
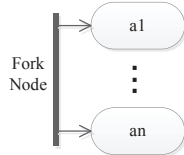
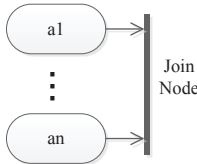
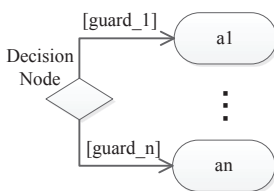
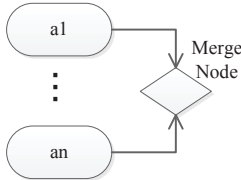
1: procedure GENERATE_LTL_FORMULAS( $n$ );
2:   extracts node information;
3:   binds input values and generates ltl formulas using the following templates:
4:   for all  $n \in \text{initial\_node} \mid n \in \text{action}$  do
5:     if  $m \in N_{\text{outgoins}}$  then
6:       ' ' '
7:       LTLSPEC  $G(\langle n \rangle \rightarrow F \langle m \rangle)$ 
8:       ' ' '
9:     end if
10:  end for
11:  for all  $n \in \text{JoinNode} \wedge i \in N_{\text{incomings}}$  do
12:    if  $(i \geq 0) \wedge V \leftarrow V \cup \{i\}$  then
13:      ' ' '
14:      LTLSPEC  $G((\langle i \rangle \ \& \ \langle i \rangle) \rightarrow F \langle n \rangle)$ 
15:      ' ' '
16:    end if
17:  end for
18: end procedure

```




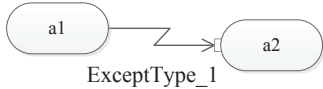
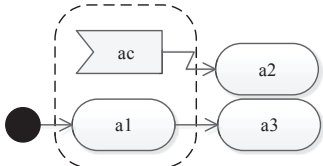
The transformation rules presented in [MTZ14] were not yet sufficient to cover complex containment relations for composite control flows. In particular, transformation rules without the explicit representation of the control nodes increase the complexity and size of LTL formulas. For instance, if a decision node occurs in between two fork nodes then LTL formulas will be of a rather complex and long form, like the following example: “ $G(a1 \rightarrow F(a2 \wedge a2 \wedge (b1 \text{ xor } (b2 \wedge b3))))$ ”. In particular, this formula does not clarify the relationships among the elements of the activity diagram. In this work, we extend and refine these transformation rules by introducing control nodes which decrease the complexity and length of the LTL formulas and provide better understandability of the relationships among the elements of the model. For instance, a **Decision Node** has two outgoing branches, we use the operator “**xor**” to describe the outgoing branches of a **Decision Node**. However, this strategy cannot be effectively generalised for **Decision Nodes** that have more than two outgoing control flows because the operator “**xor**” with n operands ($n \geq 3$) is an odd function which yields **true** not only when one of its operands is **true** but also when the odd numbers of the operands are **true** [PH08]. Therefore, a **Decision Node** can be implemented using “**xor**” operator or its equivalent but more complex form “ $(a \wedge \neg b) \vee (\neg a \wedge b)$ ”.

Similar to the **Decision Node**, the “**xor**” operator is used to describe the incoming guard condition of a **Merge Node**, but we implement it to its equivalent but more complex form. The LTL formula for sequential order of actions is formalised as “ $(a1 \rightarrow F a2)$ ” which describes that each time $a1$ is executed it is eventually followed by the execution of $a2$. The semantics of sequential order

defines rather loose temporal relationship; particularly, in the low-level model new action(s) can be inserted between two directly succeeding actions. If an action is enabled immediately after the previous element terminates, the **X** operator is used, for instance, activity parameter nodes lead to immediate execution of connected nodes. Please note that most of the formulas for different constructs are surrounded by the **G** operator to express the meaning of all possible execution scenarios.

UML Constructs	Modeling Notation	LTL-Based Transformation Rules
Sequencing of Actions: A set of actions (transitively) executed in sequential order. For instance, the execution of <i>a1</i> will trigger the execution of <i>a2</i> .		$G (a1 \rightarrow F a2)$
Fork Node: The execution of a Fork Node leads to the parallel execution of subsequent actions (<i>a1...an</i>) [Gro11b, p. 387].		$G (ForkNode \rightarrow F (a1 \& \dots \& an)) \& G ((a1 \& \dots \& an) \rightarrow O ForkNode)$
Join Node: The concurrent execution of multiple actions (<i>a1...an</i>) leads to the execution of a Join Node [Gro11b, p. 393]. Specifically, a Join Node is used to synchronize incoming concurrent flows. The semantics states that all actions have to be completed before the execution of a Join Node.		$G ((a1 \& \dots \& an) \rightarrow F JoinNode)$
Decision Node: The semantic represents the case in which the execution of a Decision Node is spawn in two or more branches, which branch is actually traversed depends on the evaluation of the guards on the outgoing edges [Gro11b, p. 370].		$G (DecisionNode \rightarrow F (a1 \text{ xor } \dots \text{ xor } an))$ <p>or equivalently, but more complex</p> $G (DecisionNode \rightarrow F ((a1 \& ! \dots \& ! an) \mid (! a1 \& ! \dots \& an)))$
Merge Node: The execution of exclusively one action among a set of alternative actions will lead to the execution of a Merge Node [Gro11b, p. 398].		$G (a1 \text{ xor } \dots \text{ xor } an \rightarrow F MergeNode)$ <p>or equivalently, but more complex</p> $G (((a1 \& ! \dots \& ! an) \mid \dots \mid (! a1 \& ! \dots \& an)) \rightarrow F MergeNode)$

(continued on next page)

UML Constructs	Modeling Notation	LTl-Based Rules	Transformation
<p>Send Signal Action: The Send Signal Action is enabled after the occurrence of the action from which it takes inputs and sends the signal to the target object. [Gro11b, p. 421].</p>		$G (a1 \rightarrow X a2)$	
<p>Accept Event Action: 1) Accept Event Action with no input causes an invocation of next action. The Accept Event Action is enabled upon entry to the activity containing it [Gro11b, p. 317].</p> <p>2) Accept Event Action with incoming flows waits to receive an input and enables only after the signal is sent by the prior action [Gro11b, p. 317]. Afterwards, Accept Event Action leads to the execution of next action.</p>	 <p>1) Accept Event Action with no input</p>  <p>2) Accept Event Action with incoming flows</p>	<p>1) $G (a1 \rightarrow X a2) \ \& \ G !(a1 \ \& \ a2)$</p> <p>2) $G (a1 \rightarrow G (a2 \rightarrow X a3))$</p>	
<p>Exception Handler: Exception Handler leads to execution of the handler body in case the specified exception occurs during the execution of the protected node [Gro11b, p. 373]. A variable, namely, <code>ExceptionType_i</code> is specified for handling the exception (<i>i</i> is an incrementally generated number).</p>		$G ((a1 \ \& \ \text{ExceptionType_i} = \text{ExceptType_1}) \rightarrow X a2)$	
<p>Interruptible Activity Region: When a token leaves an interruptible region via edges designated by the region as interrupting edges, all tokens and behaviours in the region are terminated [Gro11b, p. 391]. If the condition “isInterrupted” evaluates to true then action connected through <code>interrupting_edge</code> is executed; otherwise, remaining activities will be executed.</p>		<p>1) $G (ac \ \& \ \text{isInterrupted} \rightarrow X \text{interrupting_edge}) \ \& \ G !(ac \ \& \ \text{interrupting_edge})$</p> <p>2) $G (\text{interrupting_edge} \rightarrow X a2)$</p> <p>3) $G (\text{InitialNode} \ \& \ !(isInterrupted) \rightarrow F a1)$</p>	

(continued on next page)

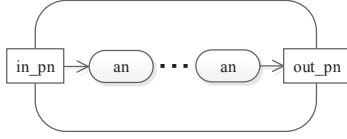
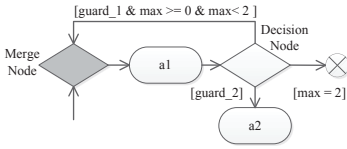
UML Constructs	Modeling Notation	LTL-Based Rules	Transformation Rules
<p>Activity Parameter Node: The execution of input Activity Parameter is enabled when the activity is invoked, to provide input values to the connected nodes through outgoing edges. During the execution of the activity an output Activity Parameter Node accepts all tokens offered to it. We abstracted activity parameter nodes into boolean.</p>		$1) (G (in_pn \rightarrow X a1) \& G (a1 \rightarrow Y in_pn))$ $2) (G (an \rightarrow X out_pn) \& G (out_pn \rightarrow Y an))$	
<p>Loop: The execution of one or more elements is repeated a number of times until a specified condition is reached. A loop can be considered equivalent to a cyclic execution flow including a Decision Node and a Merge Node.</p>		$(DecisionNode \rightarrow F a2) \mid$ $(DecisionNode \& (max \geq 0 \& max < 2) \rightarrow F MergeNode) \mid ((DecisionNode \& max = 2) \rightarrow F FlowFinal) \& ! F(MergeNode) \mid F(a1)$	

TABLE 3.1: Transformation Rules for Generating LTL Formulas for UML Activity Diagrams

Beyond the basic constructs of activity diagrams, we consider complex structures such as exception handlers, interruptible activity region, accept event actions, send signal actions, activity parameter nodes and loops in this chapter. In our implementation, we consider a loop equivalent to a cyclic execution flow including a decision node and a merge node. In particular, the decision node (termination condition) decides whether to continue the repetition process or terminate the process. In the UML 2 specification [Gro11b, p.396], a **Loop Node** represents a loop with setup, body and test sections. The test section may precede or follow the body. The setup section executes only once, when first entering the loop whilst the test and body sections execute each time through the loop until the test section evaluates to false. The test section is similar to the decision condition whereas the setup section is similar to the incoming (e.g., action) of the merge node (that executed once before entering the loop). The condition (i.e., maximum number of iterations) can be applied with a guard, for instance the edge with condition $(max \geq 0 \& max < 2)$ leads to the merge node shown in Table 3.1.

We note that an LTL formula can be as simple as $G(a1 \rightarrow F a2)$ in case of representing the temporal relationship between two actions. Nevertheless, an LTL formula can also be quite complex, like $G((JoinNode \wedge \neg a1 \wedge \neg a2) \vee (\neg JoinNode \wedge a1 \wedge \neg a2) \vee (\neg JoinNode \wedge \neg a1 \wedge a2) \rightarrow F MergeNode)$ for describing composite structures of complex models that embraces two or more control structures or actions. Our approach supports the automated generation of formal constraints for the combination of control structures.

3.2.2 Step 2: Mapping a UML Activity Diagram into SMV Descriptions

In this step, a low-level UML activity diagram is automatically transformed into formal descriptions accepted by the NuSMV model checker. The language that underpins these formal descriptions is hereafter called SMV language. The generated descriptions will be used as input for the NuSMV model checker to verify against the LTL-based constraints generated from the high-level counterparts (as explained in the previous step). On the one hand, the translation of UML activity diagrams into SMV descriptions should comply with the informal semantics of UML activity diagrams as described in UML 2 specification [Gro11b]. On the other hand, as we discussed two tasks in Section 3.1 the encoding of the low-level activity diagram in terms of SMV description language should enable better interpretation of the model checking results (e.g. counterexamples) and effectively communicate the results to the developers. In summary, the translation of a UML activity diagram to SMV descriptions should provide the infrastructure to facilitate the verification of the containment relationship, and especially, analysing verification results to provide useful feedbacks for aiding the developers in resolving containment inconsistencies.

We achieve the mapping of a UML activity diagram into SMV descriptions using an extended version of the breadth-first search. We have developed another algorithm similar to Algorithm 1 for mapping of a UML activity diagram into SMV descriptions. In particular, we developed three helper functions, namely, `get_initial_nodes()`, `get_outgoing_nodes()`, and `generate_smv_code()`. First two functions are mentioned in Algorithm 1, whereas the most important function `generate_smv_code(n)`, responsible for generating SMV descriptions for each construct of a UML activity diagram is depicted in Algorithm 3.

The UML activity diagram is translated into one main module. In our approach, each node of the UML activity diagram will be represented by a boolean state variable in the section **VAR** and its corresponding state transitions will be defined in the section **ASSIGN** by a combination of two functions provided by NuSMV, that are `init()`—for assigning the initial state of a variable— and `next()`—for describing the transition to the next state. The function `next()` is often combined with the branching structure “**case/esac**” for selecting one of many possible choices. Normally a state variable will be initialized with a **false** value (except **Initial Nodes** discussed in the next subsection). It can move to a different state (e.g. **true**) if the incoming conditions are satisfied (see Line 11 in Algorithm 3). The incoming guard conditions can comprise a guard expression and/or the finishing of preceding nodes. A state variable evaluating to **true** implies that the corresponding node of the UML activity diagram is activating. After finishing its execution, the node’s state will be switched back to **false** (see Line 12 in Algorithm 3).

We note that `generate_smv_code(n)` is not realised as a single function but rather a polymorphism of multiple functions. That is, depending on the type of the input node n , a particular function for generating SMV descriptions for that node type will be invoked. This can be achieved in traditional

Algorithm 3 Generating SMV Descriptions for a Modelling Construct n of a UML Activity Diagram

```

1: procedure GENERATE_SMV_CODE( $n$ );
2:   extracts node information;
3:   binds input values and generates SMV descriptions using the following templates:
4:   ' '
5:   ' '
6:   VAR
7:     « $n$ » : boolean;                                ▷ State variable declaration
8:   ASSIGN
9:     init(« $n$ ») := «node-initial-state»              ▷ Definitions of state transitions
10:    next(« $n$ ») := case
11:      «incoming-guard-condition( $s$ )» : TRUE;
12:      « $n$ » : FALSE;
13:    esac;
14:   ' '
15: end procedure

```

programming languages by using a typical “if/then/else” or “switch/case” construct. In our prototypical implementation, we leverage the powerful polymorphic method invocation technique provided by Xtend², which is used to realise the transformation of UML activity diagram to SMV descriptions. Using this technique, we devise multiple functions for generating SMV descriptions with respect to the input node types. The functions have the same name but require different input types. According to the particular type of the input node at execution time, Xtend will dispatch the execution to the corresponding function.

In the subsequent sections, we will present and discuss in detail the rules for generating SMV descriptions for each node type that constitutes the individual function `generate_smv_code(n)`.

Dealing with Data The UML activity diagram can contain variables such as integer, real, or string [Gro11b]. Thus, directly mapping variables of these types to NuSMV increases a finite state space which might lead to the state space infinite [EW02]. Nevertheless, we note that the constraints that are associated with nodes or edges can affect the behaviour of a UML activity diagram. The range of constraints in a UML activity diagram, regardless of their domains, is $\mathbb{B} = \{\mathbf{true}, \mathbf{false}\}$. Therefore, we need to explore execution paths corresponding to both truth values yielded by each constraint, which can be done automatically by the NuSMV model checker.

An efficient encoding strategy would therefore be to abstract all data and to introduce for each constraint expression a boolean representative variable [EW02]. A constraint evaluates to **true** (resp. **false**) iff its boolean representative is **true** (resp. **false**). This encoding decision can help reducing significantly the state space under consideration. In this work, we opt to abstract and encode each constraint respectively in SMV as a boolean variable, for example, as shown in the

²See https://eclipse.org/xtend/documentation/202_xtend_classes_members.html#polymorphic-dispatch

LTL formula corresponding to a loop in Table 3.1. However, in cases when the different types of variables may have dependencies among constraints, temporary variables of enumerated types can be introduced to handle them.

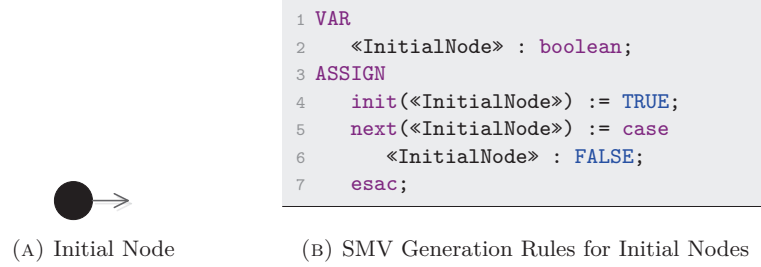


FIGURE 3.2: Translation of Initial Nodes into SMV Descriptions

Initial Node The mapping of a UML activity diagram to SMV starts with the **Initial Nodes** and follow their outgoing nodes using a breadth-first search. An **Initial Node** is special node that denotes a starting point of a UML activity diagram and does not have any incoming edges. Thus, each **Initial Node** is represented by a boolean state variable whose initial state would be assigned as `true` (see Figure 3.2).

Action, Fork Node, Join Node, and Final Node In this section, we consider a set of nodes including **Action**, **Fork Node**, **Join Node**, and **Final Node** that can be encoded rather similarly in SMV because they will be triggered with respect to their incoming control flows. Please note that UML allows for multiple incoming edges of the nodes. In case a node has multiple incoming edges, the semantics of triggering the node’s execution is *implicit join* [Gro11b]. Therefore, we use the logical *AND* operator (“&”) to represent the implicit “*and-join*” guard for all tokens going through the incoming control flows. Figure 3.3 describes the translation of **Action**, **Fork Node**, **Join Node**, and **Final Node** into SMV descriptions based on the templates shown in Algorithm 3. The **ASSIGN** section defines the transition relation of nodes. The node is initially set to false. However, if the incoming condition(s) are satisfied, it is changed to a true state (see Line 11 in Algorithm 3. The node’s state shall be switched back to false after the execution.

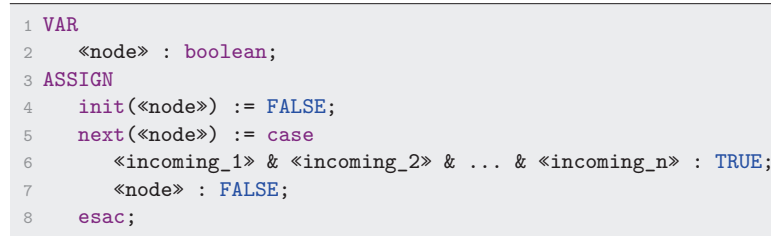


FIGURE 3.3: Generic Rules for Mapping UML Constructs to SMV Descriptions

According to the UML 2 specification [Gro11b] the semantics of **Send Signal Action** and **Accept Event Action** inherit from an **Action**. Thus, we can implement the state transitions of **Send Signal Action** and **Accept Event Action** similar to that of an **Action** node. Note that an **Accept Event Action** can be enabled with or without incoming flows. If an **Accept Event Action** has no incoming flows, it is always enabled to accept events. Moreover, it does not stop after accepting an event and providing output, but continues to wait for other events. This indicates an exception to the normal execution rules in activity diagrams. Thus, in our approach, we consider both an **Accept Event Action** that is enabled using an incoming control flow and the case of **Accept Event Action** with no incoming edge. We introduce a boolean variable, `isEventOccur` to capture semantics of **Accept Event Action** with no incoming edge. The variable `isEventOccur` initialises to `false` and is set to `true` when an event arrives.

Merge Node A **Merge Node** brings together multiple alternative control flows and exclusively accepts one among them [Gro11b, p. 398]. In case a **Merge Node** has two incoming control flows, a straightforward naive encoding strategy is to use the logical exclusive OR operator “ $a1 \text{ xor } a2$ ” (or its equivalent but longer form “ $(a1 \wedge \neg a2) \vee (\neg a1 \wedge a2)$ ”) to describe the incoming guard condition of a **Merge Node**. However, this strategy cannot be effectively generalised for **Merge Nodes** that have more than two incoming control flows because the operator “**xor**” with n operands ($n \geq 3$) yields `true` not only when one of its operands is `true` but also when the odd numbers of the operands are `true` [PH08]. This semantics does not precisely reflect the (semi-)formal description of **Merge Nodes** presented in the UML 2 specification [Gro11b, p. 399]. Moreover, in case of some k of the incoming nodes $a1...an$ ($k \leq n$) are simultaneously activated, the UML 2 specification states that b should be activated k times respectively [Gro11b, p. 400]. As the UML 2 specification does not define clearly the execution order of multiple instances of b in this particular case, we can assume it follows an interleaving execution semantics.

In this chapter, we devise a novel encoding of **Merge Nodes** in SMV that satisfies the “*exclusive choice of multiple alternate flows*” semantics described in the UML 2 specification as shown in Figure 3.4b. For each **Merge Node**, we introduce a temporary variable, namely, `merge_flag_i`, where i represents an incrementally generated number to avoid name conflicts. This temporary variable has an enumerated type that comprises “**undetermined**”—to denote its normal state—and “**in_ax**” where $x = 1, \dots, n$ —to represent the state values that correspond to the incoming control flows from $a1$ to an , respectively. The variable `merge_flag_i` will be used to handle the case when some of the incoming nodes $a1...an$ are simultaneously activated, i.e., some k , where $1 \leq k \leq n$, of the corresponding state variables yield `true` at the same time. In this case, `merge_flag_i` will choose non-deterministically and exclusively one among these activated nodes as shown in Line 12. We note that the non-deterministic assignment (Line 12) is a powerful means provided by the NuSMV model checker for exhaustively exploring multiple possible execution paths yielded by the values of an enumerated state. That is, in order to verify in case some k incoming nodes

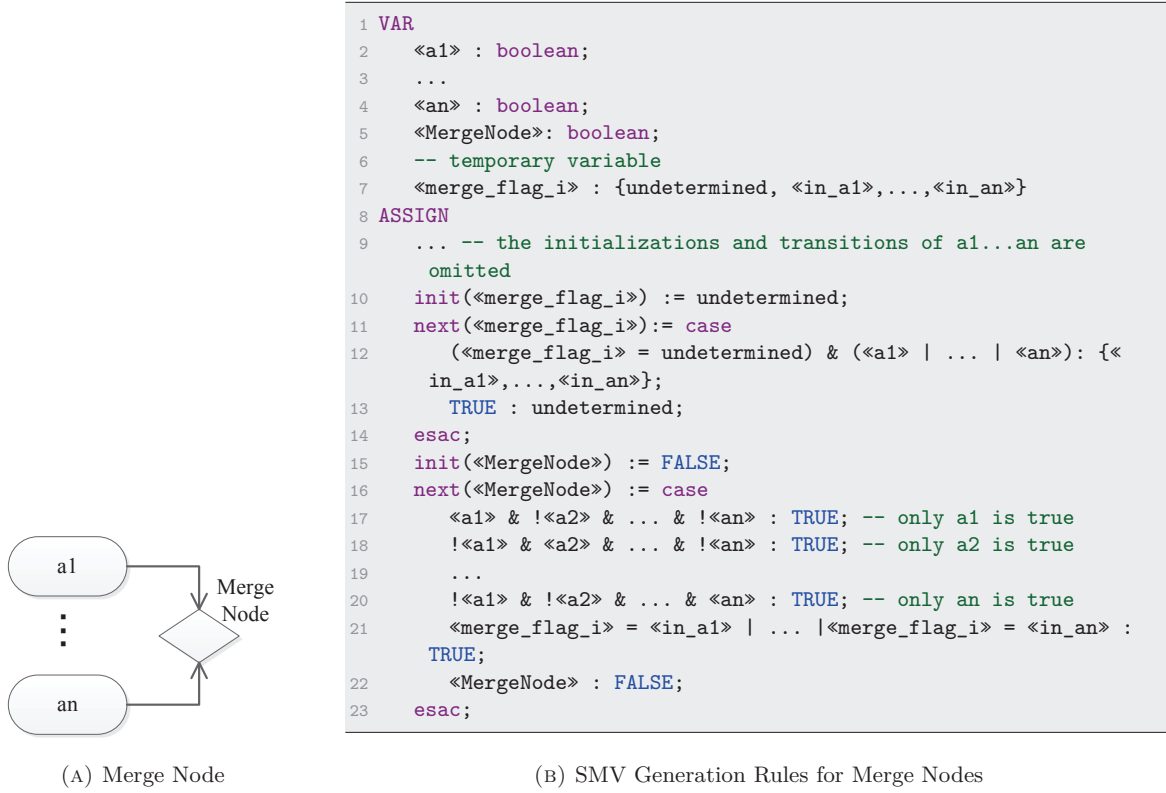
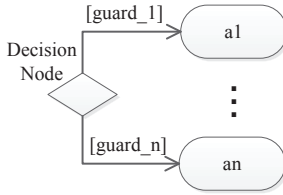


FIGURE 3.4: Translation of Merge Nodes into SMV Descriptions

are activated, NuSMV will bind `merge_flag_i` to a certain value “`in_ax`” in the first place, to “undetermined” in the next transition, then to another value “`in_ay`” in the subsequent transition, and so forth. In combination with the branching construct “`case/esac`” (Line 16–23), we can see that the Merge Node is activated if and only if either one of the incoming nodes is `true` or the variable `merge_flag_i` is assigned to a state value “`in_ax`”, where $x = 1, \dots, n$.

Decision Node A Decision Node is a special case in which its execution will trigger one of the outgoing control flows according to the corresponding guard conditions. In theory, more than one outgoing nodes can be activated following a Decision Node if their guard conditions evaluate to `true`. However, the UML 2 specification states that the execution traversing through a Decision Node will be passed to only one outgoing node but does not dictate the order of evaluation and execution in case multiple guard constraint hold simultaneously [Gro11b, p. 371]. In reality, it is often assumed that the developers are responsible for the exclusiveness of the guard conditions of the outgoing control flows. We opt for this assumption and assume that only one of the guards of the outgoing control flows evaluates to `true` at a time.

Figure 3.5 illustrates the rules for mapping a Decision Node into SMV descriptions. Similar to the case of a Merge Node, we introduce a temporary variable, namely, `post_decision_i` (i is an incrementally generated number) for exclusively choosing one of many alternative outgoing



(A) Decision Node

```

1  VAR
2    «DecisionNode»: boolean;
3    «a1»: boolean;
4    ...
5    «an»: boolean;
6    -- temporary variable
7    «post_decision_i»: {undetermined, «guard_1», ..., «guard_n»};
8  ASSIGN
9    -- if this Decision Node is not an initial node, FALSE must
    be used instead.
10   init(«DecisionNode») := TRUE;
11   next(«DecisionNode») := case
12     «DecisionNode»: FALSE;
13   esac;
14   init(«post_decision_i») := undetermined;
15   next(«post_decision_i») := case
16     «DecisionNode» & («post_decision_i = undetermined»): {«
    guard_1», ..., «guard_n»};
17     TRUE: undetermined;
18   esac;
19   -- the first outgoing branch
20   init(«a1») := FALSE;
21   next(«a1») := case
22     «post_decision_i» = «guard_1»: TRUE;
23     «a1»: FALSE;
24   esac;
25   ...
26   -- the n(th) outgoing branch
27   init(«an») := FALSE;
28   next(«an») := case
29     «post_decision_i» = «guard_n»: TRUE;
30     «an»: FALSE;
31   esac;

```

(B) SMV Generation Rules for Decision Nodes

FIGURE 3.5: Translation of Decision Nodes into SMV Descriptions

control flows. The variable `post_decision_i` has an enumerated type including a normal state “undetermined” and the values corresponding to the outgoing control flows (guard conditions) (Line 7). The evaluation of the guard conditions is made using a “case/esac” construct (Line 14–17). The next state of `post_decision_i` will be either `guard_1`, `guard_2` or `guard_n` (Line 22, 29). For the purpose of verification, it is possible to initialise guard/constraint with the boolean values that are evaluated at execution time. Nevertheless, we can leverage the ability of NuSMV to exhaustively inspect all execution paths with respect to two boolean values of each variable to verify the satisfaction between the generated SMV descriptions and LTL constraints. However, in cases when the different types of variables may have dependencies among constraints, temporary variables of enumerated types can be introduced to handle them.

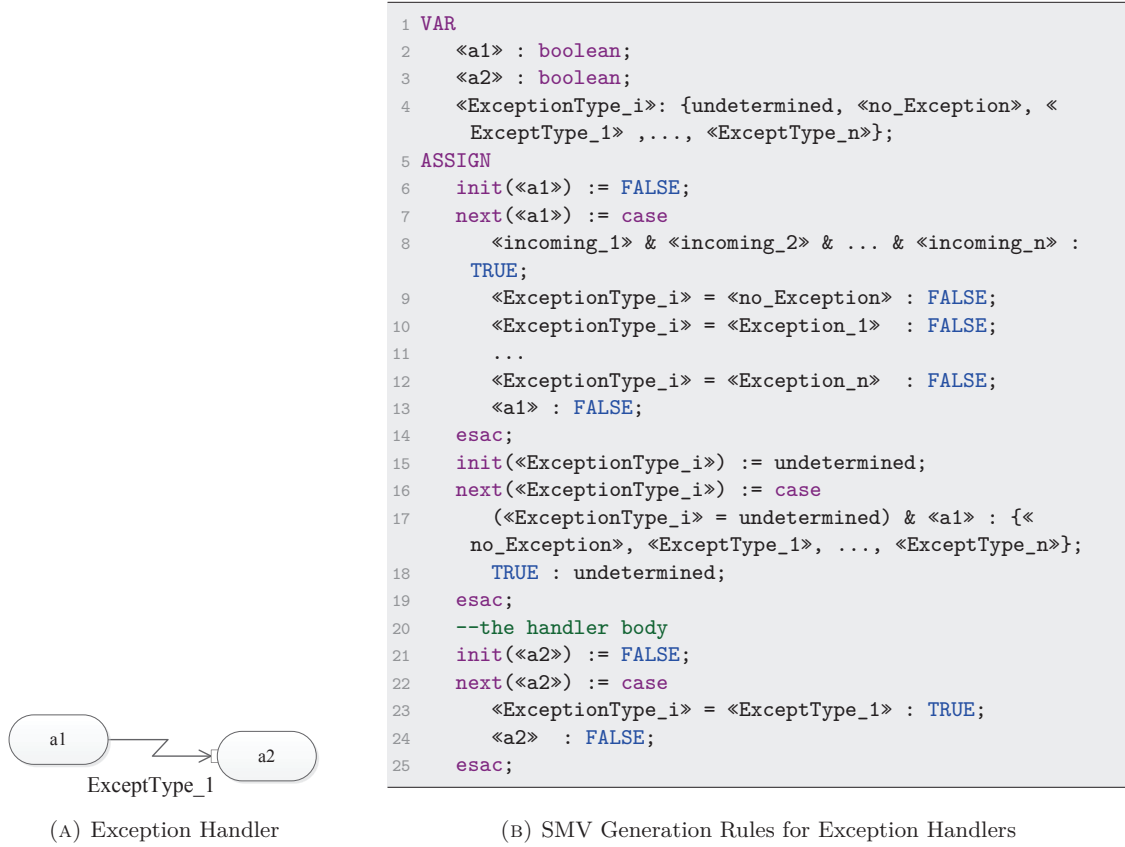


FIGURE 3.6: Translation of Exception Handlers into SMV Descriptions

Exception Handler An Exception Handler describes a body to execute when a particular exception is caught. In particular, if an exception occurs during the execution of the action (protected node), the set of execution handlers on the action is examined for a handler that matches the exception. If a match is found, the handler catches the exception and executes its body. The exception object is placed in the `exceptionInput` node as a token to start execution of the handler body. The execution of the handler body may access the caught exception via the `exceptionInput` node. A handler matches if the type of the exception is the same as, or a descendant of, one of the `exceptionTypes` of the handler [Gro11b, p. 374]. However, theory proposed in the standard does not clearly define how to declare the type of the exception or its parameters. An **Exception Handler** contains a **Protected Node** (where the exception is raised), a **Handler Body** (where it is handled), the **Exception Input** (ObjectNode), and the **Exception Type** (Classifier) of the exception. An exception in the protected node could be raised either by triggering external event or as a consequence of a branch trigger. A **Handler Body** is not enabled to execute in any case other than in response to an exception being caught by its handler.

Figure 3.6 illustrates the rules for mapping an Exception Handler into SMV descriptions. For handling the exception, we introduce a temporal variable, namely, `ExceptionType_i` (i is an incrementally generated number) for more than one handlers connected to the protected node.

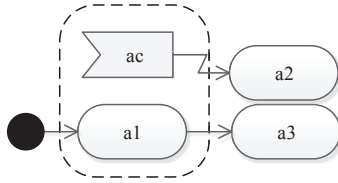
ExceptionType_i has an enumerated type including an initial state “undetermined” and types of exception that a handler have, for instance, **ExceptType_1** and so on (Line 4). The absence of an exception is represented **no_Exception** by assigning the value to **ExceptionType_i**. The next state of **ExceptionType_i** will be either **no_Exception**, **ExceptType_1** or **ExceptType_n**. The execution of the **Handler Body (a2)** starts when **ExceptType_1** matches with a exception raised by a protected node (Line 23).

Interruptible Activity Region An **Interruptible Activity Region** is a group of nodes, where all tokens and behaviours in the region are terminated, if an edge (designated by the region as interrupting edge) traverses an interruptible activity, before leaving the region [Gro11b, p. 391]. During the execution of an **Interruptible Activity Region**, the reception of an event triggers the block abort of that part of the **Activity** and resumes execution with another action (target node) outside the interruptible region. However, other actions outside the interruptible region can not be executed before the handling of particular event.

Figure 3.7 illustrates the rules for mapping an **Interruptible Activity Region** into SMV descriptions. For handling the interrupting event, we introduce a boolean variable, namely, **isInterrupted** that evaluates to **true** if the event occurs; otherwise, it evaluates to **false**. More specifically, the execution of an *interrupting_edge* starts when **Accept Event Action (ac)** receives an event (Line 8–17) which leads to the execution of action *a2* outside the region (Line 20–23). If **isInterrupted** is **false** (i.e., negation of **isInterrupted** is **True**), then all the nodes within the region complete their execution (Line 25–33) other than the source and target nodes of the *interrupting_edge*.

Activity Parameter Node The **Activity Parameter Nodes** are the “Object Nodes” that provide the means of an activity to accept inputs and supply outputs. When the activity is invoked the input values are placed as tokens on input activity parameter nodes (with no incoming edges) and are accessible within the activity via the outgoing edges of those nodes. After completing the execution of an activity, the output values held by output activity parameter nodes (with no outgoing edges) are given to the corresponding parameters of the activity [Gro11b, p. 346].

Activity Parameter Nodes are abstracted as boolean variables (Line 2–3). As mentioned in section 3.2.2 directly mapping of a node that contains data leads to the state explosion. Therefore, we introduce temporary variables namely, **in_par** and **out_par** for holding the actual parameter values (Line 5–6). These temporary variables have an enumerated type that comprises “undetermined”—to denote its normal state—and “**in_val x** ” and “**out_val x** ” where $x = 1, \dots, n$ —to represent the input values and output values that correspond to the input activity parameter node and output activity parameter node, respectively. The transformation rule for **Activity Parameter Nodes**



(A) Interruptible Activity Region

```

1 VAR
2   «a1» : boolean;
3   ...
4   «ac» : boolean;
5   «isInterrupted» : boolean;
6   «interrupting_edge» : boolean;
7 ASSIGN
8   init(«isInterrupted») := {TRUE, FALSE};
9   init(«ac») := FALSE;
10  next(«ac») := case
11    «isInterrupted» : TRUE;
12    «ac» = : FALSE;
13  esac;
14  init(«interrupting_edge») := FALSE;
15  next(«interrupting_edge») := case
16    «ac» & «isInterrupted» : TRUE;
17    «interrupting_edge» : FALSE;
18  esac;
19  --the target node outside the interruptible region
20  init(«a2») := FALSE;
21  next(«a2») := case
22    «interrupting_edge» : TRUE;
23    «a2» : FALSE;
24  esac;
25  init(«a1») := FALSE;
26  next(«a1») := case
27    «InitialNode» & ! «isInterrupted» : TRUE;
28    «a1» : FALSE;
29  esac;
30  init(«a3») := FALSE;
31  next(«a3») := case
32    «a1» : TRUE;
33    «a3» : FALSE;
34  esac;

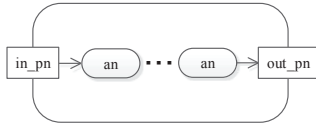
```

(B) SMV Generation Rules for Interruptible Activity Region

FIGURE 3.7: Translation of Interruptible Activity Region into SMV Descriptions

with no incoming edges i.e., `in_pn` (Line 8–17) and Activity Parameter Nodes with no outgoing edges i.e., `out_pn` (Line 20–29) are shown in Figure 3.8.

Dealing with Loops Describing loops in terms of state-based formal descriptions like SMV is a very challenging task. Generally, a loop allows repeated execution of one or more actions until a specified condition is not met. Loops may produce fixed or variable cyclic execution flows. In the latter case, a loop might execute indefinitely and hence cause a state space explosion for model checking. However, it is unrealistic for most loops that they really execute indefinitely. The model checking techniques are not able to prove the correctness of the model, unless an upper limit is known, that unfolds all loops to their maximum iteration. In order to prevent an indefinite loop and ensure that loop will eventually stop, we created a way to terminate the loop. In particular, we consider a loop equivalent to a cyclic execution flow including a Decision Node and a Merge Node, as illustrated in Figure 3.9 as inspired by [EW02; GM05]. Eshuis and Wieringa use strong



(A) Activity Parameter Node

```

1  VAR
2    «in_pn» : boolean;
3    «out_pn»: boolean;
4    -- temporary variables
5    «in_par» : {undetermined, «in_val1»,..., «in_valn»};
6    «out_par»: {undetermined, «out_val1»,..., «out_valn»};
7  ASSIGN
8    init(«in_par») := undetermined;
9    next(«in_par») := case
10     («in_par» = undetermined) : {«in_val1»,..., «in_valn»
11     };
12     TRUE : undetermined;
13   esac;
14   -- the input parameter node
15   init(«in_pn») := FALSE;
16   next(«in_pn») := case
17     «in_par» = «in_val1» & ... & «in_par» = «in_valn» :
18     TRUE;
19     «in_pn» : FALSE;
20   esac;
21   ...
22   init(«out_par») := undetermined;
23   next(«out_par») := case
24     («out_par» = undetermined) : {«out_val1»,..., «
25     out_valn»};
26     TRUE : undetermined;
27   esac;
28   -- the output parameter node
29   init(«out_pn») := FALSE;
30   next(«out_pn») := case
31     «out_par» = «out_val1» & ... & «out_par» = «out_valn»
32     : TRUE;
33     «out_pn» : FALSE;
34   esac;

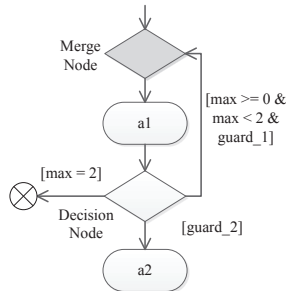
```

(B) SMV Generation Rules for Activity Parameter Nodes

FIGURE 3.8: Translation of Activity Parameter Nodes into SMV Descriptions

fairness (also known as compassion) constraints to exit loops eventually [EW02], whilst Guelfi and Mammar only consider loops with predefined limited iterations [GM05].

The execution of a loop would be repeated only if the maximum number of allowed iterations is not yet reached. To enable model checking, we impose a limit on the number of loops such that a loop will iterate at least once and at most a defined maximum number of iterations. The condition (i.e., maximum number of iterations) might be applied to an additional guard. The conditional edges labelled with $\text{max} \geq 0 \ \& \ \text{max} < 2$ and $\text{max} = 2$ are shown in Figure 3.9a. If the maximum number of iterations is not specified by the user, then the loop will terminate after repeating three times. To deal with loops, we initialise the control variable, namely *max*. Its value is incremented each time the action *a1* is executed (Line 22–25). The transformation rule for loops is shown in Figure 3.9. We apply the similar rules for a decision node (choice of outgoing Line 38–41) and a merge node (Line 10–20) as we presented in Figures 3.5 and 3.4, respectively. If *max* become equals to maximum number, a new iteration cannot start and execution of the loop will terminate.



(A) Loop Structure

```

1  VAR
2    «MergeNode»: boolean;
3    «DecsisonNode»: boolean;
4    «a1»: boolean;
5    ...
6    «max»: «0..2»;  -- control variable
7    «merge_flag_i»: {undetermined, «in_a1», «in_guard_1»};
8    «post_decision_i»: {undetermined, «guard_1», «guard_2», «guard_3»};
9  ASSIGN
10   init(«merge_flag_i») := undetermined;
11   next(«merge_flag_i») := case
12     («merge_flag_i» = undetermined) & («incoming_1» | «DecsisonNode»): {«in_incoming_1», «in_guard_1»};
13     TRUE : undetermined;
14   esac;
15   init(«MergeNode») := FALSE;
16   next(«MergeNode») := case
17     «incoming_1» & !«DecsisonNode» : TRUE;
18     !«incoming_1» & «DecsisonNode» : TRUE;
19     «merge_flag_i» = «in_incoming_1» | «merge_flag_i» = «in_guard_1» : TRUE;
20     «MergeNode» : FALSE;
21   esac;
22   init(«a1») := FALSE;
23   next(«a1») := case
24     «MergeNode» : TRUE;
25     «a1» : FALSE;
26   esac;
27   init(«DecisionNode») := FALSE;
28   next(«DecisionNode») := case
29     «a1» : TRUE;
30     «DecisionNode» : FALSE;
31   esac;
32   init(«max») := (0);
33   next(«max») := case
34     («max» >= 0) & («max» < 2) : «max» + 1;
35     («max» = 2) : «max»;
36     TRUE : «max»;
37   esac;
38   init(«post_decision_i») := undetermined;
39   next(«post_decision_i») := case
40     «DecisionNode» & («post_decision_i» = undetermined) : {guard_1, guard_2, guard_3};
41     TRUE : undetermined;
42   esac;
43   ....
44   init(«FlowFinal») := FALSE;
45   next(«FlowFinal») := case
46     «post_decision_i» = «guard_3» & «max» = 2 : TRUE;
47     «FlowFinal» : FALSE;
48   esac;

```

(B) SMV Generation Rules for Loops

FIGURE 3.9: Translation of Loops into SMV Descriptions

As mentioned in Section 3.2.1, the test and setup sections of a UML 2 **Loop Node** are similar to the decision condition and the incoming (e.g., action) of the merge node. Therefore, aforementioned mapping rules for the loop can be applied on the **Loop Node**.

3.2.3 Step 3: Containment Checking Using NuSMV Model Checker

The main goal of our approach is to assess the containment relationship between a high-level and low-level activity diagram. More specifically, containment checking aims to verify whether the elements and structures of a high-level model, such as actions, control nodes and edges/guards correspond to those of a refined low-level model. Note that the refined and extended low-level model may contain new actions that are inserted between two directly succeeding actions (serial insert), in parallel to another one using fork and join (parallel insert), and/or with in separate path using decision and merge (conditional insert). In addition, the low-level model can have new loops, exception handlers and event actions, and so on, but the elements should not be inserted arbitrarily. Therefore, it is necessary to check the containment consistency between the low-level model and its high-level counterpart to correctly build the software system.

In our approach, containment checking is performed using the NuSMV model checker. NuSMV takes the LTL properties (generated in Step 1) and the SMV descriptions (generated in Step 2), and exhaustively explores violations of a property by traversing the complete state space. In case the SMV descriptions satisfy the LTL properties, it implies that the behaviour described in the high-level model can be embraced by the low-level model's behaviour. Otherwise, the low-level model deviates improperly from the high-level counterpart. In particular, each LTL formula/property represents a part of the high-level model. If a certain property does not satisfied the SMV descriptions, it means that the corresponding part of the high-level model is not contained in the low-level model. In this case, NuSMV will generate a counterexample that consists of the execution traces of the SMV descriptions leading to the violation. The counterexample can help the developers to locate and resolve the containment inconsistencies. Note that the counterexample provides only limited information for understanding the causes of inconsistencies but not how to fix the inconsistencies.

3.3 Scenario from Industrial Case Study

In this section, a representative case, namely, the *Loan Approval*, will be described in detail to illustrate how our approach works to detect and resolve the containment inconsistencies. This scenario is extracted from an e-business application in the banking sector. The banking domain must enforce security and must be in conformity with the regulations in effect. The core functionality of the loan approval system can be described as follows: It starts after receiving a new customer's loan

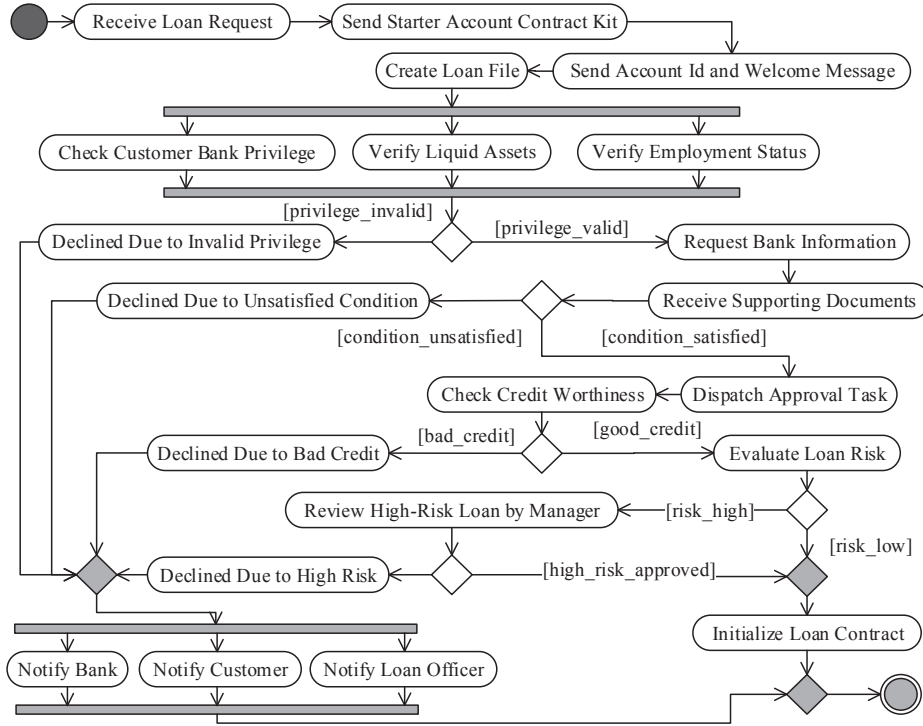


FIGURE 3.10: High-Level Activity Diagram of Loan Approval System

request, and then preliminary inspections are performed to assure that the customer has provided valid information about the credit (e.g., saving or debit account). Afterwards, the customer's credit worthiness is evaluated. Finally, the evaluation of loan risk is carried out. If the loan inquired by the customer is low, the loan contract is initialised otherwise a loan declined. The first two subsections describe the automated translation of high-level loan approval system into LTL formulas and low-level (refined and enhanced) loan approval system into SMV descriptions respectively. The last subsection presents analysis of containment checking results.

3.3.1 Generating LTL Formulas from the High-Level Model

The high-level representation of the loan approval system in terms of a UML activity diagram is shown in Figure 3.10. LTL formulas are automatically generated from the high-level activity diagram of the loan approval system using our LTL-based transformation rules presented in Table 3.1. For instance, the LTL formula “**LTLSPEC** (DecisionNode -> **F** ((DeclinedDueToInvalidPrivilege & ! RequestBankInformation) | (! DeclinedDueToInvalidPrivilege & RequestBankInformation)))” is generated for the Decision Node (as discussed in Section 3.2.1). The high-level loan approval activity diagram contains 78 elements including 14 control nodes and 21 actions. For these elements, 27 LTL formulas are generated. Each generated LTL formula is used as input for containment checking. Table 3.2 shows the generated LTL formulas from the high-level loan approval system.

UML Constructs	Generated LTL Formulas
Sequence	<pre> LTLSPEC G (InitialNode -> F ReceiveLoanRequest); LTLSPEC G (ReceiveLoanRequest -> F SendStarterAccountContractKit); LTLSPEC G (SendStarterAccountContractKit -> F CreateLoanFile); LTLSPEC G (CreateLoanFile -> F SendAccountIdandWelcomeMessage); LTLSPEC G (SendAccountIdandWelcomeMessage -> F ForkNode); LTLSPEC G (JoinNode -> F DecisionNode); LTLSPEC G (RequestBankInformation -> F ReceiveSupportingDocuments); LTLSPEC G (ReceiveSupportingDocuments -> F DecisionNode1); LTLSPEC G (DispatchApprovalTask -> F CheckCreditWorthiness); LTLSPEC G (CheckCreditWorthiness -> F DecisionNode2); LTLSPEC G (EvaluateLoanRisk -> F DecisionNode3); LTLSPEC G (ReviewHigh_RiskLoanbyManager -> F DecisionNode4); LTLSPEC G (MergeNode -> F InitializeLoanContract); LTLSPEC G (MergeNode1 -> F ForkNode2); LTLSPEC G (MergeNode2 -> F ActivityFinalNode); </pre>
Fork Node	<pre> LTLSPEC G (ForkNode -> F (CheckCustomerBankPrivilege & VerifyLiquidAssets & VerifyEmploymentStatus)) & G ((CheckCustomerBankPrivilege & VerifyLiquidAssets & VerifyEmploymentStatus) -> O ForkNode); LTLSPEC G (ForkNode2 -> F (NotifyBank & NotifyCustomer & NotifyLoanOfficer)) & G ((NotifyBank & NotifyCustomer & NotifyLoanOfficer) -> O ForkNode2); </pre>
Join Node	<pre> LTLSPEC G ((CheckCustomerBankPrivilege & VerifyLiquidAssets & VerifyEmploymentStatus) -> F JoinNode); LTLSPEC G ((NotifyBank & NotifyCustomer & NotifyLoanOfficer) -> F JoinNode2); </pre>
Decision Node	<pre> LTLSPEC (DecisionNode -> F ((DeclinedDuetoInvalidPrivilege & ! RequestBankInformation) (! DeclinedDuetoInvalidPrivilege & RequestBankInformation))); LTLSPEC (DecisionNode1 -> F ((DeclinedDuetoUnsatisfiedCondition & ! DispatchApprovalTask) (! DeclinedDuetoUnsatisfiedCondition & DispatchApprovalTask))); LTLSPEC (DecisionNode2 -> F ((DeclinedDuetoBadCredit & ! EvaluateLoanRisk) (! DeclinedDuetoBadCredit & EvaluateLoanRisk))); LTLSPEC (DecisionNode3 -> F ((ReviewHigh_RiskLoanbyManager & ! MergeNode) (! ReviewHigh_RiskLoanbyManager & MergeNode))); LTLSPEC (DecisionNode4 -> F ((DeclinedDuetoHighRisk & ! MergeNode) (! DeclinedDuetoHighRisk & MergeNode))); </pre>
Merge Node	<pre> LTLSPEC (G (DecisionNode3 & ! DecisionNode4) (! DecisionNode3 & DecisionNode4) -> F MergeNode); LTLSPEC (G (DeclinedDuetoInvalidPrivilege & ! DeclinedDuetoUnsatisfiedCondition & ! DeclinedDuetoBadCredit & ! DeclinedDuetoHighRisk) (! DeclinedDuetoInvalidPrivilege & DeclinedDuetoUnsatisfiedCondition & ! DeclinedDuetoBadCredit & ! DeclinedDuetoHighRisk) (! DeclinedDuetoInvalidPrivilege & ! DeclinedDuetoUnsatisfiedCondition & DeclinedDuetoBadCredit & ! DeclinedDuetoHighRisk) (! DeclinedDuetoInvalidPrivilege & ! DeclinedDuetoUnsatisfiedCondition & ! DeclinedDuetoBadCredit & DeclinedDuetoHighRisk) -> F MergeNode1); LTLSPEC (G (JoinNode2 & ! InitializeLoanContract) (! JoinNode2 & InitializeLoanContract) -> F MergeNode2); </pre>

TABLE 3.2: LTL Formulas Generated from the High-Level Loan Approval Activity Diagram

3.3.2 Generating SMV Descriptions from the Low-Level Model

The low-level loan approval model is a refined and extended version of the high-level model that provides more detailed information about the system. For instance, the contract documents are sent to the customer for a final decision. If the customer agrees with loan terms and contract then manager and customer both officially sign the loan contract. Otherwise loan terms and contract are revised. Finally, if no negative reports have been filed, the loan settlement task is performed otherwise the loan approval process will be closed. The low-level model shown in Figure 3.11, is automatically converted into SMV descriptions using our aforementioned transformation rules.

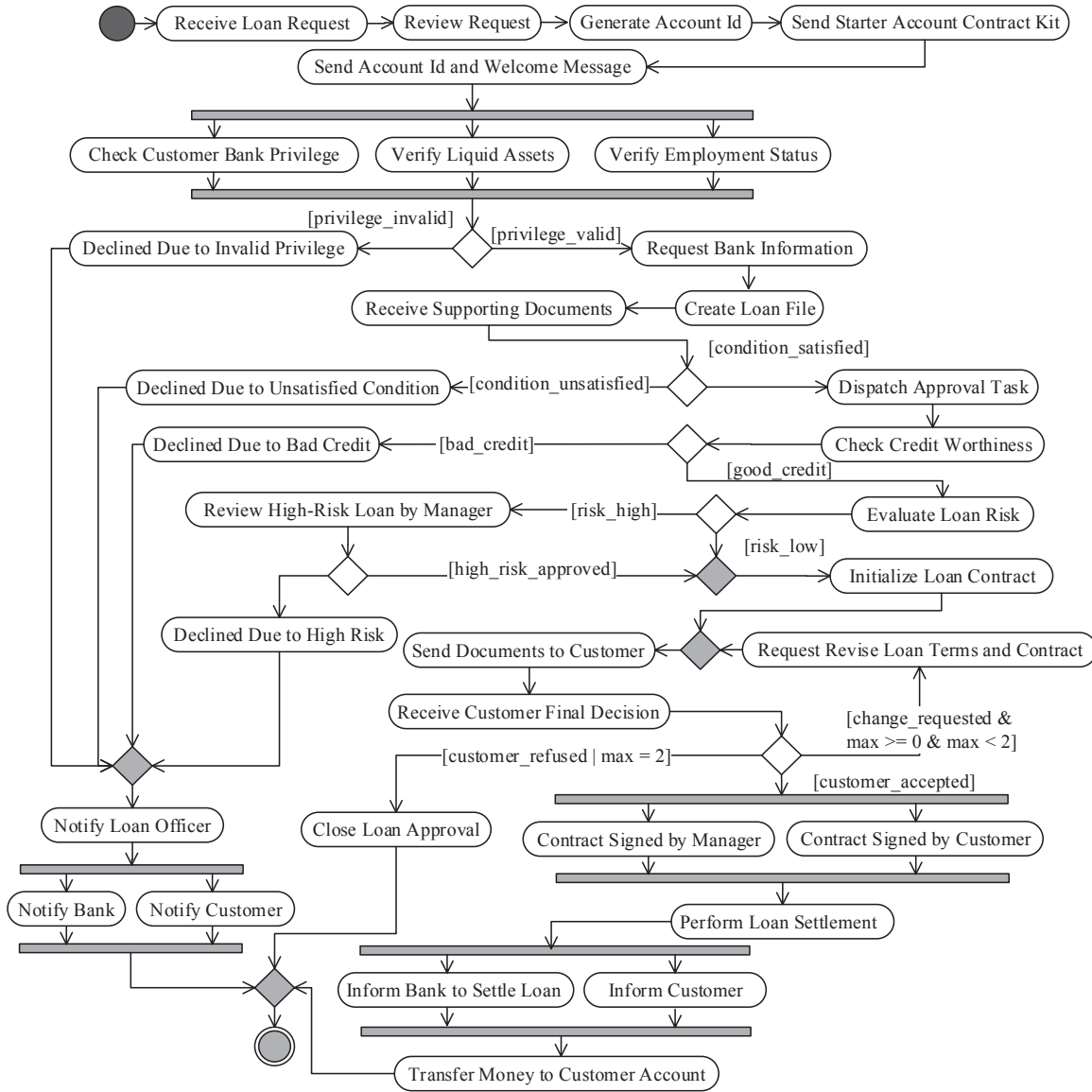


FIGURE 3.11: Low-Level Activity Diagram of Loan Approval System

Listing 3.1 shows an excerpt of the SMV descriptions resulting from the translation of the low-level loan approval system. Some repetitive parts that produced from the similar types of nodes, have been omitted in the listing. Without going into excessive detail, the SMV description can be summarised as follows: The SMV description consists of two parts, the variables declaration part and the variable assignment part. All the elements of the loan approval activity diagram are declared as variables under the “VAR” keyword. The nodes of the loan approval activity diagram are represented by boolean variables except temporary variables for decision and merge nodes to handle outgoing branching and incoming nodes, respectively. These temporary variables are represented as scalar variables (enumerative type) as discussed in Section 3.2.2. Furthermore, the control variable *max* is initialised for handling loop. The corresponding present states and next states of these variables are declared under the keyword “ASSIGN”. Inside the next expression of the variable, a “case...esac” expression is created for every state that lists all possible subsequent states.

3.3.3 Containment Checking Results

The containment checking is achieved by using the NuSMV model checker to check the generated SMV descriptions (as illustrated in Listing 3.1) against the LTL formulas generated from the high-level model (as presented in Table 3.2). The LTL formulas and SMV descriptions are combined into one NuSMV input file and executed by the NuSMV model checker. Listing 3.2 shows the verification result including the list of satisfied and unsatisfied LTL formulas. The NuSMV model checker generates a counterexample demonstrating a sequence of permissible state executions leading to a state in which the violation occurs in LTL formula. By looking at the violation reported as a counterexample by NuSMV, we find that LTL formulas “G (SendStarterAccountContractKit -> F CreateLoanFile)”, “G (CreateLoanFile -> F SendAccountIdandWelcomeMessage)” and “G (ForkNode2 -> F (NotifyBank & NotifyCustomer & NotifyLoanOfficer)) & G ((NotifyBank & NotifyCustomer & NotifyLoanOfficer) -> O ForkNode2)” are false. It means that this sequence of formal properties specified by the high-level loan approval model is not contained in its low-level counterpart. Despite the size and execution traces of this counterexample, the exact cause of the containment inconsistency is unclear, for instance, “*is the containment violation caused by a missing element, or a misplacement of elements, or both of them?*”. After the generation of the counterexample, it is important to analyse the generated counterexample to find the actual source of the inconsistency and correct the responsible elements in the model.

```

1 MODULE main
2   VAR
3     InitialNode      : boolean;
4     ReceiveLoanRequest : boolean;
5     ReviewRequest    : boolean;
6     GenerateAccountId : boolean;
7     ForkNode         : boolean;
8     JoinNode         : boolean;
9     MergeNode        : boolean;
10    merge_flag_1      : {undetermined, in_DecisionNode4, in_DecisionNode3};
11    .....
12  ASSIGN
13    init(InitialNode) := TRUE;
14    next(InitialNode) := case
15      InitialNode : FALSE;
16      TRUE       : InitialNode;
17    esac;
18    init(ReceiveLoanRequest) := FALSE;
19    next(ReceiveLoanRequest) := case
20      InitialNode : TRUE;
21      ReceiveLoanRequest : FALSE;
22      TRUE       : ReceiveLoanRequest;
23    esac;
24    .....
25    init(SendStarterAccountContractKit) := FALSE;
26    next(SendStarterAccountContractKit) := case
27      GenerateAccountId : TRUE;
28      SendStarterAccountContractKit : FALSE;
29      TRUE       : SendStarterAccountContractKit;
30    esac;
31    init(SendAccountIdandWelcomeMessage) := FALSE;
32    next(SendAccountIdandWelcomeMessage) := case
33      SendStarterAccountContractKit : TRUE;
34      SendAccountIdandWelcomeMessage : FALSE;
35      TRUE       : SendAccountIdandWelcomeMessage;
36    esac;
37    .....
38    init(RequestBankInformation) := FALSE;
39    next(RequestBankInformation) := case
40      post_decision_975 = guard_2 : TRUE;
41      RequestBankInformation : FALSE;
42      TRUE       : RequestBankInformation;
43    esac;
44    init(CreateLoanFile) := FALSE;
45    next(CreateLoanFile) := case
46      RequestBankInformation : TRUE;
47      CreateLoanFile : FALSE;
48      TRUE       : CreateLoanFile;
49    esac;
50    init(ReceiveSupportingDocuments) := FALSE;
51    next(ReceiveSupportingDocuments) := case
52      CreateLoanFile : TRUE;
53      ReceiveSupportingDocuments : FALSE;
54      TRUE       : ReceiveSupportingDocuments;
55    esac;
56    .....

```

LISTING 3.1: Excerpts of the SMV Description for the Low-level Loan Approval Model

```

$ NuSMV LoanApproval.smv
-- specification G (InitialNode -> F ReceiveLoanRequest) is true
-- specification G (ReceiveLoanRequest -> F SendStarterAccountContractKit) is true
-- specification G (SendStarterAccountContractKit -> F CreateLoanFile) is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
InitialNode = TRUE
ReceiveLoanRequest = FALSE
.....
-- Loop starts here
-> State: 1.18 <-
NotifyBank = FALSE
NotifyCustomer = FALSE
merge_flag_3 = undetermined
-> State: 1.19 <-
merge_flag_3 = in_JoinNode1
-> State: 1.20 <-
merge_flag_3 = undetermined
-- specification G (CreateLoanFile -> F SendAccountIdandWelcomeMessage) is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
.....
-- specification G (MergeNode1 -> F ForkNode2) is true
-- specification ( G (ForkNode2 -> F ((NotifyBank & NotifyCustomer) & NotifyLoanOfficer)) & G (((
    NotifyBank & NotifyCustomer) & NotifyLoanOfficer) -> 0 ForkNode2)) is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
.....

```

LISTING 3.2: NuSMV Containment Checking Result along with a Counterexample

3.4 Performance Evaluation

So far, we have presented a scenario from industrial case study representing real systems from the banking sector that illustrate how our approach works and demonstrate its applicability in real cases. As our approach aims to support the developers to verify the containment relationship during their development tasks, it is crucial to assess whether our approach's performance is reasonable in a normal working environment. The sizes of the input models used for performance evaluation are ranging from a few dozens to hundreds of elements, which are the typical sizes of software behaviour models that developers can efficiently work with [Stö14].

We evaluate the performance of our approach using five different cases. Four of them are taken from the industrial scenarios. One of them is the *Loan Approval* mentioned in the previous section. The other three are *Itinerary Management*, *Customer Relationship Management* (CRM) fulfilment system [Tra+12] and *Billing Renewal* [Tra+11]. The itinerary management provides the services for booking airline tickets, hotels, and cars, respectively. The CRM fulfilment scenario is a part of a customer relationship management system concerning the customer care, billing, and provisioning of an Austrian Internet Service Provider. The billing renewal system concerns a billing and provisioning system of a domain registrar and hosting provider. Furthermore, we artificially increased the size of the billing renewal system by adding a number of control nodes, actions and edges to evaluate a model with close to 100 elements in the high-level model and more than 300 elements in

the low-level model. This case is called Synthetic Larger Model. While the Synthetic Larger Model is in our point of view already a model that is too large for efficiently working with it, this model provides a useful data point in terms of an upper bound. The performance evaluation is conducted on a regular computer equipped with an 2.6 GHz i5 processor and eight gigabytes of memory running Windows 8. The approach under consideration is implemented using Java and executed with the Java VM 1.7, in particular, the automated transformation UML activity diagrams into formal descriptions/consistency constraints has been realised using Eclipse Xtend³. Note that we used the NuSMV model checker version 2.5.4 for verifying the containment relationship.

Input size	Itinerary management		CRM fulfillment		Billing renewal		Loan approval		Synthetic larger model	
	HL	LL	HL	LL	HL	LL	HL	LL	HL	LL
Control Nodes	12	14	7	8	12	17	14	19	23	53
Action Nodes	7	13	12	16	14	22	21	33	29	92
Edges	21	35	23	30	35	54	43	61	70	179
Total Elements	30	59	42	67	61	93	78	113	122	324
Model Loading (ms)	1.308±0.54	2.442±0.28	2.680±0.65	2.955±0.72	3.115±0.41	3.457±0.57	3.374±0.72	3.396±0.64	3.559±0.54	6.904±0.58
Translation Time (ms)	0.202±0.07	0.437±0.34	0.274±0.07	0.467±0.14	0.464±0.22	0.411±0.04	0.460±0.12	3.115±0.41	0.697±0.15	4.798±0.31

TABLE 3.3: Model Size and Translation Time of UML Activity Diagrams

The time taken for model loading, transforming and verification of models are measured in milliseconds. Before measuring each task, sufficient warming up executions are performed to eliminate potential confounding factors of class loading in Java. The first part of Table 3.3 shows the complexity of the input UML activity diagrams (HL = high-level model, LL = low-level model) with regard to their elements including control nodes (i.e., those nodes that can change the flow of the execution), action nodes (including special actions that represent data handling tasks), and edges (representing the links between nodes). The second part of the table presents the evaluation results for loading and transforming activity diagrams into formal constraints and descriptions. Table 3.4 shows the containment checking time, total time (i.e., verification plus loading and translation time) and violated formulas.

The evaluation results show that the containment checking time spent by NuSMV for the loan approval is longer than for the itinerary management, CRM fulfillment system and billing renewal system. It is because NuSMV found inconsistencies between the formal properties of the low-level model and the LTL formulas of the high-level model and thus NuSMV needed to generate a counterexample. We note that the NuSMV model checker consumes more time for verification than is used for translation of the models. The results show that the loading and translation times of the itinerary management system are lower than for the other models because the activity models of the other three systems contain a greater number of control and action nodes. In summary, all realistic cases are handled in a total time below or around two seconds which is quite acceptable for practical purposes.

We also applied our approach on a Synthetic Larger Model which contains a wide range and larger number of model elements. The time taken for containment checking of the Synthetic Larger

³See <http://www.eclipse.org/xtend>

Model (HL=122 and LL=324) is around 35630 ms (i.e., about 0.59 minutes). The model checking time, although it grows considerably for such a large number of elements, is still reasonable in the context of a typical working environment. The verification of the Synthetic Larger Model has allowed us to evaluate the scalability of our containment checking approach. The evaluation results show that our approach efficiently translates activity diagrams into formal specifications for supporting containment checking. The analysis and evaluation results also demonstrate that our approach works well for larger realistic scenarios. Our approach can also handle composite controls (combinations of two or more control structures) quite well. For instance, activity diagrams of the loan approval system contain a Merge Node after a Decision Node that have been adequately mapped into LTL formulas. Moreover, after locating the causes of containment inconsistencies for loan approval system and systematic larger model, the low-level activity diagrams of these models are updated and re-mapped to their formal descriptions, and then re-verified. As expected, the rerunning of the containment checking process on these models yielded no further violations.

Containment checking	Itinerary management	CRM fulfillment	Billing renewal	Loan approval	Synthetic larger model
Verification Time (ms)	177.5±12.817	97.5±4.629	190±16.035	1728.75±64.017	35615±1319.762
Total Time (ms)	181.889	103.876	220.258	1739.095	35630.958
Violated Formulas	0 out of 11	0 out of 11	0 out of 18	3 out of 27	2 out of 36
Reachable States	98 (2 ⁶ .61471)	40 (2 ⁵ .32193)	221 (2 ⁷ .7879)	5.6776e+006 (2 ²² .4369)	931 (2 ⁹ .86264)
Total States	2.37763e+013 (2 ⁴⁴ .4346)	3.71085e+012 (2 ⁴¹ .7549)	2.00386e+015 (2 ⁵⁰ .8317)	1.89108e+022 (2 ⁷⁴ .0016)	5.68419e+036 (2 ¹²² .096)

TABLE 3.4: Performance Evaluation Results for UML Activity Diagrams

One of the issues of our approach is to rely on model checkers which can suffer from the “*state space explosion*” problem. The number of states examined by the model checkers can grow exponentially with the size of the input [CW96; CGP99]. This problem can be partially alleviated by using predicates instead of enumeration types [EW02]. The size of models can negatively affect the verification time of containment checking. This is because symbolic model checking allows verification of large systems that have over 10^{20} states [Bur+92]. In this work constraints are abstracted and encoded as boolean representative (see Section 3.2.2). This encoding decision can help reducing significantly the state space under consideration.

3.5 Discussion

In this section, we discuss various aspects of our model checking based approach for the containment relationship between a low-level activity diagram and its high-level counterpart. The research questions are revisited and briefly discussed below:

Q1: Can we achieve formal specifications and descriptions for containment checking fully automatically?

We have introduced a set of transformation rules to facilitate the automated transformation of high-level and low-level UML activity diagrams into LTL constraints and SMV descriptions, respectively. Our proposed translation techniques provide effective means for automated generation of formal specifications for large and complex behavioural models. These techniques aim at reducing the burden on the developers for manually specifying the consistency constraints and formal descriptions of behaviour diagrams to check for containment inconsistencies. In the scope of this chapter, we covered complex structures of UML activity diagrams along with basic constructs. The more complex structures of activity diagrams are often used for modelling complex software system behaviour but have not been adequately considered by most of the existing studies in the literature so far.

Because containment checking is performed on generated SMV descriptions and LTL constraints, we present a simple sketch of a proof. The idea is to prove that an activity diagram and SMV descriptions derived from it are behaviourally equivalent. Let's assume that an activity diagram \mathcal{A} is a tuple (N, E, G) where N is a finite set of nodes derived from the UML 2 specification [Gro11b, Sec. 12] *InitialNode*, *ActivityFinalNode*, *FlowFinalNode*, *Action*, *DecisionNode*, *MergeNode*, *ForkNode*, *JoinNode*, *ParameterNode* and *ExceptionHandler*, $E \subseteq N \times N$ is an ordered finite set of edges, and G is a finite set of guard expressions. An edge e connects a source node s to a target node t : this is represented by $e(s, t)$ in which $e \in E$ and $s, t \in N$. Note that the semantics of SMV corresponds to a finite state machine tuple $M = (V, S, R, I)$ where S and V are set of states and variables respectively. Specifically, a state s assigns a value $s(v)$ to each variable $v \in V$. The transition relation $R \subseteq S \times S$ specifies the possible state to state transitions; $I \in S$ is the initial state. The SMV translation rules mentioned in Section 3.2.2 might also be considered for mapping of activity diagrams to finite state machines. An execution (or trace) of finite state machine is a finite sequence of states (s_0, s_1, \dots, s_n) so that $s_0 = I$ and each pair s_i, s_{i+1} in the sequence, $(s_i, s_{i+1}) \in R$. Therefore, the SMV descriptions for the activity diagrams are derived by translating each N into a state S in M – they have same content. The next sequential node is replaced by the next state. That is an edge $e = (s, t)$ will have the same semantics to $(s_i, s_{i+1}) \in R$. It is however rather straightforward that activity diagrams and derived SMV descriptions are behaviourally equivalent. In LTL a path is an ordered sequence of states, such that each state is followed by its next state via a transition. Accordingly, we have derived several formulas from the high-level model that have similar semantics to the derived SMV descriptions.

Our approach makes use of the existing model checkers for formally verifying the containment relationship. Unfortunately, one of the drawbacks of model checking techniques is that they are not scalable to very large systems. The number of states in the finite state representation increases exponentially with the number of variables (i.e., the state explosion problem). The NuSMV model checker used in our study is based on the symbolic model checking technique, and therefore, is able to support the verification of large systems up to 10^{20} states [Bur+92]. One of the biggest advantages of using the NuSMV model checker is that SMV's finite state based

encoding of the input behaviour models is rather straightforward. That is, each model element is represented by a boolean variable in the SMV description and LTL formulas. However, this also implies that our technique is only applicable for other model checkers that support similar encoding techniques, such as SPIN⁴. Like NuSMV, the SPIN model checker has similar concepts regarding LTL formulas and exhaustive verification options; hence, it is possible to easily modify the encoding with reasonable extra efforts.

Q2: Can we design a containment checking approach that offers an acceptable performance for realistically sized input models?

In order to illustrate the applicability and feasibility of the overall approach, we conducted industrial case studies in the banking sector and e-business domains from previous industry projects of our team [Tra+11; Tra+12] and also performed performance evaluations of our approach in these cases. By analysing the evaluation results we found that our approach efficiently translates activity diagrams into formal specifications and works well for larger realistic scenarios. The time taken for both transformation and verification of all these realistic scenarios is less than 2 seconds. Our study of a synthetic larger model also shows the exponential growth for much larger models, so that we suggest to perform performance evaluations if the approach should be used for very large models and immediate results are needed. However, our evaluation results show that even with the input models having about 300 model elements, the verification time stays within less than 1 minute. Therefore, the proposed containment checking approach can be used for practical purposes.

One of the most challenging issues, among others, in comparing behaviour models is to deal with various types of loops. In our approach we currently consider one case to handle loops which are formed by combining decision nodes and merge nodes. However, the derived mapping rules for the loop can be easily applied on the **Loop Node** as mentioned in the UML 2 specification [Gro11b]. Another issue is that a loop structure, especially an unconditional loop (i.e., the number of iterations may be nondeterministic), cannot be efficiently described by temporal logics. A loop with a predetermined number of iterations can be represented using temporal logics such as k -bounded existence [DAC98]. Nevertheless, Dwyer et al. [DAC98] show that even in the case $k = 2$, the bounded existence structure already becomes rather sophisticated. Thus, automated generation of complex temporal formulas like our approach does can help to efficiently deal with such complex structures. Furthermore, the containment relationship between two behaviour models at different abstraction levels is based on the assumption that element names of a high-level model and its corresponding low-level counterparts are aligned to a common ontology respected by all stakeholders. The assumption is rather realistic because a low-level model is mainly achieved through a refinement of a high-level model where existing high-level elements are often enriched with more details and elements [TZD10].

⁴See <http://spinroot.com>

3.6 Related Work

Several approaches have been proposed in the literature for model consistency checking. In the subsequent sections, we briefly summarise the existing works related to our approach, in particular, approaches that focus on consistency checking and containment checking as well as those approaches that provide the mapping of models into formal descriptions, specification of formal constraints and formalisations of activity diagram.

3.6.1 Existing Approaches for Consistency Checking

In the literature, many approaches tackled different types of models and/or model checking techniques [LMT09; SZ01; MTZ17a]. Some of them focus on checking the consistency of behavioural models against structural models or checking different types of behaviour models (models and other representations of the same reality such as the requirements or implementations). For instance, Yeung proposes an approach for checking the consistency between UML class diagram (i.e., structural model) and statechart model (i.e., behavioural model) by applying a pair of integrated formal methods, namely B method and Communicating Sequential Processes (CSP) [Yeu04]. The models are manually translated into B and CSP and also no discussion regarding the automation level of verification is provided. Van der Straeten et al. present an approach for checking the consistency of different UML models by using description logic [Str+03]. This approach tackled the UML version 1.5 and concentrates on evolution consistency (i.e., consistency between different versions of the same model). Graaf and van Deursen introduce a model-driven consistency checking approach for checking the consistency between several behaviour models [GD07]. The approach first normalises the input models, then performs an automated model transformation to UML statecharts, and afterwards compares the different statecharts to identify inconsistencies. Knapp et al. check the consistency between two views of a system by verifying the state machines with sequence diagram [SKM01; KMR02]. The authors propose an encoding system, namely, HUGO, for translating state machines into PROMELA (Process or Protocol Meta Language), the formal input language of the SPIN⁵ model checker [SKM01]. In another study, the authors present the transformation of timed UML state machines and collaborations into timed automata using the HUGO/RT tool, and consistency checking is performed using the UPPAAL⁶ model checker [KMR02]. Amálio et al. [ASP04] developed a modelling framework to construct UML class, state and snapshot diagrams (i.e., object diagrams) and analyse models of sequential systems using Z notation. However, models are manually translated into Z but formal analysis is supported by the Z/Eves⁷ theorem prover.

⁵<http://spinroot.com>

⁶<http://www.uppaal.org>

⁷<http://czt.sourceforge.net/eclipse/zeves/>

Lam and Paget propose an algebraic approach for verifying the consistency between sequence and statecharts. In this approach, behaviour models are encoded into π -calculus to support the verification of model consistency [LP05]. Wang et al. introduce an approach for consistency checking between UML 1.5 statecharts and sequence diagrams [Wan+05]. In this approach finite state processes are used to express statecharts whereas a trace of messages is used for sequence diagrams. Heimdahl et al. propose deviation analysis approach using existing model checkers, related to the identification and analysis of changes in system behaviour between two similar control systems in slightly distinct environments [HCW05]. The $RSML^{-e}$ language is used for the specification of the systems which is then translated into NuSMV⁸ input language, whereas constraints are manually specified in Computation Tree Logic (CTL).

The aforementioned approaches concentrate on consistency between different models, typically from different development activities (such as requirements elicitation and implementation) or between different views of the system. The major difference of these approaches and our approach is that we consider the consistency of the same model at different levels of abstraction, i.e., “vertical consistency” [Str05]. In particular, we focus on checking the consistency of the containment of the high-level model in the low-level model, rather than checking the consistency of elements of two different representations. These approaches introduce the formalisations for class, sequence, or statecharts but not for activity diagrams. Their formalisations might be used when it is important to consider the connection among different diagrams in the verification process but not for verifying the containment relationship for activity diagrams.

In the field of business process management, several approaches on computing the similarity of process models have been presented [BL12]. Some approaches concentrate on retrieving process models in the large repository that are similar or even identical models of a given process model or fragment thereof (the search model) [DDGBn09; Dij+11]. Dijkman et al. focus on three perspectives of similarity: text similarity based on a comparison of the labels, structural similarity based on the topology of the process models, and behavioural similarity based on the causal relations between activities in a process model. However, a search query involves determining the degree of similarity between the search model and each model in the repository.

While other approaches focus on a trace theory to validate the conformance of two process models or process models against their execution traces recorded in the event logs. An approach presented in [AMW06] measures the degree of behavioural similarity between Petri net based process models and their execution traces ranging from “completely different” to “identical”. Another approach aims at verifying whether two process models are similar using their corresponding event traces mined from process execution logs [Aal+08]. Bae et al. propose quantitative process similarity metric for measuring mining process similarity and difference [Bae+07]. In this method, dependency graphs are extracted from process models and converted into normalized matrices. Afterwards,

⁸ <http://nusmv.fbk.eu>

metric space distances are calculated based on the difference between the normalized matrices. However, these approaches produce an estimated degree of similarity of these models and are useful for finding similar or alternative behavioural descriptions but not applicable for verifying the containment relationship.

3.6.2 Containment Checking

In recent years, the notion of behaviour inheritance has been studied in the realm of consistency checking of behaviour diagrams, in particular, the inheritance of object life cycles in statecharts. Stumptner and Schrefl introduce specialisations of object life cycles by examining extension and refinement in the context of UML statecharts. The approach introduced a one-to-one rule-based mapping of statecharts into some semantic domains but does not support automation [SS00]. Van der Aalst presents a theoretical framework for defining the semantics of behaviour inheritance [Aal02]. In this work, four different inheritance rules, based on hiding and blocking principles, are defined for UML activity diagram, statechart and sequence diagram. However, the application of these inheritance concepts in the context of actual scenarios is not clarified, such as the formal semantics of the consistency constraints or the automatic generation of these formal descriptions. In addition, this approach only considers some selected parts of UML diagrams. Engels et al. [EKG02] deal with the consistency of models made up of different submodels: UML-RT capsule and protocol statecharts. They used CSP as a mathematical model for describing the consistency requirements with respect to deadlock freeness and protocol conformance; the FDR⁹ (Failures-Divergences Refinement) tool is used for checking purposes. However, the trace model can only be used for expressing that some actions will not occur and also not all actions indeed occur. CSP does not support state variables; however, they can be simulated to some extent by using a recursive process with parameters.

A closely related work is proposed by Egyed for structural models. In particular, Egyed's approach aims to check whether a UML class diagram conforms to another more abstract class diagram based on structural transformation rules [Egy02]. This approach alone is not applicable for containment checking, as behaviour models contain control constructs that cannot be matched only structurally. Arcaini et al. [AGR16] and Krings and Leuschel [KL16] focus on mathematical proof of a logical relation between an abstract model and its refined models. The former approach concentrates on a proof of stuttering refinement between two Abstract State Machines (ASMs). Specifically, the translation of ASM to SMT (Satisfiability Modulo Theories) instances is proposed, and the SMT solver Yices¹⁰ is used to prove refinement correctness. The later focusses on the implementation of BMC, k-Induction and IC3 symbolic model checking algorithms for B and Event-B. In particular, they included the proof information to improve the algorithms' performance and integrated them in the nightly builds of ProB¹¹. However, these approaches consider formal specification languages

⁹ <https://www.cs.ox.ac.uk/projects/fdr>

¹⁰ <http://yices.csl.sri.com/>

¹¹ <http://www3.hhu.de/stups/prob/>

(i.e., B, Event-B and ASM) to describe a system architecture instead of software behavioural models. Therefore, in contrast to our approach, these approaches require a strong mathematical background from the user.

So far, none of the published studies have considered the containment checking problem for behaviour models. The idea with this research is to resolve the particular problem for a comprehensive set of modelling constructs used for describing software behaviour such as exception handlers, interruptible activity regions, parameterized tasks, event actions, and loops.

3.6.3 Mapping of Models into Formal Descriptions and Specification of Formal Constraints

The existing consistency checking approaches mainly focus on the manual mapping of input models into formal descriptions, in addition, the consistency constraints (in the form of formal specifications) are provided by the users. These approaches require a substantial time and effort, and therefore, are not favourable by developers and non-technical stakeholders. For example, Koehler et al. propose a model checking approach for a business process model and its implementation [KTK02]. The approach uses automata as the basis of the operational models through which the behaviour of the processes and their implementing systems are specified; however, automated transformation of automata into the input language of NuSMV is not supported. Engels et al. check contracts between business processes (modelled as UML activity diagrams) and Web services (specified by visual contracts), i.e., they check the consistency between behavioural and structural properties [Eng+08]. The approach uses graph-based algorithms for checking the consistency. Martens' technique verifies the consistency between a locally specified executable model and a globally described abstract process model based on the simulation relation [Mar05]. To perform verification, BPEL process models are manually transformed into Petri nets. Ehrig and Tsiolakis use attributed graphs to represent sequence diagrams and attributed type graphs with graphical constraints for class diagrams [TE00]. This approach considers only the existence, correct multiplicity, and valid scoping for checks of model elements. In contrast to these approaches our approach introduces transformation rules grounded on formal expressions. Therefore, the high-level behaviour models are automatically translated into consistency constraints (i.e., LTL formulas) and low-level behaviour models into formal descriptions using the transformation rules. The generated consistency constraints and formal descriptions are used by model checkers to verify the containment relationship.

There are only few approaches that introduce process patterns to provide help for specifying the formal constraints, for instance, Förster et al. introduce an approach to visualise the modelling of business process constraints through the Process Pattern Specification Language (PPSL) [F+07]. To perform the conformance checking, the corresponding process constraints are further translated into temporal logics whilst business processes are translated to transition systems by using DMM+

rules and GROOVE¹² (Graphs for Object-Oriented Verification). Janssen et al. present the business query patterns to support specification of process requirements, afterwards, these patterns are translated into LTL formulas, whereas, business processes modelled using Testbed modelling language AMBER (Architectural Modelling Box for Enterprise Redesign) that are further translated into PROMELA for performing verification [Jan+99]. Wasylkowski and Zeller propose the concept of operational preconditions along with a tool named Tikanga [WZ09]. In the approach, first a Java program is provided as an input and Kripke structures are derived from them. Then, for each Kripke structure a set of CTL formulas are derived from user-given templates. Afterwards, these specifications are generalised into operational preconditions for detecting violations. Despite of the few attempts to support the specification of requirements or formal constraints using patterns, these approaches still require a considerable amount of knowledge of formal specifications/patterns. In particular, these approaches do not eliminate the need for translating the requirements directly into formal constraints.

In our approach we considered LTL, as its formulas are defined over Kripke structures such that every state must have at least one successor state. Furthermore, LTL past operators provide a more concise and intuitive way to reason about previous states and transitions. A major difference between our approach and existing works is that our approach provides automatic translation of the high-level input model into formal consistency constraints (i.e., LTL formulas) thus, does not require the strong knowledge of formal methods.

3.6.4 Formalisations of Activity Diagram

Many attempts have been made to give a formal definition for UML activity diagrams to enable model checking. For instance, Yang and Zhang present an approach for formalising the UML 1.4 activity diagrams into π -calculus to check the consistency between requirements and business processes, modelled using activity diagrams [DS03]. Guelfi and Mammar introduce formal semantics for activity diagrams enriched with timing aspects and translate them into PROMELA. In this approach the temporal logics for verification of activity diagrams are specified by the users [GM05]. Eshuis and Wieringa present the formal semantics of UML 1.3 activity diagrams for workflow models based on STATEMATE semantics (see [HN96]) of statecharts and also introduce data integrity constraints [EW01]. In [EW02], they also propose a verification approach for verifying workflow models represented in UML 1.4 activity diagrams. In this approach, first an activity diagram is converted into an activity hypergraph, then the hypergraph is encoded into a Kripke structure. In another work [Esh06], Eshuis extends prior work by refining existing translations and introducing another translation based on the statechart semantics of UML 1.5 to check data integrity constraints for an activity diagram and a set of class diagrams. In the translation, the author handles fork and join nodes as transitions rather than nodes. These approaches tackle UML 1.x semantics

¹² <http://groove.cs.utwente.nl>

Reference	Consistency Type	Model Type	UML Version	Translation to Formal Descriptions	Support for Consistency Checking
Yeung [Yeu04]	horizontal	statechart, class diagram	UML 1.4	manual	-
Van der Straeten et al. [Str+03]	evolution-horizontal	statechart, class diagram, sequence diagram	UML 1.5	manual	automated
Graaf and Van Deursen [GD07]	horizontal	statechart, sequence diagram	UML 1.4	semi-automated	manual
Amálio et al. [ASP04]	horizontal	statechart, class diagram, snapshot diagram	—	manual	automated
Knapp et al. [SKM01; KMR02]	horizontal	statechart, collaboration diagram	UML 1.4	semi-automated	automated
Lam et al. [LP05]	horizontal	statechart, sequence diagram	UML 2.0	semi-automated	automated
Wang et al. [Wan+05]	horizontal	statechart, sequence diagram	UML 1.5	semi-automated	automated
Heimdahl et al. [HCW05]	horizontal	$RSML^{-e}$	—	semi-automated	automated
Dijkman et al. [DDGBn09; Dij+11]	horizontal	process model	—	semi-automated	automated
Bae et al. [Bae+07]	horizontal	process model	—	semi-automated	automated
Stumptner and Schrefl [SS00]	vertical	statechart	UML 1.3	semi-automated	automated
Van der Aalst [Aal02]	vertical	statechart, activity diagram, sequence diagram	UML 1.4	semi-automated	automated
Egyed [Egy02]	vertical-structure containment	class diagram	UML 1.3	automated	automated
Arcaini et al. [AGR16]	vertical	abstract state machine	-	automated	automated
Krings and Leuschel [KL16]	vertical	-	-	-	automated
Van der Aalst et al. [AMW06]	horizontal	process model	—	manual	automated
Van der Aalst et al. [Aal+08]	horizontal	process model	—	semi-automated	automated
Engels et al. [Eng+08]	horizontal	activity diagram, visual contracts	UML 2.1	semi-automated	automated
Engels et al. [EKG02]	horizontal-vertical	capsule and protocol statecharts	UML 1.4	manual	automated
Koehler et al. [KTK02]	vertical	process model	—	semi-automated	automated
Förster et al. [F+07]	horizontal	process model	UML 2.0	automated	automated
Wasylikowski and Zeller [WZ09]	horizontal	Java program	—	semi-automated	automated
Janssen et al. [Jan+99]	horizontal	process model	—	automated	automated
Martens [Mar05]	vertical	process model	—	semi-automated	automated
Tsiolakis and Ehrig [TE00]	horizontal	class diagram, sequence diagram	UML 1.3	manual	automated
Yang and Shensheng [DS03]	horizontal	activity diagram	UML 1.4	manual	automated
Eshuis and Wieringa [EW01]	—	class diagram, activity diagram	UML 1.3	—	—
Eshuis [Esh06]	horizontal	class diagram, activity diagram	UML 1.5	semi-automated	automated
Eshuis and Wieringa [EW02]	horizontal	activity diagram	UML 1.4	semi-automated	automated
Guelfi and Mammar [GM05]	horizontal	activity diagram	UML 2.0	manual	automated
Lam [Lam08]	horizontal	activity diagram	UML 2.1	manual	—
Lam [Lam07]	horizontal	activity diagram	UML 2.1	manual	automated
Our Approach	vertical-behaviour containment	activity diagram	UML 2.4.1	automated	automated

TABLE 3.5: Overview and Comparison of Related Work

based on statecharts; although activity diagrams and statecharts seem syntactically similar, every activity diagram cannot be transformed into a statechart [GM05]. Besides that, UML 2 activity diagrams have different semantics based on Petri nets [Esh06; GM05]. Furthermore, the modelling capability of UML 2.0 allows unrestricted parallelism, whereas in UML 1.x, the entire state machine (activity) performed a run-to-completion step. For these reasons the translation of activity diagrams into formal languages is not considered as feasible and applicable [GM05].

Lam introduces theoretical foundations for specifying and classifying different equivalence notions of a subset of UML activity diagrams [Lam08]. In [Lam07] the author also provides the semantics of UML 2 activity diagram into input language of NuSMV model checker to check the correctness of the activity diagram. In the formalisation activity edges are directly encoded as tokens of typed boolean variable. The aforementioned approach uses similar concepts for formalisation of activity diagram, however, comparing to our proposal, it does not provide clear information for dealing with data.

In our approach, we create a state variable of type boolean in the SMV descriptions (i.e., input language of NuSMV model checker) for each construct of a UML activity diagram. Similar to Eshuis and Wieringa's technique [EW02], we abstract and encode constraints that are associated with nodes or edges as boolean variables. Compared to these previous works, our formalisation makes several different choices, such as our treatment of merge and decision nodes. In this research, we not only focus on a fundamental set of modelling constructs, but also complex structures such as exception handlers, interruptible activity regions, parameterized tasks, event actions, and loops. The complex structures are, although highly useful in certain modelling situations to enhance the maintainability of the overall system, rarely considered in existing other approaches. For instance, exception handlers and interruptible activity regions can be used to describe exceptional circumstances that might occur during the execution of a system [Ler+10], and loops are used for handling the situations that require certain action(s) to be executed more than once. Our approach is based on UML 2 activity diagram rather than statechart based semantics. None of these semantics, in our opinion, achieves the desired simplicity and conceptual clarity of verifying the containment relationship for activity diagrams. They are only useful for verification of activity diagrams against safety and/or liveness properties such as deadlock freedom. Besides the automated transformation of the low-level activity diagram into formal SMV descriptions, we consider high-level activity diagram as an input model which is automatically translated into formal consistency constraints.

Table 3.5 summarises the related works and compares them to our work in the context of model consistency checking, especially containment checking, activity diagram formalisation, modelling support and mapping of models into formal descriptions and consistency checking. However, the details for manually specifying the formal consistency constraints (e.g., rules specified through Assertions, LTL and CTL etc.) are not presented in this table. The minus (−) symbols in the table represents the unavailability of information regarding particular activity. The table demonstrates

that none of the published approaches have considered the containment checking problem for behaviour models. It might be noted that automatic translation to formal consistency constraints have also not been considered in existing approaches.

3.7 Summary

This chapter has investigated the problem of containment checking for software behaviour models at different levels of abstraction in order to improve the quality and correctness of the software system, which is based on the Research Question RQ2. The containment checking of behaviour models using formal verification techniques requires both formal descriptions and consistency constraints of these models. In our approach, on the one hand, automated transformation of high-level UML activity diagrams into LTL formulas is provided. On the other hand, low-level activity diagrams, often resulting from various steps of refinement and enriching of the high-level counterparts, are transformed into formal SMV descriptions. Therefore our automated translation strategy is useful to bridge the gap between manual specification of formal properties as well as consistency constraint for containment checking. To illustrate the applicability of the proposed approach, we realized use case scenarios of loan approval system, itinerary management, CRM fulfilment, billing renewal system and synthetic larger model; the performance evaluation is also carried out in particular cases. The evaluation results demonstrate that the proposed approach performs for typical activity diagram size reasonably well in a typical working environment. Through the evaluation of industrial scenarios, we also show that automated transformation of activity diagrams into formal constraints and/or descriptions significantly increases the usability of formal languages in practice. In contrast to activity diagrams, the sequence diagrams represent different perspectives of a system and have different semantics. The next chapter will investigate the sequence diagrams in order to support containment checking throughout the behavioural views of a software system model.

4

Model Checking Based Containment Checking of UML Sequence Diagrams

This chapter presents a model checking based approach for containment checking between UML 2 sequence diagrams at different levels of abstraction. Previous studies have not adequately addressed the containment checking problem in sequence diagrams. This problem addresses Research Question RQ2. Furthermore, the chapter is based on a peer-reviewed conference paper in the proceedings of the 23rd Asia-Pacific Software Engineering Conference [MTZ16].

4.1 Introduction

In software development process, scenarios are often modelled using sequence diagrams to describe the interactions among environment (e.g., human beings) and components (aka lifelines) for analysing the behaviour of software systems. In the course of software system modelling, as models are created and evolved independently by different stakeholders, inconsistencies among models often occur. Therefore, it is crucial to detect and fix the inconsistencies at early phases of the software development process. To address the particular problems, we developed a technique which allows to automatically check containment consistency of UML 2 sequence diagrams based on model checking techniques that has not been addressed adequately so far. The idea behind containment checking is to verify whether the behaviour (or functions) described by the refined and extended low-level sequence model conforms those specified in the high-level counterpart. It allows the stakeholders to improve the quality of the complex systems by determining and resolving the deviations at design phase, before the systems are actually implemented and deployed.

Although, some semantics have been proposed for the verification of sequence diagrams against safety properties such as deadlock freedom [Stö04; Hau+05; JS15], they do not cover the containment relationship between sequence diagrams. In this work we therefore provide the efficient and simpler formalisations of sequence diagrams to track the execution state of an interaction involving send/receive events of messages and combined fragments without compromising the containment relationship. Specifically, we introduced a fully automated technique to translate both high-level and low-level sequence diagrams into temporal logic based constraints (LTL) [Pnu77] and formal

behaviour descriptions (i.e., symbolic model language (SMV) [Cim+99]), respectively. The model checker NuSMV is used to verify the containment relationship. If the consistency constraints do not satisfy formal descriptions then the model checker produces a counterexample as a trace of states. We have developed a tool to implement all of the techniques and have validated our approach by detecting inconsistencies in three realistic scenarios.

The chapter is structured as follows: Section 4.2 discusses the model checking based approach for containment checking of UML sequence diagrams. In Section 4.3 a realistic use case scenario is described in detail to illustrate our approach along with a performance evaluation of three realistic scenarios. Finally, Section 4.4 presents related work and Section 4.5 summarises the chapter.

4.2 Containment Checking Approach for UML Sequence Diagrams

In this section, we describe our model checking based approach for addressing the problem of containment checking of UML 2 sequence diagrams. An overview of our approach is presented in Figure 3.1 located at Chapter 3, consists of the following steps: (i) mapping the high-level sequence diagram into formal consistency constraints (i.e., LTL formulas), (ii) translating the low-level sequence diagram into formal SMV descriptions, (iii) verifying whether the generated constraints and descriptions satisfy the containment relationship using the NuSMV model checker [Cla+96]. The subsequent sections describe the steps involved in our containment checking approach.

4.2.1 Automated Transformation of Sequence Diagrams into LTL and SMV Descriptions

As the definitions and semantics of UML 2 sequence diagrams are rather informal and ambiguous [Gro11b], we facilitate the automated creation of the formal constraints and descriptions by defining transformation rules for formally representing constructs of sequence diagrams based on containment relationship. The main objective is to represent the high-level sequence diagram's constructs and their relationships in an LTL formalism such that the execution order of the interactions will become the consistency constraints for the corresponding low-level sequence model. Furthermore, the encoding of the low-level sequence diagram in terms of the SMV description language should provide the foundation to facilitate the verification of the containment relationship.

Our approach takes advantage of the standards that are widely used in industry, such as Eclipse Papyrus¹, which is an Eclipse based open source UML 2 tool. In order to translate the high-level and low-level sequence diagrams into LTL formulas and SMV descriptions, respectively, we

¹See <https://www.eclipse.org/papyrus>

Algorithm 4 Mapping UML Sequence Diagram *SeqD* into SMV Descriptions / LTL

```

1: procedure TRANSLATE(SeqD)
2:    $Q \leftarrow \emptyset$  ▷  $Q$  is the queue of non-visited interactions
3:    $V \leftarrow \emptyset$  ▷  $V$  is the queue of visited interactions
4:    $Q \leftarrow Q \cup \text{get\_lifelines}(\text{Lf})$ 
5:   for all  $i \in Q$  do
6:      $V \leftarrow V \cup \{i\}$ 
7:      $Q \leftarrow Q \setminus \{i\}$ 
8:     generate_smv_code( $i$ )/generate_ltl( $i$ )
9:      $I_{\text{interactions}} \leftarrow \text{get\_interactions}(i)$ 
10:    for all  $e \in I_{\text{interactions}}$  do
11:      if ( $e \notin V$ ) then
12:         $Q \leftarrow Q \cup \{e\}$ 
13:      end if
14:    end for
15:  end for
16: end procedure

```

leverage the Eclipse Xtend framework². We achieve the mapping of sequence diagram into SMV descriptions and LTL formulas using an extended version of the breadth-first search algorithm as shown in Algorithm 4. In this algorithm, we develop four helper functions, namely, `get_lifelines()`, `get_interactions()`, `generate_smv_code()` and `generate_ltl()`. The function `get_lifelines(Lf)` returns a set of lifelines. The function `get_interactions(i)` extract all interactions i , i.e., messages along with sending and receiving *OccurrenceSpecifications* (*OSs*) covered by lifelines in temporal order, associated *CombinedFragments* included operands. An interaction e is called “receiving event” of i “sending event” if there is a link from i to e that synchronize with the appropriate sending and receiving lifelines.

Algorithm 5 Generating SMV Specifications for an Interaction i of a UML Sequence Diagram *SeqD*

```

1: procedure GENERATE_SMV_CODE( $i$ );
2:   extracts interaction information;
3:   binds input values and generates SMV descriptions using the following templates:
4:   ' ' '
5:   VAR
6:     «i» : boolean; ▷ State variable declaration
7:   ASSIGN
8:     init(«i») := «interaction-initial-state»
9:     next(«i») := case
10:       «incoming-condition(s)» : TRUE;
11:       «i» : FALSE;
12:     esac;
13:   ' ' '
14:   ' ' '
15: end procedure

```

²See <https://eclipse.org/xtend>

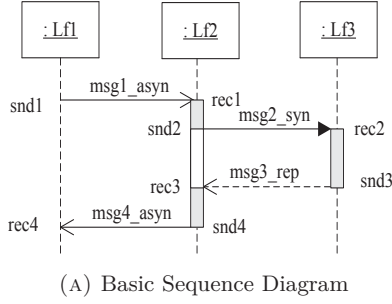
The most important function is `generate_smv_code(i)`, which is responsible for generating SMV descriptions for each interaction of a UML sequence diagram. We illustrate the skeleton of the function `generate_smv_code(i)` in Algorithm 5. The pair of triple apostrophes (`'''`) denotes the string templates used for generating code in the our implementation. For the sake of readability, we opt to omit the verbosity of the transformation code realized using the Xtend language and use a pair of guillemots i.e., “«” and “»” to denote the parameterized placeholders that will be bound to and substituted with the actual values extracted from the low-level input model interactions by the Xtend engine, as we see in Figure 4.1c. In our formalisations, we map a message and its sending and receiving *OSs* (i.e., events) as a tuple $\langle Lf, msg, snd/rec \rangle$, where *Lf* represents the lifeline that is responsible for sending or receiving messages. *msg* denotes the message name. *snd* and *rec* represent the sending *OS* and receiving *OS* of the corresponding message on a lifeline, respectively. To represent multiple inputs for an element (i.e., state or event) the logical *AND* operator (“&”) is used.

We note that `generate_smv_code(i)` is not realized as a single function but rather a polymorphism of multiple functions. That is, depending on the type of the input interaction *i*, a particular function for generating SMV descriptions for that *OS* or fragment type will be invoked. This can be achieved in traditional programming languages by using a typical “if/then/else” or “switch/case” construct. In our prototypical implementation, we leverage the powerful polymorphic method invocation technique provided by Xtend³, which is used to realize the transformation of UML sequence diagram to SMV descriptions. Using this technique, we devise multiple functions for generating SMV descriptions with respect to the input types. Due to space limitations and similar technical details we do not discuss `generate_ltl(i)` function here.

Basic Sequence Diagram A sequence diagram without a *CombinedFragment* is referred to as a basic sequence diagram (such as the one in Figure 4.1a). The following rules for sending and receiving messages must be considered as the semantics of a basic sequence diagram.

- The *OSs* on the same lifeline must occur in the same order in which they are described [Gro11b, p.505].
- “The semantics of a complete message is simply the trace $\langle sendEvent, receiveEvent \rangle$ ” [Gro11b, p.507]. A receiving *OS* of a message is enabled for execution if and only if the sending occurrence of the same message has already occurred [Gro11b, p.507].
- If sending and receiving *OSs* of the same message are on the same lifeline then the sending event of a message must exist before its receiving event [Gro11b, p.506].

³See https://eclipse.org/xtend/documentation/202_xtend_classes_members.html



```

1. G (Lf1 -> F (Lf1_msg1_async_snd1
))
2. F (Lf2_msg3_rep_rec3) -> ((!
Lf2_msg3_rep_rec3) U (
Lf2_msg2_async_snd2 &
Lf3_msg3_rep_snd3 & wait_rep)
)
3. G ((Lf2_msg3_rep_rec3) & ! (
wait_rep) -> F (
Lf2_msg2_async_snd4))
4. F (Lf1_msg4_async_rec4) -> ((!
Lf1_msg4_async_snd4) U (
Lf2_msg4_async_snd4 &
Lf1_msg1_async_snd1))

```

(B) LTL Generation Rules

```

1 VAR
2   «Lf1_msg1_async_snd1» : boolean;
3   «wait_rep» : boolean;
4   ...
5 ASSIGN
6   init(«Lf1») := TRUE;
7   next(«Lf1») := case
8     «Lf1» : FALSE;
9   esac;
10  --sending OS on a lifeline Lf1
11  init(«Lf1_msg1_async_snd1») := FALSE;
12  next(«Lf1_msg1_async_snd1») := case
13    «Lf1» : TRUE;
14    «Lf1_msg1_async_snd1» : FALSE;
15  esac;
16  --receiving OS on a lifeline Lf1
17  init(«Lf1_msg4_async_rec4») := FALSE;
18  next(«Lf1_msg4_async_rec4») := case
19    «Lf1_msg1_async_snd1» & «Lf2_msg4_async_snd4» :
    TRUE;
20    «Lf1_msg4_async_rec4» : FALSE;
21  esac;
22  ...
23  init(«wait_rep») := FALSE;
24  next(«wait_rep») := case
25    «Lf2_msg2_syn_snd2» : TRUE;
26    «wait_rep» : FALSE;
27  esac;
28  --sending OS after receiving reply
29  init(«Lf2_msg4_async_snd4») := FALSE;
30  next(«Lf2_msg4_async_snd4») := case
31    «Lf2_msg3_rep_rec3» & ! «wait_rep» : TRUE;
32    «Lf2_msg4_async_snd4» : FALSE;
33  esac;

```

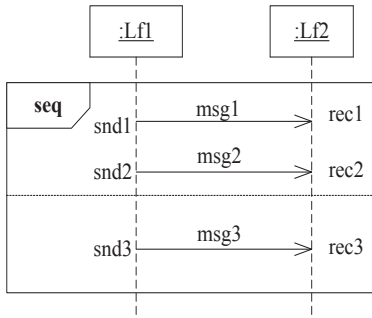
(C) SMV Generation Rules

FIGURE 4.1: Translation of Basic Sequence Diagram

Figure 4.1a shows a basic sequence diagram where *msg4_async* is a message received on the lifeline *Lf1* which is sent from the lifeline *Lf2*. The receiving *OS* is only enabled when its sending *OS* (*snd4*) on *Lf2* and its prior *OS* (*snd1*) on *Lf1* have already occurred. The particular receiving rule is shown in Figure 4.1c (Line 17–21). In case the synchronous message is sent, a condition *wait_rep* is used indicating that the *OS* can not be sent when the lifeline is waiting for the reply (Line 23–33). Figure 4.1b shows the LTL generation rules for sending *OSs* of messages *msg1* and *msg4*, and receiving *OSs* of messages *msg3* and *msg4*.

Weak Sequencing Combined Fragment The **seq** interaction operator imposes the order of the execution of operands associated with the same lifeline with the following constraints [Gro11b, p.483]:

- The ordering of events (i.e., *OSs*) within each of the operands are maintained.
- *OSs* on different lifelines from different operands may execute in any order.
- *OSs* on the same lifeline from different operands are ordered such that an *OS* of the first operand comes before that of the second operand.



(A) Weak Sequencing Example

```

1. G (seq -> F opd1) & G (opd1 ->
   F opd2)
2. G (opd1 -> F (Lf1_msg1_snd1))
3. G (Lf1_msg1_snd1 -> F (
   Lf1_msg2_snd2))
4. G ((opd2 & Lf1_msg2_snd2) -> F
   (Lf1_msg3_snd3))
5. F (Lf2_msg3_rec3) -> ((!
   Lf2_msg3_rec3) U (
   Lf1_msg3_snd3 & opd2 &
   Lf2_msg2_rec2))

```

(B) LTL Generation Rules

```

1 ASSIGN
2 ...
3 init(«opd1») := FALSE;
4 next(«opd1») := case
5   «seq» : TRUE;
6   «opd1» : FALSE;
7 esac;
8 --OSs in the first operand on Lf1
9 init(«Lf1_msg1_snd1») := FALSE;
10 next(«Lf1_msg1_snd1») := case
11   «opd1» : TRUE;
12   «Lf1_msg1_snd1» : FALSE;
13 esac;
14 init(«Lf1_msg2_snd2») := FALSE;
15 next(«Lf1_msg2_snd2») := case
16   «Lf1_msg1_snd1» : TRUE;
17   «Lf1_msg2_snd2» : FALSE;
18 esac;
19 init(«opd2») := FALSE;
20 next(«opd2») := case
21   «opd1» : TRUE;
22   «opd2» : FALSE;
23 esac;
24 --OS in the second operand on Lf1
25 init(«Lf1_msg3_snd3») := FALSE;
26 next(«Lf1_msg3_snd3») := case
27   «opd2» & «Lf1_msg2_snd2» : TRUE;
28   «Lf1_msg3_snd3» : FALSE;
29 esac;

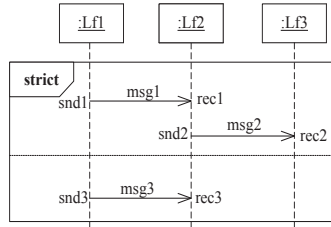
```

(C) SMV Generation Rules

FIGURE 4.2: Translation of Weak Sequencing Combined Fragment

Figure 4.2a shows an example of a *Weak Sequencing* combined fragment. Figure 4.2c illustrates the rules for mapping a *Weak Sequencing* combined fragment into SMV descriptions. The combined fragment, corresponding operands, each of its covered lifelines, and *OSs* are mapped into state variables. The choice of the order of *OSs* is made using a “case/esac” construct. For instance, the sending *OS* of a message *msg3* on a lifeline *Lf1* within the second operand *opd2* (i.e., *Lf1_msg3_snd3*) cannot occur until the last *OS* of the first operand *opd1* on the same lifeline (i.e., *Lf1_msg2_snd2*) completes its execution (Line 25–29). Figure 4.2b shows the LTL generation rules for sending messages within both operands on a lifeline *Lf1* (Line 2–4) and receiving a message within a second operand on a lifeline *Lf2* (Line 5).

Strict Sequencing Combined Fragment The semantics of *Strict Sequencing* (i.e., **strict** interaction operator) imposes the total order between adjacent operands. It contains a stronger version of the second rule introduced for *Weak Sequencing*, in particular, *OSs* on different lifelines from different operands have strict order of execution [Gro11b, p.483]. In other words, the first *OS* in a succeeding operand cannot be enabled until all the *OSs* on all the covered lifelines within the preceding operand have completed. Any covered lifeline needs to wait for other lifelines to enter the second or subsequent operand. For instance, sending *OS* of a message *msg3* within the second operand covered by a lifeline *Lf1* will not be executed until the last *OS* that is *rec2* within a first operand on a lifeline *Lf3* finishes its execution, as shown in Figure 4.3c (Line 15–19). Figure 4.3b shows LTL generation rules for the *Strict Sequencing*.



(A) Strict Sequencing Example

```

1. G (opd1 -> F (Lf1_msg1_snd1))
2. F (Lf3_msg2_rec2) -> ((!
  Lf3_msg2_rec2) U (Lf2_msg2_snd2
))
3. G ((Lf1_msg1_snd1 &
  Lf3_msg2_rec2 & opd2) -> F (
  Lf1_msg3_snd3))

```

(B) LTL Generation Rules

```

1 ASSIGN
2 ...--OSs within the first operand
3   init(«Lf1_msg1_snd1») := FALSE;
4   next(«Lf1_msg1_snd1») := case
5     «opd1» : TRUE;
6     «Lf1_msg1_snd1» : FALSE;
7   esac;
8   ...
9   init(«Lf3_msg2_rec2») := FALSE;
10  next(«Lf3_msg2_rec2») := case
11    «Lf2_msg2_snd2» : TRUE;
12    «Lf3_msg2_rec2» : FALSE;
13  esac;
14  --the first OS within the second operand
15  init(«Lf1_msg3_snd3») := FALSE;
16  next(«Lf1_msg3_snd3») := case
17    «Lf3_msg2_rec2» & «Lf1_msg1_snd1» & «opd2»
18    : TRUE;
19    «Lf1_msg2_snd2» : FALSE;
20  esac;

```

(C) SMV Generation Rules

FIGURE 4.3: Translation of Strict Sequencing Combined Fragment

Alternatives In the UML 2 specification [Gro11b, p.482], an *Alternative* combined fragment describes a branching operation in a sequence diagram. The **alt** operator of the combined fragment represents a choice of behaviour where at most one of the operands will be selected whose interaction constraint (guard condition) evaluates to **True** (i.e., an if-then-else statement). The **else** guard is the negation of the disjunction of all other constraints in the enclosing combined fragment. If none of the operands has a guard that evaluates to **True**, none of the operands will be executed and the remainder of the enclosing *InteractionFragment* will be performed.

Figure 4.4a shows an example of an *Alternative* combined fragment, whose **guard** is encoded as a boolean variable. If the **guard** of the first operand evaluates to **True**, the *OSs* enclosed within the first operand are executed, otherwise the whole operand is skipped. We introduce a temporary

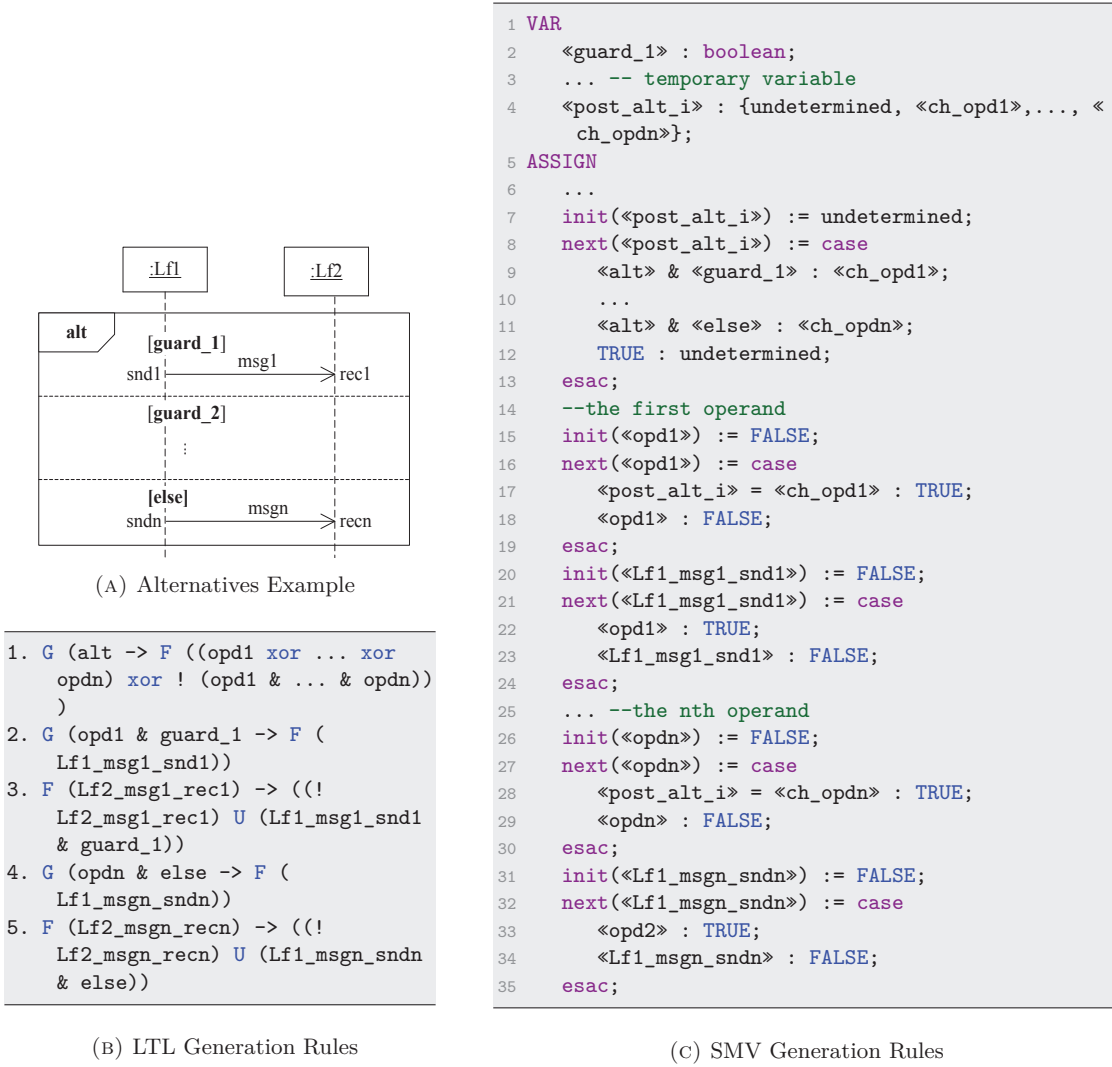
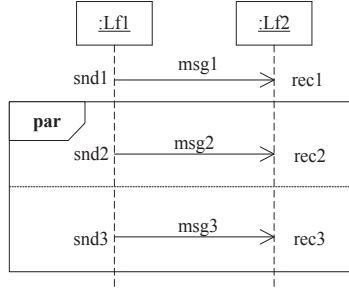


FIGURE 4.4: Translation of Alternative Combined Fragment

variable, namely, `post_alt_i` (i is an incrementally generated number) for exclusively choosing one of many alternative operands. The variable `post_alt_i` has an enumerated type including a normal state “undetermined” and the values corresponding to the operands (Line 4). The choice among alternatives is made using a “case/esac” construct as shown in Figure 4.4c (Line 7–13). The LTL generation rules for the **alt** operator enumerate all possible choices of executions; that is, only *OSs* of one of the operands, whose guard evaluates to **True**, will happen, as shown in Figure 4.4b.

Parallel Combined Fragment A *Parallel* combined fragment is denoted by an interaction operator **par** which defines potentially parallel merge execution of behaviours of the operands [Gro11b, p.483]. The *OSs* of different operands can be interleaved in any way as long as the ordering imposed



(A) Parallel Combined Fragment

```

1. G (par -> F (opd1 & opd2))
2. G (Lf1_msg1_snd1 & opd1 -> F (
  Lf1_msg2_snd2)) & G (
  Lf1_msg1_snd1 & opd2 -> F (
  Lf1_msg3_snd3))
3. F (Lf2_msg2_rec2) -> ((!
  Lf2_msg2_rec2) U (Lf2_msg1_rec1
  & Lf1_msg2_snd2)) & F (
  Lf2_msg3_rec3) -> ((!
  Lf2_msg3_rec3) U (Lf2_msg1_rec1
  & Lf1_msg3_snd3))

```

(B) LTL Generation Rules

```

1 ASSIGN
2 ...
3 init(«Lf1_msg1_snd1») := TRUE;
4 next(«Lf1_msg1_snd1») := case
5   «Lf1» : TRUE;
6   «Lf1_msg1_snd1» : FALSE;
7 esac;
8 ... --the first operand on Lf1
9 init(«opd1») := TRUE;
10 next(«opd1») := case
11   «par» : TRUE;
12   «opd1» : FALSE;
13 esac;
14 init(«Lf1_msg2_snd2») := FALSE;
15 next(«Lf1_msg2_snd2») := case
16   «opd1» & «Lf1_msg1_snd1» : TRUE;
17   «Lf1_msg2_snd2» : FALSE;
18 esac;
19 ... --the second operand on Lf1
20 init(«opd2») := TRUE;
21 next(«opd2») := case
22   «par» : TRUE;
23   «opd2» : FALSE;
24 esac;
25 init(«Lf1_msg3_snd3») := FALSE;
26 next(«Lf1_msg3_snd3») := case
27   «opd2» & «Lf1_msg1_snd1» : TRUE;
28   «Lf1_msg3_snd3» : FALSE;
29 esac;

```

(C) SMV Generation Rules

FIGURE 4.5: Translation of Parallel Combined Fragment

by each operand is preserved. In other words, *OSs* of messages within the same operand respect the order along a lifeline whilst *OSs* of messages on the same lifeline from different operands are ordered such that the first message occurrence of the operands has the same preceding *OS*. Figure 4.5c shows the translation of a *Parallel* combined fragment into SMV descriptions where a sending *OS* of a message *msg1* (*Lf1_msg1_snd1*) on a lifeline *Lf1* leads to the execution of sending *OSs* of messages in both operands (i.e., *Lf1_msg2_snd2* and *Lf1_msg3_snd3*). LTL generation rules for *Parallel* combined fragments for the covered lifelines *Lf1* and *Lf2* are presented in Figure 4.5b.

Loop Combined Fragment Describing *Loop* combined fragments in terms of state-based formal specifications like SMV is a very challenging task. The interaction operator **loop** defines that its sole operand will be repeated for at-least the minimum (*minint*) number of times and at-most maximum (*maxint*) number of times as long as the guard condition remains *True* [Gro11b, p.485]. If the loop has no bounds, this means that an indefinite loop (with *minint* = 0 and *maxint* = *infinite*) is executed, which can cause a state space explosion for model checking. However, it is

unrealistic for most loops that they really execute indefinitely, and therefore, we assume that loops will eventually stop.

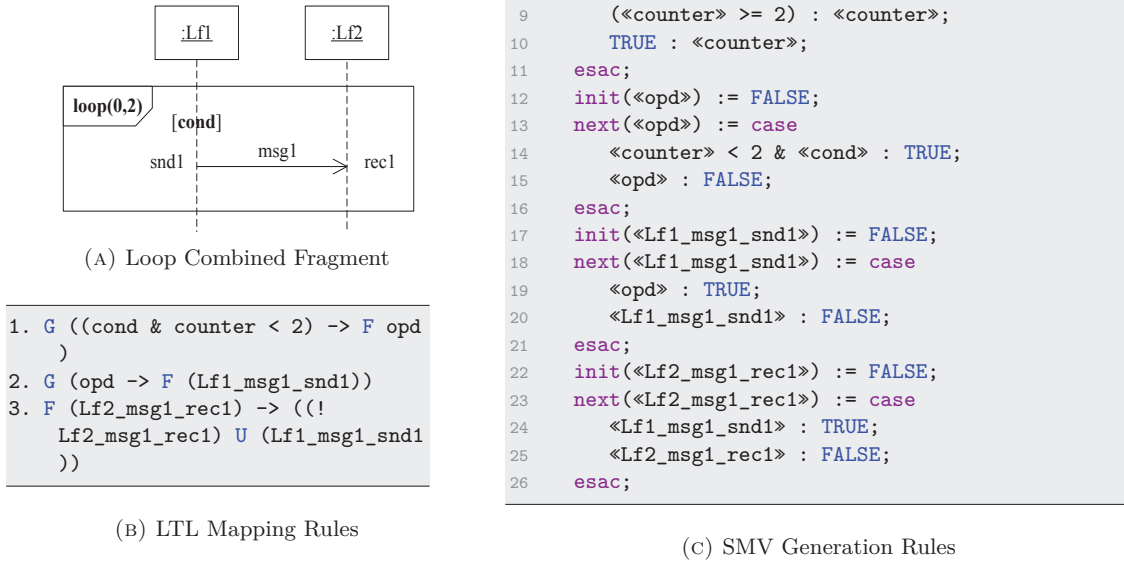


FIGURE 4.6: Translation of Loop Combined Fragment

Figure 4.6a shows an example of a *Loop* combined fragment having `minint` = 0 and `maxint` = 2. To deal with *Loop* combined fragments, we initialize the control variable, namely *counter*, which is 0 initially (i.e., equal to `minint`). After the end of the current iteration, the *counter* is increased by one at the beginning of the next iteration shown in Figure 4.6c (Line 8). Furthermore, the loop condition and *counter* are checked at the beginning of each iteration (Line 5–11). If the condition is evaluated to **False** or *counter* is greater or equals to `maxint`, a new iteration cannot start and execution of the loop will terminate. The *OSs* of all the messages within the operand among iterations execute sequentially along a lifeline. Figure 4.6b presents the LTL generation rules for the *Loop* fragment.

Option Combined Fragment The interaction operator **opt** designates that the combined fragment represents a branching operation (i.e., a simple if-then statement) in a sequence diagram where either the (sole) operand executes or nothing happens [Gro11b, p.483]. The *Option* combined fragment is semantically similar to an *Alternative* combined fragment, except that it has one operand with non-empty content and there is no **else** guard. If the guard condition evaluates to **True**, all

the *OSs* of messages within an enclosing *Option* combined fragment take place. Otherwise, the *Option* combined fragment is excluded, and its succeeding messages will be executed. Figure 4.7c illustrates the rules for mapping an *Option* combined fragment into SMV descriptions corresponding to the lifeline *Lf1*; whereas Figure 4.7b shows the LTL-based translation rules for the diagram in Figure 4.7a.

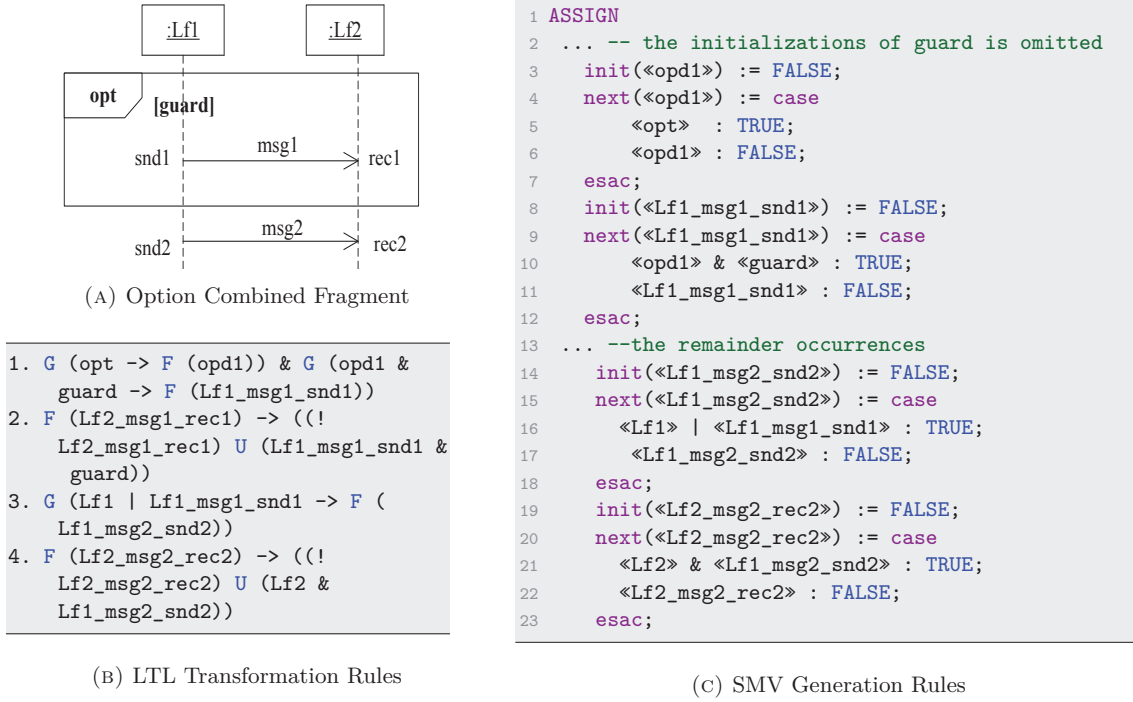


FIGURE 4.7: Translation of Option Combined Fragment

Break Combined Fragment The interaction operator **break** represents a breaking scenario in the sense that if the interaction constraint (guard condition) evaluates to **True** its sole operand executes [Gro11b, p.483]. When the constraint evaluates to **False**, the break operand will be ignored and the remainder of the enclosing *InteractionFragment* will be performed. Figure 4.8a shows an example of a *break* combined fragment in which **guard** is encoded as a boolean variable. If the **guard** of the break operand evaluates to **True**, the *OSs* enclosed within the break operand are executed, otherwise the whole operand is skipped. The constraint after the break operand is the negation of the break operand's constraint (i.e., **!guard is True**). Figure 4.8c and Figure 4.8b illustrate the rules for mapping a *Break* combined fragment into SMV descriptions and LTL formulas, respectively.

Ignore and Consider Combined Fragments The interaction operator **ignore** defines that there is a set of messages that needs to be ignored within the combined fragment [Gro11b, p.487]. Conversely, the interaction operator **consider** specifies a set of messages that are to be considered

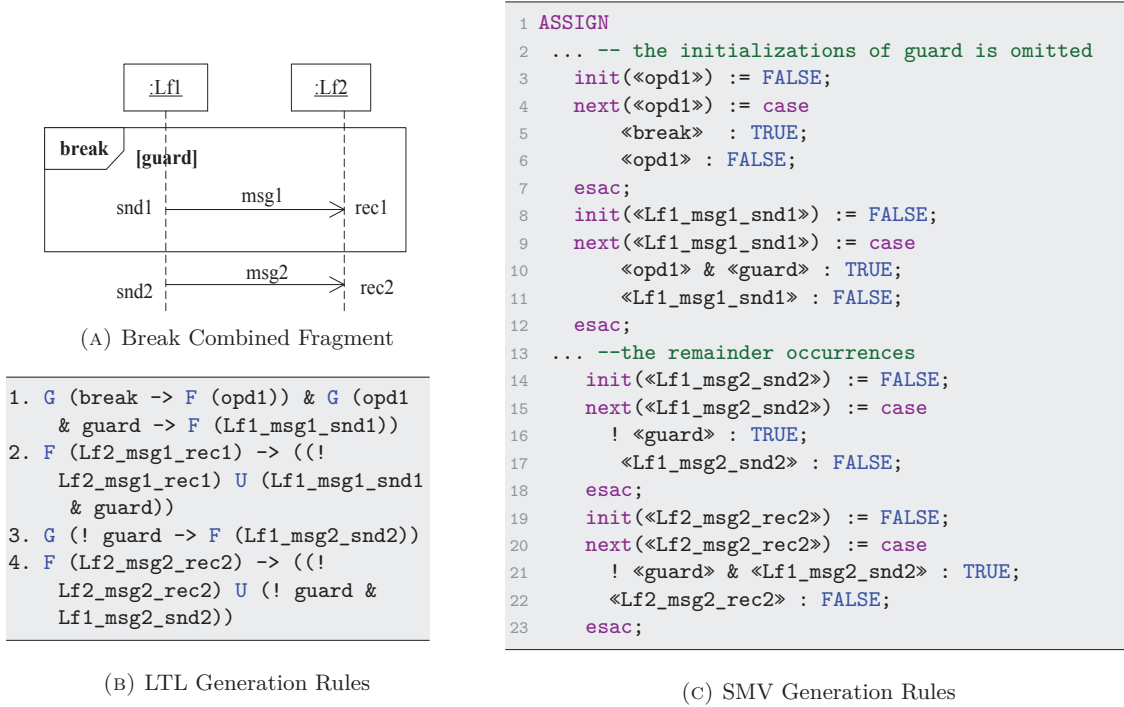


FIGURE 4.8: Translation of Break Combined Fragment

within the combined fragment; all other messages are ignored. Due to space limitations and similar technical details, the mapping rules for *Ignore* and *Consider* combined fragments into LTL and SMV descriptions are omitted.

4.2.2 Checking Containment between Sequence Diagrams via NuSMV

The main goal of this chapter is to assess whether the “execution” of the low-level model includes the “execution” prescribed in the high-level model (in the same order of executed elements). More specifically, containment checking for sequence diagrams aims to verify whether the elements and structures of a high-level sequence diagram (e.g., lifelines, sending and receiving events of messages and combined fragments) correspond to those of a refined and extended low-level sequence diagram. In this research, containment checking is achieved by utilizing the NuSMV model checker. NuSMV takes as inputs the generated SMV descriptions and LTL formulas, and exhaustively explore all executions of the SMV descriptions by traversing the complete state space to determine whether the temporal logic properties hold. In case the SMV descriptions satisfy the LTL formulas, this implies that the behaviour described in the high-level sequence diagram is contained in the low-level sequence diagram’s behaviour. Otherwise, the low-level sequence diagram deviates improperly from the high-level counterpart. In this case, NuSMV will generate a counterexample that consists of the execution traces of the SMV descriptions leading to the violation.

4.3 Evaluation

4.3.1 ATM System Scenario

This section presents a realistic scenario, namely, the *Automated Teller Machine* (ATM) to validate whether the proposed approach helps identifying containment inconsistencies in UML sequence diagrams. The high-level representation of the ATM system in terms of a UML sequence diagram is shown in Figure 4.9. For performing the containment checking first the high-level sequence diagram of the ATM system is automatically translated into LTL formulas. Afterwards, the low-level ATM system—a refined version of the high-level model (see Figure 4.10) is automatically converted into SMV descriptions using our translation tool. Finally, the containment checking is achieved by using the NuSMV model checker.

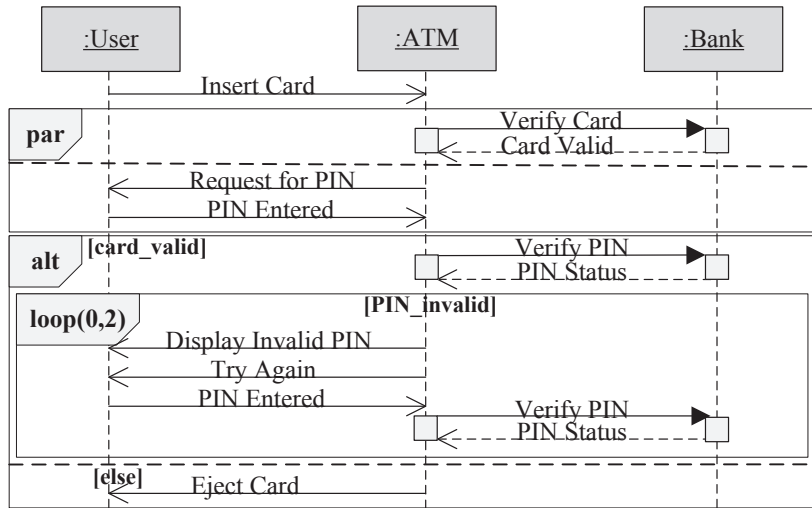


FIGURE 4.9: High-Level Sequence Diagram of ATM System

Listing 4.1 shows an excerpt of a violation trace generated by NuSMV including the list of satisfied and unsatisfied LTL formulas, i.e., a counterexample. By looking at the violation reported as a counterexample, we found that LTL formulas “ $G (User_DisplayInvalidPIN_Rec8 \rightarrow F ATM_TryAgain_Sn-d9)$ ” and “ $(F User_TryAgain_Rec9 \rightarrow (!User_TryAgain_Rec9 \vee (ATM_TryAgain_Snd9 \& User_DisplayInvalid-PIN_Rec8)))$ ” are violated. This means that this sequence of formal properties specified by the high-level ATM system is not contained in its low-level counterpart. After the generation of the counterexample, it is important to analyse the generated counterexample to find the actual source of the inconsistency and correct the responsible elements in the sequence diagram.

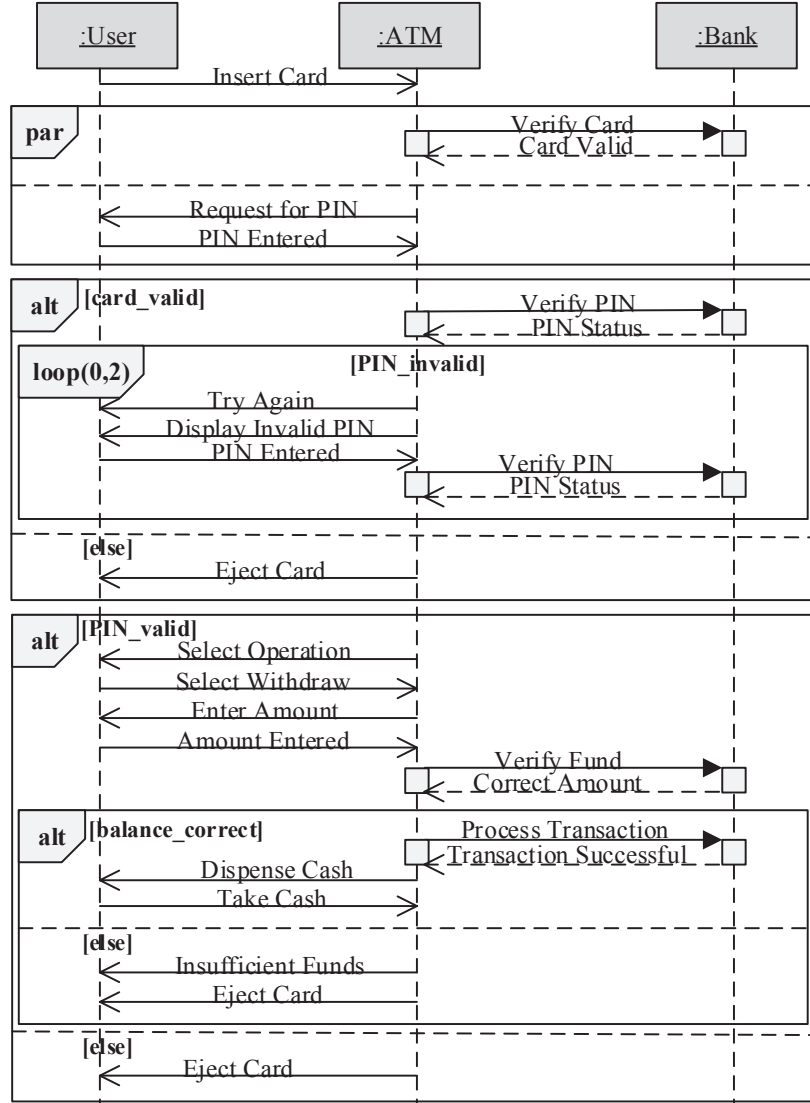


FIGURE 4.10: Low-Level Sequence Diagram of ATM System

4.3.2 Performance Evaluation

The main idea behind our performance evaluation is to validate whether the proposed approach provides considerable support for typical models used in real-world settings. The performance evaluation is conducted on a regular computer equipped with an 2.6 GHz i5 processor and 8GB of memory using NuSMV 2.5.4 running under Windows 8. In addition to the *Automated Teller Machine* (ATM) system presented in Section 4.3.1, we perform the evaluation on two other industrial scenarios with different sizes and complexity, in particular, *Order Processing* (OP) and *Itinerary Management* (IM), adapted from our previous projects in e-business domain [Tra+12]. The reported times include model loading, generating and verification of models measured in milliseconds.

```

$ NuSMV ATM.smv
...
-- specification G ((PIN_invalid & max < 2) -> F opd5) is true
-- specification G (opd5 -> F ATM_DisplayInvalidPIN_Snd8) is true
-- specification (F User_DisplayInvalidPIN_Rec8 -> (!User_DisplayInvalidPIN_Rec8 U
  ATM_DisplayInvalidPIN_Snd8)) is true
-- specification G (ATM_DisplayInvalidPIN_Snd8 -> F ATM_TryAgain_Snd9) is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
-- Loop starts here
-> State: 1.1 <-
User = FALSE
ATM = FALSE
Bank = FALSE
-- specification (F User_TryAgain_Rec9 -> (!User_TryAgain_Rec9 U (ATM_TryAgain_Snd9 &
  User_DisplayInvalidPIN_Rec8))) is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
-- Loop starts here
-> State: 2.1 <-
User = FALSE
...

```

LISTING 4.1: NuSMV Containment Checking Result

Table 4.1 shows the complexity of the input sequence diagrams (HL = high-level model, LL = low-level model) with respect to their elements including *OSs* of messages, interaction operators and operands, and covered lifelines.

Input size	Order processing		ATM system		Itinerary management	
	HL	LL	HL	LL	HL	LL
Occurrence Specifications	20	30	26	52	34	72
Interaction Operators	2	3	3	5	4	8
Interaction Operands	5	7	5	9	7	22
Lifelines	4	4	3	3	5	6
Total Elements	31	44	37	69	50	108
Model Loading (ms)	2.344±0.25	2.680±0.49	2.515±0.41	4.223±1.28	3.374±0.57	5.126±0.65
Translation Time (ms)	0.289±0.03	0.464±0.08	0.341±0.05	0.646±0.14	0.462±0.21	0.815±0.41

TABLE 4.1: Model Size and Translation Time of UML Sequence Diagrams

Table 4.2 shows the total execution time of three models, reachable states and violated formulas. The evaluation results indicate that the containment checking time spent by NuSMV for the ATM system is longer than for the Itinerary management and Order processing. This is the case because NuSMV found inconsistencies between the formal descriptions of the low-level model and LTL formulas of the high-level model and thus NuSMV needed to generate a counterexample for two violated LTL formulas. The evaluation results demonstrate that our approach efficiently translates sequence diagrams into formal descriptions and consistency constraints for supporting containment checking. In particular, all realistic scenarios are handled in a total time around a second which is quite reasonable for practical purposes. Our analysis and evaluation results based on the aforementioned use case scenarios show the feasibility of our approach for larger realistic scenarios.

Containment checking	Order processing	ATM system	Itinerary management
Verification Time (ms)	127.55±10.350	1011.25±22.320	768.57±53.049
Total Time (ms)	133.327	1018.975	774.511
Violated Formulas	0 out of 22	2 out of 30	1 out of 38
Reachable States	6 ($2^{2.58496}$)	4056 ($2^{11.9858}$)	32 (2^5)
Total States	2.2518e+015 (2^{51})	1.95846e+026($2^{87.3399}$)	5.10424e+038($2^{128.585}$)

TABLE 4.2: Performance Evaluation Results for UML Sequence Diagrams

4.4 Related Work

Some attempts have been made to give a formal definition for UML sequence diagrams to enable model checking. For instance, Alawneh et al. introduce a unified paradigm to verify and validate prominent diagrams, including sequence diagrams, using NuSMV [Ala+06]. The proposed semantics is not in full accordance with the standard semantics specified in UML 2 due to the lack of send and receive events. Moreover, the approach only supports alternatives and parallel combined fragments. Störrle [Stö04] proposes the semantics for sequence diagrams in terms of the set of valid and invalid traces for “plain InteractionFragments”, i.e., ones without combined fragments. Haugen et al. present the formal semantics of sequence diagram through an approach named STAIRS [Hau+05]. STAIRS focuses on the refinement for interactions, as a tuple $\langle action, sender, receiver, messagename \rangle$. However, this notation cannot describe the order of occurrences, where same message appears twice on the same lifelines. In contrast our work considers that each OS is unique within a sequence diagram. The aforementioned approaches have ignored the guard conditions, which compromises soundness of containment reasoning. Lima et al. provide a tool to translate sequence diagrams into PROMELA and verify using SPIN model checker [Lim+09]. Their translation does not support strict sequencing, consider and ignore combined fragments, as well as synchronous messages. Leue et al. translate the Message Sequence Charts (MSCs), especially branching and iteration of high-level MSC into PROMELA to verify MSCs using the XSPIN tool [LL96]. Jacobs and Simpson present the translation of sequence diagram into process algebra CSP to investigate whether a potential design meets its description using FDR refinement checker [JS15]. None of these semantics, in our opinion, achieves the simplicity and conceptual clarity of verifying the containment relationship for sequence diagrams. They are only useful for verification of sequence diagrams against safety properties such as deadlock freedom.

4.5 Summary

This chapter addressed the Research Question RQ2. It presents a model checking based approach to automatically detect containment inconsistencies between UML 2 sequence diagrams at different levels of abstraction in order to improve the system’s quality. To this end, we proposed a translation technique for automated generation of consistency constraints (i.e., LTL formulas) and SMV

descriptions from high-level and low-level sequence diagrams, respectively. The NuSMV model checker is employed for verifying containment relationship. In order to illustrate the applicability of the proposed approach, we realized realistic scenarios from various domains and also evaluated the performance our approach in these cases. The next chapter will investigate the containment checking problem between global and local choreography models.

This chapter presents a model checking based approach for containment checking between service choreography models at different levels of abstraction, in order to improve the quality and correctness of the service oriented systems. To date, however, previous studies have not considered the containment relationship between global and local choreography models. This problem addresses Research Questions RQ2 and RQ3. The chapter is based on a peer-reviewed conference paper in the proceedings of the 14th IEEE International Conference on Services Computing [Mur+17].

5.1 Introduction

In choreography-based service-oriented systems, a typical design and development scenario is that the global model (aka interaction model) is created by business analysts to agree on interaction scenarios from a global perspective. The global model will then be refined during detailed design phase into the public visible behaviour and hence forms a local choreography model (aka interconnection model) of each participant. The local choreography model shows an abstraction of orchestration internal actions/activities. The local choreography models often deviate from the global model due to the involvement of different stakeholders and independent evolutions. Hence, detecting model inconsistencies in early phases of the service development life cycle is crucial to eliminate as many anomalies as possible before service-oriented systems are actually implemented and deployed.

The literature discusses two possible ways to alleviate such problems: (i) the local models (i.e., representing implementation of individual services) can be generated from the global model (i.e. public view) [DW07; Zah+06]; (ii) the global and orchestration models can be created separately and then checked against each other [Bus+06; Fos+05; Yeu07]. The former strategy, although helpful to certain extent, did not prevent the overriding of manual changes that are made to complete the models. The later strategy focuses on the assessment of model inconsistencies that require formal descriptions and consistency constraints of the models. However, it is a challenging task to accurately and correctly express such formal descriptions and consistency constraints due to

the substantial amount of knowledge and specialized training required for the underlying formalisms and formal techniques.

In this chapter, we proposed a containment checking approach that verifies whether the behaviour (or interactions) described in the local choreography models collectively encompasses those specified in the global model. This improves the quality and correctness of the service oriented systems. To date, however, previous studies have not considered the containment relationship between global and local choreography models. Specifically, we have performed automated translation of global choreography model into temporal logic based consistency constraints (i.e. LTL) [Pnu77] and local choreography models into formal behaviour descriptions (i.e., SMV language); whereas the NuSMV model checker [Cim+99; Cla+96] is used for verification. This way, our approach helps to alleviate the burden of manually encoding consistency constraints, and therefore, increase productivity and avoid potential translation errors. Moreover, we investigated the performance of the proposed approach on use case scenarios of ATM machine, travel booking and order processing systems. This is done to ensure whether the stakeholders are supported to verify the containment relationship during their development tasks.

The rest of this chapter is organized as follows: Section 5.2 motivates the necessity of containment checking in service choreographies and explains a running example modelled using business process model and notation (BPMN) diagrams. Section 5.3 describes a novel approach for assessment of containment violations in service choreographies. Section 5.4 describes the performance evaluation of the proposed approach. Section 5.5 presents the related work. Finally, Section 5.6 summarises the chapter.

5.2 Motivation and Running Example

Service choreography is a set of interrelated service interactions at the high-level of abstraction, which represents message exchanges, interaction rules and agreements between web service partners. Figure 5.1 shows a global model of the *Travel Booking* application modelled using the BPMN 2.0 choreography notation [Gro11a]. The sender and receiver of a message are collapsed into one choreography activity; the unshaded and gray shaded bands represent the sender (initiating participant) and the receiver (non-initiating participant) of a message, respectively. The collaboration in travel booking choreography process involves six partners/participants, namely traveller, travel agency, acquirer, airline, hotel and rent a car agencies. More specifically, the process starts when a traveller sends an itinerary request to the travel agency which, in turn, contacts the acquirer for validation of a credit card. If the traveller has enough credit, then approval is sent to the travel agency that further triggers airline, hotel and rent a car partners. Accordingly, the purchase confirmation, reservation confirmation and vehicle assign are sent. Otherwise, the traveller is informed about unauthorized credit card.

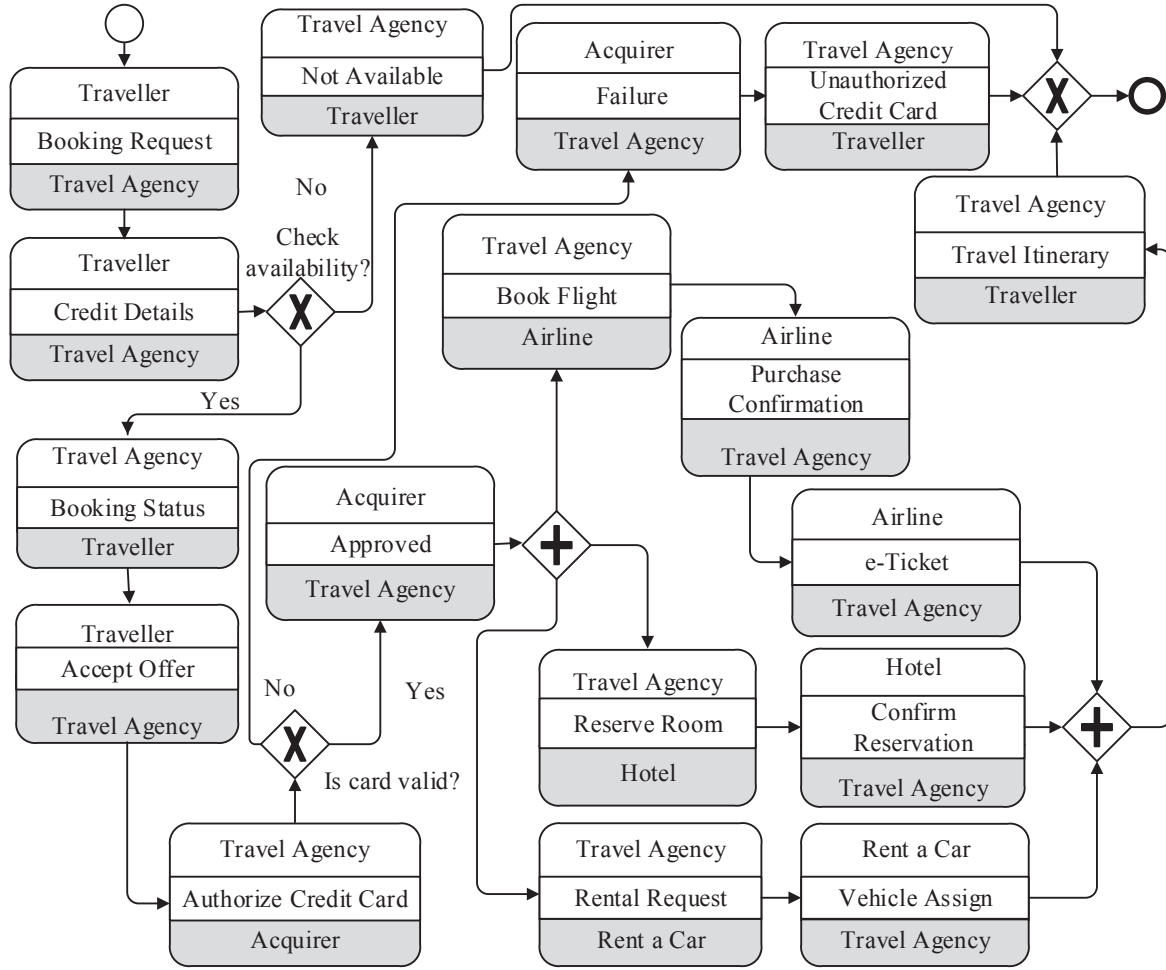


FIGURE 5.1: Travel Booking System: Global Choreography Model

The local choreography models of all participants (i.e., pools) involved in the travel booking system are shown in Figure 5.2. The message flow indicates the exchange of a message between two participants; whereas the sequence flow reflects the order in which activities are performed within a pool. It is crucial to sequence the choreography activities in such a way that the participants involved in the service choreography know when they are responsible for initiating the interactions. For instance, the **BookingRequest** and **CreditDetails** messages in the global model meant to be received in a sequential order (i.e., **CreditDetails** message follows **BookingRequest** message), as shown in Figure 5.1. However, the local model of the *travel agency* participant shown in Figure 5.2 specifies that the **CreditDetails** message precedes **BookingRequest** message. Furthermore, the sequential order of **PurchaseConfirmation** and **e-Ticket** messages is replaced by the parallel order using fork and join in the local choreography models, in particularly *travel agency* and *airline*. Please note that the containment checking not only deals with the missing participant or interaction but also misplacement of elements among the models. The undesired containment violations would cause severe problems; for example, improper identification of services and their

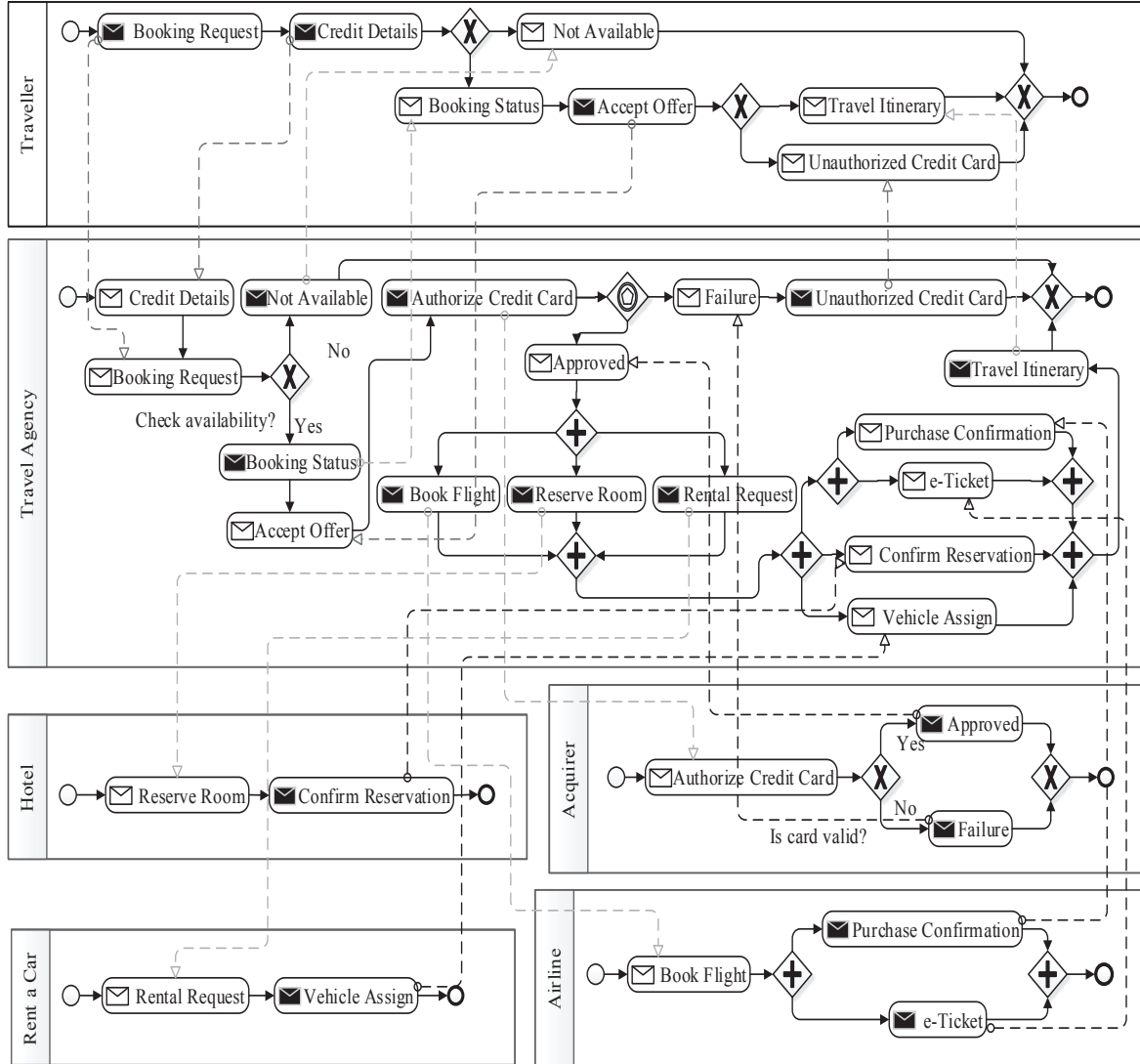


FIGURE 5.2: Travel Booking System: Local Choreography Models

corresponding service providers, and therefore affect the delivery of services. In order to eliminate such problems, containment checking shall be performed.

5.3 Approach

In this section, we address the problem of checking whether the message exchange behaviour (or interactions) described in the joint local choreography models encompasses those specified in the global model. Formally, the containment relationship between service choreographies is defined in such a way that $(GCM \mapsto LTL) \prec (LCMi = (LCM_1 \dots LCM_n) \mapsto SMV)$, where GCM denotes the global choreography model that is mapped to LTL formulas and $LCMi$ denotes a joint set of local choreography models that is mapped to SMV descriptions. Note that a symbolic model

checker named NuSMV [Cim+99] is used to verify the containment relationship between generated constraints and descriptions. If the generated formal descriptions of the local choreography models do not satisfy certain constraints generated from the global choreography model, then counterexamples are produced by the NuSMV model checker. An overview of our approach is shown in Figure 5.3. In the following sections, we describe each steps of our approach.

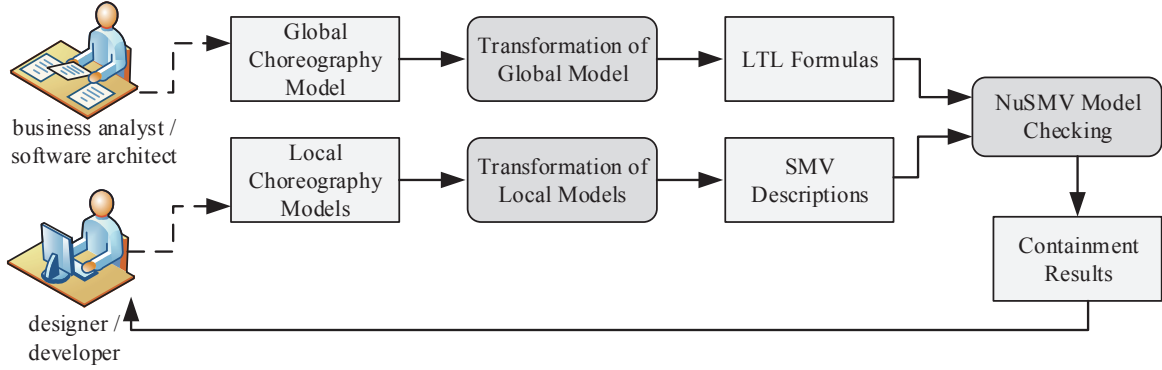


FIGURE 5.3: Overview of the Containment Checking Approach

5.3.1 Generating LTL Constraints from Global Choreography Model

The section is concerned with the automated transformation of global choreography model (*GCM*) into formal consistency constraints. From the containment checking perspective, the control flow relations between choreographic activities or interactions need to be represented in an appropriate formalism so that the execution order of interactions will become the consistency constraints for all local choreography models. In this context, a certain execution path is derived from the global model for describing the temporal relationships among the elements (e.g., choreography tasks, senders and receivers of the interactions, and guard conditions) using Linear Temporal Logic (LTL) [Pnu77].

This research focuses on both future and past temporal operators because LTL with past operators is exponentially more succinct than its future-only counterpart [LMS02]. The exclusive decision and merge gateways are implemented as $(a \wedge \neg b) \vee (\neg a \wedge b)$ instead of using logical “xor” operator. This is because, xor operator yields **true** not only when one of its operands is **true** but also when the odd numbers (i.e., $n \geq 3$) of the operands are **true** [PH08]. The generated formulas for different path constructs i.e., fork and join are enclosed by the **G** and **H** operators to express that all possible execution scenarios of the formulas are satisfied.

The construction of LTL formulas for containment checking is a highly knowledge intensive endeavour. In this context, the LTL-based transformation rules are defined to formally represent the constructs of BPMN 2.0 choreography model. Therefore, the input *GCM* is automatically translated into corresponding LTL formulas using the LTL-based transformation rules. In our

Algorithm 6 Translate Global Choreography Model *GCM* into LTL Formulas

```

1: procedure TRANSLATE(GCM)
2:    $Q \leftarrow \emptyset$  ▷  $Q$  is the queue of non-visited interactions
3:    $V \leftarrow \emptyset$  ▷  $V$  is the queue of visited interactions
4:    $Q \leftarrow Q \cup \text{get\_start\_events}(e)$ 
5:   for all  $i \in Q$  do
6:      $V \leftarrow V \cup \{i\}$ 
7:      $Q \leftarrow Q \setminus \{i\}$ 
8:     generate_ltl_code( $i$ )
9:      $I_{\text{succeeding\_interactions}} \leftarrow \text{get\_interaction}(i)$ 
10:    for all  $j \in I_{\text{succeeding\_interactions}}$  do
11:      if ( $j \notin V$ ) then
12:         $Q \leftarrow Q \cup \{j\}$ 
13:      end if
14:    end for
15:  end for
16:  extracts interaction information;
17:  binds input values and generates ltl formulas using the following templates:
18:  for all ( $1 \geq i \geq j$ ) do
19:    if  $i \in \text{AND} - \text{Split} \wedge j \in I_{\text{succeeding\_interactions\_snd}}$  then
20:      ' ' '
21:      (LTLSPEC  $G(\langle i \rangle \rightarrow F \langle j \rangle \ \& \ \langle j \rangle) \ \& \ H(\langle j \rangle \ \& \ \langle j \rangle \rightarrow O \langle i \rangle))$ 
22:      ' ' '
23:    end if
24:  end for
25:  for all ( $i \geq 0$ )  $\wedge V \leftarrow V \cup \{i\}$  do
26:    if  $i \in \text{AND} - \text{Join} \wedge i \in I_{\text{preceding\_interactions\_rec}}$  then
27:      ' ' '
28:      (LTLSPEC  $G(\langle i \rangle \ \& \ \langle i \rangle \rightarrow F \langle j \rangle) \ \& \ H(\langle j \rangle \rightarrow O \langle i \rangle \ \& \ \langle i \rangle))$ 
29:      ' ' '
30:    end if
31:  end for
32: end procedure

```

formalisation, we map a choreography interaction as a 3-tuple $\langle \text{participant_name}, \text{msg}, \text{snd}/\text{rec} \rangle$; where (i) *participant_name* indicates the corresponding participant; (ii) *msg* represents a message that describes communication contents between two participants; and (iii) *snd* and *rec* describe the sending and receiving actions of the corresponding message, respectively. However, the initiating participants of the choreography activities must have been involved in the previous activity (excluding first activity).

The Eclipse Xtend¹ framework is leveraged to translate the *GCM* into LTL formulas. Specifically, the breadth-first search algorithm is extended with three helper functions, namely `get_events(e)`, `get_interaction(i)` and `generate_ltl_code(i)`, as shown in Algorithm 6. The function `get_events(e)` returns a set of start events. A start event indicates the starting point of a

¹See <https://eclipse.org/xtend>

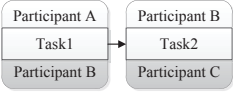
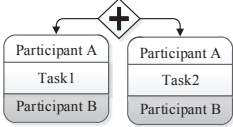
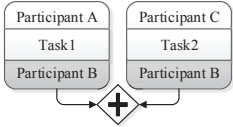
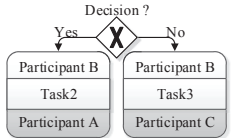
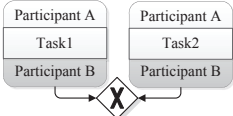
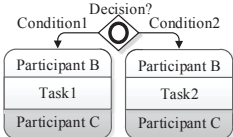
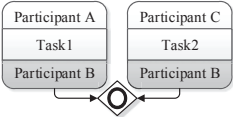
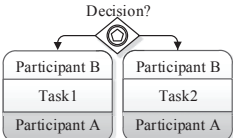
BPMN Choreography	Modelling Notation	LTl-Based Rules	Transformation
Sequence: (i) The sending action of a choreography task must exist before its receiving action. (ii) The initiator of a choreography task (excluding first activity) can not send a message to the receiver until it has received the prior message.		<pre> 1) G (ParticipantA_Task1_Snd -> F (ParticipantB_Task1_Rec)) & H (ParticipantB_Task1_Rec -> O (ParticipantA_Task1_Snd)) 2) F (ParticipantB_Task2_Snd) -> (! ParticipantB_Task2_Snd U (ParticipantB_Task1_Rec)) </pre>	
Parallel Fork: The execution of a Parallel Fork (AND-Split) leads to the parallel execution of subsequent choreography tasks. The initiators of all choreography tasks immediately following the Parallel Fork must be same as the common sender or receiver of choreography tasks preceding the gateway.		<pre> G (Fork -> F (ParticipantA_Task1_Snd & ParticipantA_Task2_Snd)) & H ((ParticipantA_Task1_Snd & ParticipantA_Task2_Snd) -> O Fork) </pre>	
Parallel Join: The concurrent execution of multiple interactions lead to the execution of a Parallel Join (AND-Join) gateway. However, all incoming branches have to be completed before the execution of a Parallel Join.		<pre> G ((ParticipantB_Task1_Rec & ParticipantB_Task2_Rec) -> F Join) & H ((Join) -> O (ParticipantB_Task1_Rec & ParticipantB_Task2_Rec)) </pre>	
Exclusive Decision: The execution of an Exclusive Decision (XOR-Split) is spawn in two or more branches, which branch is actually traversed depends on the evaluation of the guards on the outgoing flows.		<pre> (ExclusiveDecision -> F ((ParticipantB_Task2_Snd & ! ParticipantB_Task3_Snd) (! ParticipantB_Task2_Snd & ParticipantB_Task3_Snd))) </pre>	
Exclusive Merge: The execution of one of the choreography receiving action among a set of alternative receiving actions will lead to the execution of an Exclusive Merge (XOR-Join).		<pre> (G (ParticipantB_Task1_Rec & ! ParticipantB_Task2_Rec) (! ParticipantB_Task1_Rec & ParticipantB_Task2_Rec) -> F ExclusiveMerge) </pre>	
Inclusive Decision: An Inclusive Decision gateway (OR-Split) represents the execution of one or more alternative branches. The traversal of branches depend on the evaluation of the guard conditions. In particular, all sequence flows with a true evaluation will be traversed.		<pre> G ((InclusiveDecision & Condition1) -> F (ParticipantB_Task1_Snd)) G ((InclusiveDecision & Condition2) -> F (ParticipantB_Task2_Snd)) </pre>	
Inclusive Merge: The alternative but also parallel execution of two or more active interactions lead to the execution of the Inclusive Merge gateway (OR-Join).		<pre> (G (ParticipantB_Task1_Rec ParticipantB_Task2_Rec) -> F InclusiveMerge) </pre>	
Event-Based: The execution of an Event-based gateway is spawn in two or more branches, which branch is actually traversed depends on a specific Event that occur. Usually, the receipt of a message or timeout determines the path that will be taken rather than the evaluation of the guards.		<pre> G (Event-basedgateway & rec_msg1) -> F (ParticipantB_Task1_Snd & (! ParticipantB_Task2_Snd)) G (Event-basedgateway & rec_msg2) -> F ((! ParticipantB_Task1_Snd) & ParticipantB_Task2_Snd) </pre>	

TABLE 5.1: LTL-Based Transformation Rules for BPMN Global Choreography Model

choreography. Hence, it has no incoming sequence flow. The function `get_interaction(i)` extract all interactions i . The choreography tasks along with the senders and receivers of the messages, as well as the message exchange dependencies (i.e., sequence flows for performing two tasks in sequence or gateways for more complex behaviours) are extracted. An interaction j is called “succeeding interaction” of i if there is a control flow going from i to j . Thus, a set of succeeding choreography activities of i can be achieved by following all of its outgoing control flows.

The `generate_ltl_code(i)` function is responsible for generating LTL formulas for each construct of a *GCM*. The pair of triple apostrophes (`'''`) represents the string templates that are used for code generation based on Eclipse Xtend framework. However, a pair of guillemots (`«` and `»`) is used to represent the parametrised place-holders that will be bound to and substituted with the actual values extracted from the input model elements by the Xtend engine. The `generate_ltl_code(i)` function is not realized as a single function but rather a polymorphism of multiple functions. That is, depending on the type of the input interaction i , a particular function for generating LTL formulas for that interaction will be invoked. The LTL-based transformation rules for Parallel Fork and Join are presented in Algorithm 6. In particular, LTL formula for Parallel Join requires visited predecessors that are joined using the logical *AND* operator (`&`) and offered to Parallel Join. The Parallel Join cannot execute until all incoming flows have been received. In case of Parallel Fork the token offers are made at each outgoing flows (i.e., initiating participant of succeeding interaction). Table 5.1 summarises the constructs of choreography models along with their informal descriptions extracted from the BPMN 2.0 specification [Gro11a] and LTL-based transformation rules that constitutes the interactions between participants.

5.3.2 Generating SMV Descriptions from Local Choreography Models

This section concerns the generation of formal descriptions (i.e., SMV language) from the local choreography models (*LCMi*). To define the interactions within a BPMN 2.0 collaboration diagram, 2-tuple $\langle participant_name, task_snd/task_rec \rangle$ is used to represent the participant name and send task/receive task. The mapping of local choreography models (*LCMi*) into SMV descriptions is attained using an extended version of the breadth-first search, as shown in Algorithm 7. Similar to global choreography model (*GCM*), three helper functions are created, namely `get_events(el)`, `get_interaction(i)` and `generate_smv_code(i)`. The function `get_events(el)` returns a set of start events concerning the input *LCMi*. However, none start event shall be used as a graphical marker for starting the pool. The function `get_interaction(i)` extracts all interactions such as choreography tasks, control nodes and connecting edges. In particular, given a certain interaction i , the outgoing interactions (within pool) can be attained using the function `get_interaction(i)`. An interaction j is called “outgoing interaction” of i if there is a sequence flow going from i to j . In a similar way, communication between two participants (pools)

can be achieved. An interaction j is called “receiving interaction” of i if there is a message flow going from i to j . Thus, a set of receiving actions of choreography tasks and outgoing interactions of i can be achieved by following all of its message flows and outgoing sequence flows, respectively.

Algorithm 7 Generating SMV Descriptions from Local Choreography Models $LCMi$

```

1: procedure TRANSLATE( $LCMi$ )
2:    $Q \leftarrow \emptyset$  ▷  $Q$  is the queue of non-visited interactions
3:    $V \leftarrow \emptyset$  ▷  $V$  is the queue of visited interactions
4:    $Q \leftarrow Q \cup \text{get\_start\_events}(el)$ 
5:   for all  $i \in Q$  do
6:      $V \leftarrow V \cup \{i\}$ 
7:      $Q \leftarrow Q \setminus \{i\}$ 
8:     generate_smv_code( $i$ )
9:      $I_{outgoing\_interactions} \leftarrow \text{get\_interaction}(i)$ 
10:     $I_{receiving\_interactions} \leftarrow \text{get\_interaction}(i)$ 
11:    for all  $j \in I_{succeeding\_interactions} | j \in I_{receiving\_interactions}$  do
12:      if ( $j \notin V$ ) then
13:         $Q \leftarrow Q \cup \{j\}$ 
14:      end if
15:    end for
16:  end for
17:  extracts interaction information;
18:  binds input values and generates SMV descriptions using the following templates:
19:  '''
20:  VAR
21:    « $i$ » : boolean; ▷ State variable declaration
22:  ASSIGN
23:    init(« $i$ ») := «interaction-initial-state»
24:    next(« $i$ ») := case
25:      «incoming-condition(s)» : TRUE;
26:      « $i$ » : FALSE;
27:    esac;
28:  '''
29: end procedure

```

The function `generate_smv_code(i)` is responsible for generating the SMV description for each interaction within the $LCMi$. In particular, the aforementioned 2-tuple, control node or event will be represented by a boolean state variable in the section `VAR` and its corresponding state transitions will be described in the section `ASSIGN` by a combination of two functions given in NuSMV. The `init()` is used for assigning the initial state of a variable and `next()` is used for defining the transition to the next state. The function `next()` is often combined with the branching structure “`case/esac`” for selecting one of many possible choices. The state is initially set to `false`. However, if the incoming condition(s) are satisfied, it is changed to a `true` state (see Line 22 in Algorithm 7). The incoming condition(s) would be a guard expression and/or finishing of the preceding interaction(s). The interaction’s state shall be switched back to `false` after finishing the execution (see Line 23 in Algorithm 7). Note that the `generate_smv_code(i)` is not realised

```

1 VAR
2   «interaction» : boolean;
3 ASSIGN
4   init(«interaction») := FALSE;
5   next(«interaction») := case
6     «incoming_1» & «incoming_2» & ... & «incoming_n» : TRUE;
7     «interaction» : FALSE;
8   esac;

```

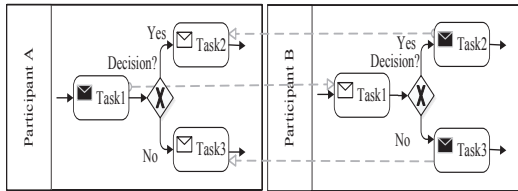
FIGURE 5.4: Generic Rules for Mapping BPMN Collaboration Constructs to SMV Descriptions

as a single function but rather a polymorphism of multiple functions. That is, depending on the type of the input interaction i , a particular function for generating SMV descriptions for that node type will be invoked. The subsequent sections discuss the rules for generating SMV descriptions for each node type that constitutes the individual function `generate_smv_code(i)`.

Task, Fork, Join, End Event and Start Event This section focuses on a set of elements that are triggered with respect to their incoming flows and are formalised rather similar in SMV. Figure 5.4 illustrates the translation of **Task**, **Fork**, **Join** and **End Event** into SMV descriptions based on the translation template shown in Algorithm 7. If an interaction has multiple incoming flows, the logical AND operator (“&”) is used to represent the implicit “and-join” guard for all tokens passing through the incoming flows. Note that **none Start Event** is a special event that denotes the starting point of a BPMN model. It does not have any incoming flows. Thus, each **none Start Event** is represented by a boolean state variable whose initial state would be assigned as **true**.

Branching The **Exclusive Decision**, **Inclusive Decision**, and **Event-based** gateways are the branching constructs in BPMN specification [Grol1a]. The execution of the **Exclusive Decision** will trigger one of the outgoing flows according to the corresponding guard conditions. The initiating participant of the messages that follow the gateway controls the decision. Figure 5.5 shows the rules for mapping an **Exclusive Decision** into SMV descriptions whose guard conditions is abstracted as boolean variables (Line 7–8). We introduce a temporary variable named `post_decision_i` in which i is an incrementally generated number for exclusively choosing one of many alternative sequence flows. The variable `post_decision_i` has an enumerated type which includes a normal state “undetermined” and the values corresponding to the sequence flows (Line 10). The choice among alternative sequence flows is made using a “case/esac” construct (Line 17–21). In case none of the guard conditions is **true**, the state transitions defined in Figure 5.5 will be stuck. This is precise, but undesired, behaviour. To avoid this stuck a “Default Condition” for one of the outgoing sequence flows can be used. The default condition is a complement of other guard conditions and will be chosen when all other conditions turn out to be false.

An **Inclusive Decision** represents the execution of any number of branches instead of one or all. The translation rules of **Inclusive Decision** are similar to **Exclusive Decision**; however, a



(A) Exclusive Decision

```

1  VAR
2    «ExclusiveDecision»: boolean;
3    «ParticipantB_Task1_Rec» : boolean;
4    ...
5    «ParticipantB_Task3_Snd» : boolean;
6    -- abstraction of boolean expressions
7    «guard_1» : boolean;
8    «guard_2» : boolean;
9    -- temporary variable
10   «post_decision_i» : {undetermined, «out_yes
    », «out_no»};
11  ASSIGN
12    init(«ExclusiveDecision») := FALSE;
13    next(«ExclusiveDecision») := case
14      «ExclusiveDecision» : FALSE;
15    esac;
16    ... -- the initializations of guards are
    omitted
17    init(«post_decision_i») := undetermined;
18    next(«post_decision_i») := case
19      «ExclusiveDecision» & «guard_1» : «
    out_yes»;
20      «ExclusiveDecision» & «guard_2» : «
    out_no»;
21      TRUE : undetermined;
22    esac;
23    init(«ParticipantB_Task2_Snd») := FALSE;
24    next(«ParticipantB_Task2_Snd») := case
25      «post_decision_i» = «guard_1» : TRUE;
26      «ParticipantB_Task2_Snd» : FALSE;
27    esac;
28    init(«ParticipantB_Task3_Snd») := FALSE;
29    next(«ParticipantB_Task3_Snd») := case
30      «post_decision_i» = «guard_2» : TRUE;
31      «ParticipantB_Task3_Snd» : FALSE;
32    esac;

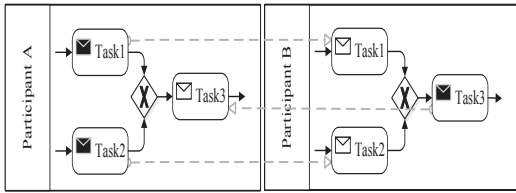
```

(B) SMV Generation Rules for Exclusive Decision

FIGURE 5.5: Translation of Exclusive Decision into SMV Descriptions

“true” evaluation of one guard condition does not exclude the evaluation of other guard conditions. All sequence flows with a “true” evaluation will be traversed by a token. The Event-based gateway represents an alternative branching point where the decision is made by two or more events; for instance, the choice for an outgoing sequence flow is made when an event will occur on the particular outgoing flow. When execution of process arrives at the particular point, the execution stops until either the message event or the timer event occurs. However, the occurrence of first event will immediately continue its outgoing sequence flow by disabling the other paths. In this case, a boolean variable *wait_event* is used to indicate that the outgoing flows can not be proceed until an event is occurred. The SMV descriptions of Inclusive Decision and Event-based gateways can be derived from the Exclusive Decision. The complete translation details cannot be presented due to space reasons and similar technical details.

Exclusive Merge and Inclusive Merge An Exclusive Merge brings together multiple alternative interactions and exclusively accepts one among them. If an Exclusive Merge has two incoming interactions, a straightforward naive encoding strategy is to use the xor operator “ $\text{ParticipantB_Task1_Rec} \text{ xor } \text{ParticipantB_Task2_Rec}$ ” to express the incoming guard condition of an Exclusive Merge. However, this strategy cannot be effectively generalised for Exclusive Merge that has more than two incoming sequence flows because the operator “xor” with n operands ($n \geq 3$) yields **true** not only when one of its operands is **true** but also when the odd numbers of the operands are **true** [PH08]. Exclusive Merge is therefore implemented by its equivalent but longer form “ $(\text{ParticipantB_Task1_Rec} \wedge \neg \text{ParticipantB_Task2_Rec}) \vee (\neg \text{ParticipantB_Task1_Rec} \wedge \text{ParticipantB_Task2_Rec})$ ”.



(A) Exclusive Merge

```

1  VAR
2    «ParticipantB_Task1_Rec» : boolean;
3    «ParticipantB_Task2_Rec» : boolean;
4    «ExclusiveMerge» : boolean;
5    -- temporary variable
6    «merge_flag_i» : {undetermined, «in_t1», «in_t2»}
7  ASSIGN
8    ... -- the initializations and transitions
          of tasks are omitted
9    init(«merge_flag_i») := undetermined;
10   next(«merge_flag_i») := case
11     («merge_flag_i» = undetermined) & («
        ParticipantB_Task1_Rec» | «
        ParticipantB_Task2_Rec»): {«in_t1», «in_t2
        »};
12     TRUE : undetermined;
13   esac;
14   init(«ExclusiveMerge») := FALSE;
15   next(«ExclusiveMerge») := case
16     «ParticipantB_Task1_Rec» & ! «
        ParticipantB_Task2_Rec» : TRUE;
17     ! «ParticipantB_Task1_Rec» & «
        ParticipantB_Task2_Rec» : TRUE;
18     «merge_flag_i» = «in_t1» | «merge_flag_i
        » = «in_t2» : TRUE;
19     «ExclusiveMerge» : FALSE;
20   esac;

```

(B) SMV Generation Rules for Exclusive Merge

FIGURE 5.6: Translation of Exclusive Merge into SMV Descriptions

In order to avoid name conflicts, a temporary variable `merge_flag_i` is introduced for each Exclusive Merge, where i represents an incrementally generated number to avoid name conflicts, as shown in Figure 5.6. This temporary variable has an enumerated type that comprises “undetermined” and “in_tx”. The former denotes the normal state; whereas the latter represent the state values that correspond to the incoming sequence flows ($x = 1, \dots, n$). As we see in Line 11, one branch will be non-deterministically and exclusively selected from the activated

branches. That is, in order to verify in case some k ($k \leq n$) incoming interactions are simultaneously activated, NuSMV will bind `merge_flag_i` to a certain value “`in_tx`” in the first place, to “`undetermined`” in the next transition, then to another value “`in_ty`” in the subsequent transition, and so forth. In combination with the branching construct “`case/esac`” (Line 14–20), we can see that the **Exclusive Merge** is activated if and only if either one of the incoming interactions is `true` or the variable `merge_flag_i` is assigned to a state value “`in_tx`”, where $x = 1, \dots, n$.

An **Inclusive Merge** has similar semantics to **Exclusive Merge**; however, it brings together not only multiple alternatives but also parallel interactions and accepts one or more among them. In the case of **Inclusive Merge**, we use the logical OR operator (“`|`”) to express the incoming guard condition instead of `xor` operator.

5.3.3 Containment Checking Using NuSMV Model Checker

This section is devoted to the identification of containment problems. The containment violations may occur due to a variety of reasons, such as (i) missing participant or interaction – a participant or interactions exist in the global model may not exist in the local choreography models; (ii) misplacement of elements – the local choreography model contains interactions with participant specified in the global choreography model but with different structure. Listing 5.1 shows an excerpt of a violation trace generated by NuSMV including the list of satisfied and unsatisfied LTL formulas, i.e., a counterexample. Four LTL formula are violated; this means that the sequence of formal properties specified by the global travel booking model is not contained in the local choreography counterparts.

```
$ NuSMV TravelBookingProcess.smv $
...
-- specification ( G (Traveler_CreditDetails_Snd -> F TravelAgency_CreditDetails_Rec) & H (
    TravelAgency_CreditDetails_Rec -> O Traveler_CreditDetails_Snd)) is true
-- specification ( F TravelAgency_CreditDetails_Rec -> (!TravelAgency_CreditDetails_Rec U (
    TravelAgency_BookingRequest_Rec & Traveler_CreditDetails_Snd))) is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
StartEvent1 = TRUE
...
-- specification ( G (TravelAgency_CreditCardNotAuthorized_Snd -> F Traveler_CreditCardNotAuthorized_Rec)
    & H (Traveler_CreditCardNotAuthorized_Rec -> O TravelAgency_CreditCardNotAuthorized_Snd))
is true
-- specification ( G (Airline_PurchaseConfirmation_Snd -> F TravelAgency_PurchaseConfirmation_Rec) & H (
    TravelAgency_PurchaseConfirmation_Rec -> O Airline_PurchaseConfirmation_Snd)) is false
...
-> State: 2.8 <-
-- specification ( G (Airline_e-Ticket_Snd -> F TravelAgency_e-Ticket_Rec) & H (TravelAgency_e-
    Ticket_Rec -> O Airline_e-Ticket_Snd)) is false
...
```



```

-> State: 3.8 <-
-- specification ( G ((Hotel_ReservationConfirmed_Snd & Fork5) -> F TravelAgency_ReservationConfirmed_Rec
) & H (TravelAgency_ReservationConfirmed_Rec -> O (Hotel_ReservationConfirmed_Snd & Fork5)))
is true
-- specification ( G ((RentaCar_VehicleAssign_Snd & Fork5) -> F TravelAgency_VehicleAssign_Rec) & H (
TravelAgency_VehicleAssign_Rec -> O (RentaCar_VehicleAssign_Snd & Fork5))) is true
-- specification ( F Airline_e-Ticket_Snd -> (!Airline_e-Ticket_Snd U (
TravelAgency_PurchaseConfirmation_Rec & Airline_PurchaseConfirmation_Snd))) is false
...

```

LISTING 5.1: NuSMV Containment Checking Result for Travel Booking Process

5.4 Evaluation

We implement containment checking approach and conduct a preliminary evaluation of its performance. The main idea is to validate whether the proposed approach performs reasonably for typical models used in industry on typical workstations used by developers. The workstation used for the performance evaluation is running under Windows 8 on a 2.6 GHz i5 processor with 8GB of memory using NuSMV 2.5.4. The evaluation is conducted through three behaviour models of different sizes and complexity that are taken from our previous industry projects. One of them is the *Travel Booking* (TB) mentioned in the previous section. The other two are *Automated Teller Machine* (ATM) and *Order Processing* (OP). ATM occupies an important position in the e-Banking domain. It allows the authenticated user to perform financial transactions such as view account balance, withdraw cash and deposit funds. The OP scenario allows the customer to order the company's products via the website. Table 5.2 shows the complexity of the input BPMN model (GCM = global choreography model, LCMi = local choreography models) with respect to their elements including tasks, gateways, and edges (sequence and message flows).

Input size	OP		TB		ATM	
	GCM	LCMi	GCM	LCMi	GCM	LCMi
Gateways	7	19	7	28	8	19
Interactions	16	32	17	34	22	44
Edges	25	72	27	83	32	89
Total Elements	48	123	51	145	62	152
Model Loading (ms)	2.351±0.59	3.387±0.65	3.215±0.39	4.984±0.27	3.278±0.14	5.620±0.87
Translation Time (ms)	0.315±0.19	0.514±0.98	0.541±0.05	0.784±0.44	0.596±0.22	0.861±0.07

TABLE 5.2: Model Size and Translation Time of Service Choreographies

Table 5.3 shows the total execution time of three models, reachable states and violated formulas. The evaluation results indicate that the containment checking time spent by NuSMV for the TB process is longer than the ATM and OP. This is because NuSMV found violations between the formal descriptions of the LCMi and LTL formulas of the GCM and thus NuSMV needed to generate a counterexample for violated LTL formula. The evaluation results demonstrate that our

approach efficiently translates service choreography models into formal descriptions and consistency constraints for supporting containment checking. In particular, all realistic scenarios are handled in a total time around a second which is quite reasonable for practical purposes. Our analysis and evaluation results based on the aforementioned use case scenarios show the feasibility of our approach for larger systems.

Containment checking	OP	TB	ATM
Verification Time (ms)	265.0±11.952	816.25±7.440	463.75±13.025
Total Time (ms)	271.567	825.744	474.607
Violated Formulas	0 out of 34	4 out of 38	0 out of 47
Reachable States	5 (2 ^{2.32193})	1.87027e+015 (2 ^{50.7322})	128 (2 ⁷)
Total States	2.15163e+023 (2 ^{77.5098})	7.3584e+039 (2 ^{132.435})	1.42772e+029 (2 ^{96.8496})

TABLE 5.3: Performance Evaluation Results for Service Choreographies

5.5 Related Work

In recent years, considerable attention has been paid to composition of web services in terms of choreography, interface behaviours, provider behaviours, or orchestration. Zaha et al. [Zah+06] present the algorithms for checking the local enforceability of global models and for generating local models (i.e., provider behaviour) from global models. Decker and Weske [DW07] introduce a Petri Net based formalism for specifying choreographies called interaction Petri Nets. They propose algorithms for deriving the behavioural interface and for verifying local enforceability. Busi et al. [Bus+06] examined the bi-simulation based correspondence between choreography and orchestration. In particular, they operationally relate choreographies to orchestration. Yu et al. [Yu+06] propose an approach for the specification of properties called PROPOLS and for verification of BPEL schemas. The approach first translated the BPEL schemas and PROPOLS into Finite State Automatas (FSAs), then compares these FSAs. However, the approach does not deal with the service choreographies.

Li et al. [LZP07] introduce two formal languages $Chor_L$ and $Orch_L$ for describing choreography and orchestration derived from WS-CDL (Web Services Choreography Description Language)² and BPEL, respectively. Based on the two languages, they give a definition of endpoint projection which is used for automatic generation of orchestrations. Lohmann and Wolf [LW09] show that the existing techniques for controllability problem of orchestrations can be reused for the verification of realizability of choreographies. In particular, they focus on the transformation of choreography specification into service orchestration.

Kwantes et al. [Kwa+15] present the translation of the BPMN collaboration diagram into an LTL formula to check conformance with local workflows as BPMN process diagrams using GROOVE³

²See <https://www.w3.org/TR/ws-cdl-10/>

³See <http://groove.cs.utwente.nl>

tool. However, the translation has been done manually. Poizat and Salaün [PS12] introduce the LOTOS NT (Language Of Temporal Ordering Specification New Technology) process algebra formalism for BPMN choreographies to validate the realizability between models using the CADP⁴ (Construction and Analysis of Distributed Processes) state space exploration tools. In particular, the interactions produced by the global choreography model and communicating peer processes are compared. Fu et al. [FBS04] present a formal specification, verification, and analysis tool for web service compositions based on Guarded Automata (GA). BPEL specifications are translated to GA and then mapped to PROMELA, the input language of the SPIN⁵ model checker. Solaiman et al. [SSMJ15] developed a BPMNverifier tool that automatically converts BPMN choreography models into PROMELA. However, the LTL properties are manually created or otherwise retrieved for the generated PROMELA models; they are stored in a repository. In [Yeu07], Yeung introduces an approach for checking the consistency of the global view (WS-CDL) with the abstract BPEL process. In this context, both WSBPEL and WS-CDL are translated into communicating sequential processes. Then, consistency checking is performed using the FDR2⁶ model checker. These approaches require a considerable amount of knowledge of temporal logics properties.

In the course of our earlier research, we have investigated the containment checking problem for activity diagrams [MTZ14] and sequence diagrams [MTZ16]. This research focuses on the containment relationship between global and local choreography models, which has not been considered in the literature. The proposed approach provides formalisation for automated transformation of global and local choreography models into consistency constraints and formal descriptions.

5.6 Summary

Motivated by the need to address the Research Questions RQ2 and RQ3, the central theme of this chapter focuses on the containment checking in service choreographies. This chapter has introduced a set of transformation rules to facilitate the automated transformation of global and local choreography models into LTL constraints and SMV descriptions, respectively. This provides efficient means for automated generation of consistency constraints and formal descriptions for large and complex choreography models. To illustrate the applicability of the proposed approach, we realized use case scenarios of ATM machine, travel booking and order processing systems; the performance evaluation is also carried out in particular cases. By analysing the evaluation results we found that our approach efficiently translates choreography models into formal specifications and works well for larger realistic scenarios.

⁴See <http://cadp.inria.fr/publications/Garavel-Lang-Mateescu-Serwe-13.html>

⁵See <http://spinroot.com>

⁶See <https://www.cs.ox.ac.uk/projects/fdr>

The results produced by the model checkers (i.e., counterexamples) are rather cryptic and verbose, and thus, tracking the entire evidence is difficult for architects/developers. In order to mitigate the need for strong background of formal techniques, the counterexample analysis mechanism is required. Therefore, the next chapter will investigate the interpretation of generated counterexample, in order to provide more informative and comprehensive feedbacks to understand and resolve the cause(s) of containment inconsistencies.

This chapter presents an approach for interpreting the containment checking results produced by model checking in order to facilitate better understanding and resolving of the violations revealed by containment checking. The approach aims to analyse the counterexamples to identify the actual causes of containment inconsistencies. Based on the analysis, it produced an appropriate set of guidelines to countermeasure the containment violations. This problem is related to the Research Question RQ4. The chapter is based on a peer-reviewed workshop paper in the proceedings of the 13th International Business Process Management Workshops [MTZ15], an article submitted to a peer-reviewed journal Science of Computer Programming, and two peer-reviewed conference papers, one in the proceedings of the 23rd Asia-Pacific Software Engineering Conference [MTZ16], and another in the 14th IEEE International Conference on Services Computing [Mur+17].

6.1 Introduction

Model checking is a powerful verification technique for detecting inconsistencies of software systems [CGP99]. In general, behaviour models are transformed into formal descriptions and verified against predefined properties/constraints. The model checker then exhaustively searches for property violations in formal descriptions of a model and produces counterexample(s) when these properties do not satisfy the formal descriptions. The ability to generate counterexamples in case of consistency violations are considered as one of the strengths of the model checking approach. Unfortunately, counterexamples produced by existing model checkers are rather cryptic and verbose. In particular, there are two major problems in analysing counterexamples. First, the developers and non-technical stakeholders who often have limited knowledge of the underlying formal techniques are confronted with cryptic and lengthy information (e.g., states numbers, input variables over tens of cycles and internal transitions, and so on) in the counterexample [KT11]. Second, because a counterexample is produced as a trace of states, it is challenging to trace back the causes of inconsistencies to the level of the original model in order to correct the flawed elements [DRS03]. As a result, the developers have to devote significant time and effort in order to identify the cause

of the violation, or they get confused about the relevance of a given trace in the overall explanation of the violation [JRS04; BNR03]. Besides that, in order to raise the practical applicability of model checking, there is a need for an automated approach to interpret the counterexamples with respect to containment checking and finding the causes of a violation.

For this purpose, one can choose to either intervene in the model checking process or analyse the model checking outcomes to achieve the necessary information. The former method is more direct and can be better optimised (due to close integration with the model checkers) but likely leads to tight dependence with a particular model checker or underlying model checking techniques, and therefore, will be less generalisable. Hence, in this work we opt to focus on analysing the outcomes of model checking. So far, some approaches have been proposed for counterexample analysis [BNR03; DRS03; KT11]. Out of these existing approaches, only a few are aiming at supporting counterexample analysis of behavioural models [KT11]. Most of these approaches focus on fault localization in source programs for safety and liveness properties or generation of proofs to aid the understanding of witnesses. As these approaches focus on model checking in a generic context, their analysis techniques can be applied in a wide range of application domains. However, this comes with a price: the analysis outcomes are rather abstract and far from helping in understanding the specific causes of a violation in a particular domain. Furthermore, these techniques have not considered to provide any annotations or visual supports for understanding the actual causes nor suggest any potential countermeasures.

In this chapter, we present an automated method for analysing the counterexamples in the context of containment checking and produce an appropriate set of guidelines to countermeasure the containment violations. In particular, we have constructed a counterexample analyser that automatically extracts the information from the counterexample trace file generated by containment checking using the NuSMV model checker [Cim+00]. Based on the extracted information along with formalisation rules for the containment relationship, our counterexample analyser identifies the cause(s) of the violation(s) and produces an appropriate set of guidelines to countermeasure the containment violations. Our approach allows the developers to focus on the immediate cause of an inconsistency without having to sort through irrelevant information. In order to make our approach more usable in practice, we devise visual supports that can highlight the involved elements in the process models. Furthermore, it provides annotations containing causes of inconsistencies and potential countermeasures shown in the input models. In the scope of this chapter, counterexample interpretation of containment checking results not only deals with the BPMN [Gro11a] process diagrams and collaboration diagrams, but also UML [Gro11b] activity diagrams and sequence diagrams.

The rest of this chapter is organized as follows: Section 6.2 describes the counterexample interpretation approach for BPMN process models in detail, while the three subsequent sections present the counterexample analysis for service choreographies (Section 6.3), activity diagrams (Section 6.4)

and sequence diagrams (Section 6.5). In Section 6.6, we review the related approaches regarding behavioural consistency checking and counterexamples interpretation. Finally, Section 6.7 summarises the chapter.

6.2 Interpretation of Containment Inconsistencies

In this research, containment checking is performed using the NuSMV model checker. NuSMV takes the LTL properties and the SMV descriptions, and exhaustively explores violations of a property by traversing the complete state space. If, however, the low-level process model deviates improperly from the high-level counterpart, NuSMV will generate a counterexample that consists of the linear (looping) paths of the SMV specification/description leading to the violation. The counterexample essentially shows the progress of the states from the beginning (i.e., all variables are initialized) until the point of violation along with the corresponding variables' values. Hence, it is time consuming and error-prone to locate relevant states because the developers may have to exhaustively walk through all of these execution traces. We note that counterexamples generated by the NuSMV model checker may contain different information depending on the selected model checking options, model encoding techniques, and the input LTL formulas. Thus, it is crucial to provide useful feedbacks to the developers and non-technical stakeholders that can reveal the causes of containment inconsistencies and suggest potential resolutions.

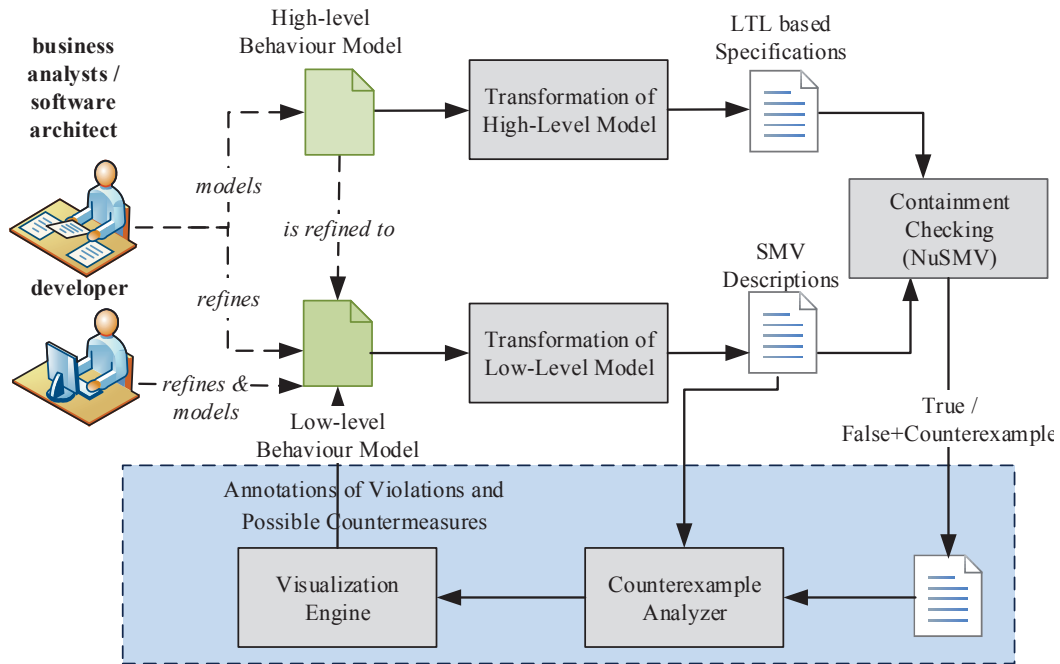


FIGURE 6.1: Overview of the Counterexample Analysis Approach

Containment inconsistencies may occur due to a variety of reasons, such as missing or misplaced elements in the low-level model, and so on. We propose a two-step approach for locating the causes of containment inconsistencies. In the first step, the counterexample analyser extracts the information from the output trace file generated by the NuSMV model checker and identifies the actual causes of the unsatisfied containment relationship and produces appropriate suggestions. In the second step, the information provided by the counterexample analyser will be annotated in the low-level process model along with concise descriptions of the violation's causes and potential countermeasures. Figure 6.1 shows an overview of model checking based containment checking research. However, the main focus of this chapter (i.e., counterexample analysis) is shown in the big blue box.

6.2.1 Counterexample Analyser for Locating Causes of Containment Inconsistencies

The counterexample analyser investigates the causes of an unsatisfied containment relationship with respect to the LTL-based transformation rules. Initially, the counterexample analyser reads the output trace file and parses the counterexamples that represent the unsatisfied LTL formulas. Afterwards, the counterexample analyser traverses the extracted information, LTL-based transformation rules and SMV description to find out why the elements and control flow structures of the high-level model are not matched by their corresponding low-level counterparts. Note that the elements that are described in the high-level model but missing or misplaced in the low-level model can be the causes of the containment inconsistencies. The counterexample analyser inspects and addresses all possible causes of an unsatisfied containment relationship defined by a specific LTL-based transformation rule and possible countermeasures.

In order to locate the causes of the inconsistency, the counterexample analyser first verifies whether all the elements (e.g., activities, events and/or gateways) that exist in the high-level model are also present in the low-level model by using function `match()` (see Algorithm 8). Note that `match()` essentially compares the nodes' names, types, and/or guard conditions. For this, the counterexample analyser locates the missing element cause (either one, multiple, or all elements could be missing) and suggests the countermeasure (i.e., insert missing element at a specific position in the model).

After that a number of rules related to unsatisfied LTL formulas for different possible kinds of elements in the BPMN model are checked. For this, the counterexample analyser matches the exact position of the corresponding elements in the high-level model related to unsatisfied LTL formulas with elements present in the low-level model. Specifically, the counterexample analyser reads the sequence (of elements of the low-level model) from the SMV description and identifies corresponding element (i.e., activity, a gateway and so on) causing the violation of the LTL formulas. The

Algorithm 8 Verifying Missing/Deleted Elements in the Low-Level Model

```

1: procedure FIND_MISSING_ELEMENTS( $HL, LL$ );
2:   extracts nodes information from output trace file, SMV descriptions and
3:   LTL-based transformation rules;
4:   takes two model elements
5:    $\triangleright$  where  $n \in HL(\text{high-level model}) \wedge m \in LL(\text{low-level model})$ 
6:   for all  $n \wedge m$  do
7:     match( $n, m$ )
8:     if  $\text{type}(n) = \text{type}(m) \wedge (n.\text{name} = m.\text{name})$  then
9:       returns true
10:    else
11:      returns false
12:    end if
13:    generate missing element cause and countermeasure (insert the missing element)
14:  end for
15: end procedure

```

preceding and succeeding elements of that element are matched with the elements of LTL formulas to locate the causes of inconsistencies (i.e., misplacement of elements) as shown in Algorithm 9.

Algorithm 9 Verify Misplacement of Elements in the Low-Level Model

```

1: procedure FIND_MISPLACEMENT_ELEMENTS( $HL, LL$ );
2:   extracts nodes information from output trace file, SMV descriptions and
3:   LTL-based transformation rules;
4:   takes elements ( $n$ ) of unsatisfied LTL formula and element ( $m$ ) from
5:   SMV descriptions
6:   for all unsatisfied( $n, m$ ) do
7:     match( $n, m_{\text{succeeding\_elements}}$ )
8:     if  $n = m_{\text{succeeding\_elements}}$  then
9:        $m \leftarrow m_{\text{succeeding\_elements}}$ 
10:      generate violation causes and countermeasures
11:    else match( $n, m_{\text{preceding\_elements}}$ )
12:       $m \leftarrow m_{\text{preceding\_elements}}$ 
13:      generate violation causes and countermeasures
14:    end if
15:  end for
16: end procedure

```

The descriptions of the possible causes for each LTL-based transformation rule and relevant countermeasures to resolve these causes are presented in Table 6.1. The right-hand side column contains the informal description of elements and second column contains the corresponding LTL-based transformation rules for formally representing these constructs. Let us consider the first one as an example: sequential order, which describes the relation that one element A2 eventually follows another element A1. As in the other rules in Table 6.1, violations occur due to a misplacement of elements. For instance, in the case of the sequence described by the LTL formula $(G (A1 \rightarrow F A2))$, a violation might happen because the element A2 (transitively) exists in the low-level model

as a preceding element of the element A1, but not as a succeeding element of A1. In this context, the counterexample analyser generates the relevant countermeasures to resolve the violation (in this case: “*swapping the occurrence of A2 to A1*” or “*add A2 after A1*”). Nevertheless, our approach provides promising results for composite controls, for instance, combinations of two or more control structures, like G (ParallelFork \rightarrow F (ExclusiveDecision & A1 & & An)).

Elements	LTl-Based Rules	Causes of Unsatisfied Rule	Possible Countermeasures
Sequence: A set of elements (transitively) executed in sequential order.	G (A1 \rightarrow F A2))	The sequential rule is violated, if element A2 does not eventually follow element A1 in the low-level model, but A2 exists as a preceding element of A1.	<ul style="list-style-type: none"> • Swap the occurrence of A2 and A1. • Add A2 after A1 in the low-level BPMN model.
Parallel Fork (AND-Split): The execution of a Fork leads to the parallel execution of subsequent activities or events (A1, A2...An). Please note that the activities or events may be executed one after the other or possibly may be executed in a real parallel enactment.	G (ParallelFork \rightarrow F (A1 & A2 & ... & An))	The Parallel Fork rule is unsatisfied, if a Fork gateway is not eventually followed by either one or all the activities/events (A1, A2,...An), or either one or all the activities/events exist as a preceding element of a Fork gateway.	<ul style="list-style-type: none"> • Put elements (A1 and/or A2 ...and/or An) after the Parallel Fork in the low-level model. • Elements (A1, A2...An) shall be triggered from the Parallel Fork.
Parallel Join (AND-Join): The execution of two or more parallel elements (A1, A2...An) leads to the execution of a Join gateway. The semantics is represented that all elements must complete before the execution of a Join gateway.	G (A1 & A2 & ... & An) \rightarrow F ParallelJoin)	The Parallel Join rule is violated, because either one or all the elements (A1, A2 ... An) exist as succeeding elements of a Join gateway, but are not followed by a Join gateway.	<ul style="list-style-type: none"> • Replace flawed elements(s) (“element’s name”) with the correct elements (“element’s name”), respectively. • Remove flawed element(s) (“element’s name”) from the low-level BPMN model. • Elements (A1, A2...An) shall be followed by a Parallel Join.
Exclusive Decision (XOR-Split): The execution of an Exclusive Decision eventually followed by the execution of at least one of the elements among the available set of elements based on condition expressions for each gate of the gateway.	G (ExclusiveDecision \rightarrow F (A1 xor A2)))	The Exclusive Decision rule is violated, if both of the branches return either FALSE or TRUE exclusively. It means that the Exclusive Decision gateway is not followed by elements (i.e., A1 and A2).	<ul style="list-style-type: none"> • Replace flawed elements (“element’s name”) with correct elements (“element’s name”) after the Exclusive Decision, respectively. • Remove flawed elements (“element’s name”) from the low-level BPMN model.
Exclusive Merge (XOR-Join): The execution of at least one element among a set of alternative elements will lead to the execution of an Exclusive Merge gateway.	(G (A1 xor A2) \rightarrow F ExclusiveMerge)	The Exclusive Merge rule is unsatisfied, because activity A1 and activity A2 are not followed by an Exclusive Merge, but one or both elements exist as the succeeding elements of an Exclusive Merge in the low-level model.	<ul style="list-style-type: none"> • Put the Exclusive Merge after A1 and A2 in the model. • Replace the flawed elements (“element’s name”) with correct elements (“element’s name”) before the Exclusive Merge, respectively.

TABLE 6.1: Tracking Back the Causes of Containment Violations and Relevant Countermeasures for BPMN Process Diagrams

6.2.2 Visual Support for Understanding and Resolving Inconsistencies

In this section, we explain how the containment checking results can be presented to the developers in a user friendlier manner in comparison to the counterexamples. The visual support aims at shows the developer the causes of containment inconsistencies that occur when the elements and structures (e.g., activities, events and/or gateways) of the high-level process model do not have corresponding parts in the low-level model and also provides relevant countermeasures to resolve the violations.

The visual support is based on the information provided by the counterexample analyser along with the input low-level process model. In particular, the element(s) that indicates the first element causing the violation of the LTL formula is highlighted in blue whilst the elements that are causes of containment violations are visualized in red, and the elements that satisfied the corresponding LTL formula appear in green. In order to improve the understandability of the counterexamples, we create annotations at the first element causing the violation to show the description of the cause(s) of the containment violation and relevant potential countermeasures to address the violation. Once the root cause of a containment violation is located, the cause is eliminated by updating the involving elements of the low-level process model. To differentiate more than one unsatisfied rule, shades of the particular colour are applied, for instance, the first unsatisfied rule is displayed in the original shade while others are gradually represented in lighter tones. The low-level process model displaying highlighted involving elements and annotation of the actual causes of the containment inconsistencies and relevant countermeasures is shown in Figure 6.3.

6.2.3 Use Case from Industrial Case Study

This section briefly discusses a use case from an industrial case study on a billing and provisioning system of a domain name registrar and hosting provider to illustrate the validity of our technique. The Billing Renewal process is taken from our previous industry projects [TZD10]. The Billing Renewal process comprises a wide variety of services, for instance, credit bureau services (cash clearing, credit card validation and payment activities, etc.), hosting services (web and email hosting, cloud hosting, provisioning, etc.), domain services (domain registration, private domain registration, transfer, website forwarding, etc.), and retail services (customer service and support, etc.). Figure 6.2 shows the high-level Billing Renewal process modelled as a BPMN 2.0 process diagram. The model is devised to capture essential control structures such as sequence and parallel execution, exclusive decision, and so on. Similarly, the low-level model of the Billing Renewal process containing detailed information is also modelled

Formal consistency constraints (i.e., LTL formulas) are automatically generated from the high-level BPMN model whilst the low-level BPMN model is transformed into SMV description. Next,

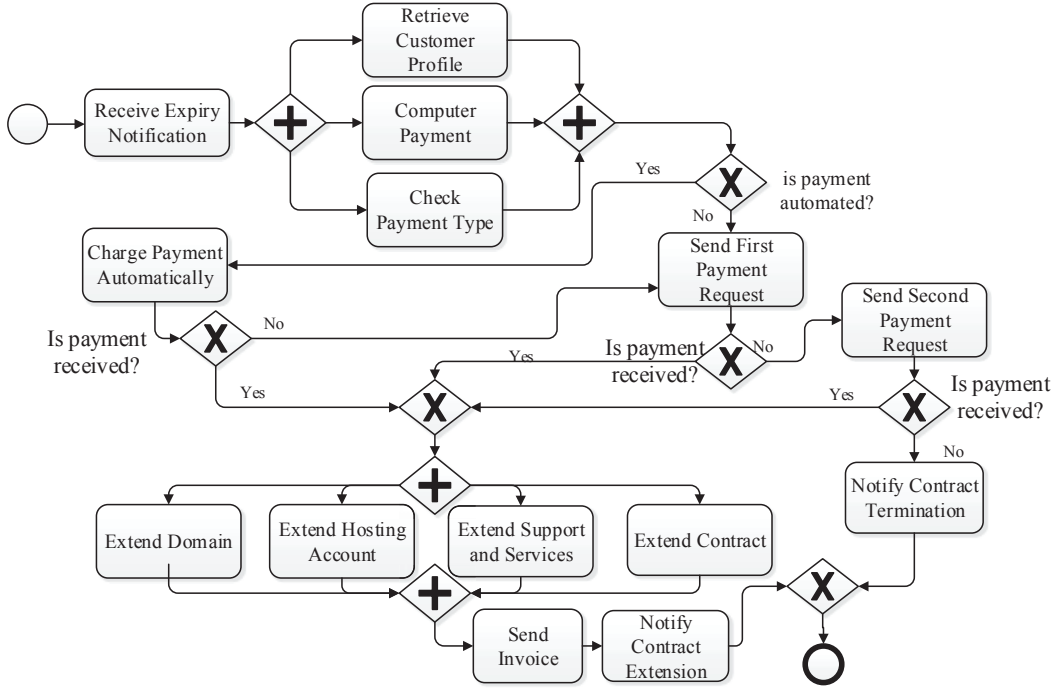


FIGURE 6.2: High-Level BPMN Model of the Billing Renewal Process

NuSMV verifies whether the formal SMV description is consistent with the generated LTL formulas. The NuSMV model checker generates a counterexample demonstrating a sequence of permissible state executions leading to a state in which the violation occurs in LTL formula. Finally, our approach for counterexample interpretation is applied to process the violation traces and visualize the involved elements in the low-level BPMN model along with annotations containing containment violation causes and suggestions. We opt to omit the verbose generated LTL formulas and SMV descriptions and focus more on the interpretation of the generated counterexample.

Listing 6.1 shows an excerpt of a violation trace generated by NuSMV including the list of satisfied and unsatisfied LTL formulas, i.e., a counterexample. Despite the size and execution traces of this counterexample, the exact cause of the inconsistency is unclear, for instance, “*is the containment violation caused by a missing element, or a misplacement of elements, or both of them?*”. It is time consuming and human labour intensive to locate the relevant states because the developers may have to exhaustively walk through all of these execution traces. The counterexample presents symptoms of the cause, but not the cause of the violation itself. Therefore, any manual refinement to the model could fail to resolve the deviation and may introduce other violations.

Figure 6.3 shows the low-level Billing Renewal process displaying the actual causes of the containment inconsistency and relevant countermeasures to address them. Using the visualizations of the violation, it is easy to see which elements of the low-level model involve in the containment inconsistency. In this case, the containment relationship is not satisfied due to the violation of a parallel fork rule and a sequential rule. The parallel fork rule $G \text{ (ParallelFork1} \rightarrow F \text{ ((ComputerPayment$

`& CheckPaymentType) & RetrieveCustomerProfile))` is unsatisfied because `ParallelFork1` is not followed by the parallel execution of the subsequent tasks (i.e., `ComputerPayment`, `CheckPaymentType` and `RetrieveCustomerProfile`) in the low-level model, which is the actual cause of the containment violation. This violation can be addressed by triggering `ComputerPayment` from `ParallelFork1` as shown in the attached comment to `ParallelFork1`. Similarly, the root cause of second violation is mainly because `SendInvoice` does not lead to `ParallelJoin4`. This might be a symptom of a misplacement of `SendInvoice` in the model as the primary cause that led to the containment inconsistency.

```
$ NuSMV BillingRenweal.smv
...
-- specification G (StartEvent -> F ReceiveExpiryNotification) is true
-- specification G (ReceiveExpiryNotification -> F ParallelFork1) is true
-- specification G (ParallelFork1 -> F ((ComputerPayment & CheckPaymentType) & RetrieveCustomerProfile))
   is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
StartEvent = TRUE
ReceiveExpiryNotification = FALSE
ParallelFork1 = FALSE
RetrieveCustomerProfile = FALSE
ComputerPayment = FALSE
.....
-- Loop starts here
-> State: 1.6 <-
CheckPaymentType = FALSE
ParallelJoin2 = TRUE
SendLastPaymentRequest = FALSE
ExclusiveDecision5 = TRUE
ParallelJoin3 = TRUE
.....
```

LISTING 6.1: NuSMV Containment Checking Result of the Billing Renewal Process

The use case illustrates that a rich and concise visualization of the inconsistency causes can allow for an easy identification of the elements that cause the violation and helps developers correct the process model accordingly. In the particular case, after following the suggested countermeasures, rerunning the containment checking process yielded no further violations. Without these supports, the developers would have to study and investigate the syntax and semantics of the trace file in order to determine the relationship between the execution traces and the process model, and then locate the corresponding inconsistency within the model, meaning that the complex matching between the variables and states in the counterexample and the elements of the models must be performed manually. This is especially cumbersome for those having limited knowledge of the underlying formal techniques.

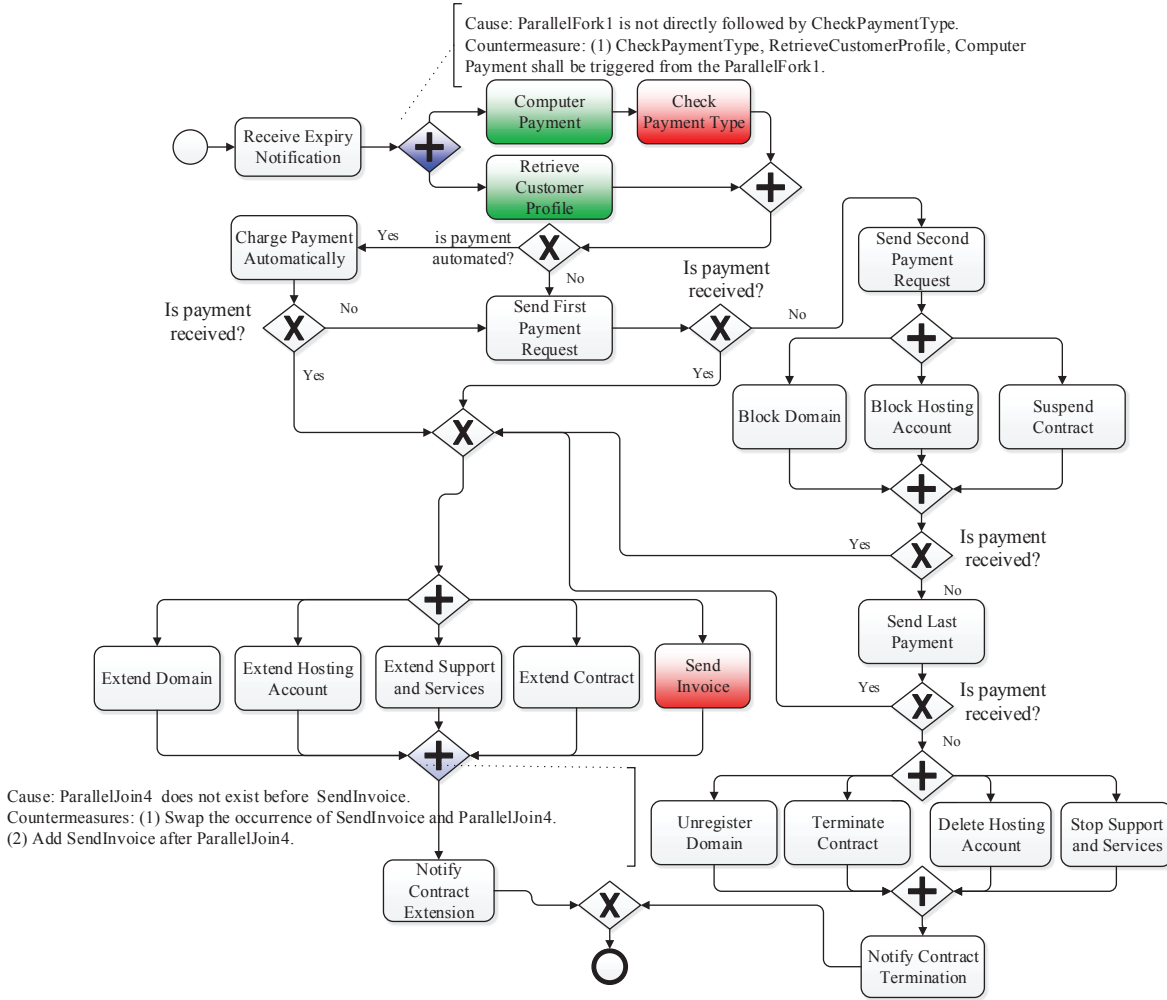


FIGURE 6.3: Visual Support for Understanding and Resolving Containment Violations in BPMN

6.3 Dealing with Containment Violations for Service Choreographies

This section is devoted to the identification of containment problems for service choreographies and their resolutions. The containment violations may occur due to a variety of reasons, such as (i) missing participant or interaction – a participant or interactions exist in the global choreography model may not exist in the local choreography models (collaboration diagram); (ii) misplacement of elements – the local choreography model contains interactions with participant specified in the global choreography model but with different structure. To alleviate containment checking problems, an analysis of the generated counterexample is supported as described in Section 6.2. The automated counterexample analysis not only detects the actual causes of the unsatisfied containment relationship but also provides appropriate guidelines to resolve the particular violations. Therefore, the output trace file (presented in Listing 5.1 of Chapter 5) is scrutinised and parsed to

determine the unsatisfied LTL formulas. The extracted formulas and SMV descriptions together with LTL-based transformation rules are traversed to find out why the elements of the global choreography model are not matched with their corresponding local choreography counterparts (collaboration diagram).

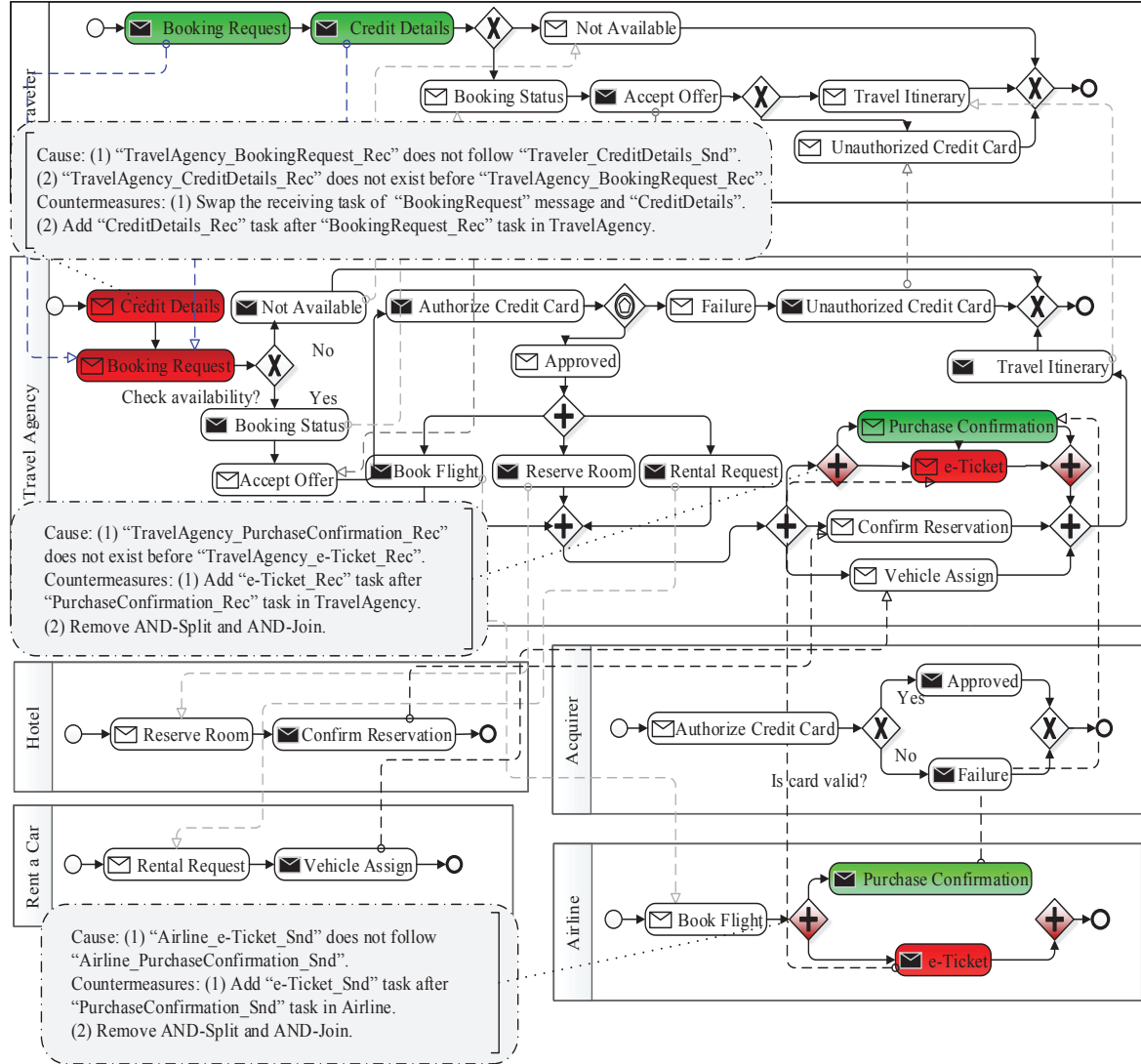


FIGURE 6.4: Local Choreography Models After Running Counterexample Analysis

The counterexample analysis results for Travel Booking application are presented in Figure 6.4. The gray boxes display the actual causes and potential countermeasures of the unsatisfied formulas. Furthermore, the elements responsible for causing the containment violation are highlighted in red; whereas the elements that satisfied the rule are highlighted in green. In this case, the containment relationship is not satisfied due to the violation of sequential rules. The receiving rule for the `RequestBooking` and `CreditDetails` messages is violated because *traveller* invokes the *travel agency* by sending `RequestBooking` message before `CreditDetails` message; however,

travel agency receives the **CreditDetails** message before the **RequestBooking** message. This implies that either receiving event of one or both tasks are misplaced. It can be resolved by putting the **CreditDetails** receive task after the **RequestBooking** receive task in the local choreography model of *travel agency*.

Similarly, the primary root causes of other violations are due to the execution of **PurchaseConfirmation** and **e-Ticket** messages in parallel order instead of sequential order. These violations can be resolved by deleting forks and joins, and putting the **e-Ticket** message after the **PurchaseConfirmation** message in the local choreography models of *travel agency* and *airline*. Once the causes are located, they are eliminated by updating the responsible elements of the choreography models and rerunning the containment checking process yielded no further violations.

6.4 Dealing with Containment Violations for Activity Diagrams

The NuSMV model checker generates a counterexample if the SMV descriptions do not satisfy the LTL properties. However, the counterexample provides only limited information for understanding the causes of inconsistencies but not how to fix the inconsistencies. Understanding the causes of containment violations is a prerequisite to resolve the inconsistencies in the models. As described in Section 6.2, the counterexample analysis approach consists of two steps. In the first step, the actual causes of the unsatisfied containment relationship are located based on the generated counterexamples and appropriate guidelines to resolve the particular violations are produced. In the second step, the concise descriptions of the isolation's causes and potential countermeasures, produced in the previous step are annotated in the low-level model. Please note that the containment inconsistencies may occur due to a variety of reasons, such as missing and misplacement of elements in the low-level model and so on.

We report typical possible causes of a containment inconsistency by presenting a set of reasons why the specific LTL formula is false and what strategies can be used to resolve it. For instance, consider a **Send Signal Action** (*a2*) in a low-level activity diagram (shown in Figure 6.5b), **Send Signal Action** (*a2*) is not preceded by Action *a1* as compared to the high-level diagram (shown in Figure 6.5a). In this case, the results produced by NuSMV indicate that $G (a1 \rightarrow Xa2)$ is false. This containment inconsistency occurs due to a violation of the containment relationship at **Send Signal Action**, because its semantics specifies that whenever Action (*a1*) is executed it will immediately enable the execution of **Send Signal Action** (*a2*). This can imply either one or both actions are misplaced. This containment inconsistency can be resolved by adding *a2* immediately after *a1* or by swapping the occurrence of *a1* and *a2* in the low-level model. If either *a1* or *a2* does not appear (i.e., deleted) in the low-level model, then the containment violation occurs due to missing Action (*a1*) or **Send Signal Action** (*a2*). In case of missing *a2* the violation can be

resolved by adding $a2$ immediately after $a1$, or in case of missing $a1$ the violation can be resolved by adding $a1$ before $a2$.

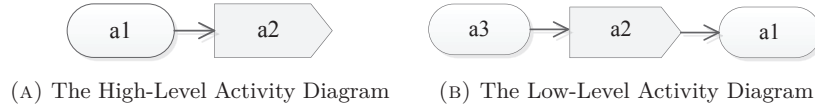


FIGURE 6.5: Send Signal Action Preceded by Different Action

6.4.1 Locating Causes of Containment Inconsistencies Due to Misplacement of Elements

Table 6.2 shows the violations occur due to a misplacement of elements for each LTL-based transformation rule and relevant countermeasures for changing the input models to satisfy the containment relationship. In Table 6.2, a state sequence is denoted by $k - 1, k, k + 1, \dots, kn$, where k is the current state of the element, $k - 1$ and $k + 1$ are preceding and succeeding states of the element respectively, and kn is the last state. The nodes $a1, \dots, an$ and $b1, \dots, bn$ will be parameterized with the actual node names.

Unsatisfied Rule	Cause of Violation Due to Misplacement	Possible Countermeasure(s)
$G (a1 \rightarrow F a2)$	Action $a1$ exists at k position but Action $a2$ does not exist in future $k + 1$ to kn (i.e., $a2$ does not eventually follow $a1$) in the low-level model. $a2$ appears before $a1$ in the low-level model.	<ul style="list-style-type: none"> • Swap the occurrence of $a2$ and $a1$. • Add $a2$ after $a1$ in the low-level model, where $(a1)$ and $(a2)$ are nodes that will be parameterized with the actual action names.
$G (\text{ForkNode} \rightarrow F (a1 \ \& \ \dots \ \& \ an)) \ \& \ G ((a1 \ \& \ \dots \ \& \ an) \rightarrow O \text{ForkNode})$	A Fork Node is activated but either one or all of the elements $(a1 \dots an)$ are not. One or all elements exist before a Fork Node or a Fork Node is not concurrently followed by either one or all of elements.	<ul style="list-style-type: none"> • Add elements $(a1 \dots an)$ in the low-level model after the particular Fork Node. • Elements $(a1 \dots an)$ must exist after the particular Fork Node.
$G ((a1 \ \& \ \dots \ \& \ an) \rightarrow F \text{JoinNode})$	The Join Node rule is violated when one or all the elements $(a1 \dots an)$ exist as succeeding elements of a Join Node, but are not followed by a Join Node.	<ul style="list-style-type: none"> • Elements $(a1 \dots an)$ shall be followed by a particular Join Node. • Remove element(s) $(a1 \dots an)$ from the low-level model. • Replace flawed elements(s) $(b1 \dots bn)$ with the corresponding correct elements $(a1 \dots an)$.

(continued on next page)

Unsatisfied Rule	Cause of Violation Due to Misplacement	Possible Countermeasure(s)
<pre>(DecisionNode -> F ((a1 & !a2) (!a1 & a2)))</pre>	<p>The Decision Node rule is violated, if both of the branches return either false or true exclusively. It means that the Decision Node is not followed by elements (i.e., <i>a1</i> and <i>a2</i>) in the low-level model.</p>	<ul style="list-style-type: none"> • Replace the flawed elements (e.g., <i>b1</i> and <i>b2</i>) with the corresponding correct elements <i>a1</i> and <i>a2</i> after the particular Decision Node. • Remove flawed elements (<i>b1</i> and <i>b2</i>) from the low-level model, respectively. • Put corresponding Decision Node before <i>a1</i> and <i>a2</i>.
<pre>G (((a1 & !a2) (!a1 & a2)) -> F MergeNode)</pre>	<p>The Merge Node rule is unsatisfied, because elements (i.e., <i>a1</i> and <i>a2</i>) are not followed by a Merge Node, but one or both elements exist as the succeeding elements of a Merge Node in the low-level model.</p>	<ul style="list-style-type: none"> • Put the particular Merge Node after <i>a1</i> and <i>a2</i> in the model. • Replace the flawed elements <i>b1</i> and/or <i>b2</i> with corresponding correct elements <i>a1</i> and <i>a2</i> before the particular Merge Node.
<pre>G (a1 -> X a2)</pre>	<p>The Send Signal Action rule is violated, if Send Signal Action <i>a2</i> either (transitively) exists in the low-level model as an eventually succeeding element of Action <i>a1</i> ($k + 2$) to (kn) or preceding element of the <i>a1</i> ($k - 1$) to ($k - n$), but not as next element of the <i>a1</i>.</p>	<ul style="list-style-type: none"> • Add <i>a2</i> immediately after <i>a1</i> in the low-level model. • Swap the occurrence of <i>a1</i> and <i>a2</i>.
<pre>1) G (a1 -> X a2) & G !(a1 & a2) 2) G (a1 -> G (a2 -> X a3))</pre>	<p>Accept Event Action <i>a2</i> (transitively) exists in the low-level model as a preceding element of Action <i>a1</i> but not after <i>a1</i>. In the second rule, the Action <i>a3</i> is not at next position $k + 1$ (i.e., <i>a3</i> exists as an eventually succeeding element of <i>a2</i> ($k + 2$) to (kn)).</p>	<ul style="list-style-type: none"> • Replace the flawed action <i>b1</i> with the corresponding correct action (<i>a1</i> or <i>a3</i>). • Put <i>a2</i> after <i>a1</i> in the low-level model. • Put <i>a3</i> immediately after <i>a2</i> in the low-level model, where <i>a1...a3</i> and <i>b1</i> are action nodes that will be parameterized with the actual action names.
<pre>G ((a1 & ExceptionType_i = ExceptType_1) -> X a2)</pre>	<p>The rule is violated, because <i>a2</i> (i.e., exception handler) does not exist after <i>a1</i> (i.e., protected node) and Exception Type.</p>	<ul style="list-style-type: none"> • Put correct handler body <i>a2</i> after corresponding protected node and Exception Type in the low-level model.
<pre>1) G (ac & isInterrupted -> X interrupting_edge) & G !(ac & interrupting_edge) 2) G (interrupting_edge -> X a2) 3) G (InitialNode & !(isInterrupted) -> F a1)</pre>	<p>The first rule is unsatisfied, because the element <i>interrupting_edge</i> is not at next position $k + 1$ of Accept Event Action <i>ac</i> (i.e., it exists as an eventually succeeding element of <i>ac</i> ($k + 2$) to (kn)). The second rule is violated, because either <i>interrupting_edge</i> and <i>a2</i> are misplaced, or <i>a1</i> is misplaced.</p>	<ul style="list-style-type: none"> • Put <i>interrupting_edge</i> immediately after Accept Event Action <i>ac</i>. • Remove <i>interrupting_edge</i> and <i>a2</i> from the low-level model. • Remove element <i>a1</i> from the low-level model. • Replace flawed elements <i>b1...b3</i> with corresponding correct elements <i>interrupting_edge</i>, <i>a1</i> and <i>a2</i>.

(continued on next page)

Unsatisfied Rule	Cause of Violation Due to Misplacement	Possible Countermeasure(s)
<pre> 1) (G (in_pn -> X a1) & H (a1 -> Y in_pn)) 2) (G (an -> X out_pn) & H (out_pn -> Y an)) </pre>	<p>The first rule is violated, because Action <i>a1</i> (transitively) exists in the low-level model as an eventually succeeding element of input Activity Parameter Node <i>in_pn</i>, but not as next element of the <i>in_pn</i>. The second rule is violated, because the output Activity Parameter Node <i>out_pn</i> (transitively) exists in the low-level model as a preceding element of Action <i>an</i>, but not as next element of <i>an</i>.</p>	<ul style="list-style-type: none"> • Put <i>a1</i> immediately after <i>in_pn</i> in the low-level model. • Put <i>out_pn</i> immediately after <i>an</i> in the low-level model.
<pre> (DecisionNode -> F a2) (DecisionNode & (max >= 0 & max < 2) -> F MergeNode) (DecisionNode & max = 2 -> F FlowFinal) & ! F(MergeNode) F(a1) </pre>	<p>The rule is unsatisfied, because the elements that consist of the loop (e.g., decision node, merge node, actions) are not present in the correct order, for instance, Action <i>a2</i> exists before a DecisionNode, but not followed by a DecisionNode. The MergeNode does not exist before Action <i>a1</i>.</p>	<ul style="list-style-type: none"> • Replace flawed elements (action, decision node, or merge node) with the corresponding correct elements, such as <i>a1</i> and <i>a2</i>. • Put action <i>a1</i> before a Decision Node. • Put action <i>a2</i> after a Decision Node. • Add corresponding MergeNode and <i>a1</i> in the low-level model. • Add the flow final in the low-level model.

TABLE 6.2: Tracking Back the Causes of Violation Due to Misplacement of Elements and Possible Countermeasures

6.4.2 Counterexample Analysis Results for Loan Approval System

This section describes the analysis of counterexample generated by NuSMV model checker for a loan approval system (presented in Listing 3.2 of Chapter 3), in order to find the actual source of the inconsistency and correct the responsible elements in the model. Here, the question arises why formulas regarding sequential order and fork node rules are violated. It is tedious and challenging for the developers to manually navigate and locate the relevant states because they have to exhaustively walk through all of these execution traces. In order to interpret the generated counterexample, we applied our counterexample analysis technique that visualises the involved elements in the low-level activity diagram along with annotations containing violation causes and suggestions. In this case, the sequential rules are violated because `CreateLoanFile` does not eventually follow the `SendStarterAccountContractKit` and does not precede the `SendAccountIdandWelcomeMessage`. However, it exists as the succeeding element of `RequestBankInformation` and preceding element of `ReceiveSupportingDocuments`. This might indicate that a misplacement of `CreateLoanFile` in the low-level model can be seen as a reason for the sequential violation. Therefore, this violation can be resolved by putting the `CreateLoanFile` after action `SendStarterAccountContractKit` and before `SendAccountIdandWelcomeMessage` in the low-level loan approval model. In addition,

involved elements are highlighted to complement the textual descriptions by easing the understanding, and not overloading the developer with text. For instance, `CreateLoanFile` causing the violation is highlighted in red colour, whereas `SendAccountIdandWelcomeMessage` action of the formula is highlighted in green colour. As we can see in Figure 6.6, the first two boxes display the actual causes and potential countermeasures of the unsatisfied sequential formulas.

In a similar way, the fork node rule $G(\text{ForkNode2} \rightarrow F(\text{NotifyBank} \ \& \ \text{NotifyCustomer} \ \& \ \text{NotifyLoanOfficer})) \ \& \ G((\text{NotifyBank} \ \& \ \text{NotifyCustomer} \ \& \ \text{NotifyLoanOfficer}) \rightarrow O \text{ ForkNode2})$ is violated because actions `(NotifyBank, NotifyCustomer and NotifyLoanOfficer)` do not follow the `ForkNode2` concurrently, in particular, `NotifyLoanOfficer` action exists before the `ForkNode2`. This violation can be addressed by adding `NotifyLoanOfficer` after the `ForkNode2` as shown in Figure 6.6, insert by in the third box. The element responsible for causing the containment inconsistency is highlighted in red, whereas the elements that satisfied the rule are highlighted in green. Once the causes are located, causes are eliminated by updating the responsible elements of the low-level activity diagram and rerunning the containment checking process yielded no further violations.

6.5 Dealing with Containment Violations for Sequence Diagrams

After the generation of the counterexample, it is important to analyse the generated counterexample to find the actual source of the inconsistency and correct the responsible elements in the sequence diagram. In order to interpret the generated counterexample, we applied our counterexample analysis technique on the output trace file (presented in Listing 4.1 of Chapter 4) that creates annotation at the first element causing the inconsistency to show the description of inconsistency causes and suggestions. In this case, the sending and receiving *OSs* rules for the `TryAgain` message are violated because the `TryAgain` message is sent and received prior to the receiving *OS* of the `DisplayInvalidPIN` message in the low-level model. These violations can be resolved by putting the `TryAgain` message after the `DisplayInvalidPIN` message in the low-level ATM system. In Figure 6.7, the blue boxes show the actual causes and potential countermeasures of unsatisfied formulas. Once the causes are located, causes are eliminated by updating the responsible elements of the low-level sequence diagram.

6.6 Related Work

The work presented in this chapter relates to the two main research areas: behaviour model consistency checking and analysis of the model checking results (i.e., counterexamples) for identifying the causes of inconsistencies.

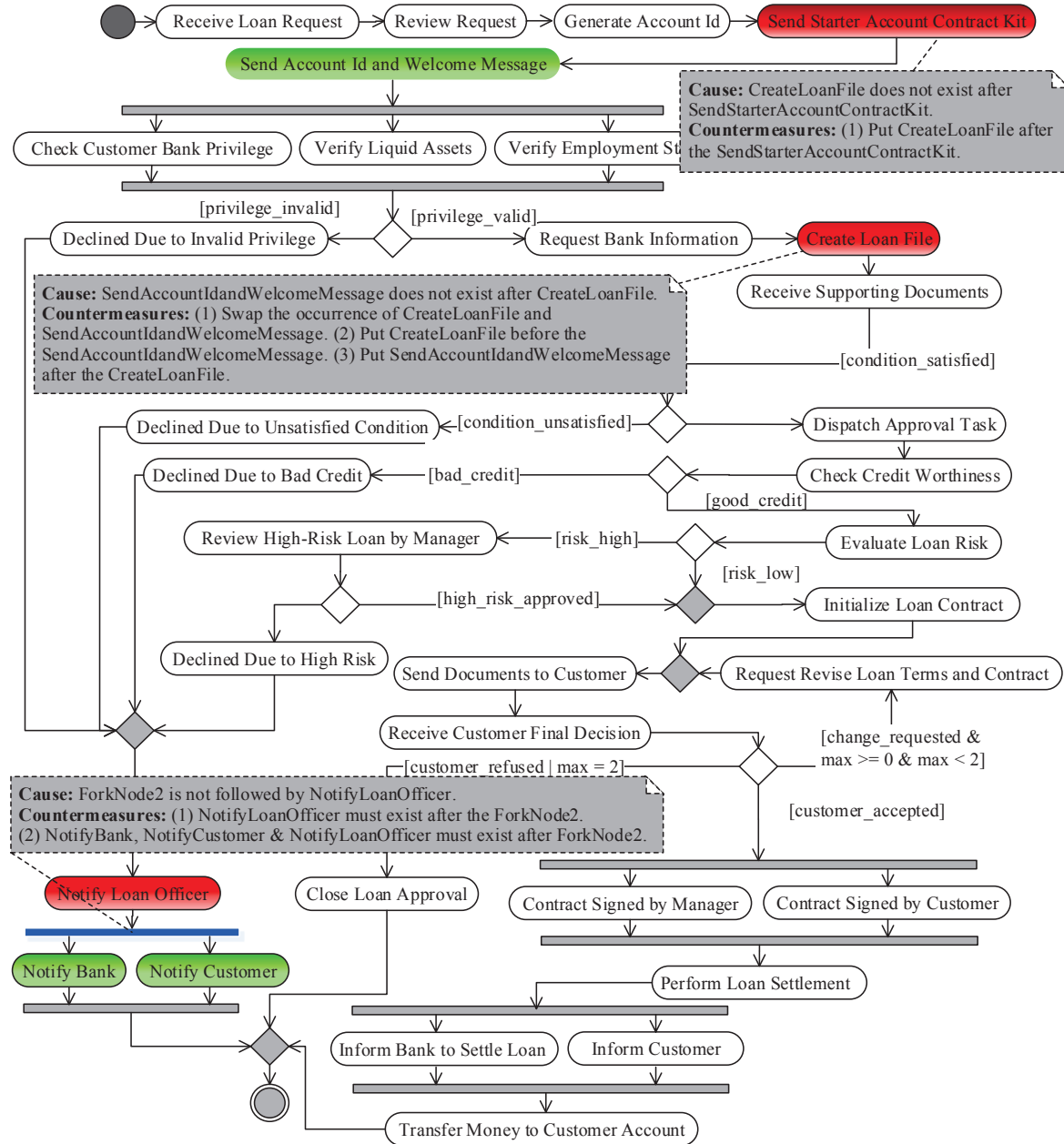


FIGURE 6.6: Low-Level Activity Diagram of Loan Approval System After Running Counterexample Analysis

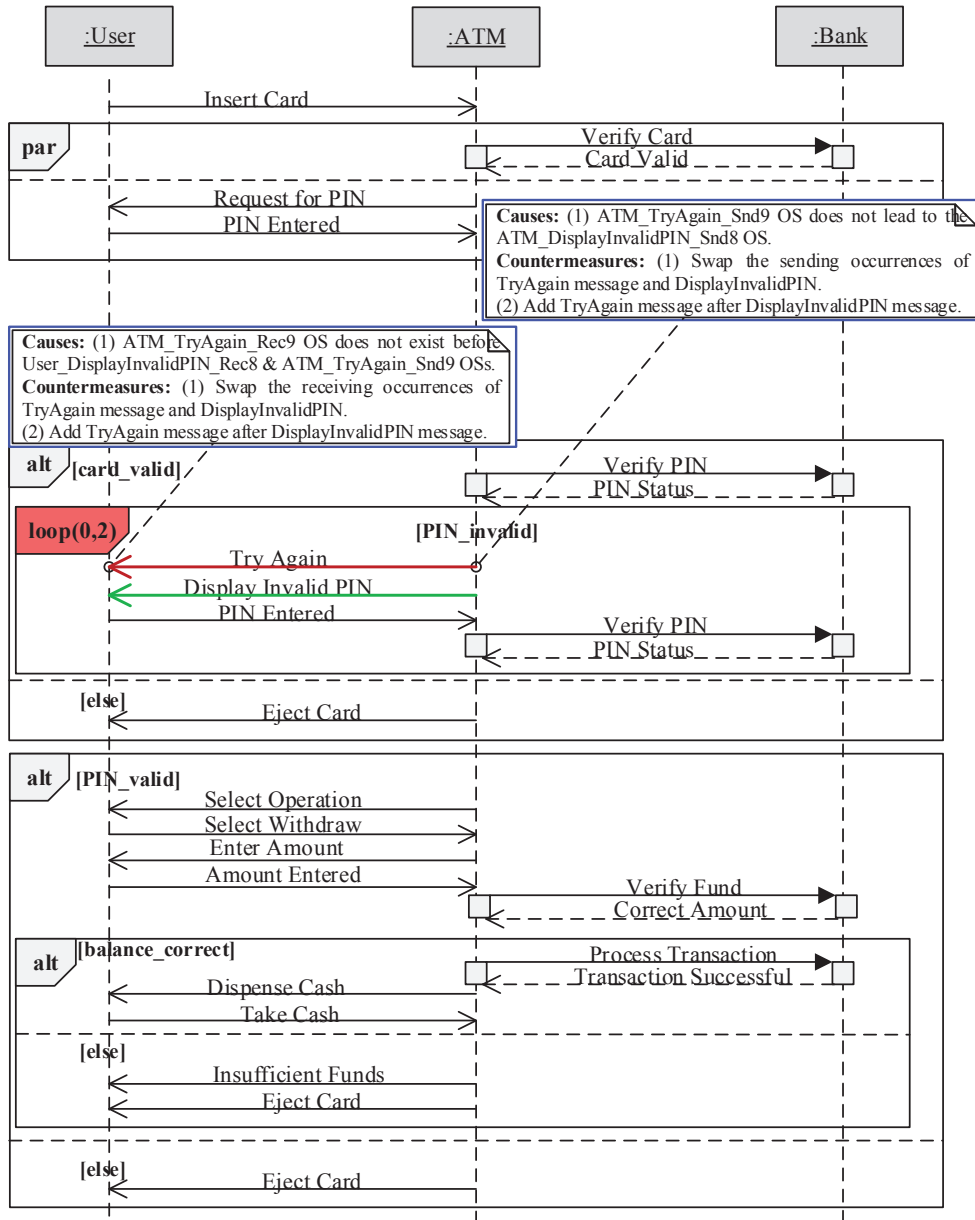


FIGURE 6.7: Low-Level ATM System After Running Counterexample Analysis

6.6.1 Behaviour Model Consistency Checking

In the literature, many approaches tackle different types of models and/or model checking techniques [LMT09]. However, very few of these studies focus on the consistency of behaviour models; for instance, van der Straeten et al. [Str+03] present an approach for checking the consistency of different UML models by using description logic. This approach considers model-instance, instance-instance, and model-model conflicts, instead of containment checking. Van der Aalst presents a theoretical framework for defining the semantics of behaviour inheritance [Aal02]. In this work, four

different inheritance rules, based on hiding and blocking principles, are defined for UML activity diagram, state-chart and sequence diagram. Similar ideas have been presented in [SS00]. In [EHK01] a general methodology is presented to deal with consistency problem of state-chart inheritance, which involves state-charts as well as the corresponding class diagrams. Communicating Sequential Processes (CSP) is used as a mathematical model for describing the consistency requirements and the FDR tool is used for checking purposes. Weidlich et al. consider the compatibility between referenced process models and the corresponding implementation based on the notion of behaviour inheritance [WDW12]. Awad et al. introduce an approach for automated compliance checking of BPMN process models using BPMN-Q queries [ADW08]. They adopted the reduction approach to verify the correctness of process models, instead of performing the detailed analysis using model checker. Unlike our approach, the aforementioned techniques do not aim at providing the analysis of the violation results for identifying the causes of inconsistencies and a set of countermeasures to resolve inconsistencies. Thus, these approaches are very useful for finding similar or alternative behavioural descriptions but not applicable for verifying the containment relationship.

6.6.2 Generating and Analysing Counterexamples

The problem of generating and analysing model-checking counterexamples are classified as follows: generating the counterexample efficiently, automatically analysing the counterexample to extract the exact cause of violations, and creating a visualization framework suitable for interactive exploration.

Several existing approaches have addressed the idea of generating proofs from the model-checking runs. Many of these techniques focus on building evidence in form of a proof and controlling the generation of information to aid the understanding of counterexamples [Cla+95; TC02]. One of the drawbacks of these approaches is their size and complexity, which can be polynomial in the number of states of the system and of exponential length in the worst case. The proof-like witness techniques also require manual extrapolation, and the developers still need certain knowledge of the underlying formalisms in order to understand the proofs.

The problem of the automated analysis of counterexamples was addressed by many researchers, for instance, Ball et al. [BNR03] describe an error trace as a symptom of the error and identify the cause of the error as the set of transitions in an error trace that does not appear in a correct trace of the program via the SLAM¹ model checker. Kumazawa and Tamai present an error localization technique LLL-S for a given behaviour model. The proposed technique identifies the infinite and lasso-shaped witnesses that resemble the given counterexample [KT11]. However, these approaches focus on finding the error causes in the program, such as deadlocks, assertion violations, and so on, but they are not applicable for verifying the causes of unsatisfied containment relationships.

¹See <https://www.microsoft.com/en-us/research/project/slam/>

Visual presentation of generated counterexamples is explored by Dong et al. [DRS03]. The authors developed a tool that simplifies the counterexample exploration by presenting evidence for modal μ -calculus through various graphical views. In particular, the highlighting correspondence between the generated counterexample and the analysed property is addressed in their visualization process. Armas-Cervantes et al. [AC+14] developed a tool for identifying behavioural differences between pairs of business process models by using Asymmetric Event Structure (AES) and verbalization of the results.

The above discussed approaches focus on general consistency checking. In contrast to our work, none of these techniques focuses on the diagnosis of counterexamples generated by a model checker with respect to containment checking. Note that, in our approach we do not need nor modify the source code of the model checker. Our interpretation process automatically extracts the information from the generated counterexample. In addition, another differentiating factor of our approach in comparison to aforementioned approaches is that our framework provides a compact and concise representation of the failure causes (such as missing or misplacement of elements) and countermeasures that are easily understandable for non-expert stakeholders. Finally, the counterexample interpretation steps are fully automated and do not require developer intervention.

6.7 Summary

This chapter addressed the Research Question RQ4. It presents, the counterexample analysis approach for locating the root causes of a containment inconsistency and producing appropriate guidelines as countermeasures based on the information extracted from counterexample trace file, formalisation rules, and the SMV descriptions of the low-level model. The advantage of this interpretation technique is twofold. On the one hand, the technique supports users who have limited knowledge of the underlying formalisms, and therefore, are not proficient in analysing the cryptic and verbose counterexamples. On other hand, by locating actual cause(s) of the inconsistency and providing the relevant countermeasures to alleviate the inconsistencies to the user, it significantly reduces the time of manually locating the causes of an inconsistency. The costly exhaustive searches employed by model checking are not always necessary for addressing the containment checking problem. Therefore, the next chapter will provide a lightweight graph-based approach for addressing the containment checking problem.

This chapter presents a lightweight graph-based approach for addressing the containment checking problem of software behaviour models at different levels of abstraction, since the costly exhaustive searches employed by model checking are not always necessary for containment checking. This problem addresses Research Questions RQ5. The chapter is based on a peer-reviewed conference paper in the proceedings of the 19th IEEE International Enterprise Distributed Object Computing Conference [TUMZ15].

7.1 Introduction

Model-based software development is maturing and potentials for solving problems with large and complex system development [Tai+04; HT06; BF14]. Thus, software engineers are increasingly using models for several development tasks such as describing and analysing software systems or generating system implementations out of these models. A typical development scenario based on models is, especially in the domain of enterprise systems, that a business analyst or software architect uses a high-level model for outlining the system and discussing with the customers and developers. The high-level model will then be refined to one or more low-level models by the development team. In the course of software system modelling and implementation, as models are created and evolved independently by different stakeholders and teams, inconsistencies among models often occur. Hence, detecting model inconsistencies in early phases of the software development life cycle is crucial to eliminate as many anomalies as possible before the systems are actually implemented and deployed. This has led to a rich body of work for checking and managing model consistency in the literature [LMT09]. Of these existing approaches, only a few are aiming at supporting consistency checking of behavioural models for software systems [LMT09], for instance, checking behavioural models against non-behavioural models [RW03; TE00; KC02] or checking different types of behaviour models [Wan+05; YS06; DM03; LP08; LP05]. Nonetheless, there are very few studies on checking the deviation of software behavioural models at different abstraction levels.

In this work, we focus on the containment relationship which is a special type of consistency relationship between software models at different levels of abstraction. The containment relationship is categorized as *vertical consistency* [Str05]. An unsatisfied containment relationship implies the deviation of the low-level descriptions from the corresponding high-level specifications and properties. To the best of our knowledge, very few existing studies have addressed the containment relationship so far.

Containment checking for software behaviour models, from a broader point of view, is related to the notion of behavioural equivalence relations between state transition systems [Par81; Mil89]. A notable challenge of behavioural equivalence checking (that can be deduced to containment checking) is that the estimated theoretical computational complexity is **NP**-hard, even for a class of simple finite communicating elements [Rab97]. Thus, it is rather costly to apply behavioural equivalence checking to complex and large scale software systems. Moreover, behavioural equivalence checking is strict in requiring a bidirectional equivalence of two behaviour models whilst the containment relationship mainly aims at unidirectional consistency.

There are some existing approaches trying to alleviate the complexity of equivalence checking by aiming at the *similarity* of behaviour models in particular application domains, for instance, workflows and business processes [RA06; Hid+05], state-charts [Nej+07], state-based models [WB13], to name but a few. Nevertheless, the outcome of these techniques is not a precise answer whether two behaviour models are equivalent or subsumed but rather an estimated degree of similarity of the input models. Hence, these approaches are useful for finding *similar* behavioural descriptions but not quite applicable for verifying the containment relationship.

Another challenge that has not been adequately addressed by the existing approaches for behavioural equivalence checking, and also consistency checking, is to assist the stakeholders in understanding the outcomes of the checking process. For instance, existing approaches for checking behavioural equivalence (aka *bisimulation*) often return a binary *true* (satisfied) or *false* (unsatisfied) answer without concrete information of the inequivalent cases [Mil89; Gla90; Gla93]. Another example are verification techniques based on model checking [CGP99; MTZ14] that produce counterexamples which are complex state based error traces [Cim+99; Hol91]. Those are, however, rather cryptic for users who often have limited knowledge of the underlying formal methods [DRS03].

We present in this chapter a lightweight approach for addressing these challenges of containment checking. First, the input behaviour models will be mapped onto intermediate representations, namely, *check models*. Based on a formal definition of the containment problem, our approach can be used to verify whether the resulting check models satisfy the containment relationship. Our proposed graph-based containment checking technique performs reasonably within the boundary of $O(n^3)$, where $n = \max(n_1, n_2)$ and n_1, n_2 are the numbers of elements of the two input models, respectively. Furthermore, our approach aims at producing concrete and helpful information about

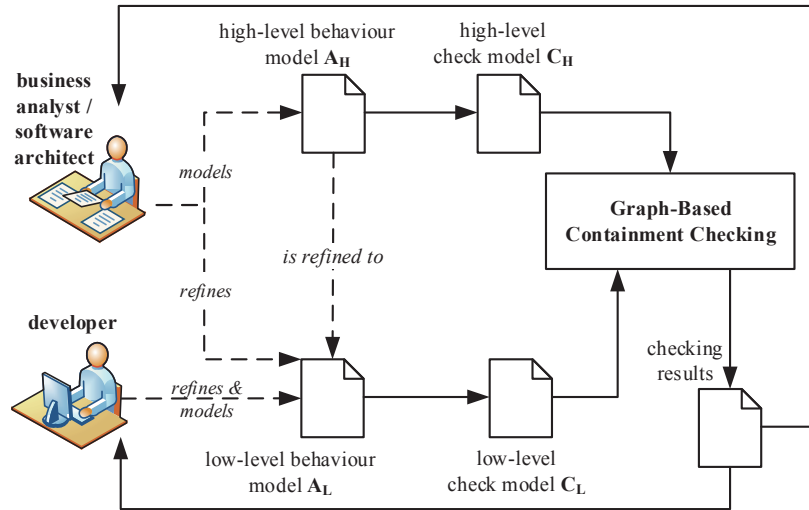


FIGURE 7.1: Overview of the Graph-Based Containment Checking Approach

the inconsistencies, such as missing elements, missing execution paths, or missing loops, in case the containment relationship is not satisfied. We illustrate our approach using UML activity diagrams—a part of the Unified Modelling Language [Gro11b]—which are widely used in both academia and industry for modelling and analysing behaviours of software systems. Nevertheless, our approach can also be applied to other types of behaviour models such as BPMN [Gro11a] that share similar notions and structures with UML activity diagrams.

The chapter is structured as follows. We describe our graph-based approach for containment checking in detail in Section 7.2. In Section 7.3 we use a realistic example extracted from industrial case studies to illustrate our approach along with its performance evaluation. Section 7.4 is dedicated for discussing the related studies on supporting behavioural consistency checking and especially containment checking. We summarise the main contributions in Section 7.5.

7.2 Approach

In this section, we describe our graph-based approach for containment checking of UML activity diagrams. An overview of the approach is shown in Figure 7.1. The main focus of the approach is depicted by the solid lines whilst the dashed lines illustrate relevant modelling and developing activities of the involved stakeholders.

Our approach starts by mapping the input UML activity diagrams at different levels of abstraction into equivalent intermediate representations, namely, *check models*. Then, our graph-based containment checking algorithm is used for verifying whether the resulting check models satisfy the containment relationship. In case the containment relationship is not satisfied (i.e., the input UML activity diagrams are inconsistent), our approach will be able to produce relevant checking results

with concrete information about the causes of inconsistencies such as missing elements, missing execution paths, or missing cycles as well as the involved model elements.

7.2.1 Activity Models and Check Models

As the definitions and semantics of UML activity diagrams are rather informal [Gro11b, Sec. 12], we derive a representative description, namely, activity model, to provide the basis for formally analysing UML activity diagrams. The definition of an activity model is based on the definition of classical transition systems [Mil89]. An activity model needs to adequately accommodate relevant concepts of a UML activity diagram such as different kinds of nodes, edges, and guards.

Definition 7.1 (Activity model). An activity model \mathcal{A} is a tuple $(N, E, G, type, guard)$ where

- N is a finite set of nodes
- $E \subseteq N \times N$ is an ordered finite set of edges,
- G is a finite set of guard expressions,
- $type : N \rightarrow \{InitialNode, ActivityFinalNode, FlowFinalNode, Action, DecisionNode, MergeNode, ForkNode, JoinNode, LoopNode, ConditionalNode\}$ is a function that maps a node to its type. Node types are derived from the UML 2 specification [Gro11b, Sec. 12].
- $guard : E \rightarrow G$ is a function that maps an edge to its guard expressions¹.

In Figure 7.2 we depict a simplified activity diagram $\mathcal{A} = (N, E, G, type, guard)$ of a travel agency system where $N = \{\text{Start, ReceiveItinerary, Fork, M1, M2, M3, BookCar, BookHotel, BookFlight, D1, D2, D3, Join, ChargeCreditCard, NotifyCustomer, Finish}\}$, $guard(e_{12}) = \text{"c_booked=no"}$, $guard(e_{15}) = \text{"c_booked=yes"}$, $guard(e_{13}) = \text{"h_booked=no"}$, $guard(e_{16}) = \text{"h_booked=yes"}$, $guard(e_{14}) = \text{"f_booked=no"}$, $guard(e_{17}) = \text{"f_booked=yes"}$, $type(\text{Start}) = InitialNode$, $type(\text{M1}) = type(\text{M2}) = type(\text{M3}) = MergeNode$, $type(\text{ReceiveItinerary}) = type(\text{BookHotel}) = type(\text{BookFlight}) = type(\text{ChargeCreditCard}) = type(\text{NotifyCustomer}) = Action$, $type(\text{Fork}) = ForkNode$, $type(\text{Join}) = JoinNode$, $type(\text{D1}) = type(\text{D2}) = type(\text{D3}) = DecisionNode$, $type(\text{Finish}) = ActivityFinalNode$. For the sake of illustration, we explicitly name the control nodes such as “Start”, “Fork”, “Join”, and “Finish”. The edges are prefixed with “e”. In reality, the developers can often accept the default identifiers generated automatically by UML modeling tools.

The main goal of our approach is to develop a lightweight graph-based algorithm for supporting the developers in checking the containment relationship between a high-level and low-level activity

¹We note that guard expressions are sub-classes of *ValueSpecification* in UML. For instance, a guard expression could be an *OpaqueExpression* such as “ $x \leq 1$ ” or a *LiteralString* such as “credit card accepted”. Here we omit the detailed formal definition of each possible kind of *ValueSpecification* as only their presence and comparability for equality is relevant for our containment checking approach, and hence, a more detailed definition is not needed.

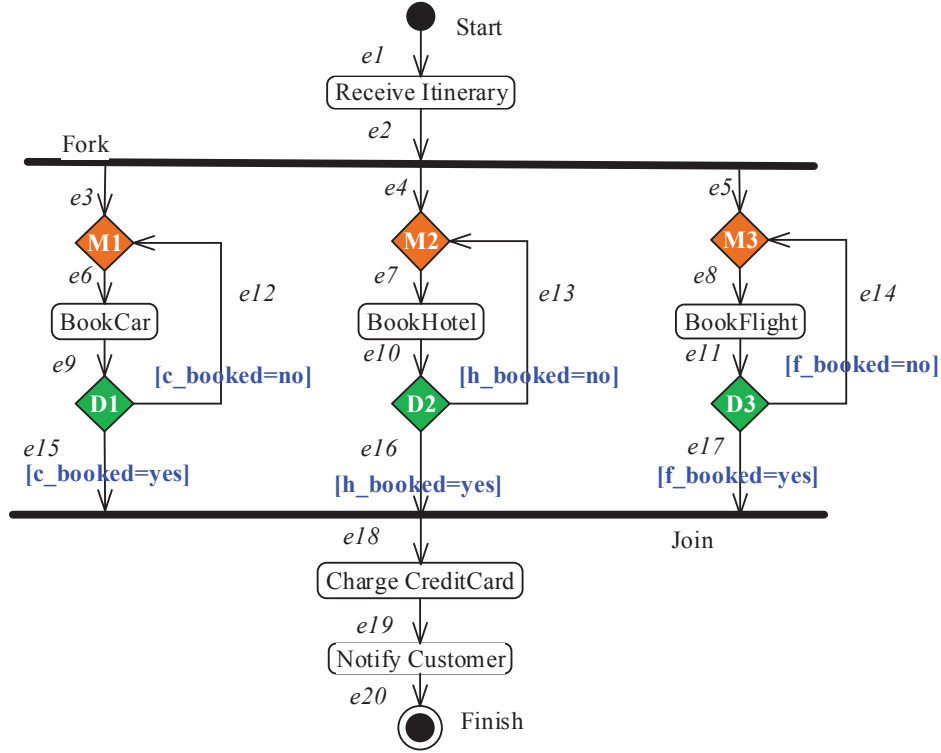


FIGURE 7.2: An Illustrative Simplified Activity Model of a Travel Agency

model. That is, our algorithm needs to verify whether the behaviour (or functions) described by the low-level activity model encompasses those specified in the high-level counterpart. Thus, the containment checking algorithm will walk through the high-level activity model to check whether its elements and structures (e.g., actions, control nodes, and edges) have corresponding parts in the low-level model.

However, the current form of an activity model might render the graph-based searching inefficient because it might also contain—apart from the essential elements such as nodes (including both actions and control nodes) and control flows that are similar to the nodes and edges of a graph—some unparalleled kinds of elements. For instance, an activity model might have edges associated with guards. A guard will determine whether the corresponding edge can be activated (i.e., leading to the execution of the edge’s target) following the execution of the edge’s source. This implies that the algorithm must dedicate special treatments for them as for the other control nodes because the guards have significant impact on the behaviour of the software systems described in the models.

Unfortunately, most of the existing approaches do not take guards into account adequately but rather assume that the control flows will be automatically activated. To alleviate this problem, we introduce an intermediate representation, namely, a check model, that can explicitly represent such kinds of elements. The essential idea is to transform the model elements associated with guards (e.g., edges) into pseudo nodes to yield an intermediate representation that poses the same

semantics as the original model and is efficient for graph searching. We show here how the edges are transformed to pseudo nodes of a check model. Similar model elements can be treated in the same manner.

Definition 7.2 (Check model). We assume that $\mathcal{A} = (N_{\mathcal{A}}, E_{\mathcal{A}}, G_{\mathcal{A}}, type_{\mathcal{A}}, guard_{\mathcal{A}})$ is an activity model. A check model \mathcal{C} derived from \mathcal{A} is represented by a tuple $(N_{\mathcal{C}}, E_{\mathcal{C}}, G_{\mathcal{C}}, type_{\mathcal{C}}, guard_{\mathcal{C}})$ where $N_{\mathcal{C}} = N_{\mathcal{A}} \cup \{n \mid type_{\mathcal{C}}(n) = GuardNode\}$, $E_{\mathcal{C}} = E_{\mathcal{A}} \cup E^g$, $G_{\mathcal{C}} = G_{\mathcal{A}}$, $type_{\mathcal{C}} = type_{\mathcal{A}} \cup \{GuardNode\}$, and $guard_{\mathcal{C}} = N_{\mathcal{C}} \cup E_{\mathcal{C}} \rightarrow G_{\mathcal{C}}$.

We use $e(s, t)$ to denote an edge $e \in E_{\mathcal{A}}$ that connects a source node s to a target node t where $s, t \in N_{\mathcal{A}}$. The construction of $N_{\mathcal{C}}$ and $E_{\mathcal{C}}$ is defined as follows. For every edge $e \in E_{\mathcal{A}}$ labeled with valid a guard constraint (i.e., $guard(e) \neq null$), we create a new node n_g , assign the guard value of e to the node, establish additional unlabeled links between n_g and the source and target nodes of e , and remove e from \mathcal{C} . Formally, the translation can be described as in Equation 7.1.

$$\begin{aligned}
 & \forall e(s, t) \in E_{\mathcal{A}} \text{ such that } guard_{\mathcal{A}}(e) \neq null \bullet \\
 & \quad (N_{\mathcal{C}} = N_{\mathcal{A}} \cup \{n_g\}) \wedge \\
 & \quad (E_{\mathcal{C}} = (E_{\mathcal{A}} \cup \{(s, n_g), (n_g, t)\}) \setminus \{(s, t)\}) \wedge \\
 & \quad (guard_{\mathcal{C}}(n_g) \equiv guard_{\mathcal{A}}(e))
 \end{aligned} \tag{7.1}$$

We note that the containment checking will be performed on the check model. Thus, it is necessary to prove that an activity model and the check model derived from it are behaviourally equivalent. We present a simple sketch of a proof as following.

Proof Sketch: Let us assume that an activity model $\mathcal{A} = (N_{\mathcal{A}}, E_{\mathcal{A}}, G_{\mathcal{A}}, type_{\mathcal{A}}, guard_{\mathcal{A}})$ is defined on a semantic domain $\mathbb{D}_{\mathcal{A}}$. Based on $\mathbb{D}_{\mathcal{A}}$ the semantics of a guarded edge $e \in E_{\mathcal{A}}$ is denoted as $\llbracket e \rrbracket$. A check model $\mathcal{C} = (N_{\mathcal{C}}, E_{\mathcal{C}}, G_{\mathcal{C}}, type_{\mathcal{C}}, guard_{\mathcal{C}})$ is mapped from \mathcal{A} using the procedure presented in Equation 7.1. We can derive a semantic domain $\mathbb{D}_{\mathcal{C}}$ for the check model \mathcal{C} based on $\mathbb{D}_{\mathcal{A}}$ such that each newly added guarded node $n \in \mathcal{C}$ corresponding to an edge $e \in E_{\mathcal{A}}$ will have the same semantics of e . That is $\llbracket n \rrbracket \equiv \llbracket e \rrbracket$. Hence, it is rather straightforward that \mathcal{C} and \mathcal{A} behave in the same way with respect to these semantic domains.

Here we give a simple example to illustrate the aforementioned proof. In the semi-formal token-based semantic domain $\mathbb{D}_{\mathcal{A}}$ of the UML 2 specification [Gro11b, pg. 371], a token is allowed to pass along an edge e if and only if $guard_{\mathcal{A}}(e)$ evaluates to *true*. Hence, $\mathbb{D}_{\mathcal{C}}$ can be defined based on $\mathbb{D}_{\mathcal{A}}$ such that a guarded node n mapped from e will pass along the token to the next node(s) if and only if $guard_{\mathcal{C}}(n) \equiv guard_{\mathcal{A}}(e)$ holds.

In Figure 7.3 we depict the check model (with omitted edges' labels) derived from the activity model shown in Figure 7.2. The two models are rather alike. The major difference is that all

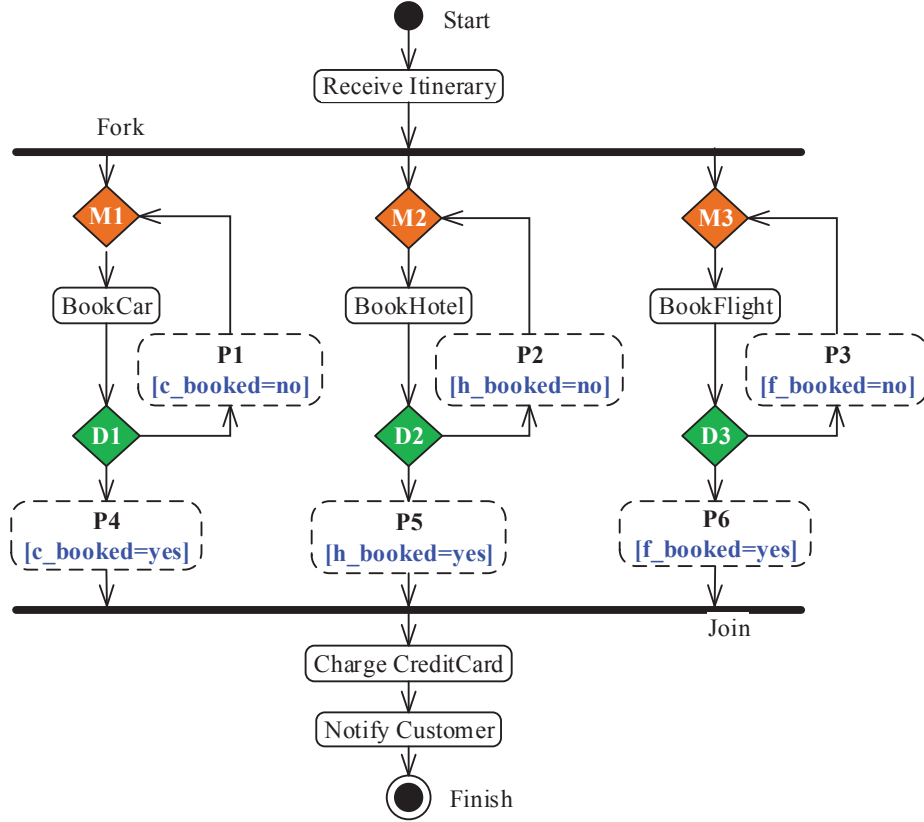


FIGURE 7.3: The Corresponding Check Model of the Travel Agency Activity Model

guarded edges of the activity model are transformed into pseudo nodes (shown with dashed border lines) associated with corresponding guard expressions in the check model. In this way, a graph search algorithm can treat all edges in the same manner.

7.2.2 Graph-Based Containment Checking

Our approach takes as inputs a high-level activity model \mathcal{A}_H and a low-level activity model \mathcal{A}_L to verify whether the containment relationship between these models are satisfied (i.e., \mathcal{A}_L “contains” \mathcal{A}_H). In case of inconsistencies, we need to inform the developers the particular causes of the violation. We use the relation symbol \prec to denote the containment relationship between these behaviour models. The containment relationship to be validated by our graph-based algorithm is defined in Equation 7.2.

$$\begin{aligned}
 \mathcal{A}_H \prec \mathcal{A}_L \stackrel{\text{def}}{=} & \text{noMissingNodes}(\mathcal{C}_H, \mathcal{C}_L) \\
 & \wedge \text{noMissingTransitiveLinks}(\mathcal{C}_H, \mathcal{C}_L) \\
 & \wedge \text{noMissingCycles}(\mathcal{C}_H, \mathcal{C}_L)
 \end{aligned} \tag{7.2}$$

where \mathcal{C}_H (resp. \mathcal{C}_L) is mapped from \mathcal{A}_H (resp. \mathcal{A}_L).

Next, we will explain in detail three functions *noMissingNodes()*, *noMissingTransitiveLinks()*, and *noMissingCycles()* presented in Equation 7.2. We note that these functions represent the (sub)tasks of containment checking. That is, our graph-based containment checking algorithm will verify whether there are any missing nodes (i.e., missing expected functions), missing transitive links (i.e., missing execution paths), and missing cycles (i.e., missing loop executions). In other words, the containment checking problem is divided into three sub-problems represented by the corresponding functions shown in Equation 7.2.

The huge advantage of this *divide-and-conquer* strategy is that our approach is able to tell specifically about the violation of the containment relationship based on the results achieved from each function. For instance, we can inform the violation of the containment relationship due to missing nodes, execution paths, or loops along with the involved elements by analysing the relevant formulas. Moreover, we note that these functions can be performed independently, and therefore, can potentially be parallelized to gain better performance.

$$\begin{aligned} \text{noMissingNodes}(\mathcal{C}_H, \mathcal{C}_L) \stackrel{\text{def}}{=} & \quad \forall n \in N_{\mathcal{C}_H} \bullet (\text{type}(n) = \text{MergeNode}) \vee \\ & (\text{type}(n) \neq \text{MergeNode} \wedge \exists m \in N_{\mathcal{C}_L} \bullet \text{match}(n, m)) \end{aligned} \quad (7.3)$$

The function *noMissingNodes()* aims to ensure that any nodes (e.g., actions, control nodes) that are present in the high-level model must also appear in the low-level counterpart. That implies the behaviour described in the low-level model can embrace the expected functions defined in the high-level model. The expected output of *noMissingNodes()* will be a set of nodes that are described in the high-level model but missing in the low-level model. If *noMissingNodes()* yields an empty set, we can assume that the functions described in the low-level model at least embrace the behaviour defined in the high-level model. As *noMissingNodes()* is rather straightforward, we present in Equation 7.3 its short formal description. Please note that *MergeNodes* can be safely ignored in this context because different combinations of merges do not lead to different ordering of the executed action nodes [Gro11b, pg. 398].

The function *match()*, which is used by *noMissingNodes()*, takes two model elements as inputs and returns *true* if two elements are matched and *false* otherwise. Intuitively, two matched elements must be of the same type and having the same identifier. Formally, *match()* can be described as shown in Equation 7.4.

$$\begin{aligned}
\text{match}(p, q) &\stackrel{\text{def}}{=} \text{type}(p) = \text{type}(q) \wedge (p.\text{name} = q.\text{name}) \\
&\wedge (\text{type}(p) = \text{DecisionNode} \implies \text{guardMatch}(p, q)) \\
&\wedge (\text{type}(p) = \text{JoinNode} \implies \text{specMatch}(p, q)) \\
&\wedge (\text{type}(p) = \text{GuardNode} \implies \text{specMatch}(p, q))
\end{aligned} \tag{7.4}$$

$$\begin{aligned}
\text{guardMatch}(p, q) &\stackrel{\text{def}}{=} \forall e_i(p, t_i) \in E_C \bullet \\
&\text{guard}(e) \neq \text{null} \implies \\
&\exists e_o(q, t_o) \in E_C \bullet \text{specMatch}(\text{guard}(p), \text{guard}(q))
\end{aligned} \tag{7.5}$$

The definition of *specMatch()* is not presented in detail here because it just defines the equality for all existing sub-classes of the class *ValueSpecification* in UML [Gro11b, pg. 139]. As this is rather trivial and lengthy, we opt to omit its definition here.

In the definition of the function *noMissingTransitiveLinks()*, we use the conventional definitions of the *adjacency matrix* and *transitive closure* of a directed graph. Let $G = (V, E)$ be a directed graph where V is the set of nodes and E is the ordered set of arcs. The adjacency matrix A_G of G is an $n \times n$ boolean matrix whose elements $A_G[i, j]$ is *true* if $e(i, j) \in E$ and *false* otherwise.

Based on the adjacency matrix A_G , we derive a *reachability matrix* $R_G = A_G^*$ to represent the transitive closure of G . It is denoted as $R_G[i, j] = \text{true}$ if there is a directed *path* from node i to node j and *false* otherwise. The function *noMissingTransitiveLinks()* is defined in Equation 7.6.

$$\begin{aligned}
\text{noMissingTransitiveLinks}(\mathcal{C}_H, \mathcal{C}_L) &\stackrel{\text{def}}{=} \forall p, q \in N_{\mathcal{C}_H} \bullet (R_{\mathcal{C}_H}[p, q] = \text{false}) \vee \\
& (R_{\mathcal{C}_H}[p, q] = \text{true} \implies (\text{type}(p) = \text{MergeNode} \vee \text{type}(q) = \text{MergeNode}) \vee \\
& (\text{type}(p) \neq \text{MergeNode} \wedge \text{type}(q) \neq \text{MergeNode}) \\
& \wedge (\exists p_{\text{match}}, q_{\text{match}} \in N_{\mathcal{C}_L} \bullet \text{match}(p, p_{\text{match}}) \\
& \wedge \text{match}(q, q_{\text{match}}) \wedge R_{\mathcal{C}_L}[p_{\text{match}}, q_{\text{match}}] = \text{true}))
\end{aligned} \tag{7.6}$$

The main idea of *noMissingTransitiveLinks()* is to verify whether any possible execution paths defined in the high-level model are missing in the low-level counterpart. Let \mathcal{C}_H (resp. \mathcal{C}_L) be input high-level and low-level check models and $R_{\mathcal{C}_H}$ (resp. $R_{\mathcal{C}_L}$) be the corresponding *reachability graph*. Let us consider an arbitrary pair of nodes (p, q) , where $p, q \in N_{\mathcal{C}_H}$. In case there are no paths between p and q in \mathcal{C}_H , i.e., $R_{\mathcal{C}_H}[p, q] = R_{\mathcal{C}_H}[q, p] = \text{false}$, we do not need to consider the corresponding links in \mathcal{C}_L . If a path between p and q exists, i.e., either $R_{\mathcal{C}_H}[p, q] = \text{true}$ or $R_{\mathcal{C}_H}[q, p] = \text{true}$, there must be a corresponding path in the low-level check model.

The function *noMissingCycles()* is responsible for verifying whether any loops described in the high-level model are not realized by the low-level counterpart. In order to define *noMissingCycles()*,

we use Tarjan's algorithm [Tar72] to obtain a set of *strongly connected components* (SCC)² in the helper function *getStronglyConnectedComponents()*.

Assuming that \mathcal{S}_H (resp. \mathcal{S}_L) is the set of the strongly connected components of \mathcal{C}_H (resp. \mathcal{C}_L), we define *noMissingCycles()* in Equation 7.7. Given an SCC s , N_s and E_s are used to denote the sets of nodes and edges of s , respectively.

$$\begin{aligned}
 \text{noMissingCycles}(\mathcal{C}_H, \mathcal{C}_L) &\stackrel{\text{def}}{=} \forall s_H \in \mathcal{S}_H \bullet \\
 &(|N_{s_H}| = 1 \wedge \exists p \in N_{s_H} \\
 &\quad \wedge \text{hasSelfLink}(p, E_{s_H}) \implies \exists s_L \in \mathcal{S}_L \bullet (|N_{s_L}| = 1 \wedge \exists q \in N_{s_L} \\
 &\quad \bullet \text{match}(p, q) \quad \wedge \quad \text{hasSelfLink}(q, E_{s_L})) \vee \\
 &(|N_{s_L}| > 1 \wedge \text{allNodesPresent}(s_H, s_L))) \\
 &\bigvee (|N_{s_H}| > 1 \wedge \exists s_L \in \mathcal{S}_L \bullet \text{allNodesPresent}(s_H, s_L))
 \end{aligned} \tag{7.7}$$

where

$$\mathcal{S}_x = \text{getStronglyConnectedComponents}(\mathcal{C}_x)$$

$$\text{hasSelfLink}(n, E) \stackrel{\text{def}}{=} \exists e = (s, t) \in E \bullet s = n \wedge t = n$$

$$\text{allNodesPresent}(G_1, G_2) \stackrel{\text{def}}{=} \forall n_1 \in N_{G_1}, \exists n_2 \in N_{G_2} \bullet \text{match}(n_1, n_2)$$

The function *noMissingCycles()* will examine two cases: either there is only one element in the nodes of an SCC or more. An SCC with only one node must have a link from that node to itself to form a cycle. If the only node p in a single node SCC has a link to itself, a containing SCC in the low-level model must exist respectively and contain the same cycle. This can be the case, either if the containing SCC is the same single node graph with a self link, or if it is a bigger SCC that contains the node p . If an SCC has more than one element, a bigger cycle is present in s_H , and an SCC $s_L \in \mathcal{S}_L$ must exist that contains all the nodes from s_H . Please note that it is sufficient to show here that the same nodes exist in cycles, as we have already established above (in *noMissingTransitiveLinks()*) that none of the transitive links between nodes is missing.

So far we have described the main ideas along with formal definitions of the containment relationship between two software behaviour models. These definitions are the basis of our lightweight graph-based approach for containment checking. In the subsequent part of the chapter, we will analyse the theoretical complexity of our approach based on the individual (sub)-functions and conduct a quantitative evaluation of the scalability and applicability of our approach using realistic industrial scenarios derived from our previous projects.

²A graph is strongly connected if every vertex is reachable from every other vertex. The strongly connected components of a directed graph form a partition into subgraphs that are themselves strongly connected.

TABLE 7.1: Estimation of Theoretical Complexity of Graph-based Containment Checking

Main function	Sub-function(s)	Worst-Case Complexity
translateAtoC	<i>addPseudoNodes</i> (G)	$O(E_G)$
noMissingNodes	<i>checkMissingModelElements</i> (G_1, G_2)	$O(V_{G_1} \times V_{G_2})$
	<i>match</i> (n_1, n_2)	$O(out(n_1) \times out(n_2))$
noMissingCycles	<i>hasSelfLink</i> (n)	$O(out(n))$
	<i>getStronglyConnectedComponents</i> (G)	$O(V_G + E_G)$
	<i>allNodesPresent</i> (S_1, S_2)	$O(V_{S_1} \times V_{S_2})$
noMissingTransitiveLinks	<i>getTransitiveClosure</i> (G)	$O(V_G ^3)$
	<i>checkMissingTransitiveLinks</i> (G_1, G_2)	$O(\mathcal{S}_{G_1} \times \mathcal{S}_{G_2})$

7.2.3 Theoretical Complexity Analysis

In this section, we present an analysis of the theoretical worst-case complexity of our approach through the constituent functions. The mapping of an activity model into a check model can be achieved by traversing the set of the activity's edges and converting guarded edges to pseudo nodes as described in Equation 7.1. This process is represented by the function *addPseudoNodes()* shown in Table 7.1.

The complexity of *noMissingNodes()* (see Equation 7.3) can be estimated as two loops over the nodes of two input check models. *noMissingNodes()* uses the function *match()* (see Equation 7.4) to check whether two arbitrary model elements are matched. We note that *match()* essentially compares the nodes' names, types, and/or guard conditions or *valueSpec*. Thus, the worst-case of *match()* occurs when we analyse the guards associated with the outgoing edges of a *DecisionNode* (we use *out*(n) to denote the set of outgoing edges of n).

In the function *noMissingTransitiveLinks()*, establishing the *transitive closure* of an input digraph G is obtained by using the traditional Warshall's algorithm [War62] that has the time complexity of $O(|V_G|^3)$ where V_G is the number of nodes of G . Then, comparing for transitive links (aka execution paths) of two check models can be quickly performed using the reachability matrix.

The function *noMissingCycles()* comprises three sub-functions. The first function *hasSelfLink()* can perform in constant time. The second function *getStronglyConnectedComponents()* is based on Tarjan's algorithm [Tar72] that has the worst-case complexity of $O(|V| + |E|)$. The third function *allNodePresent()* compares the nodes of two corresponding *strongly connected components* (SCC), and therefore, is bounded by $O(|V_{S_1}| \times |V_{S_2}|)$ where S_1 and S_2 are the aforementioned SCCs. The worst-case complexity of *allNodePresent()* occurs when a strongly connected component under consideration becomes the whole input graph (which rarely, or even never, happens because this implies all nodes of the graph must be connected).

7.3 Quantitative Evaluation and Discussion

7.3.1 Quantitative Evaluation

We implement our graph-based approach for containment checking and conduct a preliminary evaluation of its performance. The main idea is to validate whether the proposed approach performs reasonably for typical models used in industry on typical workstations used by developers. We also aim to compare with existing techniques for containment checking. Unfortunately, apart from our early work based on model checking [MTZ14], we cannot find any working tools for comparison purpose. Nevertheless, we note that the containment checking approach presented in [MTZ14] is based on model checking techniques that can support exhaustively exploring the state space to find out any behavioural inconsistencies. Therefore, this technique, when enabling the exhaustive state space search options, can be considered as an estimated upper boundary of the complexity of the containment checking problem. As a result, we will perform the comparison of our graph-based approach and the model checking based approach [MTZ14] to see how well our approach scales with respect to that upper boundary.

The workstation used for the performance evaluation is running Linux on a CPU Quad-Core 2.66 GHz with 2048 megabytes of memory. The two approaches under consideration are implemented in Java and executed with the Java VM 1.7. We note that the model checking based approach uses the NuSMV model checker [Cim+99] version 2.5.4 for verifying the containment relationship. Hence, the NuSMV's source code has been instrumented for measuring the corresponding model checking time. The evaluation is conducted through four behaviour models extracted from industrial scenarios in network service and banking sectors. These behaviour models are, namely, *Order Processing* (OP), *Travel Booking* (TB), *Customer Fulfillment* (CF), *Billing Renewal* (BP), and *Loan Approval* (LA) with different sizes and complexity.

For measuring and comparing the performance of our approach and the exhaustive exploring techniques, we focus on the worst-case scenarios. In theory, the worst-case execution of containment checking taking two input behavioural models is, as shown in the theoretical complexity analysis, when these input models have approximate or equal sizes (the *size* of a behavioural model refers to the number of elements of different types such as nodes, paths, and so on). Thus, in our experiments we take each behaviour model and perform the containment checking using different techniques to verify the model against itself. This way, we can estimate the worst-case performance of both approaches. We report in Table 7.2 our measurement and analysis results based on the aforementioned cases. We also summarise the total execution time of the two techniques and visualize them in Figure 7.4 to analyse how fast the execution times grow.

In the first part of Table 7.2, we present the complexity of the input UML activity diagrams in terms of their elements including actions—representing computational or data handling tasks,

	OP	TB	CF	BR	LA
Input Size					
Action Nodes	11	7	16	22	29
Control Nodes	8	12	8	17	22
Edges	22	24	30	54	63
Guarded Edges	3	8	4	10	19
Total Elements	41	43	54	93	114
Graph-Based Approach					
Model Loading (ms)	3159 ± 112	3377 ± 77	3890 ± 161	5666 ± 137	5142 ± 417
AtoC (ms)	311 ± 38	821 ± 91	447 ± 74	1048 ± 105	1776 ± 239
cNMN (ms)	34 ± 2	24 ± 7	49 ± 3	116 ± 10	216 ± 57
cNMC (ms)	23 ± 2	37 ± 6	30 ± 5	56 ± 3	85 ± 17
cNML (ms)	575 ± 19	723 ± 11	1145 ± 92	6124 ± 97	15 045 ± 1349
Total Time (ms)	4102	4982	5560	13 009	22 264
Model Checking Based Approach					
Model Loading (ms)	3167 ± 882	3357 ± 80	3862 ± 122	5481 ± 190	5364 ± 135
UMLtoSMV (ms)	535 ± 38	564 ± 30	665 ± 31	1190 ± 29	1590 ± 48
ModelChecking (ms)	22 635 ± 1859	32 978 ± 1394	24 632 ± 715	56 606 ± 2985	438 174 ± 35 814
Total Time (ms)	26 336	36 899	29 159	63 277	445 128

TABLE 7.2: Performance Evaluation and Comparison

control nodes—representing the nodes that can change the flow of execution described in the activity diagrams, and edges—representing the links between nodes. In measuring the execution of the graph-based containment checking approach, we observe individual tasks such as model loading, translating activity models to check models (AtoC), checking for missing nodes (cNMN), missing transitive links (cNML), and missing cycles (cNMC), respectively. The execution time of the model checking based approach can be broken down to model loading, translating activity models to NuSMV descriptions (UMLtoSMV), and performing model checking. The corresponding time consumed by each task is the average time out of 1000 rounds of execution. Before measuring each task, sufficient warming-up executions are performed to reduce potential confounding factors of class loading and instantiation in Java. The execution time is measured in nanoseconds but rounded and shown in milliseconds for the sake of readability. We also include in [Table 7.2](#) the corresponding unbiased standard deviation of the execution time of each case.

In accordance with the theoretical complexity analyzed in [Section 7.2.3](#), the costly aspect of our approach is *noMissingTransitiveLinks()* that consumes reasonable time for building the transitive closure matrix. Nevertheless, the total execution time of the graph-based technique, to the best of our knowledge, is still reasonable for a typical working environment. In the model checking based technique, performing model checking is often a time-consuming task and it grows rather exponentially (see [Figure 7.4](#)).

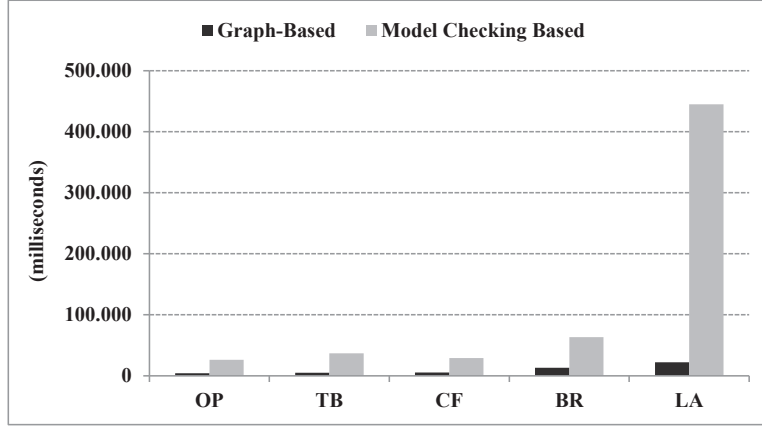


FIGURE 7.4: Comparison of the Scalability of Two Approaches

7.3.2 Discussions

The containment relationship between two behaviour models at different abstraction levels is based on the assumption that a high-level model element and its low-level corresponding have the same name and type. That reflects in the function *match()* where two input elements are compared with respect to their names, types, and/or other associated properties such as guards. The aforementioned assumption is rather realistic because a low-level model is mainly achieved through a refinement of a high-level model where existing high-level elements are often enriched with more details and elements [TZD10]. Nevertheless, in cases of mismatches of their names and types, one possibility to alleviate this problem, like in the approaches on checking behaviour similarity [BL12], is to employ supporting text matching techniques [Nav01].

Breaking down the problem of containment checking into smaller tasks (or functions) as shown in Equation 7.2 brings a number of advantages. First, these tasks are independent from each other, and therefore, can be performed in any order. For instance, we can perform *noMissingNodes()* to quickly validate whether the low-level model covers all functions specified in the high-level model or *noMissingTransitiveLinks()* to assess that no execution link in the high-level model is missing. Moreover, these tasks can also be executed in parallel to gain better performance. Last but not least, each of these tasks can inform the developers precisely about the causes of the violation of the containment relationship, for instance, due to missing nodes, cycles, or execution paths, along with the involved model elements. To the best of our knowledge, none of existing related approaches has addressed this aspect yet.

The most challenging issues in comparing behaviour models are to deal with loops (e.g., a combination of decisions nodes and backward edges or a structured loop node [Gro11b, pg. 396]) and parallel execution branches (e.g., a combination of fork and join nodes). A loop, especially a conditional loop, cannot be efficiently described by existing property specification logics such as temporal logics [Wol83] that are the formal basis of several model checking based techniques. Parallel execution

branches lead to the *state explosion* problem for existing techniques that are based on exhaustive searches through state spaces of the behaviour models [CGP99]. We aim to address these issues in our graph-based approach by using the *noMissingTransitiveLinks()* to verify the presence of all possible execution branches and *noMissingCycles()* to verify the presence of corresponding loops in both behaviour models.

There is a considerable overhead (growing towards $O(V^3)$) of building the transitive closure (TC) used by our graph-based containment checking technique in *noMissingTransitiveLinks()*. Our implementation of *noMissingTransitiveLinks()* is based on Warshall’s algorithm [War62]. Nevertheless, Nuutila has presented heuristics for improving Tarjan’s algorithm [Tar72] in detecting strongly connected components (SCC) and uses the improved SCC detection techniques (plus a special representation of successor sets) to achieve better TC finding [Nuu95]. Our approach is well in line with Nuutila’s techniques with respect to the use of Tarjan’s algorithm for *noMissingCycles()* and Warshall’s algorithm for *noMissingTransitiveLinks()*. However, we opted not to integrate Nuutila’s techniques in order to better analyze individual performance. Moreover, tight integration of Nuutila’s techniques implies the dependency between *noMissingTransitiveLinks()* and *noMissingCycles()*, and hence, may nullify the potential parallelizability of our approach.

7.4 Related Work

The consistency checking problem has been extensively studied in the literature [LMT09]. However, very few of these studies focus on the consistency of behaviour models [LMT09]. To the best of our knowledge, none of them considers the containment checking problem for behaviour models, which verifies whether an expected software behaviour specified at a higher level of abstraction is satisfied by a corresponding behaviour description at a lower level of abstraction.

Containment checking for behaviour models, to a broader extent, is related to the notion of behavioural equivalence [Mil89]. However, behavioural equivalence based techniques are rather not applicable for checking the containment relationship. On the one hand, these techniques produce a binary “*true*” (satisfied) or “*false*” (unsatisfied) answer but do not provide further concrete information of the inequivalent cases [Mil89; Gla90; Gla93; GW89]. Several studies have been conducted to alleviate the aforementioned complexity by measuring the degree of behavioural equivalence. On the other hand, Rabinovich showed that the complexity of the behavioural equivalence checking on directed graphs is **NP**-hard, even for a class of simple finite communicating elements [Rab97]. It leads to a considerable number of approaches for alleviating the aforementioned complexity by relaxing the equivalence relationship and, instead, computing the similarity of behaviour models in different application domains. For instance, Nejati et al. propose an approach to matching hierarchical state-charts models [Nej+07]. Walkinshaw and Bogdanov propose two techniques to compare state machines in terms of language perspective—the externally observable sequences of

events (transition labels) and in terms of structure perspective—to compute the precise difference of their actual states and transitions [WB13].

In the field of business process management, there also is a research trend focusing on computing the similarity of process models [BL12]. Some approaches concentrate on searching for process models in a large repository that match a given process model or a process fragment thereof [Dij+11; DDGBn09]. Also in this field, there are a considerable number of approaches using trace theory to validate the conformance of two process models or process models against their execution traces recorded in the event logs. An approach presented in [AMW06] checks the conformance between Petri net based process models and their execution traces and returns a degree of behavioural similarity ranging from “*completely different*” to “*identical*”. Another approach aims at verifying whether two process models are similar using their corresponding event traces mined from process execution logs [Aal+08]. Wang et al. measure the similarity of behaviour of process models, based on the *coverability* tree of labeled Petri nets [Wan+10]. Bae et al. propose to use distance measures metric for measuring mining process similarity and difference [Bae+07]. In this method, dependency graphs are extracted from process models and converted into normalized matrices. Afterwards, metric space distances are calculated based on the difference between the normalized matrices. Weidlich, Dijkman, and Weske consider the compatibility between referenced process models and the corresponding implementation based on the notion of behaviour inheritance [WDW12]. We note that, unlike our approach, the aforementioned techniques do not aim at providing precise answers whether two behaviour models are equivalent or subsumed nor concrete information about any inconsistencies. These approach rather produce an estimated *degree of similarity* of these models. Therefore, they are very useful for finding similar or alternative behavioural descriptions but not applicable for verifying the containment relationship.

A closely related work is proposed by Eshuis and Grefen for structurally matching BPEL [OAS07] business process models [EG07]. The main idea of this approach is to transform a BPEL process model into a tree structure and exhaustively compare two trees to find out whether the underlying process models fall into one of four matching categories, namely, “*exact matching*”, “*plug-in matching*”, “*inexact matching*”, and “*mismatched*”. This approach leverages the tree structures that are only possible to derive from block-structured behaviour models like BPEL processes³. Thus, it is not applicable to a wide range of behaviour models that allows flexible connections between model elements (e.g. UML activity diagrams [Gro11b, Sec. 12], BPMN [Gro11a]). Moreover, the tree structure in this approach does not allow cyclic paths, and therefore, cannot be used to verify loop matching as our approach⁴.

³Please note that the authors consider links in BPEL but restrict the boundary of links inside the containing block in order to create the underlying tree structures.

⁴Please note that the authors introduce loop sensitive matching but only for loops nodes, which is fundamentally different from loop structures formed by cyclic paths.

Our graph-based containment checking approach presented in this chapter is illustrated via UML activity diagrams. Like many other behaviour models, an activity diagram embraces fundamental constructs to express sequential and concurrent executions, choices, merges, and iterations. Unfortunately, the semantics described in the UML 2 specification [Gro11b] is rather informal. As a result, our earlier work presented in [MTZ14]—among the others based on model checking techniques—must derive a transformation of the input activity diagrams to equivalent formal descriptions [Shi06; TMH08]. One challenging aspect is that the non-determinism of decision nodes and loop nodes make the translation of the input models to formal properties, e.g., in temporal logics, very difficult and inefficient. Another challenge is that parallel structures, for instance, formed due to the combination of “*ForkNodes*” and “*JoinNodes*” in UML activity diagrams, often cause the *state explosion* problem [CGP99] as we discussed above. In contrast to our approach, most of the model checking based approaches have not considered to provide adequate and concrete information about the causes of the non-equivalence between behaviour models. In case inconsistencies exist, the outcomes, for instance, counterexamples [CGP99], are very cryptic for the stakeholders who often have limited knowledge about the underlying formal methods [DRS03]. Last but not least, most of the approaches based on model checking techniques are very time-consuming as shown in our evaluation, and therefore, rather not realistic to apply in complex and large software development settings.

7.5 Summary

This chapter addressed the Research Question RQ5. It presents a graph-based approach for addressing the problem of containment checking of software behaviour models at different levels of abstraction. In this approach, the input behaviour models are mapped to a formal intermediate representation that can be handled efficiently by graph search algorithms. The containment relationship is formally defined and divided into smaller problems that are resolved by three tasks: finding missing nodes, missing execution paths, and missing loops, respectively. The advantage of this *divide-and-conquer* strategy is twofold. On the one hand, these tasks can be performed independently, and therefore, can be parallelized to gain better performance. On the other hand, each task produces concrete and precise information about the violation of the containment relationship accordingly. The prototypical implementation of our approach performs within the boundary of $O(n^3)$ where n is the size of the inputs. The quantitative evaluation on industrial scenarios shows that the proposed approach performs reasonably on a typical working environment and scales well within the complexity upper bound of an exhaustive model checking based approach. Next chapter will investigate whether the behaviour of an architectural pattern is consistent in terms of the artefacts produced in the various activities of the software development process.

This chapter aims to verify whether the behaviour of architectural patterns is consistent in terms of the artefacts produced in the various activities of the software development process, such as requirements, software architecture, detailed design and implementation. To date, however, none of the published studies have considered the containment inconsistencies of architectural patterns' behaviour. This problem addresses Research Question RQ6. Furthermore, the chapter is based on a peer-reviewed conference paper in the proceedings of the 22nd European Conference on Pattern Languages of Programs [MTZ17b].

8.1 Introduction

A typical development scenario for modelling architectural patterns' behaviour is that a business analyst or software architect uses a high-level model for outlining the system and discussing it with the customers and developers. The development team will expand the high-level model to include one or more low-level models. The low-level model, for example a sequence diagram showing the detailed interactions, is closer to or even very closely related to the source code of the system. As the software development process involves various activities, such as requirements elicitation, software architecture design, detailed design and implementation that are created and evolved independently by different stakeholders and teams, inconsistencies often occur among them. For instance, high-level models might be changed according to new requirements, and low-level models are changed as the implementation is modified. If each change is not systematically propagated to all other models of the same system (or reality), the evolved models may become inconsistent. Hence, detecting inconsistencies in early phases of the software development life-cycle is crucial to eliminate as many anomalies as possible before the systems are actually deployed. Such inconsistencies concern all kinds of constraints that a high-level model imposes on the low-level model. This is important for architectural patterns, as they impose various kinds of design constraints on the detailed designs and implementations that should not be violated. To date, however, none of the published studies have considered the consistency checking of architectural patterns' behaviour.

The main idea of architectural patterns is to resolve the recurring design problems that arise in a specific context at the level of software architectures, including those related to helping in documentation of architectural design decisions, facilitating the communication between stakeholders through a common vocabulary, and describing the quality attributes of a software system [AZ05]. There have been many attempts at modelling the structure of architectural patterns [Gam+95; SG96; MT00; ZA08]; however, only a very few studies have focused on behaviour modelling of patterns [GAO94; KA08; PTT06]. In practice, the most popular languages for modelling of architectural patterns and pattern variants in software design are various kinds of informal and semi-formal box-and-line diagrams [RW05], the Unified Modelling Language (UML) [Gro11b], Architecture Description Language (ADL) [SG96; MT00], and Domain Specific Language (DSL) [MHS05].

This work focuses on a special type of consistency checking, containment checking, which can be categorized as *vertical consistency* [Str05; MTZ17a]. The idea of containment checking is to verify whether the high-level design constraints described by an architectural pattern are contained in the low-level design and implementation. Therefore, the containment checking of architectural patterns' behaviour not only deals with missing elements or interactions but also misplacement of elements at different levels of abstraction. Please note that there are severe negative effects of containment inconsistencies that may cause serious delays in and therefore increased costs of the system development process, jeopardize properties related to the quality of the system, and make it more difficult to maintain the system [SZ01]. In order to support containment checking, we conducted a systematic literature review on behaviour consistency checking research [MTZ17a]. In addition, we have investigated the containment relationship for various behaviour models in our previous work, including activity diagrams [MTZ14; TUMZ15], sequence diagrams [MTZ16] and BPMN¹ choreography and collaboration models [MTZ15; Mur+17]. We have also investigated possible solutions that are based on model-checking techniques [MTZ14; MTZ16; MTZ15; Mur+17] and graph algorithms [TUMZ15]. In this research on modelling and analysing behaviours of architectural patterns, we illustrate the containment checking solution using sequence diagrams [Gro11b]. Our solution provides informative and comprehensive feedback to the stakeholders to identify the violation causes and their resolutions. Unfortunately, modelling or mapping of architectural patterns' behaviour to sequence diagram is also a challenging task due to different variants of architectural patterns and different semantics of pattern elements and UML. In order to guide the user to follow a specific architectural pattern and its variants, we extend the UML metaclasses using UML profile mechanism. In particular, we use stereotypes to extend the properties of existing UML metaclasses. The applicability of the proposed solution is demonstrated for the MODEL-VIEW-CONTROLLER, LAYERS, and PIPE AND FILTER patterns. The proposed solution can also be applied to other types of behaviour models such as BPMN, UML activity diagrams and state machines.

¹See <http://www.omg.org/spec/BPMN/2.0.2/PDF>

The remainder of this chapter is structured as follows: Section 8.2 describes the proposed containment checking solution in detail. Section 8.3 demonstrates the applicability of the proposed solution to the MODEL-VIEW-CONTROLLER, LAYERS, and PIPE AND FILTER patterns. Section 8.4 summarises the related work on modelling and formalisation of architectural patterns, and behavioural consistency checking. Section 8.5 summarises the chapter.

8.2 Approach Overview

This research aims at identification and resolution of containment checking problems for architecture patterns' behaviour, in particular whether the generic behaviour described by the low-level model of a software system that is based on an architectural pattern encompasses those specified in the high-level counterpart of that particular pattern. Typically, a high-level model is created by a business analyst or software architect for outlining the system and discussing with the customers and developers. The low-level models are created by the development team or otherwise reverse-engineered from the source code; they are closer to or even very closely related to the source code of the system. In the course of software system modelling and implementation, as models are created and evolved independently by different stakeholders and teams, inconsistencies among models often occur. Therefore, containment checking improves the quality and reduces the complexity of big and complex system by determining and resolving the deviations between the low-level behaviour models and a high-level counterpart in the design phase. To guide the user to follow a specific architectural pattern and its variants, we extend UML metaclasses using the UML profile mechanism. In particular, we use stereotypes to extend the properties of existing UML metaclasses. An overview of our containment checking approach is shown in Figure 8.1. The main focus of the approach is depicted by the solid lines whilst the dashed lines illustrate relevant modelling and developing activities of the involved stakeholders.

Two possible ways of containment checking are graph-based search and model checking techniques. The graph based techniques might require intermediate representation of behaviour models, i.e., mapping of elements to nodes and edges [FK07; Tru+09; TUMZ15]. Subsequently, the graph search algorithms are used to verify the containment relationship between input models. The model checking techniques require the transformation of high-level behaviour model into design constraints, whereas the low-level behaviour model can be mapped into formal descriptions. This can be done using either manual mapping of input models into formal descriptions and consistency constraints (e.g., specifying the transformation rules) or automated techniques. In [MTZ14; MTZ16] we have introduced the transformation rules grounded on formal expressions that can support the automated transformation of the high-level behaviour models into design constraints and low-level behaviour models into formal descriptions. In particular, the behaviour models are

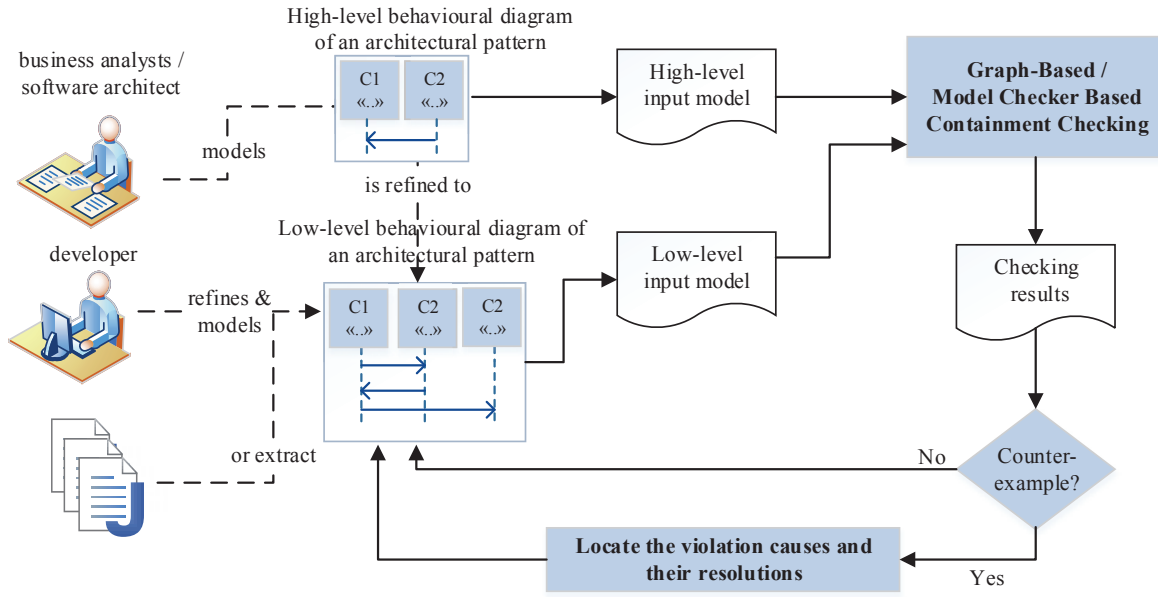


FIGURE 8.1: Approach Overview

created in Eclipse Papyrus² and the Eclipse Xtend framework³ is used to realise the transformation of behaviour models to formal descriptions and design constraints. In case the containment relationship is not satisfied (i.e., the input behaviour models are inconsistent), the checking results have to provide concrete information about the causes of inconsistencies and their resolution such as missing elements and/or misplacement of elements. In the subsequent section we will describe our approach in detail.

8.2.1 Approach Details

As emphasized, this chapter deals with the problem of checking the containment between generic behaviour of architectural patterns at different level of abstraction. We modelled the generic behaviour of architectural patterns using UML2 sequence diagrams as these diagrams can be used to capture the interaction between architectural pattern elements. Sequence diagrams show the sequence of methods/operations entailed by the architectural patterns, occurrence of events to invoke specific methods/operations, and use messages to represent the interaction among pattern elements [KA08]. In particular, a sequence diagram consists of lifelines/objects representing the individual participants in the interaction that communicate via messages. A message is sent from its source object to its target object (represents an operation/method on the objects) and has two endpoints. Each endpoint is an intersection with an object and is called an *OccurrenceSpecification* (*OS*). In particular, each message associates normally two *OSs* (aka events): one is the sending *OS* and the other is the receiving *OS*. An *OccurrenceSpecification* is a specialization of a *MessageEnd*.

²See <https://www.eclipse.org/papyrus>

³See <https://eclipse.org/xtend>

ExecutionOccurrence is represented by two event occurrences, the start event occurrence and the finish event occurrence. Messages can be asynchronous or synchronous. The source object continues to send and receive other messages after an asynchronous message is sent. In contrast, when a synchronous message is sent, the source object blocks and waits to receive a response (i.e., reply message) from the target object. As the definitions and semantics of UML sequence diagrams are rather informal [Gro11b], we derive a representative description of sequence diagrams constructs, to provide the basis for formally analysing the generic behaviour of architectural patterns.

Definition 8.1 (Sequence Model). A sequence model *Seq* is a tuple (*Objects*, *Interactions*) to express the generic behaviour of architectural pattern where

$$\begin{aligned}
 \textit{Object} &::= \text{is a finite set of objects/lifelines} \\
 \textit{type : Interaction} &::= (\textit{Object_Source}, \textit{Message}, \textit{Asynchronous/Synchronous}, \textit{Snd}) \mid \\
 &\quad (\textit{Object_Target}, \textit{Message}, \textit{Asynchronous/Synchronous}, \textit{Rec}) \\
 \textit{Snd} &::= \text{is a sending } \textit{OccurrenceSpecification} \text{ of a message on a lifeline} \\
 &\quad \textit{Object_Source} \\
 \textit{Rec} &::= \text{is a receiving } \textit{OccurrenceSpecification} \text{ of a message on a lifeline} \\
 &\quad \textit{Object_Target}
 \end{aligned}$$

The main goal of our approach is to verify whether the generic behaviour described by the low-level sequence model of an architectural pattern encompasses those specified in the high-level counterpart. Thus, our approach takes as inputs a high-level sequence model *Seq_H* and a low-level sequence model *Seq_L* to verify whether the containment relationship between these models are satisfied (denoted below as: *Seq_H* \prec *Seq_L*). In particular, the containment checking algorithm will walk through the high-level sequence model to check whether its elements and structures (e.g., objects, messages, sending and receiving occurrence specifications) have corresponding parts in the low-level model. In case of inconsistencies, we need to inform the developers the particular causes of the violation and their resolution. The following rules for sending and receiving messages must be considered as the generic behaviour specification of patterns:

- The sending and receiving occurrence specifications of messages on the same object must occur in the same order in which they are described.
- A receiving occurrence specification *Rec* of a message is enabled for execution if and only if the sending occurrence *Snd* of the same message has already occurred.
- In the case that a synchronous message is sent, the source object cannot send or receive the other messages until it has received the reply message from the target object.

In order to perform containment checking, it is necessary to extract all elements that represent the generic behaviour of architectural patterns. To this end, we use a breadth-first search algorithm as shown in Algorithm 10. In this algorithm, we use two helper functions, namely, `get_objects()` and `get_interactions()`. The function `get_objects()` returns a set of objects. The function `get_interactions(i)` extract all interactions i , i.e., messages, message sorts (i.e., asynchronous, synchronous) along with sending and receiving *OccurrenceSpecifications* (*OSs*) covered by the objects in temporal order. An interaction e is called “receiving event” of i “sending event” if there is a link from i to e that synchronize with the appropriate source and target objects.

Algorithm 10 Extract Elements from UML Sequence Diagram *Seq*

```

1: procedure EXTRACT_ELEMENTS(Seq)
2:    $Q \leftarrow \emptyset$  ▷  $Q$  is the queue of non-visited interactions
3:    $V \leftarrow \emptyset$  ▷  $V$  is the queue of visited interactions
4:    $Q \leftarrow Q \cup \text{get\_objects}(\emptyset)$ 
5:   for all  $i \in Q$  do
6:      $V \leftarrow V \cup \{i\}$ 
7:      $Q \leftarrow Q \setminus \{i\}$ 
8:      $I_{\text{interactions}} \leftarrow \text{get\_interactions}(i)$ 
9:     for all  $e \in I_{\text{interactions}}$  do
10:      if ( $e \notin V$ ) then
11:         $Q \leftarrow Q \cup \{e\}$ 
12:      end if
13:    end for
14:  end for
15: end procedure

```

Algorithm 11 Matching Elements of *Seq_H*, *Seq_L*

```

1: procedure NOMISSINGELEMENTS(SeqH, SeqL);
2:   extract interactions information from SeqH, SeqL;
3:   for all  $i \in I_{\text{interactions}}$  do ▷  $I_{\text{interactions}}(p, q)$ , where  $p \in \text{Seq}_H \wedge q \in \text{Seq}_L$ 
4:      $\text{match}(p, q)$ 
5:     if  $\text{type}(p) = \text{type}(q) \wedge (p.\text{name} = q.\text{name})$  then
6:       returns true
7:     else
8:       returns false
9:     end if
10:  end for
11: end procedure

```

The containment inconsistencies occur due to several reasons, such as improper insertion of elements, deletion and/or misplacement of elements in the concrete design of the software system. The containment relationship to be validated is defined in the following Equation 8.1:

$$\begin{aligned}
 \text{Seq}_H \prec \text{Seq}_L = & \text{noMissingElemnts}(\text{Seq}_H, \text{Seq}_L) \\
 & \wedge \text{noMisplacementElements}(\text{Seq}_H, \text{Seq}_L)
 \end{aligned}
 \tag{8.1}$$

The relation symbol \prec is used to denote the containment relationship between the behavioural models. Please note that the *noMissingElements()* and *noMisplacedElements()* functions represent the (sub)tasks of containment checking. The function *noMissingElements()* aims to ensure that all the elements (e.g., objects, messages) existing in the high-level model are also present in the low-level model. That implies the behaviour described in the low-level model can embrace the expected functions defined in the high-level model. The expected output of *noMissingElements()* will be a set of elements that are described in the high-level model but missing in the low-level model. For this, name-based matching could be performed using the function *match()* described in Algorithm 11. The *match()* function takes two model elements (from high-level and low-level) as inputs and returns *true* if two elements are matched and *false* otherwise. Intuitively, two matched elements must be of the same type and have the same identifier. In order to identify the misplacement of elements, the sequence of elements of the low-level model is matched with the high-level model. If the position of elements in the high-level model are not matched with elements in the low-level model, then the preceding and succeeding elements of the unmatched elements are used to identify the misplacement of elements. The function *noMisplacedElements()* is defined in Algorithm 12.

Algorithm 12 Sequence of Elements

```

1: procedure NOMISPLACEDELEMENTS( $Seq_H, Seq_L$ );
2:   extract interactions inforamtion from  $Seq_H, Seq_L$ ;
3:   for all unmatched( $p, q$ ) do  $\triangleright$  where  $p \in Seq_H \wedge q \in Seq_L$ 
4:      $match(p, q_{succeeding\_interactions})$ 
5:     if  $p = q_{succeeding\_interactions}$  then
6:        $q \leftarrow q_{succeeding\_interactions}$ 
7:       generate violation causes and countermeasures
8:     else  $match(p, q_{preceding\_interactions})$ 
9:        $q \leftarrow q_{preceding\_interactions}$ 
10:      generate violation causes and countermeasures
11:    end if
12:  end for
13: end procedure

```

Containment checking can be done in several ways. For example, we can map both models onto formal specifications and constraints, and verify the containment relationship using model checkers. If the consistency constraints do not satisfy formal specifications then the model checker produces a counterexample as a trace of states. Another way is to apply graph-based containment checking to verify whether there are any missing nodes (i.e., missing expected functions), missing transitive links (i.e., missing execution paths), and missing cycles (i.e., missing loop executions) [TUMZ15]. In both cases, the analysis for locating the cause(s) of inconsistency is performed to aid stakeholders in identification and resolution of containment inconsistencies. The containment relationship to be validated by the graph-based algorithm is defined in the following Equation 8.2:

$$\begin{aligned}
Seq_H \prec Seq_L = & \text{noMissingNodes}(Seq_H, Seq_L) \\
& \wedge \text{noMissingTransitiveLinks}(Seq_H, Seq_L) \\
& \wedge \text{noMissingCycles}(Seq_H, Seq_L)
\end{aligned} \tag{8.2}$$

The function *match()*, which is used by *noMissingNodes()*, takes two model elements as inputs and returns *true* if two elements are matched and *false* otherwise, as shown in Algorithm 11. For the function *noMissingTransitiveLinks()* the conventional definitions of the *adjacency matrix* and *transitive closure* of a directed graph can be used. Let $G = (V, E)$ be a directed graph where V is the set of nodes and E is the ordered set of arcs. The adjacency matrix A_G of G is an $n \times n$ boolean matrix whose elements $A_G[i, j]$ is *true* if $e(i, j) \in E$ and *false* otherwise. Based on the adjacency matrix A_G , a *reachability matrix* $R_G = A_G^*$ can be derived to represent the transitive closure of G . It is denoted as $R_G[i, j] = \text{true}$ if there is a directed *path* from node i to node j and *false* otherwise. In order to define *noMissingCycles()*, Tarjan's algorithm [Tar72] can be used to obtain a set of *strongly connected components* (SCC)⁴. For more details see [TUMZ15].

8.3 Application of Our Approach to Architectural Patterns

8.3.1 Containment Checking in MODEL-VIEW-CONTROLLER

The section discusses the identification and resolution of containment inconsistencies in the MODEL-VIEW-CONTROLLER (MVC) pattern's behaviour at different levels of abstraction. In the MVC pattern the system is divided into three different parts: The *Model* concerns the object or objects that encapsulate some application data and the logic that manipulates that data independently of the user interfaces. One or multiple *Views* display a specific portion of the data to the user. The *Controller* associated with each view receives user input and translates it into a request to the model. In particular, the views and controllers constitute the user interface. The users interact strictly through the views and their controllers, independently of the model, which in turn notifies all different user interfaces about updates. There are many variations of the MVC pattern, for instance, passive model and classic MVC⁵[SLG13]. The former is used when one controller manipulates the model exclusively. The controller modifies the model and then notifies the view about the changed model, which should be updated. The later is employed when the model changes state, and it notifies the view without the controller involvement.

UML sequence diagrams need to be extended to express the specific semantics of MVC, especially to denote which objects are which participants of MVC and which kinds of messages are sent.

⁴A graph is strongly connected if every vertex is reachable from every other vertex. The strongly connected components of a directed graph form a partition into subgraphs that are themselves strongly connected.

⁵See <https://msdn.microsoft.com/en-us/library/ff649643.aspx>

Accordingly, a set of stereotypes is required to help architects and developers in correctly mapping the elements of the MVC architectural pattern to UML sequence diagrams. This will also reduce the occurrence of containment violations between the high-level and low-level behaviour models. With the consideration of semantics for sequence diagram elements (i.e., lifelines/objects and messages) and their use in modelling different variants of the MVC pattern, we selected five stereotypes from the existing vocabulary of design elements [KA08]. The «Model», «View», «Controller» stereotypes are used to extend the semantics of lifelines/objects in UML sequence diagrams for modeling the model, view and controller participants of the pattern, respectively. UML sequence diagram support asynchronous messaging; however, the semantics defined in the UML standard do not clearly define the difference between the return values from the receiver lifeline (i.e., target object). Without using the «AsynchMessage» stereotype, it is difficult to identify whether the return value is merely a notification/acknowledgement event about the receipt of a message or the actually processed data. Similarly, the «SynchMessage» stereotype is used when the source object blocks and waits to receive a response (i.e., reply message) from the target object to update the status of the operation that invoked the synchronous communication. In summary:

- The «Model» stereotype extends the Lifeline/Object metaclass of UML and contains occurrence specifications for interaction with Controller and/or View objects.
- The «View» stereotype extends the Lifeline metaclass of UML and contains occurrence specifications for interaction with Model and Controller objects.
- The «Controller» stereotype extends the Lifeline metaclass of UML and contains occurrence specifications for interaction with Model and View objects.
- The «SynchMessage» stereotype extends the Message metaclass and uses the existing UML synch-message operations to ensure that an end-to-end connection is established with the receiver lifeline (target object), which covers the receiving occurrence specification (event end). A return operation is necessary for the synchronous communication/messaging to update the status of the operation that invoked the synchronous messaging.
- The «AsynchMessage» stereotype extends the Message metaclass to ensure that the invocation flag is active when an operation is invoked. The asynchronous communication is further constrained to ensure that the method invoking the operation is not bound to receive the reply message and only a notification/acknowledgement can inform about the receipt of message.

Furthermore, there is need for additional stereotypes to cover the missing aspects concerning the containment relationships. For instance, the User/Client object is covered by any stereotypes explained so far. Furthermore, we also consider the case in which the object(s) in the low-level model are broken down into multiple entities, so that each of them not necessarily receives an update message, in particular, if the View is broken down into *mobileView* and *desktopView*, the

message will be sent to only one specific view. For this purpose we introduce the «ViewPart» stereotype. Similarly, model and controller can be broken down into sub-components.

- The «Actor» stereotype extends the Object/Lifeline metaclass of UML and contains occurrence specifications for the interaction with View.
- The «ViewPart» stereotype extends the Lifeline metaclass of UML and divides the View into parts for interaction with a Controller and/or Model.
- The «ModelPart» stereotype extends the Lifeline metaclass of UML and divides the Model into parts for interaction with a Controller and/or View.
- The «ControllerPart» stereotype extends the Lifeline metaclass of UML and divides the Controller into parts for interaction with a Model and Views.

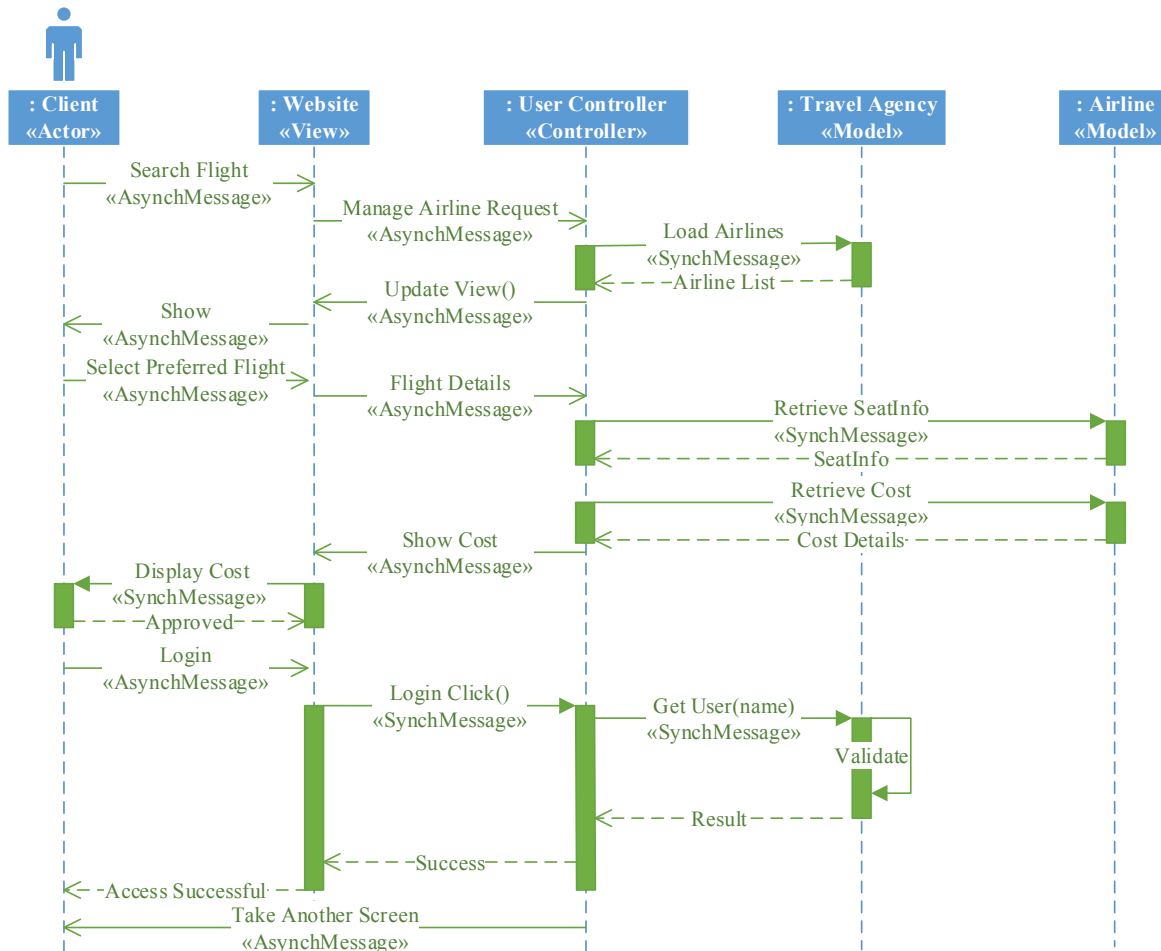


FIGURE 8.2: High-Level Model of Itinerary Management System

The high-level sequence diagram of an itinerary management system is shown in Figure 8.2. It follows the MVC pattern and involves five objects, namely client, website, user controller, travel

agency, and airline. Specifically, airline and travel agency objects represent the models; website concerns view which forward client requests to user controller; and user controller stores and retrieves data from airline and travel agency models and updates the website view accordingly. The core functionality of the itinerary management system can be described as follows: the process starts when client sends search flight request by invoking an event through the website, the user controller, in turn, contacts the travel agency model for loading airlines. The travel agency replies with a flight list to the controller which in turn is asked to update the view. When the client selects the preferred flight, the user controller asks the airline model about seat info and cost details, respectively.

The low-level itinerary management system is a refined and extended version of the high-level diagram that provides more detailed information about the system (an example of a low-level model is shown in Figure 8.3 in the context of detected violations and their causes). For instance, it contains additionally a payment model to calculate the price of an itinerary, as well as an email model and service for managing the user registration and login strategy. The low-level model may contain new messages that can be inserted in-between existing ones, or new objects and messages in parallel with existing ones, and so on, but the elements should not be inserted arbitrarily. Our containment checking for generic behaviour of the MVC pattern aims to verify whether the elements of high-level model correspond to those of a detailed design of a system.

The containment checking solution presented in the aforementioned Section 8.2.1 first verifies whether all the objects (i.e. model, view, and controller) that exist in the high-level sequence diagram of the itinerary management system (*Seq_H*) are also present in the low-level sequence diagram (*Seq_L*). For each object the respective interactions (e.g., *Client_SearchFlight_Asyn_Snd*, *Client_Show_Asyn_Rec*) are also matched. If an object present in the *Seq_H* no longer exists in the *Seq_L* it means that interactions corresponding to this object are also deleted. For this, the “missing element cause” (either one, multiple, or all elements could be missing) is detected and a corresponding countermeasure (i.e., insert the missing element at a particular position in the low-level model) is suggested. In this case, the **TakeAnotherScreen** message is sent from the UserController «Controller» to Client «Actor» present in the high-level model, but does not exist in the low-level model, can be seen as a reason for the containment violation. As we can see in Figure 8.3, the third box displays the actual cause and potential countermeasures (i.e., add **TakeAnotherScreen** after **AccessSuccessful** message – connecting UserController «Controller» to Client «Actor»).

If all objects present in *Seq_H* are also present in *Seq_L*, the next check is, whether the corresponding interactions have a different structure. The preceding and succeeding interactions of a corresponding object of *Seq_L* are matched with the interaction of *Seq_H* to locate the causes of inconsistencies. In our example, the sending *OS* of the **RetrieveCost** «SynchMessage» message and the receiving occurrence of the **CostDetails** reply message covered on the **UserController**

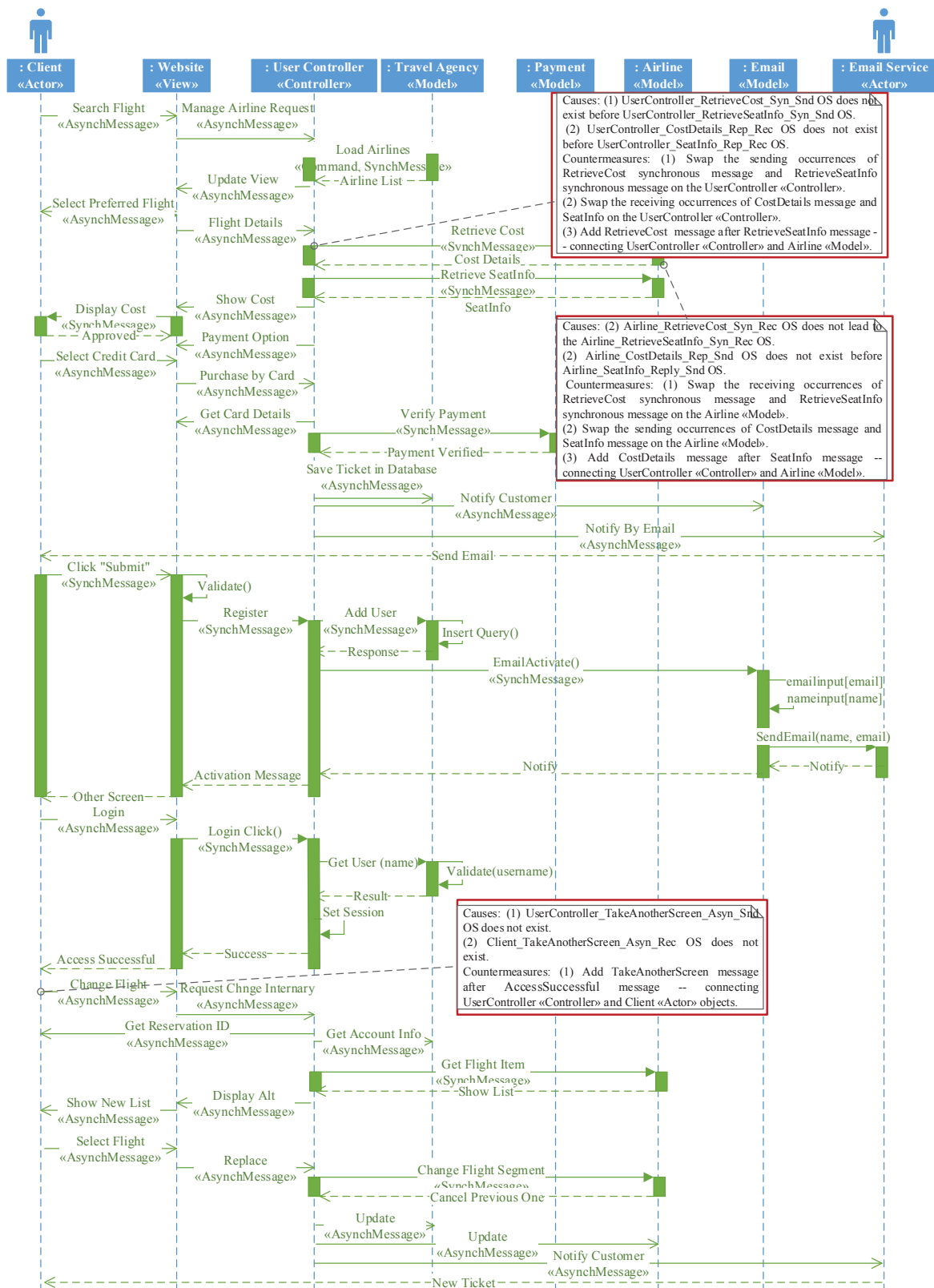


FIGURE 8.3: Feedback of Containment Results in the Low-Level Model

«Controller» are violated because the `RetrieveCost` and `CostDetails` message are sent and received prior to the sending *OS* of the `RetrieveSeatInfo` message and receiving occurrence of the `SeatInfo` message in the low-level model. Similarly, a misplacement of message occurrences exists for the `Airline` «Model». These violations can be resolved by putting the `RetrieveCost` and `CostDetails` messages after the `RetrieveSeatInfo` and `SeatInfo` messages – connecting the `UserController` «Controller» and `Airline` «Model» in the *Seq_L*. In Figure 8.3, the first and second box show the actual causes and potential countermeasures of misplacement of elements. Once the causes are located, causes are eliminated by updating the responsible elements of the low-level sequence diagram.

8.3.2 Containment Checking in LAYERS

So far, we have presented a scenario from a realistic use case representing the MVC pattern that illustrates how our proposed solution works to identify and resolve the containment inconsistencies of the MVC pattern at different levels of abstraction. As our proposed solution aims to support the software architects and/or developers to verify the containment relationship during their development tasks, it is crucial to assess whether our solution is also applicable for other architecture patterns, like the LAYERS architecture pattern, as well.

In the LAYERS pattern a system is structured into *Layers* in which each *Layer* provides a set of services to the layer above and uses the services of the layer below [AZ05]. Within each layer all constituent components work at the same level of abstraction and can interact through connectors. Between two adjacent layers a clearly defined interface is provided. In the pure form of the pattern, layers should not be by-passed: higher-level layers access lower-level layers only through the layer beneath. However, a relaxed layered scheme loosens the constraints and allows the by-passing such that a component can interact with components from any lower-level layer. The components in the layer should be organized in such a way that they share a set of common behaviours and one layer member cannot be part of multiple layers.

The generic behaviour of the LAYERS pattern at different levels of abstraction satisfies the containment relationship if elements or behaviours of the high-level model are contained in the detailed design of a LAYERS-based architecture. It is also important that the detailed design follows the same definition of LAYERS patterns as the high-level model. The semantics of UML sequence diagram elements (i.e., lifelines/objects and messages) again need to be extended for modelling the concerns of LAYERS pattern. Therefore, we used the two stereotypes «SynchMessage» and «AynchMessage» from the previous section to map the LAYERS pattern into sequence diagrams. Specifically, they support synchronous and asynchronous communication between two adjacent layers and members within a layer. The «Actor» stereotype might also be used to model the client/user. In addition, we need to ensure that interactions between lifelines residing in different layers do not allow

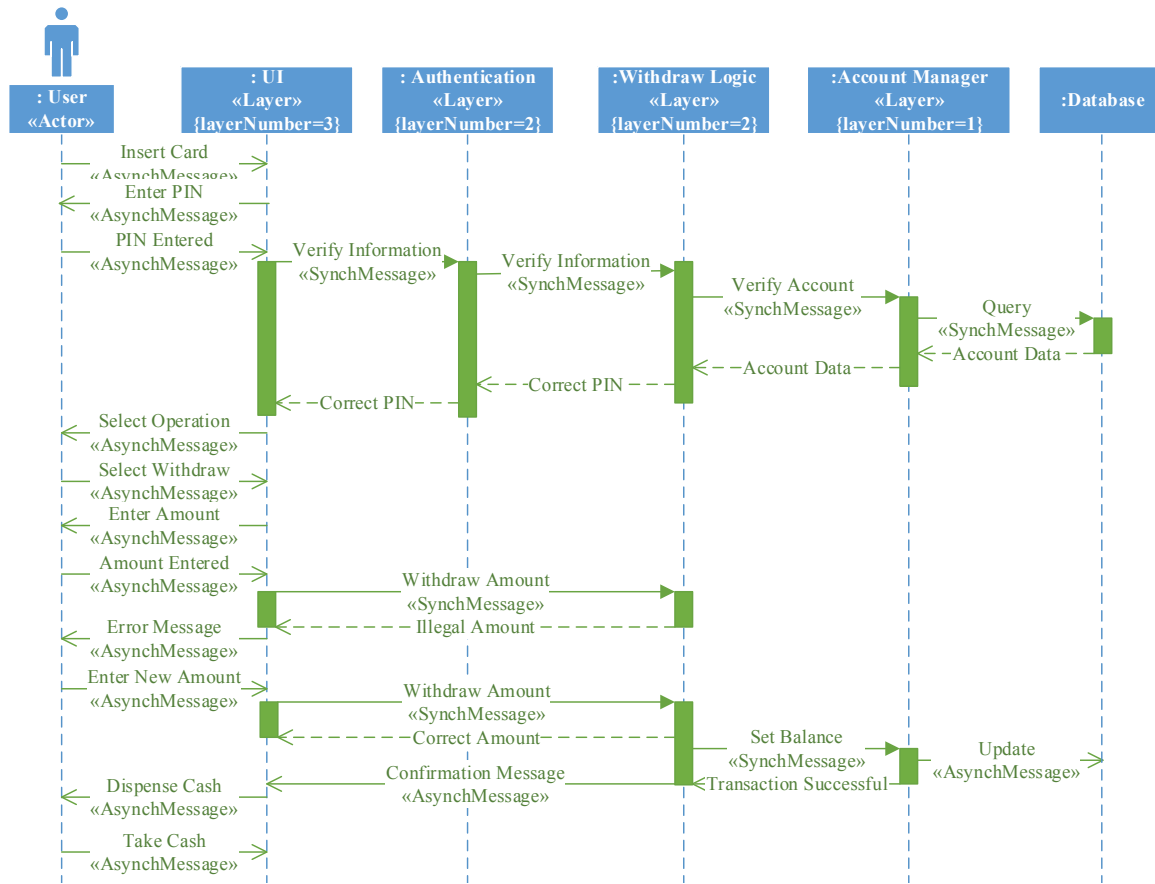


FIGURE 8.4: Modelling Layers Architecture Pattern Using Stereotypes

by-passing; also one layer member cannot be part of multiple layers. Therefore, an additional stereotype is needed to cover the missing aspects concerning these containment relationships – a similar stereotype for components is proposed in [ZA08]. Figure 8.4 shows an example of using the stereotypes used for expressing the LAYERS pattern.

- «Layer»: A stereotype that extends the Lifeline metaclass of UML and owns occurrence specifications for interaction with lower layer. The «Layer» stereotype allows lifelines who are members of the upper layer (e.g., layer N) to interact with their fellow lifelines in layer N, as well as lifelines in layer N-1 but does not allow them to communicate with other layers (e.g., N-2 and below). We also support the tag definition *layerNumber* (**+layerNumber:Integer**) for layers – representing the number of the layer in the ordered structure of layers.

Using these extensions to UML sequence diagrams it is possible to support containment checking for the explained containment relationships akin to the support for MVC explained before.

8.3.3 Containment Checking in PIPE AND FILTER

In this section, we describe the identification and resolution of containment inconsistencies in the PIPE AND FILTER pattern's behaviour at different levels of abstraction. In a PIPE AND FILTER architecture a complex task is divided into several sequential subtasks. Each of these subtasks is implemented by a separate, independent component (or filter), which handles only this task. Filters are connected through pipes, each of which transmits outputs of one filter to the inputs of another filter [Bus+96]. A Data Source produces an output stream without any input and supplies data streams to the first pipe. A Data Sink consumes an input stream but does not produce any output. The elements in the PIPE AND FILTER pattern can vary in the functions they perform. For instance, filters can be characterised into active and passive filters based on their input/output behaviour. An active filter starts processing on its own as a separate program or thread. It pulls in data and pushes out the transformed data. A passive filter is activated by being called either as a function (pull output data) or as a procedure (push input data). Pipes can buffer data between filters, form feedback loops or synchronize the filters.

Similar to the MVC and LAYERS patterns, UML sequences diagrams need to be extended in order to enable containment checking. Based on this, the containment checking can be performed to ensure that the generic behaviour described by the low-level model of a software system that is based on the PIPE AND FILTER pattern encompasses those specified in the high-level counterparts, akin to the support for the MVC pattern described above. Here, we require at least an «Filter» and «Pipe» stereotype to denote the participants of the pattern (as also introduced in [KA08]), and again the «AynchMessage» and «SynchMessage» stereotypes described above. The *Filter* stereotype is used to depict the lifelines/objects that transmit streams of data, and *Pipe* is used for message interaction from source object (filter) to target object (adjacent filter). The *AsynchMessage* and *SynchMessage* stereotypes are used to specify asynchronous and synchronous communication from one filter to the next filter in the chain, respectively. Figure 8.5 shows the stereotypes used for expressing the PIPE AND FILTER pattern in an example.

- The «Filter» stereotype extends the Lifeline metaclass of UML and covers the occurrence specification of the associated pipes.
- The «Pipe» stereotype extends the Message metaclass of UML and connects the occurrence specification of a sender lifeline to the occurrence specification of a receiver lifeline.

8.4 Related Work

This section gives an overview of existing work on modelling and formalisation of architectural patterns and consistency checking of behavioural models.

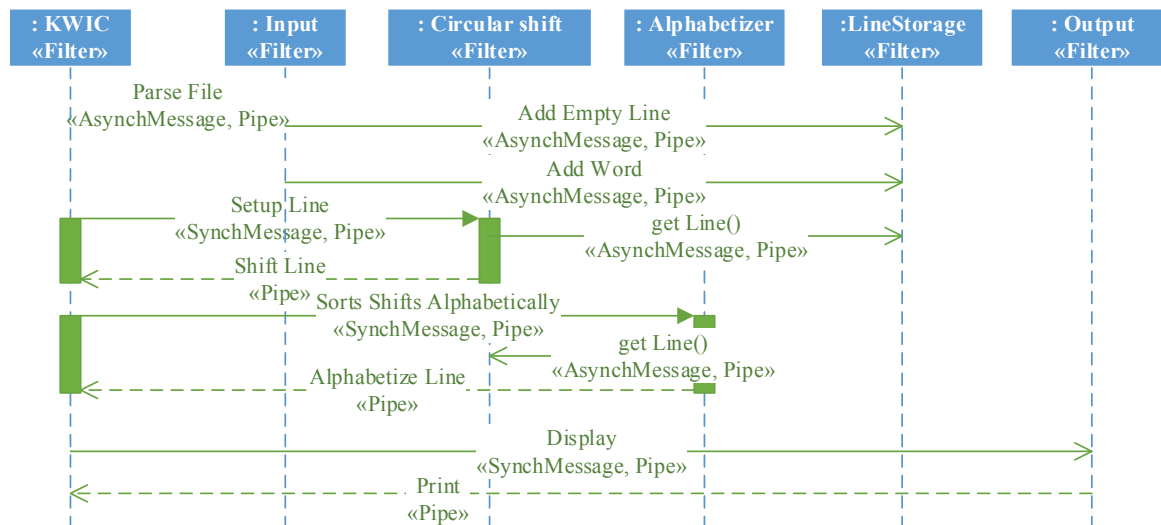


FIGURE 8.5: Modelling Pipe And Filter Pattern Using Stereotypes

8.4.1 Modelling and Formalisation of Architectural Patterns

The literature describes a number of attempts to model the structure of patterns [Gam+95; MT00; ZA08]. For instance, Giesecke et al. [Gie+07] extend the UML metamodel by creating profiles based on patterns. Their work maps the MidArch ADL to the UML metamodel for describing patterns in software design. Garlan et al. [GK00] introduce four strategies (classes and objects, classes and classes, UML components, and subsystems) for encoding the architectural elements in UML typically found in modern ADLs. Clements et al. [Cle+03] demonstrate how UML can be used to represent the fundamental architectural concepts in a number of architectural views. Selic [Sel98] describe a UML profile for real-time systems, which demonstrates several architectural concepts such as components, connectors, and ports.

Mehta and Medvidovic [MM03] propose an approach, called Alfa, for composing elements of architectural patterns using a small set of architectural primitives. In particular, they identified eight forms and nine functions as architectural primitives. Similarly, Bass et al. [BCK03] have also proposed a predefined set of unit operations, such as abstraction, compression, separation and resource sharing as the building blocks for all architectural and design patterns. Zdun et al. [ZA05; KAZ08] present a generic and extensible approach for modelling architectural patterns by means of architectural primitives. They use a vocabulary of pattern elements in parallel to architectural primitives to capture the missing semantics of architectural patterns.

There are many approaches for modelling or representing software patterns, which focuses on the design patterns from [Gam+95]. A number of such approaches attempted to formally specify the patterns (see for instance [Mik98; EHL99; SH04; MCL04]). These approaches have not been used

for architectural patterns or whole pattern languages, but just for some isolated patterns from [Gam+95].

There have also been attempts to support the modelling of patterns' behaviour in software design. For instance, Garlan et al. [GAO94] propose an object model for representing architectural designs. The authors characterise architectural patterns as specialization of the object models. Perronne et al. [PTT06] describe a modelling framework consists of two design patterns to support behaviour specification of patterns. The first, polymorphic behaviour pattern provides the integration and the execution of new behaviours for a system. The second, structured behaviour pattern provides the means to use finite state machines for behaviour switching. Kamal and Avgeriou [KA08] describe the use of primitives for systematically modelling the behaviour of architectural patterns. However, the published studies have not considered the consistency checking of architectural patterns' behaviour so far.

8.4.2 Behavioural Consistency Checking

Many approaches tackled different types of models and/or model checking techniques [SZ01; LMT09; MTZ17a]. Some of them focus on checking the consistency of behavioural models against structural models [RW03; TE00; KC02] or checking different types of behaviour models (models and other representations of the same reality such as the requirements or implementations) [LP05; YS06; GK07; Mar05]. To the best of our knowledge, very few of them consider the consistency checking problem for behaviour models at levels of abstraction. The major difference of these approaches and our approach is that we consider the consistency of the same model at different levels of abstraction, i.e., "vertical consistency" [Str05]. In particular, we focus on checking the consistency of the containment of the high-level model in the low-level model, rather than checking the consistency of elements of two different representations.

In some studies, the notion of behaviour inheritance has been studied in the realm of consistency checking of behaviour diagrams, in particular, the inheritance of object life cycles in statecharts. Stumptner and Schrefl introduce specialisations of object life cycles by examining extension and refinement in the context of UML statecharts [SS00]. Van der Aalst presents a theoretical framework for defining the semantics of behaviour inheritance [Aal02]. In this work, four different inheritance rules, based on hiding and blocking principles, are defined for UML activity diagram, statechart and sequence diagram. However, the application of these inheritance concepts in the context of actual scenarios is not clarified.

In our earlier work, we have investigated the containment checking problem for various behavioural models. Particularly, our previous research not only supports automated transformation of activity diagrams [MTZ14], sequence diagrams [MTZ16], and BPMN choreography and collaboration diagrams [MTZ15; Mur+17] into equivalent formal specifications and consistency constraints, but

also interprets the counterexamples for locating the cause(s) of inconsistencies and their resolutions [MTZ15; MTZ16; Mur+17]. Besides model checking techniques, graph-based solutions for addressing the problem of containment checking are also investigated [TUMZ15]. This research deals with the checking of architecture patterns' behaviour at different levels of abstraction, which has not been addressed so far.

8.5 Summary

Motivated by the need to address the Research Question RQ6, the central theme of this chapter focuses on the identification and resolution of containment violations in architecture patterns' behaviour at different abstraction levels. In this context, we have performed a systematic review of behaviour consistency checking research, investigated various behaviour models including activity diagrams, sequence diagrams and BPMN models, as well as possible solutions based on model-checking techniques and graph algorithms. This research help in identification of violation causes of an architectural patterns' behaviour in various activities of the development process; the software architecture, detailed design and implementation are based on the specific architecture pattern. The applicability of the proposed solution is demonstrated for MODEL-VIEW-CONTROLLER, LAYERS, and PIPE AND FILTER patterns.

This chapter presents the conclusions of this dissertation. At first, the answers to the research questions introduced in Chapter 1 are presented – they primarily concern containment checking problems in software behaviour models. After that, the major contributions of the dissertation are summarised. Finally, ongoing and future research directions are discussed.

9.1 Research Questions and Answers

Chapter 1 summarised the containment checking problems tackled in this dissertation. This led to the formulation of six research questions. They are addressed in Chapters 2 to 8. The research questions are revisited and briefly discussed below:

RQ1: What is the current state-of-the-art of software behaviour model consistency checking and potential gaps for future research?

We have performed a Systematic Literature Review (SLR) in the area of software behaviour model consistency checking. In our study, the identification and selection of the primary studies was based on a well-planned search strategy. The search process identified a total of 1770 studies, out of which 96 have been thoroughly analysed according to our predefined SLR protocol. Through in-depth analysis and interpretation of the collected data, many interesting findings along with a number of gaps and open problems are obtained that provide insights for further investigation. More specifically, the containment checking which is categorized as vertical consistency has not received special treatment in the literature. The results also indicate that a considerable number of studies either assume the existence of formal logic constraints or require manual efforts in specifying the input consistency constraints (e.g., rules specified in LTL/CTL etc.). There is also a need to improve the quality of study design and conducting evaluations to achieve solid and repeatable scientific results that have greater impact in both academia and industry. In future studies, an appropriate strategy for inconsistency handling, better tool support for consistency checking and/or development tool integration should be considered. The categories and encoding proposed in the SLR would also be considered as a guideline or recommendation for practitioners in

evaluating relevant methods and techniques as well as for researchers in designing rigorous studies and evaluations that aim for higher practical impact.

RQ2: How to perform automated transformation of behavioural diagrams into formal specifications and consistency constraints?

At the time we began working on this dissertation, none of the published studies has performed a comprehensive exploration of the containment checking problem in software behaviour models as a model checking problem. The containment checking of behaviour models using model checking techniques requires both formal consistency constraints and specifications/descriptions of these models. Unfortunately, creating formal consistency constraints and specifications was done manually, and therefore, labour-intensive and error prone.

We have investigated the problem of containment checking for activity diagrams, sequence diagrams and service choreographies at different levels of abstraction. In our proposed approaches, on the one hand, automated transformation of high-level models into LTL formulas is provided. On the other hand, low-level models, often resulting from various steps of refinement and enriching of the high-level counterparts, are transformed into formal SMV descriptions. The NuSMV model checker is employed for verifying containment relationship. Therefore, the automated translation strategy is useful to bridge the gap between manual specification of formal properties as well as consistency constraint for containment checking. In order to illustrate the applicability of the proposed approaches, we realized realistic scenarios from various domains and also evaluated the performance of approaches in particular cases. Through the evaluation of use case scenarios from industrial case studies, we also show that automated transformation of behaviour model into formal constraints and/or descriptions significantly increases the usability of formal languages in practice.

RQ3: How to verify that the behaviour (or interactions) described in the local choreography models collectively encompasses those specified in the global model?

Service choreography is a set of interrelated service interactions at a high-level of abstraction, which represents message exchanges, interaction rules and agreements between web service partners. The undesired containment violations in service choreographies would cause severe problems; for example, improper identification of services and their corresponding service providers, and therefore affect the delivery of services. Previous studies have not considered the containment relationship between global and local choreography models. Accordingly, we proposed an approach for containment checking in service choreographies that verifies whether the message exchange behaviour (or interactions) described in the joint local choreography models encompasses those specified in the global model. More specifically, a set of transformation rules are introduced to facilitate the automated transformation of global and local choreography models into LTL constraints and SMV specifications, respectively. The generated formal descriptions and properties are used by existing

model checkers for detecting any discrepancy between global and local choreography models and yield corresponding counterexamples.

RQ4: Is it possible to provide better support to the stakeholders for identification of containment problems and their resolutions?

The erroneous results reported by the model checkers (i.e., counterexamples) are not quite informative to the users. On the one hand, it requires reasonable knowledge of the underlying formalism to analyse the counterexamples. On the other hand, they show only parts of the chain of execution states leading to the cause of the inconsistency. Therefore, it is difficult for developers to track the entire evidence. This process requires considerable amount of time and effort to identify the root causes of the containment inconsistencies in order to fix the input models.

To address this problem, we introduce counterexample analysis approach for locating the root causes of a containment inconsistency and producing appropriate guidelines as countermeasures based on the information extracted from counterexample trace file, formalisation rules, and the SMV descriptions of the low-level model. Automatically analysing and presenting the root causes of inconsistencies in intuitive forms support the developers to better comprehend and resolve the problems and it also significantly reduces the time and effort of manually locating the causes of an inconsistency. The counterexample analysis problem is resolved for the activity diagrams, sequence diagrams, BPMN process models and service choreographies.

RQ5: Does graph-based containment checking provide better support for dealing with the non-determinism of decision nodes and loop nodes, as well as the state explosion problems than model checking based techniques?

Although the containment checking can be realized based on model checking, but not always the costly exhaustive searches employed by model checking are necessary for addressing the containment checking problem, leading to potentials for optimization. Accordingly, we have proposed a graph-based approach for addressing the problem of containment checking of software behaviour models at different levels of abstraction. In the approach, the input behaviour models are mapped to a formal intermediate representation that can be handled efficiently by graph search algorithms. The containment relationship is formally defined and divided into smaller problems that are resolved by three tasks: finding missing nodes, missing execution paths, and missing loops, respectively. The advantage of this divide-and-conquer strategy is twofold. On the one hand, these tasks can be performed independently, and therefore, can be parallelized to gain better performance. On the other hand, each task produces concrete and precise information about the violation of the containment relationship accordingly. The prototypical implementation of our approach performs within the boundary of $O(n^3)$ where n is the size of the inputs. The quantitative evaluation on industrial scenarios shows that the proposed approach performs reasonably on a typical working

environment and scales well within the complexity upper bound of an exhaustive model checking based approach.

RQ6: How to ensure that the behaviour of an architectural pattern is consistent across the artefacts produced in the various activities of the software development process?

Containment checking is important for architectural patterns, as they impose various kinds of design constraints on the detailed designs and implementations that should not be violated. To date, however, none of the published studies have considered the containment checking of architectural patterns' behaviour. The proposed solution aims at identifying and resolving the containment checking problems in architecture patterns' behaviour, in particular whether the generic behaviour described by the low-level model of a software system that is based on an architectural pattern encompasses those specified in the high-level counterpart of that particular pattern. In order to guide the user to follow a specific architectural pattern and its variants, we extend UML metaclasses using the UML profile mechanism. In particular, we use stereotypes to extend the properties of existing UML metaclasses. The applicability of the proposed solution is demonstrated for MODEL-VIEW-CONTROLLER, LAYERS, and PIPE AND FILTER patterns.

9.2 Major Contributions

The central theme of this dissertation focuses on a special type of vertical consistency, in particular containment checking. We have performed an SLR in the area of software behaviour model consistency checking. A total of 1770 studies have been identified by combining automated searches and manual snowballing, out of which 96 have been studied in-depth according to our predefined SLR protocol. In particular, we have studied the targeted software models, types of consistency checking, consistency checking techniques, inconsistency handling, type of study and evaluation, automation support, and practical impact.

Three model checking based techniques are proposed to address the containment checking problems in activity diagrams, sequence diagrams and service choreographies. More specifically, we introduced a set of transformation rules to facilitate the automated transformation of high-level models into formal consistency constraints (i.e., LTL formulas), whereas the low-level models are transformed into formal SMV descriptions. The formal consistency constraints and SMV descriptions are inputs to the model checker; however, the results produced by existing model checkers (e.g., counterexamples) are rather cryptic and verbose. Therefore, a counterexample analysis mechanism has been developed that provides more informative and comprehensive feedbacks to the stakeholders for identification of containment problems and their resolutions.

A lightweight graph-based approach has also been proposed because the costly exhaustive searches employed by model checking are not always necessary for addressing the containment checking

problem. First, we have investigated containment checking generically for the named design models and studied the application in realistic use case scenarios, taken mainly from enterprise information systems. Second, we have studied applying containment checking in the context of architectural patterns, in particular problems with the modelling of architectural patterns' behaviour and comprehensive feedbacks have been addressed.

9.3 Future Research Directions

Model checkers require formal descriptions and consistency constraints, which are automatically generated for activity diagrams, sequence diagrams and service choreographies. In the future, the proposed transformation rules might be adapted to support the other behaviour models such as UML state machines, communication diagrams, and EPCs. Similarly, the support for other behaviour models might be investigated for the graph-based solution. Although the prototypical implementation shows reasonable performance, further integration of improved graph algorithms, for instance for transitive closure finding and strongly connected component detection, is promising in order to gain more improvement in performance.

A controlled experiment is an analysis of a testable hypothesis in which one or more independent variables are manipulated to measure their effect on one or more dependent variables. We plan to conduct controlled experiments to empirically validate whether the proposed containment checking and counterexample analysis approaches significantly support the human analysts in identification and resolution of containment inconsistencies. Another important future work area is to develop a process related to refactoring, a catalogue of applicable refactorings, and evaluations for that catalogue of refactorings. The intention is that the architecture should not be violated during code or model refactorings.

Appendices

A

Overview of Selected Primary Studies

SID	Studied Artifacts	Semantic Domains	Checking Techniques	Studied Domains
S1	Statechart, Activity Diagram (fUML)	Process Algebra (CSP)	Model Checking	fUML
S2	Sequence Diagram, Statechart	State Transition (Transition Matrix)	Specialized Algorithm (Transition Set/Matrix, Super State Analysis)	UML
S3	Statechart	State Transition (Z)	Theorem Proving	UML
S4	Disjunctive Modal Transition Systems (DMTS)	Temporal Logic (LTL), State Transition	Model Checking+Specialized Algorithm (synthesis)	General
S5	Modal Transition System	State Transition, Quantified Boolean formulae (QBF)	Specialized Algorithm	General
S6	Statechart, Sequence Diagram	Petri Net (Generalized Stochastic PNs)	Specialized Algorithm (Analysable PNs)	UML
S7	Process Model	Guard-Action-Trigger (based on Event-Condition-Action)	Specialized Algorithm (Guard-Action-Trigger)	BPM+SOA
S8	Process Model (BPEL)	Process Algebra (D-LOTOS), Durational Action Timed Automata (DATA)	Model Checking	BPM+SOA
S9	Message Sequence Chart (MSC)	State Transition (Timed Message-Passing Automata)	Model Checking	UML
S10	Object Behavior Logic Models	State Transition (OBL)	Logical Inference	Object-Oriented Design
S11	Modal Transition Systems	Temporal Logic (CTL), State Transition	Model Checking, Specialized Algorithm (Repair)	General
S12	SCR Behavioral Requirements, Control-flow based PDL	State Transition, Graph (DFG)	Specialized Algorithm (Dataflow analysis)	SCR/PDL
S13	SDL, Message Sequence Chart (MSC)	Temporal Logic (LTL), State Transition (Finite State Automata)	Model Checking (SPIN)	Distributed Systems
S14	Service Behavioral Interface (STS)	State Transition (Symbolic Transition System STS)	Model Checking (Maude)	SOC/SOA
S15	Statechart, Sequence Diagram (UML-RT)	Process Algebra (CSP)	Model Checking	Embedded/Real-time
S16	Activity Diagram, Visual Contract	State Transition, Graph	Model Checking	BPM+SOA
S17	Activity Diagram	Typed Graph	Specialized Algorithm (Graph Transformation)	UML
S18	Activity Diagram	Temporal Logic, State Transition	Model Checking	UML

(Continued on next page)

SID	Studied Artifacts	Semantic Domains	Checking Techniques	Studied Domains
S19	Workflow	Ontology, Description Logic (DL)	Ontology Logic Reasoning	Workflow
S20	Activity Diagram (UML-MARTE)	State Transition (Time Transition System TTS)	Specialized Algorithm	Real-time
S21	Sequence Diagram, Statechart (UML/SPT)	Temporal Logic (CTL), Timed Automata	Specialized Algorithm (Schedulability Analysis)	Embedded/Real-time
S22	Statechart	Temporal Logic (CTL), State Transition	Model Checking (AGAVE)	UML
S23	Statechart	Temporal Logic (ACTL), Automata (Hierarchical Automata)	Model Checking (JACK)	UML
S24	Statechart, Sequence Diagram	State Transition (State Machine)	Specialized Algorithm	UML
S25	Sequence Diagram, Use Case	Modal Sequence Diagram (based on Harel's LSC)	Specialized Algorithm (Play-out + Synthesis)	UML
S26	Process Model (BPMN)	Description Logic, GRL (i*+NFR)	Specialized Algorithm	BPM
S27	Live Sequence Chart (LSC)	Global System Automaton	Specialized Algorithm	UML
S28	Use Case, Activity Diagram, Collaboration Diagram	State Transition (Typed Graph)	Specialized Algorithm (Graph Transformation)	UML
S29	Statechart	Process Algebra (CSP)	Model Checking (FDR)	UML
S30	Sequence Diagram, Interface Automaton	State Transition (Interface Automata)	Specialized Algorithm (Existential Consistency)	Embedded/Real-time
S31	Sequence Diagram w. Timing Constraints	State Transition (Real-time Interface Automata RIA)	Specialized Algorithm	Embedded/Real-time
S32	Statechart, Sequence Diagram	Temporal Logic (LTL), State Transition	Model Checking	Software Architecture
S33	Statechart, LTS	State Transition (Labeled Transition System)	Model Checking	UML
S34	Sequence Diagram, Activity Diagram	Temporal Logic of Actions (cTLA)	Model Checking	UML
S35	Statechart (Timed Resource-Oriented Statechart TRoS)	Process Algebra (ACSR, for Real-time)	Specialized Algorithm	Embedded/Real-time
S36	Statechart, Sequence Diagram	Timed Automata	Model Checking	Real-time
S37	Sequence Diagram	State Transition (csTS, Modal TS)	Specialized Algorithm (synthesis)	UML
S38	Statechart, Sequence Diagram (UML-RT)	State Transition	Specialized Algorithm	Embedded/Real-time
S39	Process Model (BPMN)	Temporal Logic, Petri Nets	Model Checking (GROOVE)	BPMN
S40	Statechart	Process Algebra (Pi-Calculus)	Specialized Algorithm (Weak Bisimulation)	UML
S41	Statechart, Sequence Diagram	Process Algebra (Pi-Calculus)	Specialized Algorithm (Open Bisimulation)	UML
S42	Sequence, Collaboration Diagram	Real-time Action Logic (RAL)	Logical Inference	Real-time
S43	Modal Transition System	State Transition	Specialized Algorithm	General
S44	Label Transition System, I/O FSM	State Transition, Modal Logic	Model Checking + Specialized Algorithm (Bit Vectors)	UML

(Continued on next page)

SID	Studied Artifacts	Semantic Domains	Checking Techniques	Studied Domains
S45	Process Model, Domain Ontology	Kripke Structure, Description Logic (DL)	Model Checking	BPM
S46	Statechart	State Transition (Automata)	Model Checking	UML
S47	Sequence Diagram, Statechart	Temporal Logic, State Transition (SMV Language)	Model Checking	BPM+SOA
S48	Statechart	State Transition, Operational Semantics	Specialized Algorithm (Assertion Checking)	UML
S49	Sequence Diagram	Trace Semantics	Specialized Algorithm	UML
S50	Statechart	Temporal Logic, State Transition (Kripke Structure)	Model Checking	Embedded/Real-time
S51	Abstract, Executable Process Model (BPEL)	Petri Net, Communication Graph	Specialized Algorithm (Simulation)	BPM
S52	Stateflow (Simulink)	Process Algebra (Circus:Z, CSP, Guarded Commands)	Specialized Algorithm (Simulation)	Simulink
S53	Activity Diagram	State Transition	Model Checking (NuSMV)	UML
S54	Process Model	Temporal Logic (CTL), State Transition (LTS)	Model Checking	BPM
S55	Statechart	State Transition	Specialized Algorithm (Matching+Merging, Bisimulation)	UML
S56	Statechart	State Transition (B)	Theorem Proving	UML
S57	Statechart	Temporal Logic, State Transition (PROMELA)	Model Checking	Adaptive System
S58	Statechart, Class Diagram	Process Algebra (CSP)	Model Checking (FDR)	UML
S59	Statechart	Process Algebra (CSP)	Model Checking	UML
S60	Activity Diagram, Statechart	State Transition	Specialized Algorithm (Rules)	BPM
S61	Statechart	First Order Logic	Logical Inference (CrocoPat)	UML
S62	Modal Transition System	Process Algebra (Modal Pi-Calculus)	Specialized Algorithm	Modal Transition Systems
S63	Statechart, Sequence Diagram	State Transition	Model Checking	UML
S64	Object Behavior Diagram	Petri Net (Object Behavior Diagram)	Specialized Algorithm	BPM
S65	Statechart	State Transition (Symbolic Transition System STS)	Specialized Algorithm	UML
S66	Statechart (Invariant Statechart)	State Transition	Specialized Algorithm (State Invariant)	UML
S67	Sequence Diagram, Statechart	First Order Logic	Logical Inference, Theorem Proving	UML
S68	Use Case, Activity Diagram, Statechart, Sequence Diagram	Petri Net (Colored PN)	Specialized Algorithm (PN consistency)	UML
S69	Statechart	State Transition (Timed Automata)	Specialized Algorithm (Reachability Analysis+Constraint Solving)	Real-time
S70	Stateflow (Simulink)	Graph (Binary Decision Diagram BDD)	Model Checking (Salsa)	Simulink
S71	Process Model	Petri Net	Specialized Algorithm	BPM
S72	Process Model (BPEL)	Petri Nets, BPEL Program Dependence Graph (BPD)	Specialized Algorithm	BPM

(Continued on next page)

SID	Studied Artifacts	Semantic Domains	Checking Techniques	Studied Domains
S73	Statechart	Petri Net (Object Behavior Diagram)	Specialized Algorithm (Observation Consistency)	UML
S74	Activity Diagram	Petri Net (Instantiable PN)	Specialized Algorithm	UML
S75	Use Case, Statechart	Graph	Specialized Algorithm (Graph Search)	UML
S76	Process Model (BPEL), Service Behavior (Message)	Petri Net	Specialized Algorithm (Conformance Checking)	SOC/SOA
S77	Use Case, Sequence Diagram, Statechart	State Transition (Petri Net)	Specialized Algorithm (Synthesis, Invariant)	UML
S78	Statechart, Sequence Diagram	State Transition, Traces	Model Checking	UML
S79	Workflow	Formal Workflow, First Order Logic	Specialized Algorithm (Rules)	Workflow
S80	Process Model (BPEL 2.0)	Process Algebra (Pi-Calculus)	Specialized Algorithm (Bisimulation)	BPM+SOA
S81	Process Model	Graph	Specialized Algorithm (Matching, Edit Distance)	BPM
S82	Process Model	Petri Net	Specialized Algorithm (Causal Behavioral Profiles)	BPM
S83	Process Model (BPMN,BPEL,EPC)	Petri Net	Specialized Algorithm (Behavioral Profiles)	BPM
S84	Statechart, Sequence Diagram	Temporal Logic, State Transition (Extended Sequence Diagram)	Model Checking	SOC/SOA
S85	Activity Diagram	Process Algebra (CSP)	Model Checking (FDR)	UML
S86	Statechart, Sequence Diagram	Petri Net (Extended Colored PN)	Specialized Algorithm (Coverage Checking)	UML
S87	Statechart	Process Algebra (CSP), Abstract Machine Notation (B)	Model Checking	UML
S88	WS-CDL, BPEL	Process Algebra (CSP)	Model Checking	BPM+SOA
S89	Process Model (ebXML BPSS, BPEL)	Process Algebra (Pi-Calculus)	Specialized Algorithm (Trace Refinement CSP)	BPM+SOA
S90	Statechart, Sequence Diagram	Automata (Split Automata)	Model Checking	UML
S91	Statechart, Message Sequence Chart (MSC)	Automata	Model Checking	UML
S92	Statechart, Sequence Diagram	Propositional Logic	Model checking (SAT)	UML
S93	Statechart	Temporal Logic (LTL), Automata (Extended Hierarchical Automata)	Model Checking	UML
S94	Statechart	Temporal Logic (CTL), State Transition (Kripke Structure)	Model Checking	UML
S95	Modal Transition System	State Transition	Specialized Algorithm	Modal Transition Systems
S96	Statechart	Process Algebra (CSP)	Model Checking	UML

TABLE A.1: Overview of 96 Selected Primary Studies

B

Selected Primary Studies

ID	Full Reference
S1	Abdelhalim, I. , Schneider, S. , and Treharne, H. Towards a Practical Approach to Check UML/fUML Models Consistency using CSP. <i>13th International Conference on Formal Methods and Software Engineering (ICFEM)</i> , Durham, UK, 2011, Springer, 33–48.
S2	Alanazi, M. N. and Gustafson, D. A. Super State Analysis for UML State Diagrams. <i>WRI World Congress on Computer Science and Information Engineering (CSIE)- Volume 07</i> , 2009, IEEE, 560–565.
S3	Amálio, N. , Stepney, S. , and Polack, F. Formal Proof from UML Models. <i>6th International Conference on Formal Engineering Methods (ICFEM)</i> , Seattle, WA, USA, 2004, Springer, 418–433.
S4	Beneš, N. , Černá, I. , and Křetínský, J. Modal Transition Systems: Composition and LTL Model Checking. <i>9th International Symposium on Automated Technology for Verification and Analysis (ATVA)</i> , Taipei, Taiwan, 2011, Springer, 228–242.
S5	Beneš, N. , Křetínský, J. , and Larsen, K. G. Refinement Checking on Parametric Modal Transition Systems. <i>Acta Informatica</i> , Vol. 52, No. 2-3 (2015), 269–297.
S6	Bernardi, S. , Donatelli, S. , and Merseguer, J. From UML Sequence Diagrams and Statecharts to Analysable Petri Net Models. <i>Third International Workshop on Software and Performance (WOSP)</i> , 2002, ACM, 35–45.
S7	Bhuiyan, J. , Nepal, S. , and Zic, J. Checking Conformance between Business Processes and Web Service Contract in Service Oriented Applications. <i>Australian Software Engineering Conference (ASWEC)</i> , 2006, IEEE, 80–89.
S8	Chama, I. E. , Belala, N. , and Saidouni, D.-E. FMEBP: A Formal Modeling Environment of Business Process. <i>20th International Conference on Information and Software Technologies (ICIST)</i> , Druskininkai, Lithuania, 2014, Springer, 211–223.
S9	Chandrasekaran, P. and Mukund, M. Matching Scenarios with Timing Constraints. <i>4th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS)</i> , Paris, France, 2006, Springer, 98–112.
S10	Chang, K. , Kung, D. C. , and Hsia, P. OBL: A Formal Deduction Method for Object-Oriented Systems. <i>23rd International Computer Software and Applications Conference (COMPSAC)</i> , 1999, IEEE, 450–455.
S11	Chatzieftheriou, G. , Bonakdarpour, B. , and Smolka, S. A. Abstract Model Repair. <i>4th International Symposium on NASA Formal Methods (NFM)</i> , Norfolk, VA, USA, 2012, Springer, 341–355.
(Continued on next page)	

ID	Full Reference
S12	Chechik, M. and Gannon, J. Automatic Analysis of Consistency between Requirements and Designs. <i>IEEE Trans. Softw. Eng.</i> , Vol. 27, No. 7 (2001), 651–672.
S13	D'Souza, D. and Mukund, M. Checking Consistency of SDL+MSC Specifications. <i>10th International SPIN Workshop on Model Checking Software</i> , Portland, OR, USA, 2003, Springer, 151–166.
S14	Durán, F. , Ouederni, M. , and Salaün, G. Checking Protocol Compatibility using Maude. <i>Electronic Notes in Theoretical Computer Science</i> , Vol. 255 (2009), 65–81.
S15	Engels, G. , Küster, J. M. , and Heckel, R. A Methodology for Specifying and Analyzing Consistency of Object-oriented Behavioral Models. <i>8th European Software Engineering Conference, Held jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)</i> , Vienna, Austria, 2001, ACM, 186–195.
S16	Engels, G. , Güldali, B. , and Soltenborn, C. Assuring Consistency of Business Process Models and Web Services Using Visual Contracts. <i>Applications of Graph Transformations with Industrial Relevance</i> , 17–31, (2008).
S17	Ermel, C. , Gall, J. , and Lambers, L. Modeling with Plausibility Checking: Inspecting Favorable and Critical Signs for Consistency between Control Flow and Functional Behavior. <i>14th International Conference on Fundamental Approaches to Software Engineering (FASE): part of the joint European Conferences on Theory and Practice of Software (ETAPS)</i> , Saarbrücken, Germany, 2011, Springer, 156–170.
S18	Eshuis, R. and Wieringa, R. Tool Support for Verifying UML Activity Diagrams. <i>IEEE Transactions on Software Engineering (TSE)</i> , Vol. 30, No. 7 (2004), 437–447.
S19	Fan, S. , Zhao, J. L. , and Dou, W. A Framework for Transformation from Conceptual to Logical Workflow Models. <i>Decis. Support Syst.</i> , Vol. 54, No. 1 (2012), 781–794.
S20	Ge, N. , Pantel, M. , and Crégut, X. Time Properties Dedicated Transformation from UML-MARTE Activity to Time Transition System. <i>SIGSOFT Softw. Eng. Notes</i> , Vol. 37, No. 4 (2012), 1.
S21	Gherbi, A. and Khendek, F. Consistency of UML/SPT Models. <i>13th international SDL Forum conference on Design for dependable systems (SDL)</i> , Paris, France, 2007, Springer, 203–224.
S22	Ghezzi, C. , Menghi, C. , and Molzam Sharifloo, A. On Requirement Verification for Evolving Statecharts Specifications. <i>Requirements Engineering</i> , Vol. 19, No. 3 (2014), 231–255.
S23	Gnesi, S. , Latella, D. , and Massink, M. Model Checking UML Statechart Diagrams Using JACK. <i>4th IEEE International Symposium on High-Assurance Systems Engineering</i> , 1999, , 46–55.
S24	Graaf, B. and van Deursen, A. Model-Driven Consistency Checking of Behavioural Specifications. <i>Fourth International Workshop on Model-Based Methodologies for Pervasive and Embedded Software (MOMPES)</i> , 2007, IEEE, 115–126.
S25	Greenyer, J. and Friebe, J. Consistency Checking Scenario-based Specifications of Dynamic Systems by Combining Simulation and Synthesis. <i>Fourth Workshop on Behaviour Modelling - Foundations and Applications (BM-FA)</i> , Kgs. Lyngby, Denmark, 2012, ACM, 2:1–2:9.
S26	Grøner, G. , Asadi, M. , and Mohabbati, B. Validation of User Intentions in Process Orchestration and Choreography. <i>Information Systems</i> , Vol. 43, No. 0 (2014), 83 - 99.
S27	Harel, D. and Kugler, H. Synthesizing State-Based Object Systems from LSC Specifications. <i>International Journal of Foundations of Computer Science</i> , Vol. 13, No. 01 (2002), 5–51.
(Continued on next page)	

ID	Full Reference
S28	Hausmann, J. H. , Heckel, R. , and Taentzer, G. Detection of Conflicting Functional Requirements in a Use Case-driven Approach: A Static Analysis Technique Based on Graph Transformation. <i>24th International Conference on Software Engineering (ICSE)</i> , Orlando, Florida, 2002, ACM, 105–115.
S29	Heckel, R. and Küster, J. M. Behavioral Constraints for Visual Models. <i>Electronic Notes in Theoretical Computer Science</i> , Vol. 50, No. 3 (2001), 257–265.
S30	Hu, J. , Yu, X. , Wang, L. , et al. Scenario-Based Specifications Verification for Component-Based Embedded Software Designs. <i>International Conference on Parallel Processing Workshops (ICPPW)</i> , 2005, IEEE, 240–247.
S31	Hu, J. , Yu, X. , and Zhang, Y. Checking Component-Based Embedded Software Designs for Scenario-Based Timing Specifications. <i>International Conference Embedded and Ubiquitous Computing (EUC)</i> , Nagasaki, Japan, 2005, Springer, 395–404.
S32	Inverardi, P. , Muccini, H. , and Pelliccione, P. Automated Check of Architectural Models Consistency Using SPIN. <i>16th IEEE International Conference on Automated Software Engineering (ASE)</i> , 2001, IEEE, 346–349.
S33	Junwei, D. , Zhongwei, X. , and Meng, M. Verification of Scenario-Based Safety Requirement Specification on Components Composition. <i>2008 International Conference on Computer Science and Software Engineering (CSSE)- Volume 02</i> , 2008, IEEE, 686–689.
S34	Kaliappan, P. and Koenig, H. An Approach to Synchronize UML-Based Design Components for Model-Driven Protocol Development. <i>IEEE 34th Software Engineering Workshop (SEW)</i> , Limerick, Ireland, 2011, IEEE, 27–35.
S35	Kim, J. , Kang, I. , and Choi, J.-Y. Formal Synthesis of Application and Platform Behaviors of Embedded Software Systems. <i>Software & Systems Modeling</i> , Vol. 14, No. 2 (2015), 839–859.
S36	Knapp, A. , Merz, S. , and Rauh, C. Model Checking - Timed UML State Machines and Collaborations. <i>7th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems: Co-sponsored by IFIP WG 2.2 (FTRTFT)</i> , 2002, Springer, 395–416.
S37	Krka, I. and Medvidovic, N. Component-Aware Triggered Scenarios. <i>IEEE/IFIP Conference on Software Architecture (WICSA)</i> , Sydney, NSW, Australia, 2014, IEEE, 129–138.
S38	Küster, J. M. and Stroop, J. Consistent Design of Embedded Real-Time Systems with UML-RT. <i>2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)</i> , Magdeburg, Germany, 2001, IEEE, 31–40.
S39	Kwantes, P. M. , Gorp, P. V. , and Kleijn, J. Towards Compliance Verification Between Global and Local Process Models. <i>8th International Conference on Graph Transformation ICGT, Held as Part of STAF 2015</i> , L'Aquila, Italy, 2015, Springer, 221–236.
S40	Lam, V. S. W. and Padget, J. Consistency Checking of Statechart Diagrams of a Class Hierarchy. <i>19th European conference on Object-Oriented Programming (ECOOP)</i> , Glasgow, UK, 2005, Springer, 412–427.
S41	Lam, V. S. W. and Padget, J. Consistency Checking of Sequence Diagrams and Statechart Diagrams Using the π-calculus. <i>5th International Conference on Integrated Formal Methods (IFM)</i> , Eindhoven, The Netherlands, 2005, Springer, 347–365.
(Continued on next page)	

ID	Full Reference
S42	Lano, K. Formal Specification using Interaction Diagrams. <i>Fifth IEEE International Conference on Software Engineering and Formal Methods (SEFM)</i> , 2007, IEEE, 293–304.
S43	Larsen, K. G. , Nyman, U. , and Wąsowski, A. On Modal Refinement and Consistency. <i>18th International Conference on Concurrency Theory (CONCUR)</i> , Lisbon, Portugal, 2007, Springer, 105–119.
S44	Lee, J.-D. , Jung, J.-I. , and Lee, J.-H. Verification and Conformance Test Generation of Communication Protocol for Railway Signaling Systems. <i>Comput. Stand. Interfaces</i> , Vol. 29, No. 2 (2007), 143–151.
S45	Letia, I. A. and Goron, A. Model Checking As Support for Inspecting Compliance to Rules in Flexible Processes. <i>Journal of Visual Languages & Computing</i> , Vol. 28, No. C (2015), 100 - 121.
S46	Diego, L. , Istvan, M. , and Mieke, M. Automatic Verification of a Behavioural Subset of UML Statechart Diagrams Using the SPIN Model-checker. <i>Formal Aspects of Computing</i> , Vol. 11, No. 637-664 (1999), Springer.
S47	Li, B. , Zhou, Y. , and Pang, J. Model-Driven Automatic Generation of Verified BPEL Code for Web Service Composition. <i>2009 16th Asia-Pacific Software Engineering Conference (APSEC)</i> , 2009, IEEE, 355–362.
S48	Liu, S. , Liu, Y. , and Sun, J. USMMC: A Self-contained Model Checker for UML State Machines. <i>9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)</i> , Saint Petersburg, Russia, 2013, ACM, 623–626.
S49	Lu, L. and Kim, D.-K. Required Behavior of Sequence Diagrams: Semantics and Conformance. <i>ACM Trans. Softw. Eng. Methodol.</i> , Vol. 23, No. 2 (2014), 15:1–15:28.
S50	Majzik, I. , Micskei, Z. , and Pintér, G. Development of Model Based Tools to Support the Design of Railway Control Applications. <i>26th International Conference on Computer Safety, Reliability, and Security (SAFECOMP)</i> , Nuremberg, Germany, 2007, Springer, 430–435.
S51	Martens, A. Consistency between Executable and Abstract Processes. <i>IEEE International Conference on e-Technology, e-Commerce and e-Service (EEE)</i> , 2005, IEEE, 60–67.
S52	Miyazawa, A. and Cavalcanti, A. Refinement-oriented Models of Stateflow Charts. <i>Science of Computer Programming</i> , Vol. 77, No. 10-11 (2012), 1151–1177.
S53	Muram, F. U. , Tran, H. , and Zdun, U. Automated Mapping of UML Activity Diagrams to Formal Specifications for Supporting Containment Checking. <i>11th International Workshop on Formal Engineering Approaches to Software Components and Architectures (FESCA)</i> , Grenoble, France, 2014, arXiv.org, 1–16.
S54	Nagel, B. , Gerth, C. , and Engels, G. Ensuring Consistency Among Business Goals and Business Process Models. <i>17th IEEE International Enterprise Distributed Object Computing Conference (EDOC)</i> , 2013, IEEE, 17–26.
S55	Nejati, S. , Sabetzadeh, M. , and Chechik, M. Matching and Merging of Statecharts Specifications. <i>29th International Conference on Software Engineering (ICSE)</i> , Minneapolis, MN, 2007, IEEE, 54–64.
S56	Ossami, D. D. O. , Jacquot, J.-P. , and Souquères, J. Consistency in UML and B Multi-view Specifications. <i>5th International Conference on Integrated Formal Methods (IFM)</i> , Eindhoven, The Netherlands, 2005, Springer, 386–405.
(Continued on next page)	

ID	Full Reference
S57	Ramirez, A. J. and Cheng, B. H. C. Verifying and Analyzing Adaptive Logic through UML State Models. <i>International Conference on Software Testing, Verification, and Validation (ICST)</i> , 2008, IEEE, 529–532.
S58	Rasch, H. and Wehrheim, H. Checking Consistency in UML Diagrams: Classes and State Machines. <i>6th IFIP WG 6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS)</i> , Paris, France, 2003, Springer, 229–243.
S59	Roscoe, A. W. and Wu, Z. Verifying Statechart Statecharts Using CSP and FDR. <i>8th International Conference on Formal Methods and Software Engineering (ICFEM)</i> , Macao, China, 2006, Springer, 324–341.
S60	Ryndina, K. , Küster, J. M. , and Gall, H. Consistency of Business Process Models and Object Life Cycles. <i>International Conference on Models in Software Engineering (MoDELS)</i> , Genoa, Italy, 2006, Springer, 80–90.
S61	Sabetzadeh, M. , Nejati, S. , and Easterbrook, S. Global Consistency Checking of Distributed Models with TReMer+. <i>30th International Conference on Software Engineering (ICSE)</i> , Leipzig, Germany, 2008, ACM, 815–818.
S62	Sassolas, M. , Chechik, M. , and Uchitel, S. Exploring Inconsistencies Between Modal Transition Systems. <i>Software and Systems Modeling (SoSyM)</i> , Vol. 10, No. 1 (2010), 117–142.
S63	Schäfer, T. , Knapp, A. , and Merz, S. Model Checking UML State Machines and Collaborations. <i>Electronic Notes in Theoretical Computer Science</i> , Vol. 55, No. 3 (2001), 357 - 369.
S64	Schrefl, M. and Stumptner, M. Behavior-consistent Specialization of Object Life Cycles. <i>ACM Trans. Softw. Eng. Methodol.</i> , Vol. 11, No. 1 (2002), 92–148.
S65	Schwarzl, C. and Peischl, B. Static- and Dynamic Consistency Analysis of UML State Chart Models. <i>13th International Conference on Model Driven Engineering Languages and Systems (MODELS) - Part I</i> , Oslo, Norway, 2010, Springer, 151–165.
S66	Sekerinski, E. Verifying Statecharts with State Invariants. <i>13th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS)</i> , 2008, IEEE, 7–14.
S67	Shan, L. and Zhu, H. A Formal Descriptive Semantics of UML. <i>10th International Conference on Formal Methods and Software Engineering (ICFEM)</i> , Kitakyushu-City, Japan, 2008, Springer, 375–396.
S68	Shinkawa, Y. Inter-Model Consistency in UML Based on CPN Formalism. <i>13th Asia Pacific Software Engineering Conference (APSEC)</i> , 2006, IEEE, 411–418.
S69	Shu, G. , Li, C. , and Wang, Q. Validating Objected-Oriented Prototype of Real-Time Systems with Timed Automata. <i>13th IEEE International Workshop on Rapid System Prototyping (RSP'02)</i> , 2002, IEEE, 99–106.
S70	Sims, S. , Cleaveland, R. , and Butts, K. Automated Validation of Software Models. <i>16th IEEE International Conference on Automated Software Engineering (ASE)</i> , San Diego, California, 2001, IEEE, 91–96.
S71	Smirnov, S. , Farahani, A. Z. , and Weske, M. State Propagation in Abstracted Business Processes. <i>9th International Conference on Service-Oriented Computing (ICSOC)</i> , Paphos, Cyprus, 2011, Springer, 16–31.
(Continued on next page)	

ID	Full Reference
S72	Song, W. , Zhang, W. , and Zhang, G. Quantifying Consistency Between Conceptual and Executable Business Processes. <i>IEEE International Conference on Services Computing (SCC)</i> , 2013, IEEE, 9–16.
S73	Stumptner, M. and Schrefl, M. Behavior Consistent Inheritance in UML. <i>19th International Conference on Conceptual Modeling (ER)</i> , Salt Lake City, Utah, USA, 2000, Springer, 527–542.
S74	Thierry-Mieg, Y. and Hillah, L.-M. Consistency Checking of UML Dynamic Models Based on Petri Net Techniques. <i>Innovations in Systems and Software Engineering (ISSE)</i> , Vol. 4, No. 3 (2008), 293-300.
S75	Truong, N.-T. , Tran, T.-M.-T. , and To, V.-K. Checking the Consistency between UCM and PSM Using a Graph-Based Method. <i>First Asian Conference on Intelligent Information and Database Systems (ACIIDS)</i> , 2009, IEEE, 190–195.
S76	van der Aalst, W. M. P. , Dumas, M. , and Ouyang, C. Conformance Checking of Service Behavior. <i>ACM Transactions on Internet Technology (TOIT)</i> , Vol. 8, No. 3 (2008), 1–30.
S77	van Hee, K. , Sidorova, N. , and Somers, L. Consistency in Model Integration. <i>Data & Knowledge Engineering</i> , Vol. 56, No. 1 (2006), 4–22.
S78	Wang, H. , Feng, T. , and Zhang, J. Consistency Check between Behaviour Models. <i>IEEE International Symposium on Communications and Information Technology (ISCIT)</i> , 2005, IEEE, 486-489.
S79	Wang, H. J. and Zhao, J. L. Constraint-centric Workflow Change Analytics. <i>Decis. Support Syst.</i> , Vol. 51, No. 3 (2011), 562–575.
S80	Weidlich, M. , Decker, G. , and Weske, M. Efficient Analysis of BPEL 2.0 Processes Using p-Calculus. <i>Second IEEE Asia-Pacific Service Computing Conference (APSCC)</i> , 2007, IEEE, 266–274.
S81	Weidlich, M. , Dijkman, R. , and Mendling, J. The ICoP Framework: Identification of Correspondences Between Process Models. <i>22nd International Conference on Advanced Information Systems Engineering (CAiSE)</i> , Hammamet, Tunisia, 2010, Springer, 483–498.
S82	Weidlich, M. , Polyvyanyy, A. , and Mendling, J. Causal Behavioural Profiles - Efficient Computation, Applications, and Evaluation. <i>Fundam. Inf.</i> , Vol. 113, No. 3-4 (2011), 399–435.
S83	Weidlich, M. , Mendling, J. , and Weske, M. Efficient Consistency Measurement Based on Behavioral Profiles of Process Models. <i>IEEE Trans. Softw. Eng.</i> , Vol. 37, No. 3 (2011), 410–429.
S84	Xie, Y. , Du, D. , and Liu, J. Towards the Verification of Services Collaboration. <i>33rd Annual IEEE International Computer Software and Applications Conference (COMPSAC)</i> , 2009, IEEE, 428–433.
S85	Xu, D. , Miao, H. , and Philbert, N. Model Checking UML Activity Diagrams in FDR. <i>8th IEEE/ACIS International Conference on Computer and Information Science (ICIS)</i> , Shanghai, China, 2009, IEEE, 1035 - 1040.
S86	Yao, S. and Shatz, S. M. Consistency Checking of UML Dynamic Models Based on Petri Net Techniques. <i>15th International Conference on Computing (CIC)</i> , 2006, IEEE, 289–297.
S87	Yeung, W. L. Checking Consistency between UML Class and State Models Based on CSP and B. <i>Journal of Universal Computer Science (JUSC)</i> , Vol. 10, No. 11 (2004), 1540–1559.
S88	Yeung, W. L. CSP-Based Verification for Web Service Orchestration and Choreography. <i>Simulation</i> , Vol. 83, No. 1 (2007), 65–74.
(Continued on next page)	

ID	Full Reference
S89	Yeung, W. L. A Formal Basis for Cross-Checking ebXML BPSS Choreography and Web Service Orchestration. <i>IEEE Asia-Pacific Services Computing Conference (APSCC)</i> , 2008, IEEE, 524–529.
S90	Zhao, X. , Long, Q. , and Qiu, Z. Model Checking Dynamic UML Consistency. <i>8th International Conference on Formal Methods and Software Engineering (ICFEM)</i> , Macao, China, 2006, Springer, 440–459.
S91	Knapp, A. and Wuttke, J. Model Checking of UML 2.0 Interactions. <i>Proceedings of the 2006 International Conference on Models in Software Engineering (MoDELS)</i> , Genoa, Italy, 2006, Springer, 42–51.
S92	Kaufmann, P. , Kronegger, M. , Pfandler, A. , Seidl, M. , and Widl, M. A SAT-Based Debugging Tool for State Machines and Sequence Diagrams. <i>7th International Conference on Software Language Engineering (SLE)</i> , Västerås, Sweden, 2014, Springer, 21–40.
S93	Dong, W. , Wang, J. , Qi, X. , and Qi, Z. Model Checking UML Statecharts. <i>8th Asia-Pacific Software Engineering Conference (APSEC)</i> , Macao, China, 2001, IEEE, 363 - 370.
S94	Zhao, Q. , and Krogh, B. H. Formal Verification of Statecharts Using Finite-State Model Checkers. <i>IEEE Transactions on Control Systems Technology</i> , Vol. 14, 2006, IEEE, 943–950.
S95	Fischbein, D. and Uchitel, S. On Correct and Complete Strong Merging of Partial Behaviour Models. <i>16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)</i> , New York, NY, USA, 2008, ACM, 297–307.
S96	Zhang, S. J. and Liu, Y. An Automatic Approach to Model Checking UML State Machines. <i>4th International Conference on Secure Software Integration and Reliability Improvement Companion (SSIRI-C)</i> , Washington, DC, USA, 2010, IEEE, 1–6.

TABLE B.1: List of Selected Primary Studies

C

Data Extraction: Data Items and Encoding

TABLE C.1: Evidence of Timing Support

Level	Description
1	Not considered
2	Implicit using of underlying timing model or rules, for instance, providing a way to represent temporal information using timers, temporal logics, CSP, etc.
3	Explicit timing model and analysis, for instance, by using explicitly timed models or real-time constraints

TABLE C.2: Evidence of Inconsistency Handling (adapted from [SZ01])

Level	Description
1	Not mentioned / Not considered
2	Systematic inconsistency diagnosis
3	Identifying handling actions
4	Evaluating costs and risks
5	Automated action selection and execution

TABLE C.3: Evidence of Automation Support

Level	Description
1	Manual: requiring human interactions or manually specifying rules, constraints
2	Semi-automated: assuming existing input models, partially human interaction
3	Fully automated

TABLE C.4: Evidence of Development Tool Support and Integration

Level	Description
1	Not mentioned / Not considered
2	Proposed/planned integration
3	Fully implemented integration

TABLE C.5: Evidence of Tool Support for Consistency Checking

Level	Description
1	Not mentioned / Not considered
2	Only using existing tools / libraries
3	Prototypes (including using existing tools / libraries)

TABLE C.6: Level of Empirical Evidence (adapted from [Kit04; Alv+10])

Level	Description
1	No evidence.
2	Evidence obtained from demonstration or working out toy examples.
3	Evidence obtained from expert opinions or observations.
4	Evidence obtained from academic studies, e.g., controlled experiments
5	Evidence obtained from industrial studies.
6	Evidence obtained from industrial practice.

TABLE C.7: Type of Study and Evaluation (adapted from [CA11])

Type	Description
Rigorous analysis (RA)	Rigorous derivation and proof, suited for formal model
Case study (CS)	An empirical inquiry that investigates a contemporary phenomenon within its real-life context; when the boundaries between phenomenon and context are not clearly evident; and in which multiple sources of evidence are used
Discussion (DC)	Provided some qualitative, textual, opinion
Example (EX)	Authors describing an application and provide an example to assist in the description, but the example is “used to validate” or “evaluate” as far as the authors suggest
Experience Report (ER)	The result has been used on real examples, but not in the form of case studies or controlled experiments, the evidence of its use is collected informally or formally
Field study (FS)	Controlled experiment performed in industry settings
Laboratory experiment with human subjects (LH)	Identification of precise relationships between variables in a designed controlled environment using human subjects and quantitative techniques
Laboratory experiment with software subjects (LS)	A laboratory experiment to compare the performance of newly proposed system with other existing systems
Simulation (SI)	Execution of a system with artificial data, using a model of the real word
Not mentioned	<i>These types of studies have been excluded</i>

TABLE C.8: Rigour (adapted from [IG11])

Aspect	Description		
	Level=Weak	Level=Medium	Level=Strong
Context described (C)	There appears to be no description of the context in which the evaluation is performed.	The context in which the study is performed is mentioned or presented in brief but not described to the degree to which a reader can understand and compare it to another context.	The context is described to the degree where a reader can understand and compare it to another context.
Study design described (S)	There appears to be no description of the design of the presented evaluation.	The study design is briefly described, e.g., “ten students did step 1, step 2 and step 3”.	The study design is described to the degree where a reader can understand, e.g., the variables measured, the control used, the treatments, the selection/sampling used etc.
Validity discussed (V)	There appears to be no description of any threats to validity of the evaluation.	The validity of the study is mentioned but not described in detail.	The validity of the evaluation is discussed in detail where threats are described and measures to limit them are detailed.

Bibliography

- [Aal+08] Wil M. P. van der Aalst, Marlon Dumas, Chun Ouyang, Anne Rozinat, and Eric Verbeek. “Conformance Checking of Service Behavior.” In: *ACM Transactions on Internet Technology* 8.3 [2008], pp. 1–30. DOI: [10.1145/1361186.1361189](https://doi.org/10.1145/1361186.1361189).
- [Aal02] Wil M. P. van der Aalst. “Inheritance of Dynamic Behaviour in UML.” In: *Proceedings of the 2nd International Workshop on Modelling of Objects, Components and Agents*. MOCA '02. Aarhus, Denmark, 2002, pp. 105–120.
- [Aal13] Wil M. P. van der Aalst. “Business Process Management: A Comprehensive Survey.” In: *ISRN Software Engineering* 2013 [2013], pp. 1–37. DOI: [10.1155/2013/507984](https://doi.org/10.1155/2013/507984).
- [AC+14] Abel Armas-Cervantes, Paolo Baldan, Marlon Dumas, and Luciano García-Bañuelos. “Behavioral Comparison of Process Models Based on Canonically Reduced Event Structures.” In: *Proceedings of the 12th International Conference on Business Process Management*. Vol. 8659. BPM '14. Haifa, Israel, 2014, pp. 267–282. DOI: [10.1007/978-3-319-10172-9_17](https://doi.org/10.1007/978-3-319-10172-9_17).
- [ADW08] Ahmed Awad, Gero Decker, and Mathias Weske. “Efficient Compliance Checking Using BPMN-Q and Temporal Logic.” In: *Proceedings of the 6th International Conference on Business Process Management*. BPM '08. Milan, Italy, 2008, pp. 326–341. DOI: [10.1007/978-3-540-85758-7_24](https://doi.org/10.1007/978-3-540-85758-7_24).
- [AGR16] Paolo Arcaini, Angelo Gargantini, and Elvinia Riccobene. “SMT-Based Automatic Proof of ASM Model Refinement.” In: *Proceedings of the 14th International Conference on Software Engineering and Formal Methods, Held as Part of STAF 2016*. SEFM '16. Vienna, Austria, 2016, pp. 253–269. DOI: [10.1007/978-3-319-41591-8_17](https://doi.org/10.1007/978-3-319-41591-8_17).
- [AIM10] Luigi Atzori, Antonio Iera, and Giacomo Morabito. “The Internet of Things: A survey.” In: *Computer Networks* 54.15 [2010], pp. 2787–2805. DOI: [10.1016/j.comnet.2010.05.010](https://doi.org/10.1016/j.comnet.2010.05.010).

- [Ala+06] Luay Alawneh, Mourad Debbabi, Yosr Jarraya, Andrei Soeanu, and Fawzi Hassayne. “A Unified Approach for Verification and Validation of Systems and Software Engineering Models.” In: *Proceedings of the 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems*. ECBS '06. Potsdam, Germany, 2006, pp. 409–418. DOI: [10.1109/ECBS.2006.17](https://doi.org/10.1109/ECBS.2006.17).
- [Alv+10] Vander Alves, Nan Niu, Carina Alves, and George Valença. “Requirements Engineering for Software Product Lines: A Systematic Literature Review.” In: *Information and Software Technology* 52.8 [2010], pp. 806–820. DOI: [10.1016/j.infsof.2010.03.014](https://doi.org/10.1016/j.infsof.2010.03.014).
- [AMW06] W. M. P. van der Aalst, A. K. Alves de Medeiros, and A. J. M. M. Weijters. “Process Equivalence: Comparing Two Process Models Based on Observed Behavior.” In: *Proceedings of the 4th International Conference on Business Process Management*. BPM' 06. Vienna, Austria, 2006, pp. 129–144. DOI: [10.1007/11841760_10](https://doi.org/10.1007/11841760_10).
- [ASP04] Nuno Amálio, Susan Stepney, and Fiona Polack. “Formal Proof from UML Models.” In: *Proceedings of the 6th International Conference on Formal Engineering Methods*. ICFEM '04. Seattle, WA, USA, 2004, pp. 418–433. DOI: [10.1007/978-3-540-30482-1_35](https://doi.org/10.1007/978-3-540-30482-1_35).
- [AZ05] Paris Avgeriou and Uwe Zdun. “Architectural Patterns Revisited - A Pattern Language.” In: *Proceedings of the Tenth European Conference on Pattern Languages of Programs*. EuroPLOP '05. Irsee, Germany, 2005, pp. 431–470. ISBN: 978-3-87940-805-4.
- [Bae+07] Joonsoo Bae, Ling Liu, James Caverlee, Liang-jie Zhang, and Hyerim Bae. “Development of Distance Measures for Process Mining, Discovery and Integration.” In: *International Journal of Web Services Research* 4.4 [2007], pp. 1–17. DOI: [10.4018/jwsr.2007100101](https://doi.org/10.4018/jwsr.2007100101).
- [BCK03] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. 2nd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003. ISBN: 0321154959.
- [BF14] Marco Brambilla and Piero Fraternali. “Large-scale Model-Driven Engineering of Web User Interaction.” In: *Science of Computer Programming* 89, Part B [2014], pp. 71–87. DOI: [10.1016/j.scico.2013.03.010](https://doi.org/10.1016/j.scico.2013.03.010).
- [BL12] Michael Becker and Ralf Laue. “A Comparative Survey of Business Process Similarity Measures.” In: *Computers in Industry* 63.2 [2012], pp. 148–167. DOI: [10.1016/j.compind.2011.11.003](https://doi.org/10.1016/j.compind.2011.11.003).

- [BNR03] Thomas Ball, Mayur Naik, and Sriram K. Rajamani. “From Symptom to Cause: Localizing Errors in Counterexample Traces.” In: *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’03. New Orleans, Louisiana, USA, 2003, pp. 97–105. DOI: [10.1145/604131.604140](https://doi.org/10.1145/604131.604140).
- [Boi+00] Eerke Boiten, Howard Bowman, John Derrick, Peter Linington, and Maarten Steen. “Viewpoint Consistency in ODP.” In: *Computer Networks* 34.3 [2000], pp. 503–537. DOI: [10.1016/S1389-1286\(00\)00114-6](https://doi.org/10.1016/S1389-1286(00)00114-6).
- [Bre+07] Pearl Brereton, Barbara A. Kitchenham, David Budgen, Mark Turner, and Mohamed Khalil. “Lessons from Applying the Systematic Literature Review Process Within the Software Engineering Domain.” In: *Journal of Systems and Software* 80.4 [2007], pp. 571–583. DOI: [10.1016/j.jss.2006.07.009](https://doi.org/10.1016/j.jss.2006.07.009).
- [Bud+11] David Budgen, Andy J Burn, O Pearl Brereton, Barbara A Kitchenham, and Rialette Pretorius. “Empirical Evidence About the UML: A Systematic Literature Review.” In: *Software—Practice & Experience* 41.4 [2011], pp. 363–392. DOI: [10.1002/spe.1009](https://doi.org/10.1002/spe.1009).
- [Bur+92] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. “Symbolic Model Checking: 10^{20} States and Beyond.” In: *Information and Computation* 98.2 [1992], pp. 142–170. DOI: [10.1016/0890-5401\(92\)90017-A](https://doi.org/10.1016/0890-5401(92)90017-A).
- [Bus+06] Nadia Busi, Roberto Gorrieri, Claudio Guidi, Roberto Lucchi, and Gianluigi Zavattaro. “Choreography and Orchestration Conformance for System Design.” In: *Proceedings of the 8th International Conference on Coordination Models and Languages*. COORDINATION ’06. Bologna, Italy, 2006, pp. 63–81. DOI: [10.1007/11767954_5](https://doi.org/10.1007/11767954_5).
- [Bus+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture - Volume 1: A System of Patterns*. Wiley Publishing, 1996. ISBN: 0471958697, 9780471958697.
- [BW84] Victor R. Basili and David M. Weiss. “A Methodology for Collecting Valid Software Engineering Data.” In: *IEEE Transactions on Software Engineering* 10.6 [1984], pp. 728–738. DOI: [10.1109/TSE.1984.5010301](https://doi.org/10.1109/TSE.1984.5010301).
- [CA11] Lianping Chen and Muhammad Ali Babar. “A Systematic Review of Evaluation of Variability Management Approaches in Software Product Lines.” In: *Information and Software Technology* 53.4 [2011], pp. 344–362. DOI: [10.1016/j.infsof.2010.12.006](https://doi.org/10.1016/j.infsof.2010.12.006).
- [Cav+05] Roberto Cavada, Alessandro Cimatti, Charles Jochim, Gavin Keighren, Emanuele Olivetti, Marco Pistore, Marco Roveri, and Andrei Tchaltsev. *NuSMV 2.5 User Manual*. <http://nusmv.fbk.eu/NuSMV/userman/v25/nusmv.pdf>. Last accessed: 2017-01-24. 2005.

- [CE81] Edmund M. Clarke and E. Allen Emerson. “Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic.” In: *Logics of Programs, Workshop*. Yorktown Heights, New York, 1981, pp. 52–71. DOI: [10.1007/BFb0025774](https://doi.org/10.1007/BFb0025774).
- [CGP99] Edmund M. Clarke Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. Cambridge, MA, USA: MIT Press, 1999. ISBN: 0-262-03270-8.
- [Cim+00] Alessandro Cimatti, Edmund Clarke, Fausto Giunchiglia, and Marco Roveri. “NUSMV: A New Symbolic Model Checker.” In: *International Journal on Software Tools for Technology Transfer* 2.4 [2000], pp. 410–425. DOI: [10.1007/s100090050046](https://doi.org/10.1007/s100090050046).
- [Cim+99] Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. “NUSMV: A New Symbolic Model Verifier.” In: *Proceedings of the 11th International Conference on Computer Aided Verification*. CAV ’99. Trento, Italy, 1999, pp. 495–499. DOI: [10.1007/3-540-48683-6_44](https://doi.org/10.1007/3-540-48683-6_44).
- [Cla+11] Edmund M. Clarke, William Klieber, Miloš Nováček, and Paolo Zuliani. “Model Checking and the State Explosion Problem.” In: *International Summer School on Tools for Practical Software Verification (LASER) - Revised Tutorial Lectures*. Elba Island, Italy, 2011, pp. 1–30. DOI: [10.1007/978-3-642-35746-6_1](https://doi.org/10.1007/978-3-642-35746-6_1).
- [Cla+95] E. M. Clarke, O. Grumberg, K. L. McMillan, and X. Zhao. “Efficient Generation of Counterexamples and Witnesses in Symbolic Model Checking.” In: *Proceedings of the 32nd Annual ACM/IEEE Design Automation Conference*. DAC ’95. San Francisco, California, USA: ACM, 1995, pp. 427–432. DOI: [10.1145/217474.217565](https://doi.org/10.1145/217474.217565).
- [Cla+96] Edmund M. Clarke, Kenneth L. McMillan, Sérgio Vale Aguiar Campos, and Vassili Hartonas-Garmhausen. “Symbolic Model Checking.” In: *Proceedings of the 8th International Conference on Computer Aided Verification*. CAV ’96. New Brunswick, NJ, USA, 1996, pp. 419–427. DOI: [10.1007/3-540-61474-5_93](https://doi.org/10.1007/3-540-61474-5_93).
- [Cle+03] Paul Clements, David Garlan, Reed Little, Robert Nord, and Judith Stafford. “Documenting Software Architectures: Views and Beyond.” In: *Proceedings of the 25th International Conference on Software Engineering*. ICSE ’03. Portland, Oregon, 2003, pp. 740–741. DOI: [10.1109/ICSE.2003.1201264](https://doi.org/10.1109/ICSE.2003.1201264).
- [CW96] Edmund M. Clarke and Jeannette M. Wing. “Formal Methods: State of the Art and Future Directions.” In: *ACM Computing Surveys* 28.4 [1996], pp. 626–643. DOI: [10.1145/242223.242257](https://doi.org/10.1145/242223.242257).
- [DAC98] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. “Property Specification Patterns for Finite-state Verification.” In: *Proceedings of the Second Workshop on Formal Methods in Software Practice*. FMSP ’98. Clearwater Beach, Florida, USA, 1998, pp. 7–15. DOI: [10.1145/298595.298598](https://doi.org/10.1145/298595.298598).

- [DDGBn09] Remco Dijkman, Marlon Dumas, and L. García-Bañuelos. “Graph Matching Algorithms for Business Process Model Similarity Search.” In: *Proceedings of the 7th International Conference on Business Process Management - Volume 5701*. BPM 2009. Ulm, Germany, 2009, pp. 48–63. DOI: [10.1007/978-3-642-03848-8_5](https://doi.org/10.1007/978-3-642-03848-8_5).
- [Dij+11] Remco Dijkman, Marlon Dumas, Boudewijn van Dongen, Reina Käärk, and Jan Mendling. “Similarity of Business Process Models: Metrics and Evaluation.” In: *Information Systems* 36.2 [2011], pp. 498–516. DOI: [10.1016/j.is.2010.09.006](https://doi.org/10.1016/j.is.2010.09.006).
- [DM03] Deepak D’Souza and Madhavan Mukund. “Checking Consistency of SDL+MSC Specifications.” In: *Proceedings of the 10th International Conference on Model Checking Software*. SPIN’03. Portland, OR, USA, 2003, pp. 151–166. DOI: [10.1007/3-540-44829-2_10](https://doi.org/10.1007/3-540-44829-2_10).
- [DRS03] Yifei Dong, C.R. Ramakrishnan, and S.A. Smolka. “Model Checking and Evidence Exploration.” In: *Proceedings of the 10th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems*. ECBS’03. Huntsville, Alabama, USA, 2003, pp. 214–223. DOI: [10.1109/ECBS.2003.1194802](https://doi.org/10.1109/ECBS.2003.1194802).
- [DS03] Yang Dong and Zhang Shensheng. “Using pi-Calculus to Formalize UML Activity Diagram for Business Process Modeling.” In: *Proceedings of the 10th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems*. ECBS’03. Huntsville, Alabama, USA, 2003, pp. 47–54. DOI: [10.1109/ECBS.2003.1194782](https://doi.org/10.1109/ECBS.2003.1194782).
- [DW07] Gero Decker and Mathias Weske. “Local Enforceability in Interaction Petri Nets.” In: *Proceedings of the 5th International Conference on Business Process Management*. BPM’07. Brisbane, Australia, 2007, pp. 305–319. DOI: [10.1007/978-3-540-75183-0_22](https://doi.org/10.1007/978-3-540-75183-0_22).
- [EG07] Rik Eshuis and Paul Grefen. “Structural Matching of BPEL Processes.” In: *Proceedings of the 5th European Conference on Web Services*. ECOWS ’07. Halle (Saale), Germany, 2007, pp. 171–180. DOI: [10.1109/ECOWS.2007.22](https://doi.org/10.1109/ECOWS.2007.22).
- [Egy02] Alexander Egyed. “Automated Abstraction of Class Diagrams.” In: *ACM Transactions on Software Engineering and Methodology* 11.4 [2002], pp. 449–491. DOI: [10.1145/606612.606616](https://doi.org/10.1145/606612.606616).
- [EHK01] Gregor Engels, Reiko Heckel, and JochenMalte Küster. “Rule-Based Specification of Behavioral Consistency Based on the UML Meta-model.” In: *Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*. Vol. 2185. UML ’01. Toronto, Canada, 2001, pp. 272–286. DOI: [10.1007/3-540-45441-1_21](https://doi.org/10.1007/3-540-45441-1_21).

- [EHL99] Amnon H. Eden, Yoram Hirshfeld, and Kristina Lundqvist. “LePUS - Symbolic Logic Modeling Of Object Oriented Architectures: A Case Study.” In: *Proceedings of the Second Nordic Workshop on Software Architecture, University of Karlskrona/Ronneby*. NOSA '99. Citeseer, 1999.
- [EKG02] Gregor Engels, Jochen M Küster, and Luuk Groenwegen. “Consistent Interaction Of Software Components.” In: *Journal of Integrated Design & Process Science - Component-Based System Development* 6.4 [2002], pp. 2–22. ISSN: 1092-0617.
- [Eng+01] Gregor Engels, Jochem M. Küster, Reiko Heckel, and Luuk Groenewegen. “A Methodology for Specifying and Analyzing Consistency of Object-oriented Behavioral Models.” In: *Proceedings of the 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ESEC/FSE-9. Vienna, Austria, 2001, pp. 186–195. DOI: [10.1145/503209.503235](https://doi.org/10.1145/503209.503235).
- [Eng+08] Gregor Engels, Baris Güldali, Christian Soltenborn, and Heike Wehrheim. “Assuring Consistency of Business Process Models and Web Services Using Visual Contracts.” In: *Proceedings of the International Symposium on Applications of Graph Transformations with Industrial Relevance*. AGTIVE '08. Kassel, Germany, 2008, pp. 17–31. DOI: [10.1007/978-3-540-89020-1_2](https://doi.org/10.1007/978-3-540-89020-1_2).
- [Esh06] Rik Eshuis. “Symbolic Model Checking of UML Activity Diagrams.” In: *ACM Transactions on Software Engineering and Methodology* 15.1 [2006], pp. 1–38. DOI: [10.1145/1125808.1125809](https://doi.org/10.1145/1125808.1125809).
- [EW01] Rik Eshuis and Roel Wieringa. *A Formal Semantics for UML Activity Diagrams - Formalising Workflow Models*. CTIT technical report series. Enschede: Centre for Telematics and Information Technology, University of Twente, 2001.
- [EW02] Rik Eshuis and Roel Wieringa. “Verification Support for Workflow Design with UML Activity Graphs.” In: *Proceedings of the 24th International Conference on Software Engineering*. ICSE '02. Orlando, Florida, 2002, pp. 166–176. DOI: [10.1145/581339.581362](https://doi.org/10.1145/581339.581362).
- [F+07] Alexander Förster, Gregor Engels, Tim Schattkowsky, and Ragnhild Van Der Straeten. “Verification of Business Process Quality Constraints based on Visual Process Patterns.” In: *Proceedings of the First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering*. TASE '07. Shanghai, China, 2007, pp. 197–206. DOI: [10.1109/TASE.2007.56](https://doi.org/10.1109/TASE.2007.56).
- [FBS04] Xiang Fu, Tevfik Bultan, and Jianwen Su. “WSAT: A Tool for Formal Analysis of Web Services.” In: *Proceedings of the 16th International Conference on Computer Aided Verification*. CAV '04. Boston, MA, USA, 2004, pp. 510–514. DOI: [10.1007/978-3-540-27813-9_48](https://doi.org/10.1007/978-3-540-27813-9_48).

- [Fin+93] Anthony Finkelsteiin, Dov M. Gabbay, Anthony Hunter, Jeff Kramer, and Bashar Nuseibeh. “Inconsistency Handling in Multi-Perspective Specifications.” In: *Proceedings of the 4th European Software Engineering Conference on Software Engineering*. ESEC ’93. Garmisch-Partenkirchen, Germany, 1993, pp. 84–99. DOI: [10.1007/3-540-57209-0_7](https://doi.org/10.1007/3-540-57209-0_7).
- [FK07] Lukasz Fryz and Leszek Kotulski. “Assurance of System Consistency During Independent Creation of UML Diagrams.” In: *Proceedings of the 2nd International Conference on Dependability of Computer Systems*. DEPCOS-RELCOMEX ’07. Szklarska Poreba, Poland, 2007, pp. 51–58. DOI: [10.1109/DEPCOS-RELCOMEX.2007.11](https://doi.org/10.1109/DEPCOS-RELCOMEX.2007.11).
- [FKV94] Martin D. Fraser, Kuldeep Kumar, and Vijay K. Vaishnavi. “Strategies for Incorporating Formal Specifications in Software Development.” In: *Communications of the ACM* 37.10 [1994], pp. 74–86. DOI: [10.1145/194313.194399](https://doi.org/10.1145/194313.194399).
- [FLP99] Pascal Fradet, Daniel Le Métayer, and Michaël Périn. “Consistency Checking for Multiple View Software Architectures.” In: *ACM SIGSOFT Software Engineering Notes* 24.6 [1999], pp. 410–428. DOI: [10.1145/318774.319258](https://doi.org/10.1145/318774.319258).
- [Fos+05] Howard Foster, Sebastian Uchitel, Jeff Magee, and Jeff Kramer. “Tool Support for Model-Based Engineering of Web Service Compositions.” In: *Proceedings of the IEEE International Conference on Web Services*. ICWS ’05. Orlando, FL, USA, 2005, pp. 95–102. DOI: [10.1109/ICWS.2005.119](https://doi.org/10.1109/ICWS.2005.119).
- [Fra02] David Frankel. *Model Driven Architecture: Applying MDA to Enterprise Computing*. New York, NY, USA: John Wiley & Sons, Inc., 2002. ISBN: 0-471-31920-1.
- [Gab87] Dov M. Gabbay. “The Declarative Past and Imperative Future: Executable Temporal Logic for Interactive Systems.” In: *Temporal Logic in Specification*. 1987, pp. 409–448. ISBN: 3-540-51803-7.
- [Gal+14] Matthias Galster, Danny Weyns, Dan Tofan, Bartosz Michalik, and Paris Avgeriou. “Variability in Software Systems—A Systematic Literature Review.” In: *IEEE Transactions on Software Engineering* 40.3 [2014], pp. 282–306. DOI: [10.1109/TSE.2013.56](https://doi.org/10.1109/TSE.2013.56).
- [Gam+95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0-201-63361-2.
- [GAO94] David Garlan, Robert Allen, and John Ockerbloom. “Exploiting Style in Architectural Design Environments.” In: *Proceedings of the 2Nd ACM SIGSOFT Symposium on Foundations of Software Engineering*. SIGSOFT ’94. New Orleans, Louisiana, USA, 1994, pp. 175–188. DOI: [10.1145/193173.195404](https://doi.org/10.1145/193173.195404).

- [GD07] Bas Graaf and Arie van Deursen. “Model-Driven Consistency Checking of Behavioural Specifications.” In: *Proceedings of the Fourth International Workshop on Model-Based Methodologies for Pervasive and Embedded Software*. MOMPES ’07. Braga, Portugal, 2007, pp. 115–126. DOI: [10.1109/MOMPES.2007.12](https://doi.org/10.1109/MOMPES.2007.12).
- [GH01] Dimitra Giannakopoulou and Klaus Havelund. *Runtime Analysis of Linear Temporal Logic Specifications*. Tech. rep. August. Research Institute for Advanced Computer Science, 2001, pp. 1–11.
- [Gie+07] Simon Giesecke, Matthias Rohr, Florian Marwede, and Wilhelm Hasselbring. “A Style-based Architecture Modelling Approach for UML 2 Component Diagrams.” In: *Proceedings of the 11th IASTED International Conference on Software Engineering and Applications*. SEA ’07. Cambridge, Massachusetts, 2007, pp. 530–538.
- [GK00] David Garlan and Andrew J. Kompanek. “Reconciling the Needs of Architectural Description with Object-modeling Notations.” In: *Proceedings of the 3rd International Conference on The Unified Modeling Language: Advancing the Standard*. UML’00. York, UK, 2000, pp. 498–512. DOI: [10.1007/3-540-40011-7_37](https://doi.org/10.1007/3-540-40011-7_37).
- [GK07] Abdelouahed Gherbi and Ferhat Khendek. “Consistency of UML/SPT Models.” In: *Proceedings of the 13th International SDL Forum Conference on Design for Dependable Systems*. Vol. 4745. SDL’07. Paris, France, 2007, pp. 203–224. DOI: [10.1007/978-3-540-74984-4_13](https://doi.org/10.1007/978-3-540-74984-4_13).
- [Gla90] Rob J. van Glabbeek. “The Linear Time-Branching Time Spectrum (Extended Abstract).” In: *Proceedings of the Theories of Concurrency: Unification and Extension*. CONCUR ’90. Amsterdam, The Netherlands, 1990, pp. 278–297. DOI: [10.1007/BFb0039066](https://doi.org/10.1007/BFb0039066).
- [Gla93] Rob J. Glabbeek. “The Linear Time - Branching Time Spectrum II.” In: *Proceedings of the 4th International Conference on Concurrency Theory*. CONCUR ’93. Hildesheim, Germany, 1993, pp. 66–81. DOI: [10.1007/3-540-57208-2_6](https://doi.org/10.1007/3-540-57208-2_6).
- [GM05] Nicolas Guelfi and Amel Mammarr. “A Formal Semantics of Timed Activity Diagrams and Its PROMELA Translation.” In: *Proceedings of the 12th Asia-Pacific Software Engineering Conference*. APSEC ’05. Taipei, Taiwan, 2005, pp. 283–290. DOI: [10.1109/APSEC.2005.7](https://doi.org/10.1109/APSEC.2005.7).
- [Got08] Greg Goth. “Ultralarge Systems: Redefining Software Engineering?” In: *Software, IEEE* 25.3 [2008], pp. 91–94. DOI: [10.1109/MS.2008.82](https://doi.org/10.1109/MS.2008.82).
- [Gro11a] Object Management Group. *Business Process Model And Notation*. <http://www.omg.org/spec/BPMN/2.0>. Last accessed: 2017-04-10. 2011.
- [Gro11b] Object Management Group. *UML 2.4.1 Superstructure Specification*. <http://www.omg.org/spec/UML/2.4.1>. Last accessed: 2017-05-20. 2011.

- [GVC15] Stijn Goedertier, Jan Vanthienen, and Filip Caron. “Declarative Business Process Modelling: Principles and Modelling Languages.” In: *Enterprise Information Systems* 9.2 [2015], pp. 161–185. DOI: [10.1080/17517575.2013.830340](https://doi.org/10.1080/17517575.2013.830340).
- [GW89] R J van Glabbeek and W P Weijland. “Branching Time and Abstraction in Bisimulation Semantics (extended abstract).” In: *proceedings of the IFIP 11th world computer congress - Information Processing 89*. IFIP Congress Series. San Francisco, USA, 1989, pp. 613–618.
- [Har87] David Harel. “Statecharts: A Visual Formalism for Complex Systems.” In: *Science of Computer Programming* 8.3 [1987], pp. 231–274. DOI: [10.1016/0167-6423\(87\)90035-9](https://doi.org/10.1016/0167-6423(87)90035-9).
- [Hau+05] Øystein Haugen, Knut Eilif Husa, Ragnhild Kobro Runde, and Ketil Stølen. “STAIRS Towards Formal Design with Sequence Diagrams.” In: *Software & Systems Modeling* 4.4 [2005], pp. 355–357. DOI: [10.1007/s10270-005-0087-0](https://doi.org/10.1007/s10270-005-0087-0).
- [HCW05] Mats P. Heimdahl, Yunja Choi, and Michael W. Whalen. “Deviation Analysis: A New Use of Model Checking.” In: *Automated Software Engineering* 12.3 [2005], pp. 321–347. DOI: [10.1007/s10515-005-2642-x](https://doi.org/10.1007/s10515-005-2642-x).
- [Hev+04] Alan R. Hevner, Salvatore T. March, Jinsoo Park, and Sudha Ram. “Design Science in Information Systems Research.” In: *MIS Quarterly* 28.1 [2004], pp. 75–105.
- [HF06] Henri Habrias and Marc Frappier. *Software Specification Methods - An Overview Using a Case Study*. London, UK: ISTE Ltd., 2006. ISBN: 1905209347.
- [Hid+05] Jan Hidders, Marlon Dumas, Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, and Jan Verelst. “When Are Two Workflows the Same?” In: *Proceedings of the 2005 Australasian Symposium on Theory of Computing - Volume 41*. CATS ’05. Newcastle, Australia, 2005, pp. 3–11.
- [HN96] David Harel and Amnon Naamad. “The STATEMATE Semantics of Statecharts.” In: *ACM Transactions on Software Engineering and Methodology* 5.4 [1996], pp. 293–333. DOI: [10.1145/235321.235322](https://doi.org/10.1145/235321.235322).
- [Hol91] Gerard J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, Inc., 1991. ISBN: 0-13-539925-4.
- [HT06] B. Hailpern and P. Tarr. “Model-driven Development: The Good, the Bad, and the Ugly.” In: *IBM Systems Journal* 45.3 [2006], pp. 451–461. DOI: [10.1147/sj.453.0451](https://doi.org/10.1147/sj.453.0451).

- [Huz+05] Zbigniew Huzar, Ludwik Kuzniarz, Gianna Reggio, and Jean Louis Sourrouille. “Consistency Problems in UML-based Software Development.” In: *Proceedings of the International Conference on UML Modeling Languages and Applications (UML) Satellite Activities – Revised Selected Papers*. Lisbon, Portugal, 2005, pp. 1–12. DOI: [10.1007/978-3-540-31797-5_1](https://doi.org/10.1007/978-3-540-31797-5_1).
- [IG11] Martin Ivarsson and Tony Gorschek. “A Method for Evaluating Rigor and Industrial Relevance of Technology Evaluations.” In: *Empirical Software Engineering* 16.3 [2011], pp. 365–395. DOI: [10.1007/s10664-010-9146-4](https://doi.org/10.1007/s10664-010-9146-4).
- [Jan+99] Wil Janssen, Radu Mateescu, Sjouke Mauw, Peter Fennema, and Petra van der Stappen. “Model Checking for Managers.” In: *Proceedings of the 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking*. 1999, pp. 92–107. DOI: [10.1007/3-540-48234-2_7](https://doi.org/10.1007/3-540-48234-2_7).
- [JCP91] Jay F. Nunamaker Jr., Minder Chen, and Titus D. M. Purdin. “Systems Development in Information Systems Research.” In: *Journal of Management Information Systems* 7.3 [1991], pp. 89–106.
- [JRS04] HoonSang Jin, Kavita Ravi, and Fabio Somenzi. “Fate and Free Will in Error Traces.” In: *International Journal on Software Tools for Technology Transfer* 6.2 [2004], pp. 102–116. DOI: [10.1007/s10009-004-0146-9](https://doi.org/10.1007/s10009-004-0146-9).
- [JS15] Jaco Jacobs and Andrew Simpson. “On a Process Algebraic Representation of Sequence Diagrams.” In: *Proceedings of the 12th International Conference on Software Engineering and Formal Methods*. Vol. 8938. SEFM ’14. Grenoble, France, 2015, pp. 71–85. DOI: [10.1007/978-3-319-15201-1_5](https://doi.org/10.1007/978-3-319-15201-1_5).
- [KA08] Ahmad Waqas Kamal and Paris Avgeriou. “Modeling Architectural Patterns’ Behavior Using Architectural Primitives.” In: *Proceedings of the 2Nd European Conference on Software Architecture*. ECSA ’08. Paphos, Cyprus, 2008, pp. 164–179. DOI: [10.1007/978-3-540-88030-1_13](https://doi.org/10.1007/978-3-540-88030-1_13).
- [KAZ08] Ahmad Waqas Kamal, Paris Avgeriou, and Uwe Zdun. “Modeling variants of architectural patterns.” In: *Proceedings of 13th European Conference on Pattern Languages of Programs*. Vol. 610. EuroPLoP ’08. Irsee, Germany, 2008, pp. 1–23.
- [KB13] Barbara Kitchenham and Pearl Brereton. “A Systematic Review of Systematic Review Process Research in Software Engineering.” In: *Information and Software Technology* 55.12 [2013], pp. 2049–2075. DOI: [10.1016/j.infsof.2013.07.010](https://doi.org/10.1016/j.infsof.2013.07.010).
- [KBB15] Barbara Kitchenham, David Budgen, and Pearl Brereton. *Evidence-Based Software Engineering, Empirical SE, Software Design*. Boca Raton, FL: CRC Press, 2015, p. 399. ISBN: 978-1-4822-2866-3.

- [KC02] Soon-Kyeong Kim and David Carrington. “A Formal Model of the UML Metamodel: The UML State Machine and Its Integrity Constraints.” In: *Proceedings of the 2Nd International Conference of B and Z Users on Formal Specification and Development in Z and B*. Vol. 2272. ZB ’02. Grenoble, France, 2002, pp. 497–516. DOI: [10.1007/3-540-45648-1_26](https://doi.org/10.1007/3-540-45648-1_26).
- [KC07] Barbara Kitchenham and Stuart Charters. *Guidelines for performing Systematic Literature Reviews in Software Engineering*. Tech. rep. EBSE 2007-001. Keele University and Durham University Joint Report, 2007.
- [Kit+09] Barbara Kitchenham, O. Pearl Brereton, David Budgen, Mark Turner, John Bailey, and Stephen Linkman. “Systematic Literature Reviews in Software Engineering - A Systematic Literature Review.” In: *Information and Software Technology* 51.1 [2009], pp. 7–15. DOI: [10.1016/j.infsof.2008.09.009](https://doi.org/10.1016/j.infsof.2008.09.009).
- [Kit04] Barbara Kitchenham. “Procedures for Performing Systematic Reviews.” In: *Keele, UK, Keele University* 33.2004 [2004], pp. 1–26.
- [KL16] Sebastian Krings and Michael Leuschel. “Proof Assisted Symbolic Model Checking for B and Event-B.” In: *Proceedings of the 5th International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z*. ABZ ’16. Linz, Austria, 2016, pp. 135–150. DOI: [10.1007/978-3-319-33600-8_8](https://doi.org/10.1007/978-3-319-33600-8_8).
- [KMR02] Alexander Knapp, Stephan Merz, and Christopher Rauh. “Model Checking - Timed UML State Machines and Collaborations.” In: *Proceedings of the 7th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems: Co-sponsored by IFIP WG 2.2*. FTRTFT ’02. Oldenburg, Germany, 2002, pp. 395–416. DOI: [10.1007/3-540-45739-9_23](https://doi.org/10.1007/3-540-45739-9_23).
- [KPP95] Barbara Kitchenham, Lesley Pickard, and Shari Lawrence Pfleeger. “Case Studies for Method and Tool Evaluation.” In: *IEEE Software* 12.4 [1995], pp. 52–62. DOI: [10.1109/52.391832](https://doi.org/10.1109/52.391832).
- [KT11] Tsutomu Kumazawa and Tetsuo Tamai. “Counter Example-based Error Localization of Behavior Models.” In: *Proceedings of the Third International Conference on NASA Formal Methods*. NFM’11. Pasadena, CA, 2011, pp. 222–236.
- [KTK02] Jana Koehler, Giuliano Tirenni, and Santhosh Kumaran. “From Business Process Model to Consistent Implementation: A Case for Formal Verification Methods.” In: *Proceedings of the Sixth International ENTERPRISE DISTRIBUTED OBJECT COMPUTING Conference*. EDOC ’02. Lausanne, Switzerland, 2002, pp. 96–. DOI: [10.1109/EDOC.2002.1137700](https://doi.org/10.1109/EDOC.2002.1137700).

- [Kwa+15] Pieter M. Kwantes, Pieter Van Gorp, Jetty Kleijn, and Arend Rensink. "Towards Compliance Verification Between Global and Local Process Models." In: *Proceedings of the 8th International Conference on Graph Transformation*. STAF '15. L'Aquila, Italy, 2015, pp. 221–236. DOI: [10.1007/978-3-319-21145-9_14](https://doi.org/10.1007/978-3-319-21145-9_14).
- [Lam07] Vitus S. W. Lam. "A Formalism for Reasoning About UML Activity Diagrams." In: *Nordic journal of Computing* 14.1 [2007], pp. 43–64. ISSN: 1236-6064.
- [Lam08] Vitus S. W. Lam. "Theory for Classifying Equivalences of Unified Modeling Language Activity Diagrams." In: *IET Software* 2.5 [2008], pp. 391–403. DOI: [10.1049/iet-sen:20070045](https://doi.org/10.1049/iet-sen:20070045).
- [Lam80] Leslie Lamport. "Sometime" is Sometimes "Not Never" - On the Temporal Logic of Programs." In: *Proceedings of the 7th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '80. Las Vegas, Nevada, USA, 1980, pp. 174–185. DOI: [10.1145/567446.567463](https://doi.org/10.1145/567446.567463).
- [Ler+10] Barbara Staudt Lerner, Stefan Christov, Leon J. Osterweil, Reda Bendraou, Udo Kannengiesser, and Alexander Wise. "Exception Handling Patterns for Process Modeling." In: *IEEE Transactions on Software Engineering* 36.2 [2010], pp. 162–183. DOI: [10.1109/TSE.2010.1](https://doi.org/10.1109/TSE.2010.1).
- [Lim+09] V. Lima, C. Talhi, D. Mouheb, M. Debbabi, L. Wang, and Makan Pourzandi. "Formal Verification and Validation of UML 2.0 Sequence Diagrams Using Source and Destination of Messages." In: *Electronic Notes in Theoretical Computer Science* 254 [2009], pp. 143–160. DOI: [10.1016/j.entcs.2009.09.064](https://doi.org/10.1016/j.entcs.2009.09.064).
- [LL96] S. Leue and P. B. Ladkin. "Implementing and Verifying MSC Specifications Using Promela/XSpin." In: *In Proceedings of the 2nd International Workshop on the SPIN Verification System, volume 32 of DIMACS Series*. 1996, pp. 65–89.
- [LLD98] Axel van Lamsweerde, Emmanuel Letier, and Robert Darimont. "Managing Conflicts in Goal-Driven Requirements Engineering." In: *IEEE Transactions on Software Engineering* 24.11 [1998], pp. 908–926. DOI: [10.1109/32.730542](https://doi.org/10.1109/32.730542).
- [LMS02] François Laroussinie, Nicolas Markey, and Ph. Schnoebelen. "Temporal Logic with Forgettable Past." In: *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*. LICS '02. Copenhagen, Denmark, 2002, pp. 383–392. DOI: [10.1109/LICS.2002.1029846](https://doi.org/10.1109/LICS.2002.1029846).
- [LMT09] Francisco J. Lucas, Fernando Molina, and Ambrosio Toval. "A Systematic Review of UML Model Consistency Management." In: *Information and Software Technology* 51.12 [2009], pp. 1631–1645. DOI: [10.1016/j.infsof.2009.04.009](https://doi.org/10.1016/j.infsof.2009.04.009).

- [LP05] Vitus S. W. Lam and Julian Padget. “Consistency Checking of Sequence Diagrams and Statechart Diagrams Using the π -Calculus.” In: *Proceedings of the 5th International Conference on Integrated Formal Methods*. IFM’05. Eindhoven, The Netherlands: Springer-Verlag, 2005, pp. 347–365. DOI: [10.1007/11589976_20](https://doi.org/10.1007/11589976_20).
- [LP08] Régine Laleau and Fiona Polack. “Using Formal Metamodels to Check Consistency of Functional Views in Information Systems Specification.” In: *Information and Software Technology* 50.7-8 [2008], pp. 797–814. DOI: [10.1016/j.infsof.2007.10.007](https://doi.org/10.1016/j.infsof.2007.10.007).
- [LW09] Niels Lohmann and Karsten Wolf. “Realizability Is Controllability.” In: *Proceedings of the 6th International Conference on Web Services and Formal Methods*. WS-FM’09. Bologna, Italy, 2009, pp. 110–127. DOI: [10.1007/978-3-642-14458-5_7](https://doi.org/10.1007/978-3-642-14458-5_7).
- [LZP07] Jing Li, Huibiao Zhu, and Geguang Pu. “Conformance Validation between Choreography and Orchestration.” In: *Proceedings of the First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering*. TASE ’07. Shanghai, China, 2007, pp. 473–482. DOI: [10.1109/TASE.2007.16](https://doi.org/10.1109/TASE.2007.16).
- [Mar05] Axel Martens. “Consistency between Executable and Abstract Processes.” In: *Proceedings of the 2005 IEEE International Conference on e-Technology, e-Commerce and e-Service (EEE’05) on e-Technology, e-Commerce and e-Service*. EEE ’05. Hong Kong, China, 2005, pp. 60–67. DOI: [10.1109/EEE.2005.53](https://doi.org/10.1109/EEE.2005.53).
- [Mat15] MathWorks. *Stateflow®: Model and simulate decision logic using state machines and flow charts*. <http://www.mathworks.com/products/stateflow>. Last accessed: 2016-06-01. 2015.
- [MCL04] Jeffrey K. H. Mak, Clifford S. T. Choy, and Daniel P. K. Lun. “Precise Modeling of Design Patterns in UML.” In: *Proceedings of the 26th International Conference on Software Engineering*. ICSE ’04. Scotland, UK, 2004, pp. 252–261. DOI: [10.1109/ICSE.2004.1317447](https://doi.org/10.1109/ICSE.2004.1317447).
- [ME+13] Frank D. Macías-Escrivá, Rodolfo Haber, Raul Del Toro, and Vicente Hernandez. “Self-adaptive Systems: A Survey of Current Approaches, Research Challenges and Applications.” In: *Expert Systems with Applications* 40.18 [2013], pp. 7267–7279. DOI: [10.1016/j.eswa.2013.07.033](https://doi.org/10.1016/j.eswa.2013.07.033).
- [MFP06] Gero Mühl, Ludger Fiege, and Peter Pietzuch. *Distributed Event-Based Systems*. 1st. Secaucus, NJ, USA: Springer, 2006, p. 406. ISBN: 9783540326519.
- [MHS05] Marjan Mernik, Jan Heering, and Anthony M. Sloane. “When and How to Develop Domain-specific Languages.” In: *ACM Computing Surveys* 37.4 [2005], pp. 316–344. DOI: [10.1145/1118890.1118892](https://doi.org/10.1145/1118890.1118892).
- [Mik98] Tommi Mikkonen. “Formalizing Design Patterns.” In: *Proceedings of the 20th International Conference on Software Engineering*. ICSE ’98. Kyoto, Japan, 1998, pp. 115–124. DOI: [10.1109/INMIC.2006.358176](https://doi.org/10.1109/INMIC.2006.358176).

- [Mil89] Robin Milner. *Communication and Concurrency*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1989. ISBN: 0131149849.
- [Min03] John Mingers. “The Paucity of Multimethod Research: A Review of the Information Systems Literature.” In: *Information Systems Journal* 13.3 [2003], pp. 233–250. DOI: [10.1046/j.1365-2575.2003.00143.x](https://doi.org/10.1046/j.1365-2575.2003.00143.x).
- [MM03] Nikunj R. Mehta and Nenad Medvidovic. “Composing Architectural Styles from Architectural Primitives.” In: *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ESEC/FSE-11. Helsinki, Finland, 2003, pp. 347–350. DOI: [10.1145/940071.940118](https://doi.org/10.1145/940071.940118).
- [MT00] Nenad Medvidovic and Richard N. Taylor. “A Classification and Comparison Framework for Software Architecture Description Languages.” In: *IEEE Transactions on Software Engineering* 26.1 [2000], pp. 70–93. DOI: [10.1109/32.825767](https://doi.org/10.1109/32.825767).
- [MTZ14] Faiz UL Muram, Huy Tran, and Uwe Zdun. “Automated Mapping of UML Activity Diagrams to Formal Specifications for Supporting Containment Checking.” In: *Proceedings of the 11th International Workshop on Formal Engineering approaches to Software Components and Architectures, FESCA ’14*. Vol. 147. EPTCS. Grenoble, France, 2014, pp. 93–107. DOI: [10.4204/EPTCS.147.7](https://doi.org/10.4204/EPTCS.147.7).
- [MTZ15] Faiz UL Muram, Huy Tran, and Uwe Zdun. “Counterexample Analysis for Supporting Containment Checking of Business Process Models.” In: *Proceedings of the 13th International Business Process Management Workshops, BPM ’15*. Vol. 256. Lecture Notes in Business Information Processing. Innsbruck, Austria, 2015, pp. 515–528. DOI: [10.1007/978-3-319-42887-1_41](https://doi.org/10.1007/978-3-319-42887-1_41).
- [MTZ16] Faiz UL Muram, Huy Tran, and Uwe Zdun. “A Model Checking Based Approach for Containment Checking of UML Sequence Diagrams.” In: *Proceedings of the 23rd Asia-Pacific Software Engineering Conference*. APSEC ’16. Hamilton, New Zealand, 2016, pp. 73–80. DOI: [10.1109/APSEC.2016.021](https://doi.org/10.1109/APSEC.2016.021).
- [MTZ17a] Faiz UL Muram, Huy Tran, and Uwe Zdun. “Systematic Review of Software Behavioral Model Consistency Checking.” In: *ACM Computing Surveys* 50.2 [2017], pp. 1–39. DOI: [10.1145/3037755](https://doi.org/10.1145/3037755).
- [MTZ17b] Faiz UL Muram, Huy Tran, and Uwe Zdun. “Towards Containment Checking of Behaviour in Architectural Patterns.” In: *Proceedings of the 22nd European Conference on Pattern Languages of Programs*. EuroPlop ’17. Kloster Irsee in Bavaria, Germany, 2017.

- [Mur+17] Faiz UL Muram, Muhammad Atif Javed, Huy Tran, and Uwe Zdun. “Towards a Framework for Detecting Containment Violations in Service Choreography.” In: *Proceedings of the IEEE International Conference on Services Computing*. SCC '17. Honolulu, Hawaii, USA, 2017.
- [Mur89] Tadao Murata. “Petri nets: Properties, analysis and applications.” In: *Proceedings of the IEEE* 77.4 [1989], pp. 541–580. DOI: [10.1109/5.24143](https://doi.org/10.1109/5.24143).
- [MVG06] Tom Mens and Pieter Van Gorp. “A Taxonomy of Model Transformation.” In: *Electronic Notes in Theoretical Computer Science* 152 [2006], pp. 125–142. DOI: [10.1016/j.entcs.2005.10.021](https://doi.org/10.1016/j.entcs.2005.10.021).
- [Nav01] Gonzalo Navarro. “A Guided Tour to Approximate String Matching.” In: *ACM Computing Surveys* 33.1 [2001], pp. 31–88. DOI: [10.1145/375360.375365](https://doi.org/10.1145/375360.375365).
- [Nej+07] Shiva Nejati, Mehrdad Sabetzadeh, Marsha Chechik, Steve Easterbrook, and Pamela Zave. “Matching and Merging of Statecharts Specifications.” In: *Proceedings of the 29th International Conference on Software Engineering*. ICSE '07. Hilton Minneapolis, MN, USA, 2007, pp. 54–64. DOI: [10.1109/ICSE.2007.50](https://doi.org/10.1109/ICSE.2007.50).
- [Nuu95] Esko Nuutila. “Efficient Transitive Closure Computation in Large Digraphs.” In: *Acta Polytechnica Scandinavia: Math. Comput. Eng.* 74 [1995], pp. 1–124. ISSN: 1237-2404.
- [OAS07] OASIS. *Web Services Business Process Execution Language (WSBPEL)*. https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel. Last accessed: 2016-09-01. 2007.
- [Par81] David Park. “Concurrency and Automata on Infinite Sequences.” In: *Proceedings of the 5th GI-Conference on Theoretical Computer Science*. 1981, pp. 167–183. ISBN: 3-540-10576-X.
- [Pef+08] Ken Peffers, Tuure Tuunanen, Marcus A. Rothenberger, and Samir Chatterjee. “A Design Science Research Methodology for Information Systems Research.” In: *Journal of Management Information Systems* 24.3 [2008], pp. 45–77. DOI: [10.2753/MIS0742-1222240302](https://doi.org/10.2753/MIS0742-1222240302).
- [PH08] Francis Jeffry Pelletier and Andrew Hartline. “Ternary Exclusive Or.” In: *Logic Journal of the IGPL* 16.1 [2008], pp. 75–83. DOI: [10.1093/jigpal/jzm027](https://doi.org/10.1093/jigpal/jzm027).
- [Pnu77] Amir Pnueli. “The temporal logic of programs.” In: *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*. SFCS '77. Washington, DC, USA, 1977, pp. 46–57. DOI: [10.1109/SFCS.1977.32](https://doi.org/10.1109/SFCS.1977.32).
- [PS12] Pascal Poizat and Gwen Salaün. “Checking the Realizability of BPMN 2.0 Choreographies.” In: *Proceedings of the 27th Annual ACM Symposium on Applied Computing*. SAC '12. Trento, Italy, 2012, pp. 1927–1934. DOI: [10.1145/2245276.2232095](https://doi.org/10.1145/2245276.2232095).

- [PTT06] Jean-Marc Perronne, Laurent Thiry, and Bernard Thirion. “Architectural Concepts and Design Patterns for Behavior Modeling and Integration.” In: *Mathematics and Computers in Simulation* 70.5-6 [2006], pp. 314–329. DOI: [10.1016/j.matcom.2005.11.004](https://doi.org/10.1016/j.matcom.2005.11.004).
- [RA06] Anne Rozinat and Wil M. P. van der Aalst. “Conformance Testing: Measuring the Fit and Appropriateness of Event Logs and Process Models.” In: *Proceedings of the Third International Conference on Business Process Management*. BPM’05. Nancy, France, 2006, pp. 163–176. DOI: [10.1007/11678564_15](https://doi.org/10.1007/11678564_15).
- [Rab97] Alexander Rabinovich. “Complexity of Equivalence Problems for Concurrent Systems of Finite Agents.” In: *Information and Computation* 139.2 [1997], pp. 111–129. DOI: [10.1006/inco.1997.2661](https://doi.org/10.1006/inco.1997.2661).
- [Roz11] Kristin Y. Rozier. “Survey: Linear Temporal Logic Symbolic Model Checking.” In: *Computer Science Review* 5.2 [2011], pp. 163–203. DOI: [10.1016/j.cosrev.2010.06.002](https://doi.org/10.1016/j.cosrev.2010.06.002).
- [Rup10] Nayan B. Ruparelia. “Software Development Lifecycle Models.” In: *ACM SIGSOFT Software Engineering Notes* 35.3 [2010], pp. 8–13. DOI: [10.1145/1764810.1764814](https://doi.org/10.1145/1764810.1764814).
- [RW03] Holger Rasch and Heike Wehrheim. “Checking Consistency in UML Diagrams: Classes and State Machines.” In: *Proceedings of the 6th IFIP WG 6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems*. FMOODS ’03. Paris, France, 2003, pp. 229–243.
- [RW05] Nick Rozanski and Eóin Woods. *Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives*. Upper Saddle River, NJ: Addison-Wesley Professional, 2005. ISBN: 0321112296.
- [Sch02] August-Wilhelm Scheer. *ARIS — Vom Geschäftsprozess zum Anwendungssystem*. 4th. Springer, 2002. ISBN: 3540658238.
- [Sel98] Bran Selic. “Using UML for Modeling Complex Real-Time Systems.” In: *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*. LCTES ’98. Montreal, Canada, 1998, pp. 250–260. DOI: [10.1007/BFb0057795](https://doi.org/10.1007/BFb0057795).
- [SG96] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1996. ISBN: 0-13-182957-2.
- [SH04] Neelam Soundarajan and Jason O. Hallstrom. “Responsibilities and Rewards: Specifying Design Patterns.” In: *Proceedings of the 26th International Conference on Software Engineering*. ICSE ’04. Scotland, UK, 2004, pp. 666–675. DOI: [10.1109/ICSE.2004.1317488](https://doi.org/10.1109/ICSE.2004.1317488).

- [Shi06] Yoshiyuki Shinkawa. “Inter-Model Consistency in UML Based on CPN Formalism.” In: *Proceedings of the XIII Asia Pacific Software Engineering Conference*. APSEC ’06. Bangalore, India, 2006, pp. 411–418. DOI: [10.1109/APSEC.2006.41](https://doi.org/10.1109/APSEC.2006.41).
- [SKM01] Timm Schäfer, Alexander Knapp, and Stephan Merz. “Model Checking UML State Machines and Collaborations.” In: *Electronic Notes in Theoretical Computer Science* 55.3 [2001], pp. 357–369. DOI: [10.1016/S1571-0661\(04\)00262-2](https://doi.org/10.1016/S1571-0661(04)00262-2).
- [SLG13] Karina Sokolova, Marc Lemercier, and Ludovic Garcia. “Android Passive MVC: a Novel Architecture Model for the Android Application Development.” In: *Proceedings of the Fifth International Conferences on Pervasive Patterns and Applications*. PATTERNS 2013. Valencia, Spain, 2013.
- [SN00] August-Wilhelm Scheer and Markus Nüttgens. “ARIS Architecture and Reference Models for Business Process Management.” In: *Business Process Management, Models, Techniques, and Empirical Studies - Part III*. 2000, pp. 376–389. DOI: [10.1007/3-540-45594-9_24](https://doi.org/10.1007/3-540-45594-9_24).
- [Som+12] Ian Sommerville, Dave Cliff, Radu Calinescu, Justin Keen, Tim Kelly, Marta Kwiatkowska, John Mcdermid, and Richard Paige. “Large-scale Complex IT Systems.” In: *Communications of the ACM* 55.7 [2012], pp. 71–77. DOI: [10.1145/2209249.2209268](https://doi.org/10.1145/2209249.2209268).
- [SS00] Markus Stumptner and Michael Schrefl. “Behavior Consistent Inheritance in UML.” In: *Proceedings of the 19th International Conference on Conceptual Modeling*. Vol. 1920. ER’00. Salt Lake City, Utah, USA, 2000, pp. 527–542. DOI: [10.1007/3-540-45393-8_38](https://doi.org/10.1007/3-540-45393-8_38).
- [SSMJ15] Ellis Solaiman, Wenzhong Sun, and Carlos Molina-Jimenez. “A Tool for the Automatic Verification of BPMN Choreographies.” In: *Proceedings of the 2015 IEEE International Conference on Services Computing*. SCC ’15. New York, USA, 2015, pp. 728–735. DOI: [10.1109/SCC.2015.103](https://doi.org/10.1109/SCC.2015.103).
- [Str+03] Ragnhild van der Straeten, Tom Mens, Jocelyn Simmonds, and Viviane Jonckers. “Using description logic to maintain consistency between UML models.” In: *Proceedings of the 6th International Conference on The Unified Modeling Language, Modeling Languages and Applications*. San Francisco, CA, USA, 2003, pp. 326–340. DOI: [10.1007/978-3-540-45221-8_28](https://doi.org/10.1007/978-3-540-45221-8_28).
- [Str05] Ragnhild van der Straeten. “Inconsistency Management in Model-Driven Engineering An Approach using Description Logics.” Doctoral Dissertation. Vrije Universiteit Brussel, 2005.
- [Stö04] Harald Störrle. “Trace semantics of interactions in UML 2.0.” In: *Visual Languages and Computing* [2004].

- [Stö14] Harald Störrle. “On the Impact of Layout Quality to Understanding UML Diagrams: Size Matters.” In: *Proceedings of the 17th International Conference on Model-Driven Engineering Languages and Systems*. MoDELS ’14. Valencia, Spain, 2014, pp. 518–534. DOI: [10.1007/978-3-319-11653-2_32](https://doi.org/10.1007/978-3-319-11653-2_32).
- [SV06] Thomas Stahl and Markus Völter. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, 2006. ISBN: 978-0-470-02570-3.
- [SZ01] George Spanoudakis and Andrea Zisman. “Inconsistency management in software engineering: Survey and open research issues.” In: *Handbook of Software Engineering and Knowledge Engineering*. Singapore: World Scientific, 2001, pp. 329–380.
- [Tai+04] H. Tai, K. Mitsui, T. Nerome, M. Abe, K. Ono, and M. Hori. “Model-driven Development of Large-scale Web Applications.” In: *IBM Journal of Research and Development* 48.5/6 [2004], pp. 797–809. DOI: [10.1147/rd.485.0797](https://doi.org/10.1147/rd.485.0797).
- [Tar72] Robert Endre Tarjan. “Depth-First Search and Linear Graph Algorithms.” In: *SIAM Journal on Computing* 1.2 [1972], pp. 146–160. DOI: [10.1137/0201010](https://doi.org/10.1137/0201010).
- [TC02] Li Tan and Rance Cleaveland. “Evidence-Based Model Checking.” In: *Proceedings of the 14th International Conference on Computer Aided Verification*. Vol. 2404. CAV ’02. 2002, pp. 455–470. DOI: [10.1007/3-540-45657-0_37](https://doi.org/10.1007/3-540-45657-0_37).
- [TE00] Aliki Tsiolakis and Hartmut Ehrig. “Consistency Analysis of UML Class and Sequence Diagrams using Attributed Graph Grammars.” In: *Proceedings of the Joint APPLIGRAPH/GETGRATS Workshop on Graph Transformation Systems*. Berlin, 2000, pp. 77–86.
- [TMH08] Yann Thierry-Mieg and Lom-Messan Hillah. “UML Behavioral Consistency Checking using Instantiable Petri Nets.” In: *Innovations in Systems and Software Engineering* 4.3 [2008], pp. 293–300. DOI: [10.1007/s11334-008-0065-0](https://doi.org/10.1007/s11334-008-0065-0).
- [Tra+11] Huy Tran, Ta’id Holmes, Uwe Zdun, and Schahram Dustdar. “Using Model-Driven Views and Trace Links to Relate Requirements and Architecture: A Case Study.” In: *Relating Software Requirements and Architectures*. 2011, pp. 233–255. DOI: [10.1007/978-3-642-21001-3_14](https://doi.org/10.1007/978-3-642-21001-3_14).
- [Tra+12] Huy Tran, Uwe Zdun, Ta’id Holmes, Ernst Oberortner, Emmanuel Mulo, and Schahram Dustdar. “Compliance in Service-oriented Architectures: A Model-driven and View-based Approach.” In: *Information and Software Technology* 54.6 [2012], pp. 531–552. DOI: [10.1016/j.infsof.2012.01.001](https://doi.org/10.1016/j.infsof.2012.01.001).
- [Tru+09] Ninh-Thuan Truong, Thi-Mai-Thuong Tran, Van-Khanh To, and Viet-Ha Nguyen. “Checking the Consistency Between UCM and PSM Using a Graph-Based Method.” In: *Proceedings of the 2009 First Asian Conference on Intelligent Information and Database Systems*. ACIIDS ’09. Dong Hoi, Vietnam, 2009, pp. 190–195. DOI: [10.1109/ACIIDS.2009.66](https://doi.org/10.1109/ACIIDS.2009.66).

- [TUMZ15] Huy Tran, Faiz UL Muram, and Uwe Zdun. “A Graph-Based Approach for Containment Checking of Behavior Models of Software Systems.” In: *Proceedings of the 2015 IEEE 19th International Enterprise Distributed Object Computing Conference*. EDOC ’15. Adelaide, Australia, 2015, pp. 84–93. DOI: [10.1109/EDOC.2015.22](https://doi.org/10.1109/EDOC.2015.22).
- [TZD10] Huy Tran, Uwe Zdun, and Schahram Dustdar. “Name-Based View Integration for Enhancing the Reusability in Process-Driven SOAs.” In: *Proceedings of the 8th Business Process Management Workshops- BPM 2010*. Vol. 66. Lecture Notes in Business Information Processing. Hoboken, NJ, USA, 2010, pp. 338–349. DOI: [10.1007/978-3-642-20511-8_32](https://doi.org/10.1007/978-3-642-20511-8_32).
- [Usm+08] Muhammad Usman, Aamer Nadeem, Tai-hoon Kim, and Eun-suk Cho. “A Survey of Consistency Checking Techniques for UML Models.” In: *Proceedings of the 2008 Advanced Software Engineering and Its Applications*. ASEA ’08. Hainan Island, China, 2008, pp. 57–62. DOI: [10.1109/ASEA.2008.40](https://doi.org/10.1109/ASEA.2008.40).
- [VK07] Vijay K. Vaishnavi and William Kuechler Jr. *Design Science Research Methods and Patterns: Innovating Information and Communication Technology*. 1st. Boston, MA, USA: Auerbach Publications, 2007. ISBN: 1420059327, 9781420059328.
- [Wan+05] Hongyuan Wang, Tie Feng, Jiachen Zhang, and Ke Zhang. “Consistency Check Between Behaviour Models.” In: *Proceedings of the IEEE International Symposium on Communications and Information Technology*. Vol. 1. ISCIT ’05. 2005, pp. 486–489. DOI: [10.1109/ISCIT.2005.1566899](https://doi.org/10.1109/ISCIT.2005.1566899).
- [Wan+10] Jianmin Wang, Tengfei He, Lijie Wen, and Nianhua Wu. “A Behavioral Similarity Measure Between Labeled Petri Nets Based on Principal Transition Sequences.” In: *Proceedings of the 2010 International Conference on On the Move to Meaningful Internet Systems - Volume Part I*. OTM’10. Hersonissos, Crete, Greece, 2010, pp. 394–401. DOI: [10.1007/978-3-642-16934-2_27](https://doi.org/10.1007/978-3-642-16934-2_27).
- [War62] Stephen Warshall. “A Theorem on Boolean Matrices.” In: *Journal of the ACM* 9.1 [1962], pp. 11–12. DOI: [10.1145/321105.321107](https://doi.org/10.1145/321105.321107).
- [WB13] Neil Walkinshaw and Kirill Bogdanov. “Automated Comparison of State-Based Software Models in Terms of Their Language and Structure.” In: *ACM Transactions on Software Engineering and Methodology* 22.2 [2013], pp. 1–37. DOI: [10.1145/2430545.2430549](https://doi.org/10.1145/2430545.2430549).
- [WDW12] M. Weidlich, R. Dijkman, and M. Weske. “Behaviour Equivalence and Compatibility of Business Process Models with Complex Correspondences.” In: *The Computer Journal* 55.11 [2012], pp. 1398–1418. DOI: [10.1093/comjnl/bxs014](https://doi.org/10.1093/comjnl/bxs014).
- [Wol83] Pierre Wolper. “Temporal Logic Can Be More Expressive.” In: *Information and Control* 56.1–2 [1983], pp. 72–99. DOI: [10.1016/S0019-9958\(83\)80051-5](https://doi.org/10.1016/S0019-9958(83)80051-5).

- [WZ09] Andrzej Wasylkowski and Andreas Zeller. “Mining Temporal Specifications from Object Usage.” In: *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*. ASE '09. Auckland, New Zealand, 2009, pp. 295–306. DOI: [10.1109/ASE.2009.30](https://doi.org/10.1109/ASE.2009.30).
- [Yeu04] W. L. Yeung. “Checking Consistency between UML Class and State Models Based on CSP and B.” In: *Journal of Universal Computer Science* 10.11 [2004], pp. 1540–1559. DOI: [10.3217/jucs-010-11-1540](https://doi.org/10.3217/jucs-010-11-1540).
- [Yeu07] W.L. Yeung. “CSP-Based Verification for Web Service Orchestration and Choreography.” In: *Simulation* 83.1 [2007], pp. 65–74. DOI: [10.1177/0037549707079227](https://doi.org/10.1177/0037549707079227).
- [Yin08] Robert K. Yin. *Case Study Research: Design and Methods (Applied Social Research Methods)*. Fourth Edition. Sage Publications, 2008. ISBN: 1412960991.
- [YS06] Shuzhen Yao and Sol Shatz. “Consistency Checking of UML Dynamic Models Based on Petri Net Techniques.” In: *Proceedings of the 15th International Conference on Computing*. CIC '06. Mexico City, Mexico, 2006, pp. 289–297. DOI: [10.1109/CIC.2006.32](https://doi.org/10.1109/CIC.2006.32).
- [Yu+06] Jian Yu, Tan Phan Manh, Jun Han, Yan Jin, Yanbo Han, and Jianwu Wang. “Pattern Based Property Specification and Verification for Service Composition.” In: *Proceedings of the 7th International Conference on Web Information Systems*. WISE'06. Wuhan, China, 2006, pp. 156–168. DOI: [10.1007/11912873_18](https://doi.org/10.1007/11912873_18).
- [ZA05] Uwe Zdun and Paris Avgeriou. “Modeling Architectural Patterns Using Architectural Primitives.” In: *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. OOPSLA '05. San Diego, CA, USA, 2005, pp. 133–146. DOI: [10.1145/1094811.1094822](https://doi.org/10.1145/1094811.1094822).
- [ZA08] Uwe Zdun and Paris Avgeriou. “A Catalog of Architectural Primitives for Modeling Architectural Patterns.” In: *Information and Software Technology* 50.9-10 [2008], pp. 1003–1034. DOI: [10.1016/j.infsof.2007.09.003](https://doi.org/10.1016/j.infsof.2007.09.003).
- [ZABT11] He Zhang, Muhammad Ali-Babar, and Paolo Tell. “Identifying Relevant Studies in Software Engineering.” In: *Information and Software Technology* 53.6 [2011], pp. 625–637. DOI: [10.1016/j.infsof.2010.12.010](https://doi.org/10.1016/j.infsof.2010.12.010).
- [Zah+06] Johannes Maria Zaha, Marlon Dumas, Arthur ter Hofstede, Alistair Barros, and Gero Decker. “Service Interaction Modeling: Bridging Global and Local Views.” In: *Proceedings of the 10th IEEE International Enterprise Distributed Object Computing Conference*. EDOC '06. Hong Kong, China, 2006, pp. 45–55. DOI: [10.1109/EDOC.2006.50](https://doi.org/10.1109/EDOC.2006.50).